Proposition de réponse

RAMTIN AMIN

PCAP

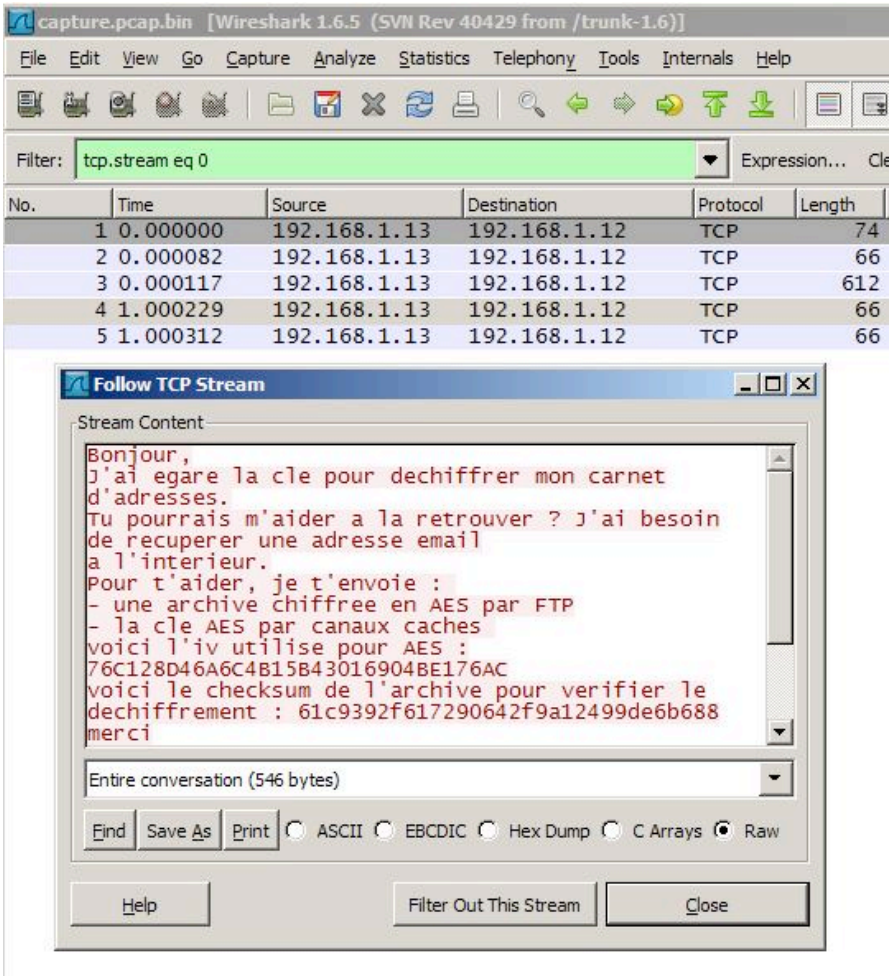
Le challenge SSTIC 2013 consiste à analyser un fichier de capture réseau :

```
$ file dump.bin
dump.bin: tcpdump capture file (little-endian) - version 2.4
```

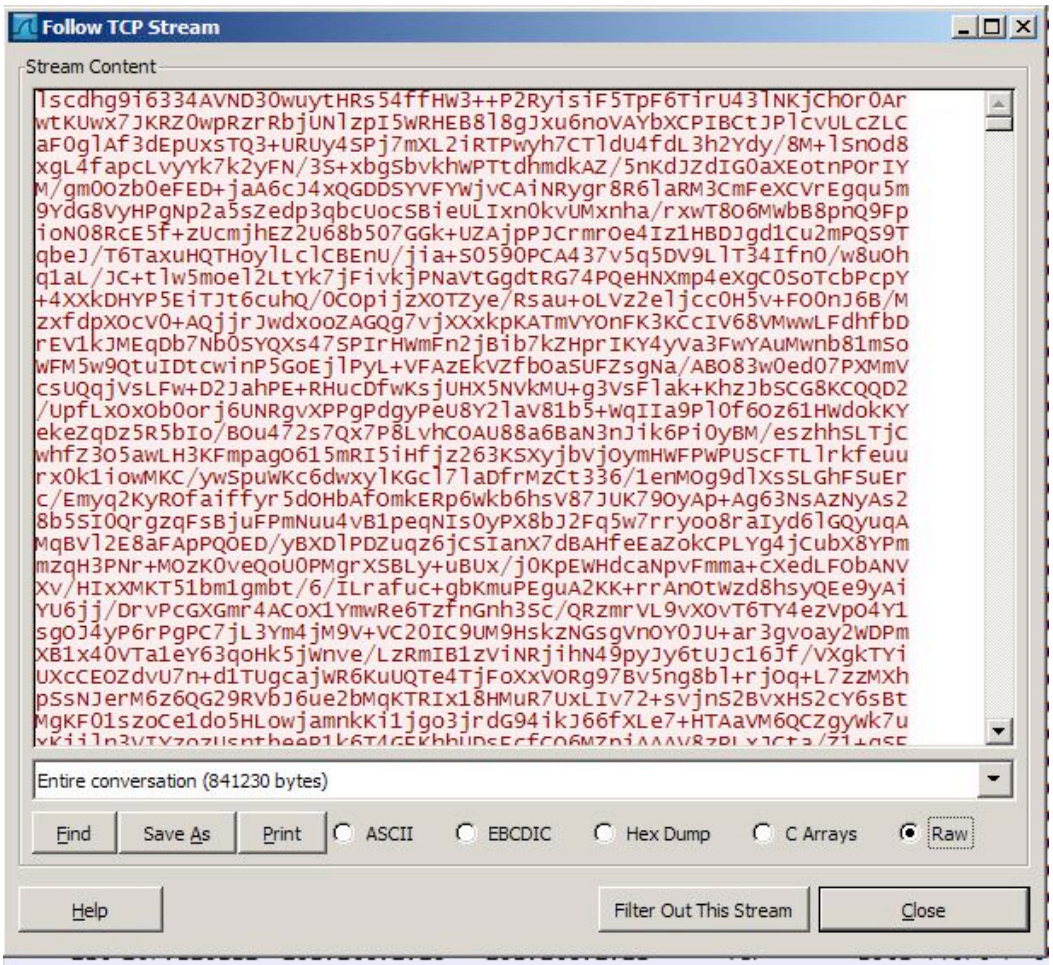
Afin de visualiser le contenu, on va pouvoir utiliser Wireshark et voir que le contenue du fichier était en fait composé de

- 1 échange TCP
- 65 paquets ICMP
- 1 échange FTP

Les premiers paquets TCP contenait une indication que nous pouvons visualiser ci dessous:



L'échange FTP:



Il contenait un fichier sstic.tgz-chiffre qui est encodé en base64

Nous pouvons directement enregistrer ce fichier afin de pouvoir le décoder plus tard. Notons que Openssl permet de décoder directement un fichier base64 avec la commande "-a"

-a/-base64 base64 encode/decode, depending on encryption flag

Les 65 paquets ICMP:

No.	Time	Protocol	Info
6	1.030155	ICMP	Echo (ping) request id=0xf132, seq=1/256, ttl=30
7	3.042155	ICMP	Echo (ping) request id=0x0233, seq=1/256, ttl=30
8	5.055383	ICMP	Echo (ping) request id=0x1333, seq=1/256, ttl=40
9	7.068428	ICMP	Echo (ping) request id=0x2433, seq=1/256, ttl=30
10	9.082048	ICMP	Echo (ping) request id=0x3533, seq=1/256, ttl=20
11	11.095455	ICMP	Echo (ping) request id=0x4633, seq=1/256, ttl=10
12	12.108802	ICMP	Echo (ping) request id=0x5533, seq=1/256, ttl=30
13	14.122642	ICMP	Echo (ping) request id=0x6633, seq=1/256, ttl=30
14	15.136050	ICMP	Echo (ping) request id=0x7533, seq=1/256, ttl=10
15	16.149389	ICMP	Echo (ping) request id=0x8433, seq=1/256, ttl=20
16	18.162944	ICMP	Echo (ping) request id=0x9533, seq=1/256, ttl=20
17	19.176277	ICMP	Echo (ping) request id=0xa433, seq=1/256, ttl=40
18	21.189804	ICMP	Echo (ping) request id=0xb533, seq=1/256, ttl=10
19	23.203362	ICMP	Echo (ping) request id=0xc633, seq=1/256, ttl=10

```
⊞ Frame 6: 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
⊟ Internet Protocol Version 4, Src: 192.168.1.13 (192.168.1.13), Dst: 192.168.
    Version: 4
    Header length: 20 bytes
  ⊟ Differentiated Services Field: 0x02 (DSCP 0x00: Default; ECN: 0x02: ECT(0)
      0000 00.. = Differentiated Services Codepoint: Default (0x00)
      .... ..10 = Explicit Congestion Notification: ECT(0) (ECN-Capable Transp
```

Nous pouvons déjà remarquer plusieurs points sur les 64 premiers paquets:

- Les paquets sont espacés d'à peu près 1 ou 2 secondes.

- Le deuxième nibble de l'ID augmente si l'écart entre 2 paquets est de 2 secondes, sinon il diminue

- Les TTL sont de 10, 20, 30 ou 40
- Au niveau de l'octet du Differenciate Service Field (DSF), nous avons soit "100" ou "010"

Étant à la recherche d'une clef de 256 bits, soit 32 octets, nous sommes à la recherche de 4 bits par paquet. L'indication que nous avons est que ces 4 bits se retrouveront sous forme de canaux dont l'un est temporel.

Sachant que l'écart de temps entre 2 paquets est souligné, nous pouvons considérer que celui ci est temporel, mais ne savons pas si 1 seconde représente un "1" ou un "0". Cela crée donc 2 possibilités.

Il y a 65 paquets donc 64 intervalles, mais nous savons pas si le premier intervalle de temps code pour le premier paquet ou le second. dans le second cas, il faudra tenter de deviner si le premier intervalle était un "0" ou un "1" car il n'y a pas de paquet ICMP avant le paquet numéro 6.

Les TTL étant de 10 20 30 ou 40, il y a donc 4 possibilités soit 2 bits. Mais nous savons pas comment la correspondance est faite. il y a 4! = 4 x 3 x 2 x 1 = 24 possibilités ici.

En ce qui concerne le DSF nous savons pas lequel des 2 bits est choisi. cela représente 2 possibilités.

J'ai utilisé scapy afin de charger le pcap et de tenter les differente possibilité. Il en resulte

Pour le TTL:

TTL	Binary
10	b01
20	b11
30	b10
40	b00

Pour le canal temporel, une difference de 2 seconde vaux 1, et une difference de 1 seconde vaux 0.

Pour le DSF, la valeur 0x02 vaut 1, et la valeur 0x04 vaux 0

Il est à noter que le canal temporel est décalé. Il faudra donc regarder au niveau du deuxième paquet ICMP pour avoir le bit correspondant au premier paquets

Temps (packet n+1)	TTL	DSF
1	(30) b10	1
1	(30) b10	1
1	(40) b00	0
1	(30) b10	0
1	(20) b11	1
0	(10) b01	0
1	(30) b10	1
0	(30) b10	1
0	(10) b01	0

Lorsque le bruteforce est terminé, la clef dont le hash est le bon est la suivante:
dd8cf2d52e69aafb734e3acd0e4a69e83ed93bc4870ecd0d5b6faad86a63ae94

Il est donc possible de déchiffrer le fichier avec openssl:

```
$ openssl enc -d -aes-256-cbc -a -in challenge.b64 -K
dd8cf2d52e69aafb734e3acd0e4a69e83ed93bc4870ecd0d5b6faad86a63ae94 -iv
76C128D46A6C4B15B43016904BE176AC -out archive.tar.gz
```

FPGA

Nous pouvons maintenant déchiffrer l'archive.

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

Nous sommes en présence de 4 fichiers:

- smp.py est un tableau de 231 valeurs
- data est un fichier crypté qui n'a pas l'air d'etre lisible
- s.ngr est un fichier RTL représentant le schematic d'un circuit dans un FPGA

- decrypt.py est un fichier python dont nous allons regarder le fonctionnement

decrypt.py import "dev"

Il fait ensuite une permutation de la clef de tel sorte que le dernier charactere se retrouve en premier, et vice versa:

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

Le script initialise alors un "device", lui transmet d'une part la clef permutée, une taille, et 224 octet provenant de data. Et il le transmet aussi la totalité du tableau contenu dans le fichier smp.py. avant de récupérer et concaténer le tout avec le tour de piste précédent.

```
1.  dev.init("sp.ngr")
2.  for i in range(0, len(d), 224):
3.    smd = d[i : (i + 224)]
4.    smd = (key, len(smd), smd)
5.    dev.send_smd(smd)
6.    dev.send_smp(smp.smp)
7.    dev.start()
8.    dev.wait_finished()
9.    result = result + dev.get_data()
```

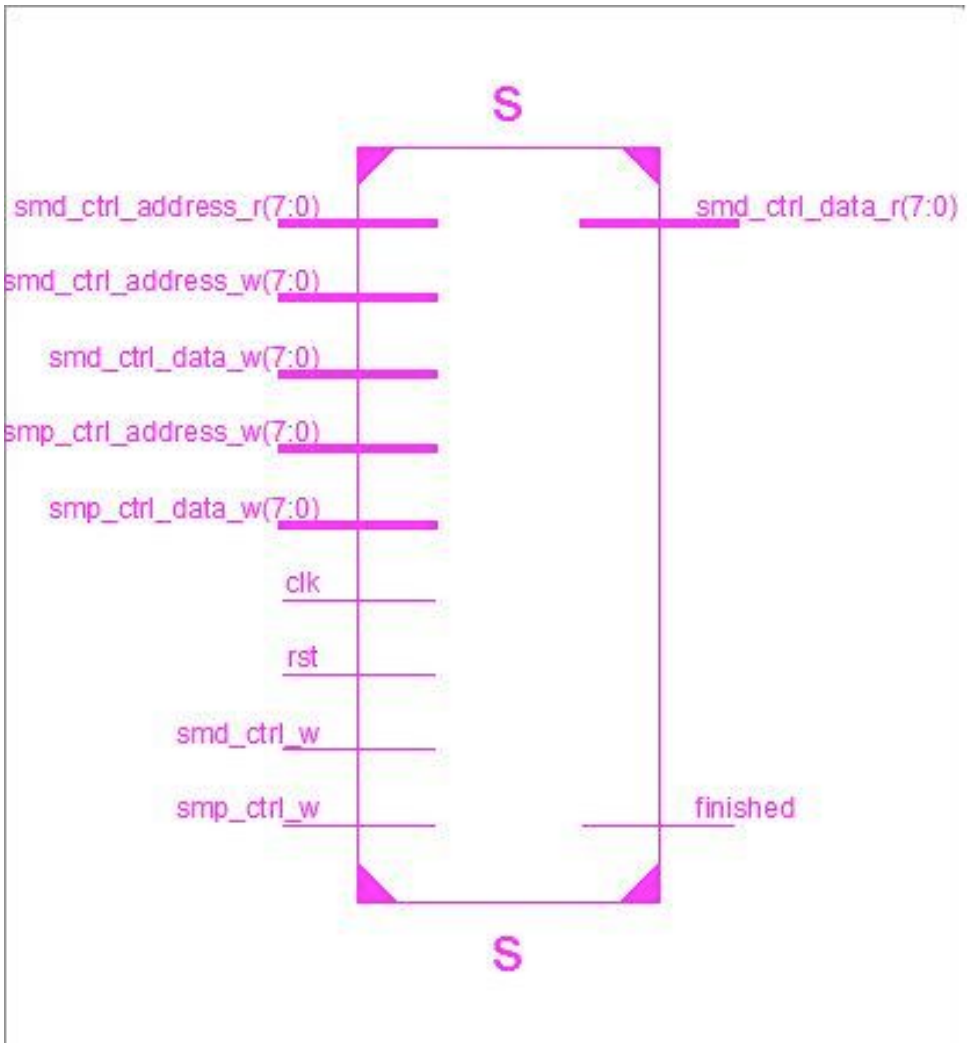
A ce stade, nous pouvons penser qu'il faudrait émuler le FPGA avec un logiciel tel que Icarus Verilog, et a l'aide d'un VPI implémenter le "dev". Mais il nous manquerai les fichier verilog.

Il est temps d'aller télécharger les 10 GB d'ISE Suite de chez Xilinx avant que d'autres participants arrivent à ce stade et commencent à saturer les liens d'Akamaï ;-)

Il existe alors des solution pour régénérer les fichier Verilog ou VHDL a l'aide de fichier de type NGC, mais pas avec les fichier NGR. La libXdm.so pourrait charger le fichier de type XDB (Xilinx Database) mais on pourrait alors tenter d'aller lire le schématique directement et d'en comprendre son fonctionnement !

C'est ce que nous allons faire.

Un brief apperçu du TOP LEVEL nous montre que le block prend en entré des liens estampillés "ctrl" qui nous font penser à "contrôle".

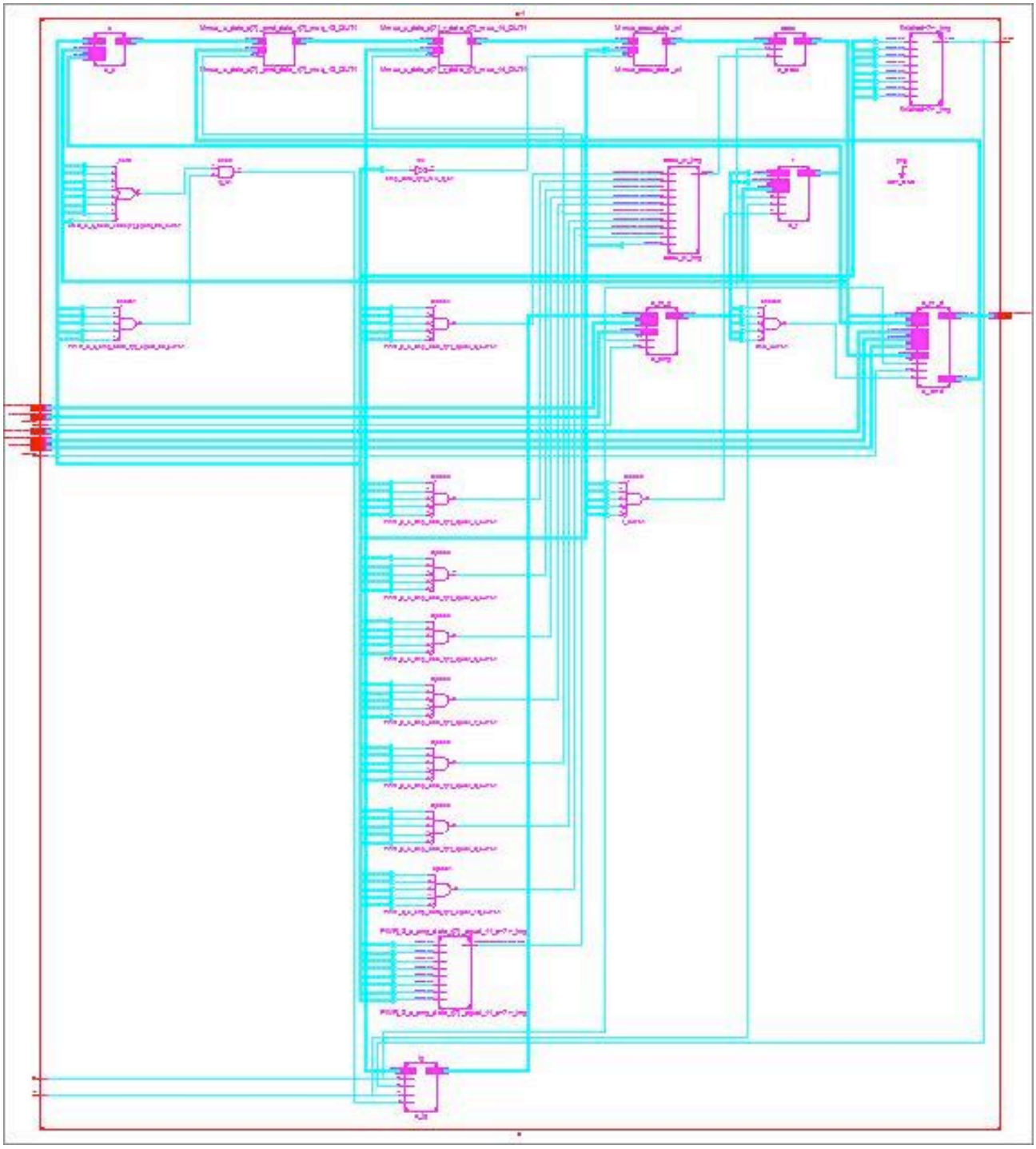


Ca serait par ces port que la connectique a “dev” se fait et que les block SMD et SMP serait transmis. un “clk” et “rst” sont la pour mettre a zéro le système et le cadencer. Et une patte “finished” nous indiquerai que le travail est fait.

Précédement, le coredump de ISE lorsque le fichier était chargé nous montre aussi ces strings:

```
/home/ealata/Recherche/SSTIC.CHALLENGE/challenge/proc/impl/proc/s.vhd
/home/ealata/Recherche/SSTIC.CHALLENGE/challenge/proc/impl/proc/s_m_d.vhd
/home/ealata/Recherche/SSTIC.CHALLENGE/challenge/proc/impl/proc/s_m_p.vhd
/home/ealata/Recherche/SSTIC.CHALLENGE/challenge/proc/impl/proc/accu.vhd
/home/ealata/Recherche/SSTIC.CHALLENGE/challenge/proc/impl/proc/r.vhd
/home/ealata/Recherche/SSTIC.CHALLENGE/challenge/proc/impl/proc/ip.vhd
/home/ealata/Recherche/SSTIC.CHALLENGE/challenge/proc/impl/proc/u.vhd
```

Nous somme probablement sur la piste d’un “proc”. Voyons maintenant ce qu’il y a à l'intérieur:



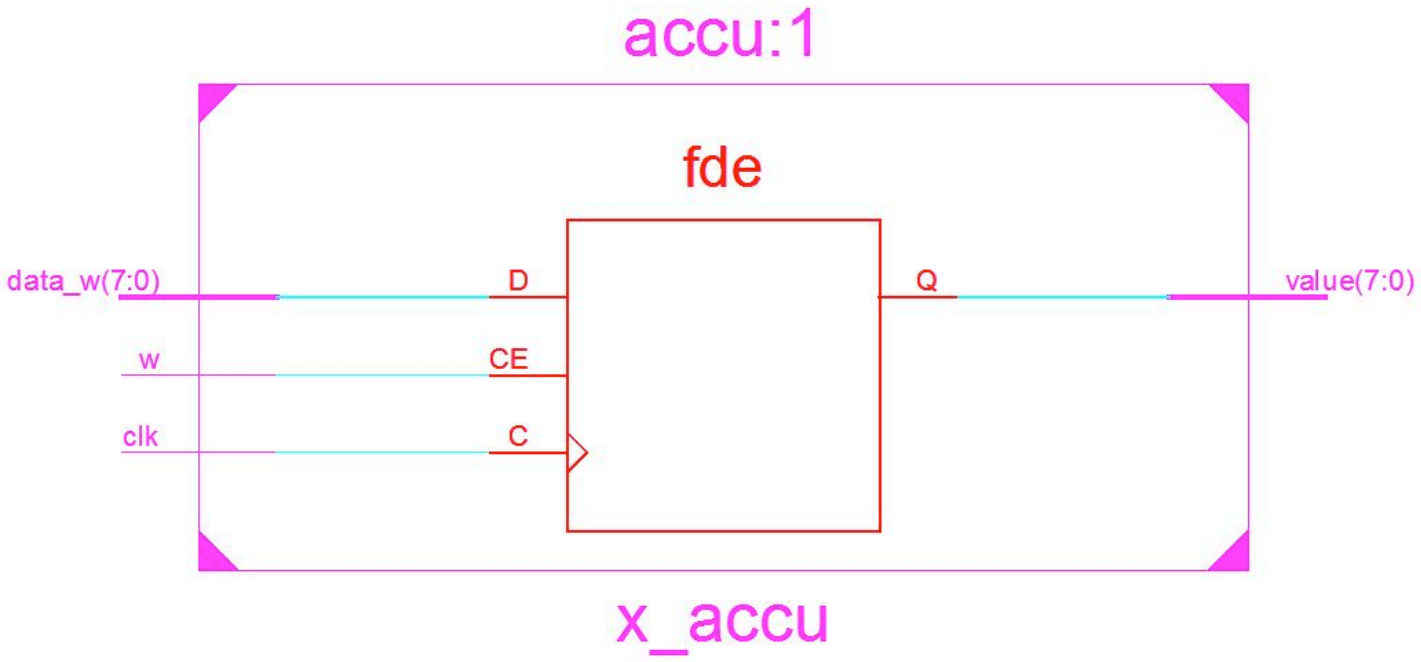
Nous sommes en présence d'un peu de porte logique et de block de niveau inferieur:

- u
- ip
- r
- accu
- smd
- smp

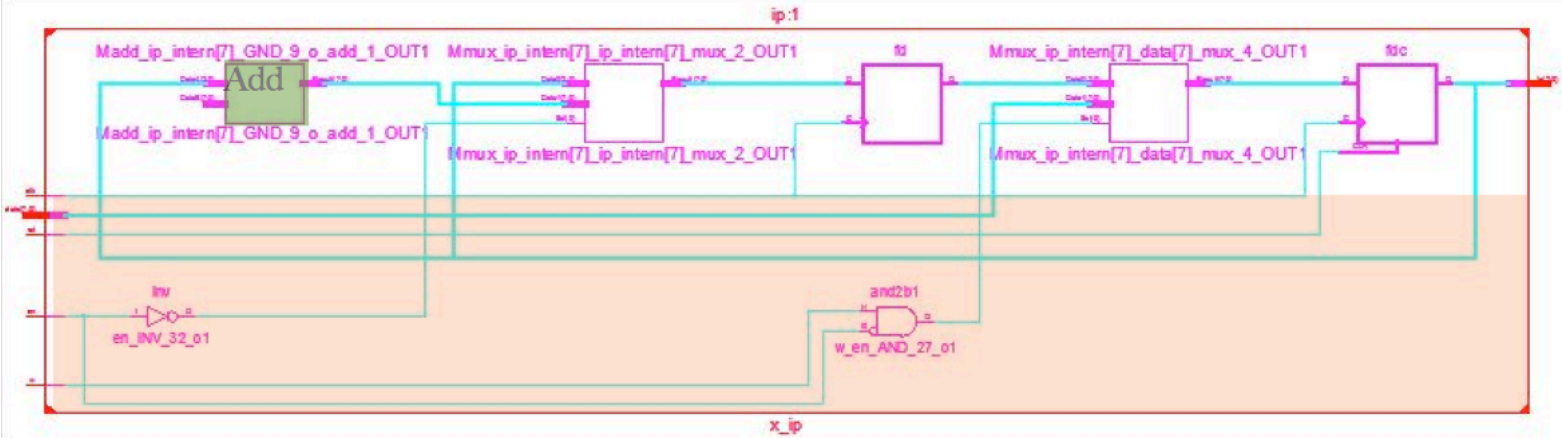
Il faudra alors aller regarder le fonctionnement de chacun pour pouvoir aller plus loin

Block accu:

Celui ci est relativement simple puisque le block utilise 8 bascule D pour en faire un accumulateur de 8 bits. Lorsque le pin "w" est a 1, un front montant sur C place dans "value" ce qu'il y avait dans "data_w".



Block IP:



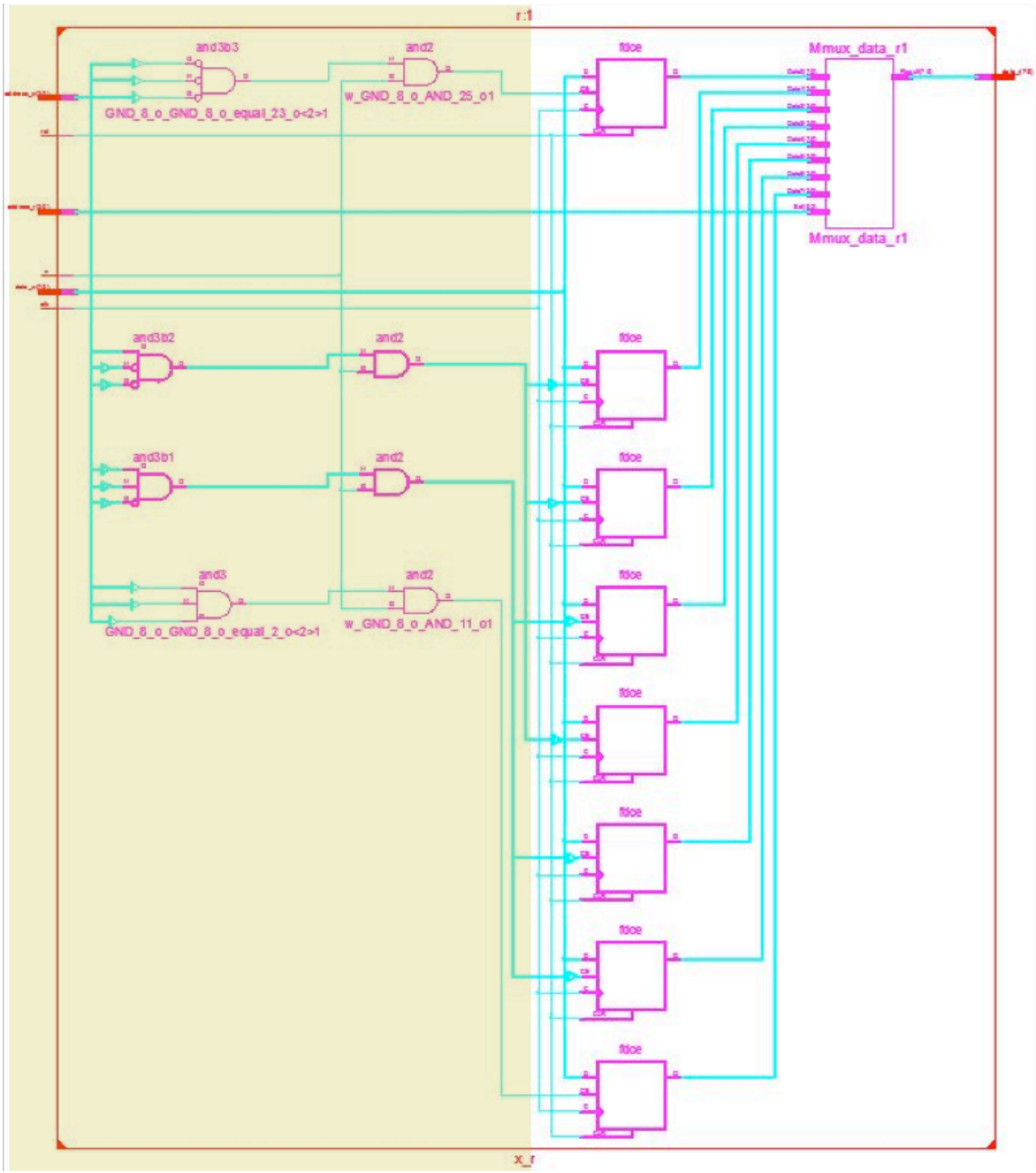
Comme son nom le dit, celui ci est un “Instruction Pointer”. Son fonctionnement est le suivant:

Nous avons un block en bas qui verifie que “finished” n’est pas actif, et qui a chaque coup d’horloge utilise un Additionneur (en vert) qui prend la valeur de sortie et lui rajoute “1” sauf si le pin “w” est actif auquel cas il met directement en sortie ce qu’il y a présent en entrée. Autrement dit, le block permet a chaque coup d’horloge d’incrémenter la valeur de sortie ou placer une valeur immédiate.

Block R:

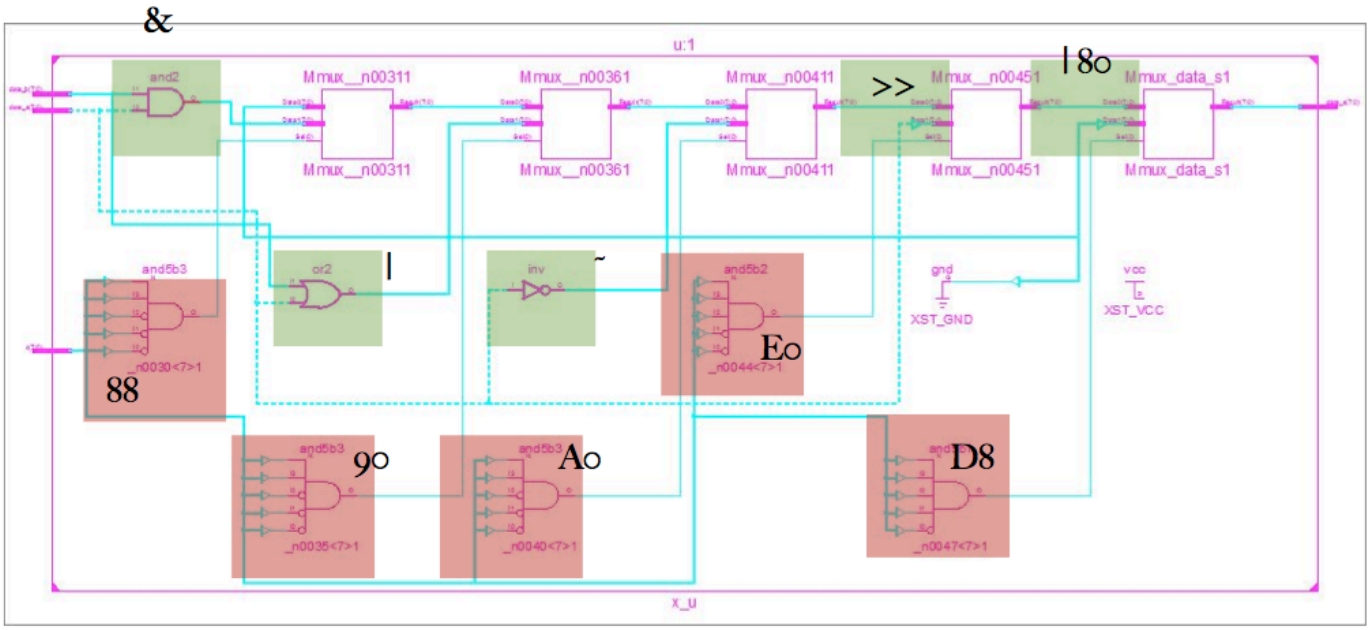
Le block R contient 8 Registres. Il est fait d’une partie de décodage d’un bus de 3 bits représenté ici en jaune. 8Accumulateur sont alors relié a un multiplexeur dont le bus de sélection permet de mettre en sortie ce qu’il y avait sur l’accumulateur sélectionné.

Ainsi on place une valeur dans un des accumulateur, et en choisissant “sel”, on récupère la valeur de l’accumulateur que l’on voulais.



Block U:

Ce block a 2 entrées: data_a et data_b d'une part. Il prend sur 5 bits une commande "C" et possède une sortie. En rouge, nous voyons des décodage d'adresse qui se font grâce à des portes AND. En vert, nous voyons l'opération faite. Le bloque U se présente alors comme un ALU. Il décode une opération qu'il fait avec ses entré data_a et data_b et place le tout sur la sortie.

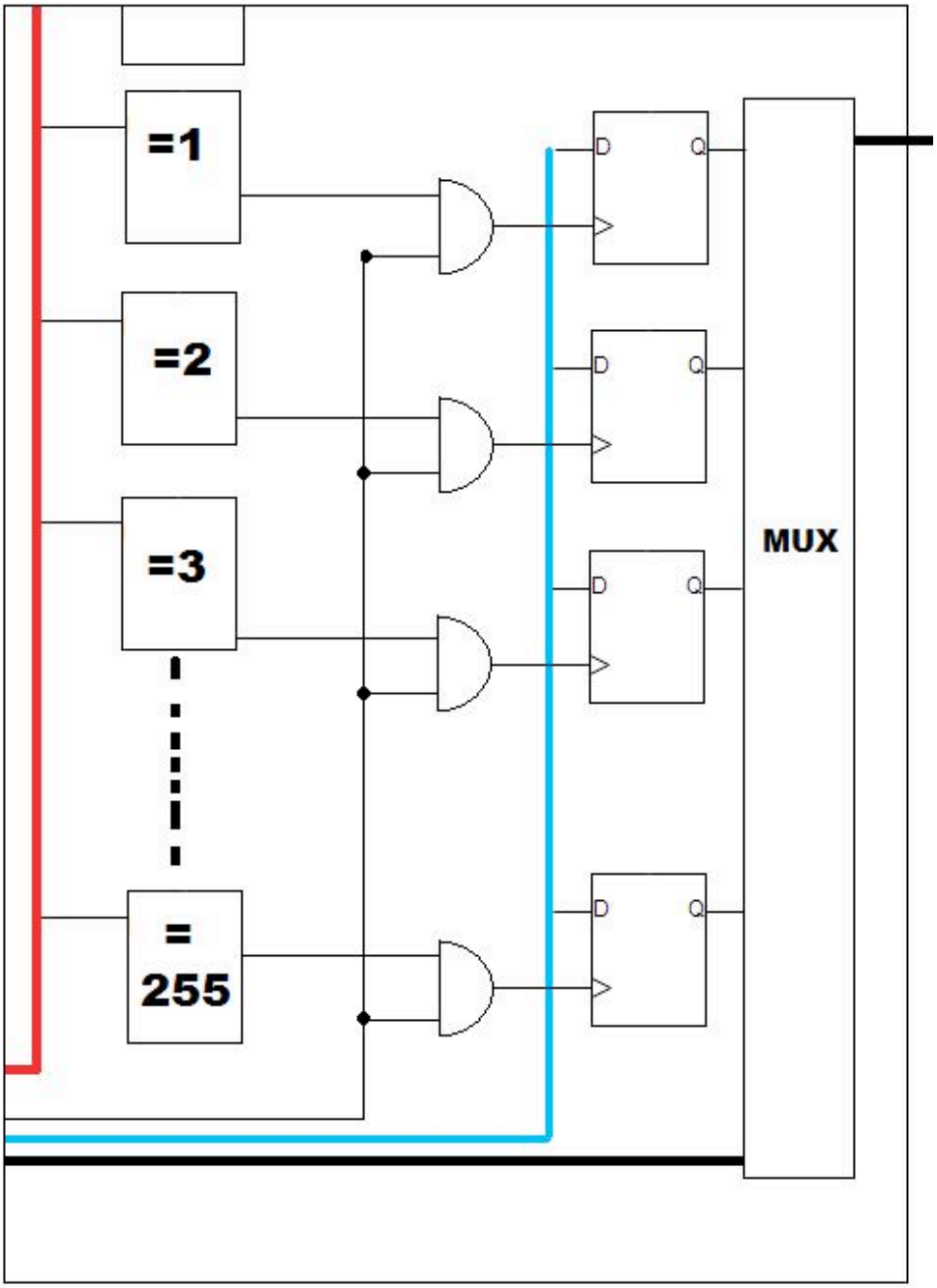


VAL	Operation
88	ET logique
9o	OU logique
Ao	NOT logique
Eo	Décalage droite de 1 bit
D8	MSB à 1

Afin de connaitre les OPCODE, il a fallu avec ISE passer sur chaque pin de chaque porte AND et regarder sa connectique.

On pourra aussi noter que seul les 5 bits de poids fort sont gardé.

Block SMD et SMP:

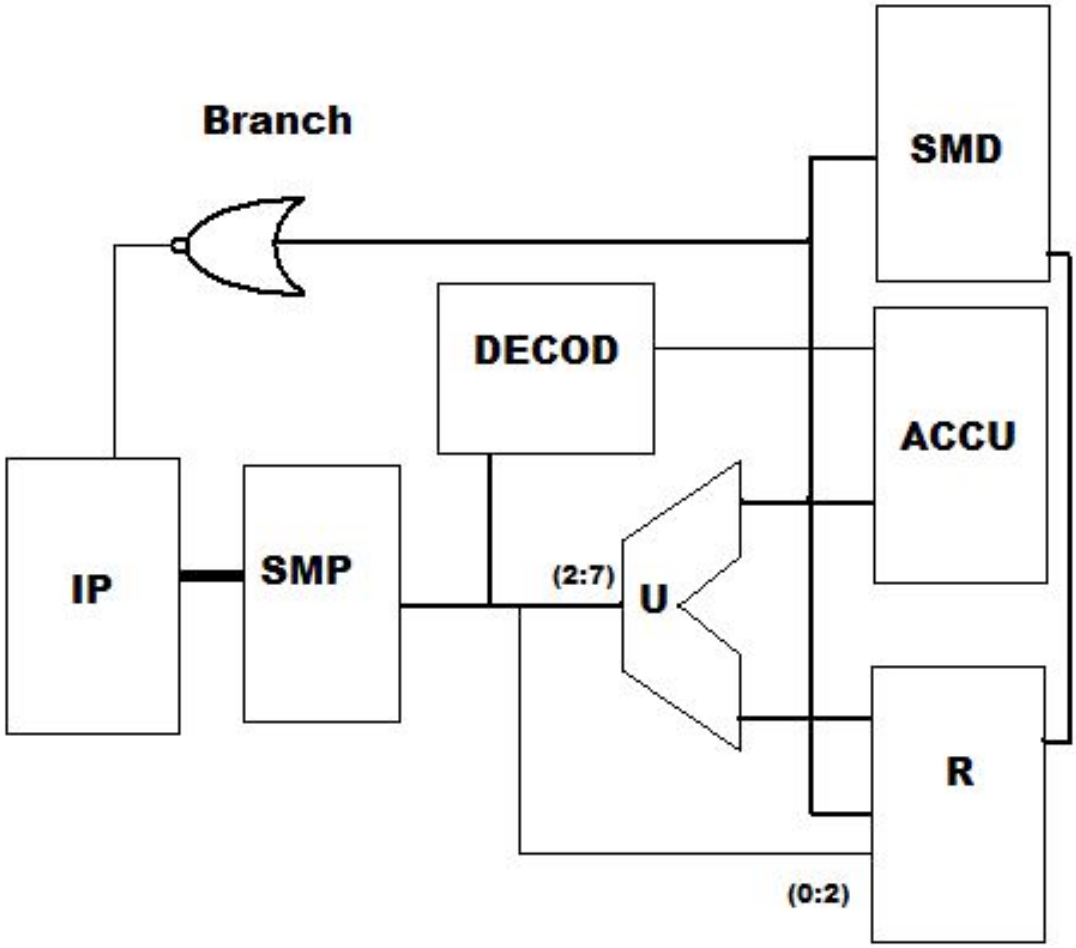


Ces bloques se ressemblent même si SMD est un peu plus complexe. Ici le block SMP a été schématisé afin de le simplifier. Il représente 256 block comparateur de décodage

composé chacun de 2 portes AND à 4 entrées dont la sortie va sur une porte AND finale. Ce qui en fait un système de décodage. Ainsi seul un seul comparateur va avoir sa sortie active pour une adresse sélectionnée. Ce qui activera une bascule D qui fait office d'accumulateur afin d'y stocker une valeur présente sur le bus de data. Ces 256 bascule 8bits sont alors reliées a un multiplexeur sur 8

bits dont le bus de sélection choisira lequel de ces accumulateur sera en sortie. Pour résumer, le block SMP correspond a une mémoire que l'on programme de l'extérieur grâce aux broches contrôle, et qu'on ne peux pas lire de l'extérieur. La lecture se fait grâce au bus de sélection. Le block SMD lui par contre permet la lecture de l'extérieure.

Nous pouvons maintenant simplifier le schéma et en déduire que c'était une implémentation de type OISC (One Instruction Set Computer). Dont voici le schéma de principe.



Ce schéma dans le FPGA était utilisé de la façon suivante: Le script python met dans le block SMD des données, et dans le block SMP un programme. Il cycle le circuit jusqu'a ce que le signal Finished soit activé. A ce moment la, il extrait de SMD sa mémoire.

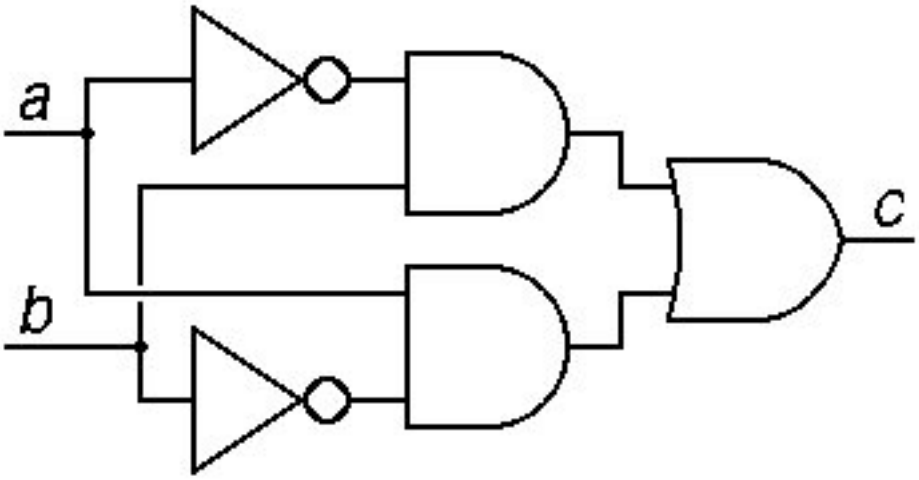
Le programme est alors capable de déchiffrer le fichier data.

Le jeu de code étant alors relativement simple, nous pouvons le désassembler et l'émuler.

Nous voyons dans le tableau suivant le code en assembleur.

Notons que l'operation exclusive OR n'est pas une opération booléenne de base et qu'elle peut etre faite a partir de ET/OU/NON logique comme montré ci dessous:

<u>XOR</u>:



```
00:   00   mov a,  00
01:   b0   mov r0, a
02:   10   mov a,  10
03:   d0   mov a,  smd[a]
04:   b7   mov r7, a
05:   a8   mov a,  r0
06:   a0   not a
07:   b6   mov r6, a
08:   0e   mov a,  0e
09:   b5   mov r5, a
0a:   71   mov a,  71
0b:   b4   mov r4, a
0c:   00   mov a,  00
0d:   bc   jna r4
0e:   01   mov a,  01
0f:   b6   mov r6, a
10:   16   mov a,  16
11:   b5   mov r5, a
12:   71   mov a,  71
13:   b4   mov r4, a
14:   00   mov a,  00
15:   bc   jna r4
16:   52   mov a,  52
17:   b6   mov r6, a
18:   af   mov a,  r3
19:   be   jna r6
1a:   11   mov a,  11
1b:   b7   mov r7, a
1c:   a8   mov a,  r0
1d:   b6   mov r6, a
1e:   24   mov a,  24
1f:   b5   mov r5, a
20:   71   mov a,  71
21:   b4   mov r4, a
22:   00   mov a,  00
23:   bc   jna r4
24:   af   mov a,  r3
25:   b1   mov r1, a
26:   a9   mov a,  r1
27:   d0   mov a,  smd[a]
28:   b6   mov r6, a
29:   0f   mov a,  0f
2a:   88   and a,  r0
2b:   d0   mov a,  smd[a]
2c:   96   or  a,  r6
2d:   b6   mov r6, a
2e:   a9   mov a,  r1
2f:   d0   mov a,  smd[a]
30:   b7   mov r7, a
31:   0f   mov a,  0f
32:   88   and a,  r0
33:   d0   mov a,  smd[a]
34:   8f   and a,  r7
35:   b7   mov r7, a
36:   af   mov a,  r3
37:   a0   not a
38:   8e   and a,  r6
39:   b7   mov r7, a
3a:   40   mov a,  40
3b:   b6   mov r6, a
3c:   53   mov a,  53
3d:   b5   mov r5, a
3e:   00   mov a,  00
3f:   bd   jna r5
40:   af   mov a,  r3
41:   c1   mov  smd[r1],  a
42:   01   mov a,  01
43:   b6   mov r6, a
44:   a8   mov a,  r0
45:   b7   mov r7, a
46:   4c   mov a,  4c
47:   b5   mov r5, a
48:   71   mov a,  71
49:   b4   mov r4, a
4a:   00   mov a,  00
4b:   bc   jna r4
4c:   af   mov a,  r3
4d:   b0   mov r0, a
4e:   02   mov a,  02
4f:   b6   mov r6, a
50:   00   mov a,  00
51:   be   jna r6
52:   c8   ret
53:   01   mov a,  01
54:   b5   mov r5, a
55:   00   mov a,  00
56:   b4   mov r4, a
57:   6d   mov a,  6d
58:   b3   mov r3, a
59:   ad   mov a,  r1
5a:   bb   jna r3
5b:   66   mov a,  66
5c:   b3   mov r3, a
5d:   ac   mov a,  r0
5e:   d8   shl  a
5f:   b4   mov r4, a
60:   ad   mov a,  r1
61:   8f   and a,  r7
62:   bb   jna r3
63:   01   mov a,  01
64:   94   or  a,  r4
65:   b4   mov r4, a
66:   ad   mov a,  r1
67:   d8   shl  a
68:   b5   mov r5, a
69:   57   mov a,  57
6a:   b3   mov r3, a
6b:   00   mov a,  00
6c:   bb   jna r3
6d:   ac   mov a,  r0
6e:   b7   mov r7, a
6f:   00   mov a,  00
70:   be   jna r6
71:   00   mov a,  00
72:   b1   mov r1, a
```

```
73:   b3   mov r3, a
74:   01   mov a,  01
75:   b2   mov r2, a
76:   63   mov a,  63
77:   e0   or  a, 80
78:   b4   mov r4, a
79:   aa   mov a,  r2
7a:   bc   jna r4
7b:   2c   mov a,  2c
7c:   e0   or  a, 80
7d:   b4   mov r4, a
7e:   af   mov a,  r3
7f:   8a   and a,  r2
80:   bc   jna r4
81:   16   mov a,  16
82:   e0   or  a, 80
83:   b4   mov r4, a
84:   ae   mov a,  r2
85:   8a   and a,  r2
86:   bc   jna r4
87:   0f   mov a,  0f
88:   e0   or  a, 80
89:   b4   mov r4, a
8a:   a9   mov a,  r1
8b:   bc   jna r4
8c:   ab   mov a,  r3
8d:   92   or  a,  r2
8e:   b3   mov r3, a
8f:   aa   mov a,  r2
90:   b1   mov r1, a
91:   59   mov a,  59
92:   e0   or  a, 80
93:   b4   mov r4, a
94:   00   mov a,  00
95:   bc   jna r4
96:   22   mov a,  22
97:   e0   or  a, 80
98:   b4   mov r4, a
99:   a9   mov a,  r1
9a:   bc   jna r4
9b:   aa   mov a,  r2
9c:   b1   mov r1, a
9d:   59   mov a,  59
9e:   e0   or  a, 80
9f:   b4   mov r4, a
a0:   00   mov a,  00
a1:   bc   jna r4
a2:   ab   mov a,  r3
a3:   92   or  a,  r2
a4:   b3   mov r3, a
a5:   00   mov a,  00
a6:   b1   mov r1, a
a7:   59   mov a,  59
a8:   e0   or  a, 80
a9:   b4   mov r4, a
aa:   00   mov a,  00
ab:   bc   jna r4
ac:   48   mov a,  48
ad:   e0   or  a, 80
ae:   b4   mov r4, a
af:   ae   mov a,  r2
b0:   8a   and a,  r2
b1:   bc   jna r4
b2:   3e   mov a,  3e
b3:   e0   or  a, 80
b4:   b4   mov r4, a
b5:   a9   mov a,  r1
b6:   bc   jna r4
b7:   aa   mov a,  r2
b8:   b1   mov r1, a
b9:   59   mov a,  59
ba:   e0   or  a, 80
bb:   b4   mov r4, a
bc:   00   mov a,  00
bd:   bc   jna r4
be:   aa   mov a,  r2
bf:   93   or  a,  r3
c0:   b3   mov r3, a
c1:   00   mov a,  00
c2:   b1   mov r1, a
c3:   59   mov a,  59
c4:   e0   or  a, 80
c5:   b4   mov r4, a
c6:   00   mov a,  00
c7:   bc   jna r4
c8:   57   mov a,  57
c9:   e0   or  a, 80
ca:   b4   mov r4, a
cb:   a9   mov a,  r1
cc:   bc   jna r4
cd:   aa   mov a,  r2
ce:   93   or  a,  r3
cf:   b3   mov r3, a
d0:   00   mov a,  00
d1:   b1   mov r1, a
d2:   59   mov a,  59
d3:   e0   or  a, 80
d4:   b4   mov r4, a
d5:   00   mov a,  00
d6:   bc   jna r4
d7:   00   mov a,  00
d8:   b1   mov r1, a
d9:   aa   mov a,  r2
da:   d8   shl  a
db:   b2   mov r2, a
dc:   a9   mov a,  r1
dd:   d8   shl  a
de:   b1   mov r1, a
df:   76   mov a,  76
e0:   b4   mov r4, a
e1:   00   mov a,  00
e2:   bc   jna r4
e3:   ab   mov a,  r3
e4:   b7   mov r7, a
e5:   00   mov a,  00
e6:   bd   jna r5
```

Une étude static du code montre par example que l'on prend le 16eme octet qui est la taille en entrée:

```
02:     10      mov a,  10
03:     d0      mov a,  smd[a]
```

Le block a l'address "0x24 -> 0x3F" nous montre un example d'XOR:

On peut tenter alors de faire un test en boite noir avec l'émulateur.

Nous allons par example prendre un block composé d'une clef de type "010000000000000000000000" et 224 fois les valeur "00" en tant que donnée.

Lorsque l'émulation est terminée, on arrive a "0x8000000000..." On suspecte alors deja un bitswap. De plus en réessayant de façon successive avec d'autre valeur (clef = "010203..." et data = "010101...") on vois qu'un XOR est appliqué entre la clef et les block de 16 octets.

Sachant que la memoire SMD ne contient que 256 octet, qu'il en faut 16 pour la clef, 1 pour la taille. il ne reste alors que 224 block de 16 possible. et 15 octet restent a la fin innutilisé. Nous comprennons alors pourquoi le programme ne prennais que des blocks de 224 octets.

Le programme suivant est la boucle principale qui a permis d'écrire l'émulateur insi que le desassembleur du code OISC.

```
1.    while(1){
2.        ip++;
3.        i = ip;
4.        opcode = smp[ip];
5.    /* Immediate value */
6.        if (opcode < 0x80){
7.          a = opcode;
8.          continue;
9.        }
10.
11.       if(opcode == 0xd0){
12.         olda = a;
13.         a = smd[a];
14.         continue;
15.       }
16.   /* Finished */
17.       if(opcode == 0xc8){
18.         finish();
19.         return 0;
20.       }
21.       opmask = opcode & 0xf8;
22.       switch(opmask){
23.   /* a = smd[r] */
24.       case 0xb0:
25.         r_addr = opcode & 0x7;
26.         r[r_addr] = a;
27.         continue;
28.         break;
29.       case 0xa8:
30.         r_addr = opcode & 0x7;
31.         a = r[r_addr];
32.         continue;
33.         break;
34.       case 0xa0:
35.         a = ~a;
36.         continue;
37.         break;
38.       case 0xb8:
39.         r_addr=opcode & 0x7;
40.         if(a == 0){
41.           ip = r[r_addr] -1;
42.         }
43.         continue;
44.         break;
45.       case 0x88:
46.         r_addr = opcode & 0x7;
47.         a = a & r[r_addr];
48.         continue;
49.         break;
50.       case 0x90:
51.         r_addr = opcode & 0x7;
52.         a = a | r[r_addr];
53.         continue;
54.         break;
          case 0xc0:
55.         r_addr = opcode & 0x7;
56.         smd[r[r_addr]] = a;
57.         continue;
58.         break;
59.       case 0xd8:
60.         a = a << 1;
61.         continue;
62.         break;
63.       case 0xe0:
64.         a = a  | 0x80;
65.         continue;
66.         break;
67.
68.       }
```

finalement, cette ligne de code nous laisse penser que le fichier une fois décodé est en BASE64:

```
result = base64.b64decode("".join(result))
```

Afin de pouvoir trouver la clef, nous allons alors utiliser de la force brute de la maniere suivant:

La clef va etre XORé avec chaque bloque de 16. Le resultat doit rester dans le set de charactere du base64 et eventuellement le retour a la ligne: “\n”.

Nous pouvont donc tester pour chaque octet de la clef, les 256 possibilité qui lui permettent une fois XORé a l’octet correspondant dans les data, de toujours avoir un resultat qui reste dans [A-Za-z0-9] , + , /, =, ‘\n’.

```
1.   #include <stdio.h>
2.   #include <stdlib.h>
3.   #include <string.h>
4.
5.   int main()
6.   {
7.     unsigned char *buf;
8.     unsigned char val, tmp;
9.     FILE *in;
10.    int i,j, n,m;
11.
12.    buf = malloc(2753 * 16);
13.    in = fopen("adat_crypt","rb");
14.    for(i = 0; i < 2753 * 16; i++){
15.      fscanf(in,"%c",&val);
16.      buf[i] = val;
17.    }
18.
19.
20.    for(n = 0; n < 16; n++){
21.      for(j = 0; j < 256; j++){
22.        m = 0;
23.        for(i = 0; i < 50; i++){
24.          tmp = buf[(i * 16) + n] ^ j;
25.          if( ((tmp >= 'A') && (tmp <= 'Z'))
               || ((tmp >= 'a') && (tmp <= 'z'))
               || ((tmp >= '0') && (tmp <= '9'))
               ||(tmp == '+') || (tmp == '/')
               || (tmp == '\n')){
26.            continue;
27.          } else {
28.            m = 1;
29.          }
30.        }
31.        if(m == 0){
32.          printf("%02x",j);
33.        }
34.      }
35.    }
36.    printf("\n");
37.  }
```

```
$ ./a.out
67c13b3d68f71a63a635c4cb78b685a4a7
```

Nous voila donc avec la clef, nous pouvons XOR cette valeur avec tous les bloques de 16 octet du fichier data et nous obtenons donc “atad”.

POSTSCRIPT

Aprés l’avoir converti de base64 -> binaire. Nous optenons un nouveau fichier Postscript.

```
/I1 currentfile 0 (cafebabe) /SubFileDecode filter /ASCIIHexDecode filter /ReusableStreamDecode filter
cf760bc77db1f282e881e...  cafebabe def

/I2 currentfile 0 (cafebabe) /SubFileDecode filter /ASCIIHexDecode filter /ReusableStreamDecode filter
123130301030304100b10...  cafebabe def

/I3 currentfile 0 (cafebabe) /SubFileDecode filter /ASCIIHexDecode filter /ReusableStreamDecode filter
142c44978faa76dae62cf...  cafebabe def

/I4 currentfile 0 (cafebabe) /SubFileDecode filter /ASCIIHexDecode filter /ReusableStreamDecode filter
8ae98ae90000000002000...  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
```

Le fichier postscript est composé de 4 définition “I1, I2, I3, I4” hexadécimale ainsi qu’un code qui sera exécuté.

Afin de pouvoir executé le script, celui ci prendra de la part de ghostscript l’instruction “shellarguments”. Ainsi la ligne de commande doit etre de type:

```
ghostscript -- script.ps 000102030405060708 0A0B0C0D0E0F
```

Il est alors nécessaire de simplifier au maximum le code et de se débarrasser des surcharge de définitions afin d’y voir plus clair.

```
errordict /handleerror { quit } put
```

Celle ci en est clairement une qui empêche de voir l'état de la pile.

Pour pouvoir avancer et debugger le code, on pourra utiliser par exemple:

```
dup ==
```

Qui dupliquera la dernière valeur de la pile et l'affichera sous forme lisible

```
pstack
```

qui affichera la pile complète.

Une fois remise en forme, on voit que les bloques I2 et I4 auront le même traitement:

On prend 2 octet de la clefs (le 2 premier pour I4, les 2 suivant pour I2) que l'on XOR avec le block. Le résultat est la sortie, et sert de nouveau pour le prochain XOR.

```
{
    3
    copy
    exch
    length
    getinterval
    2 index
    mark
    (---------\n) print
    3 1 roll
    2 index ==
    1 index ==
    dup ==

    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
...
```

le resultat sera dans les deux cas appliqués a ces deux lignes:

```
cvx
exec
```

Ce qui sous entend que les block I2 et I4 sont des block de données Postscripts eux même, et seront transformé en code exécutable après être déchiffré par les la boucle de XOR.

Nous pouvons alors chercher la clef par un bruteforce, qui cherche toutes les possibilités de clef dont le résultat final appartiendra systématiquement aux set de caractères du langage Postscript.

```
for(...){
do_xor()...
if( ((val[0] > 126)
 || (val[1] > 126)
 || (val[2] > 126)
 || (val[3] > 126)
 || (val[0] < 8)
 || (val[1] < 8 )
 || (val[2] <8 )
 || (val[3] <8))){
       m = 1;
       break;
 }
}
```

En mettand la sortie dans un fichier et en faisant un grep sur des mots clefs de Postscript, nous trouvons rapidement les clefs pour I4 et I2:

- I4: 0xbac9
- I2: 0xf7a8

Le debut de la clef semble alors etre de la forme:

BAC9F7A8----------

I2:

```
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
```

I4:

```
/I4 (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 ad
d } 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 x
or 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 s
tring 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) def
/error { (%stderr)(w) file exch writestring } bind def
%errordict /handleerror { quit } put
```

L'analyse de I2 nous laisse voir un tableau: `8#32732522170 8#35061733526 8#4410070333...` Avec des valeurs octale. Apres conversion en hexadecimale et une recherche, nous pouvons voir que ces valeurs sont celle des tables MD5.

I2 serait alors une implémentation de MD5 en Postscript qui serait appelé par la fonction "calc" défini dedans.

A ce stade, nous avons un tableau de 77 bloques de chacun 128 octets qui représentent un découpage de I1 et ses 9856 octets:

1	0	1	2	3	4	5	6	7	8	9	10	11	125	126	127
2	0	1	2	3	4	5	6	7	8	9	10	11	125	126	127
3	0	1	2	3	4	5	6	7	8	9	10	11	125	126	127
4	0	1	2	3	4	5	6	7	8	9	10	11	125	126	127
5	0	1	2	3	4	5	6	7	8	9	10	11	125	126	127
6	0	1	2	3	4	5	6	7	8	9	10	11	125	126	127
7	0	1	2	3	4	5	6	7	8	9	10	11	125	126	127
8	0	1	2	3	4	5	6	7	8	9	10	11	125	126	127
9	0	1	2	3	4	5	6	7	8	9	10	11	125	126	127
10	0	1	2	3	4	5	6	7	8	9	10	11	125	126	127
11	0	1	2	3	4	5	6	7	8	9	10	11	125	126	127
12	0	1	2	3	4	5	6	7	8	9	10	11	125	126	127
75	0	1	2	3	4	5	6	7	8	9	10	11	125	126	127
76	0	1	2	3	4	5	6	7	8	9	10	11	125	126	127
77	0	1	2	3	4	5	6	7	8	9	10	11	125	126	127

Le bloque I4 va faire un le déchiffrement du block I1. Pour se faire il fa prendre un DWORD étant la clef initiale et appliquer une première transformation Trans(K) de la façon suivante:

```
b = (a >> 7) ^ ( a >> 3) ^ (a >> 2)  ^ a;
b = b & 1
b = (a >> 1) | (b << 31);
```

La version postscript étant représenté ici:

```
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
```

0x00000000 - 0xFFFFFFFF est alors séparé en 3 parties égales et on regarde dans quelle tranche tombe Trans(K).

```
<55555555>
le
{1}
{
    dup
    <aaaaaaaa>
    le
    {-1}
    {0} ifelse
} ifelse
```

Le tableau de correspondance suivant donne alors la valeur de décalage dec1.

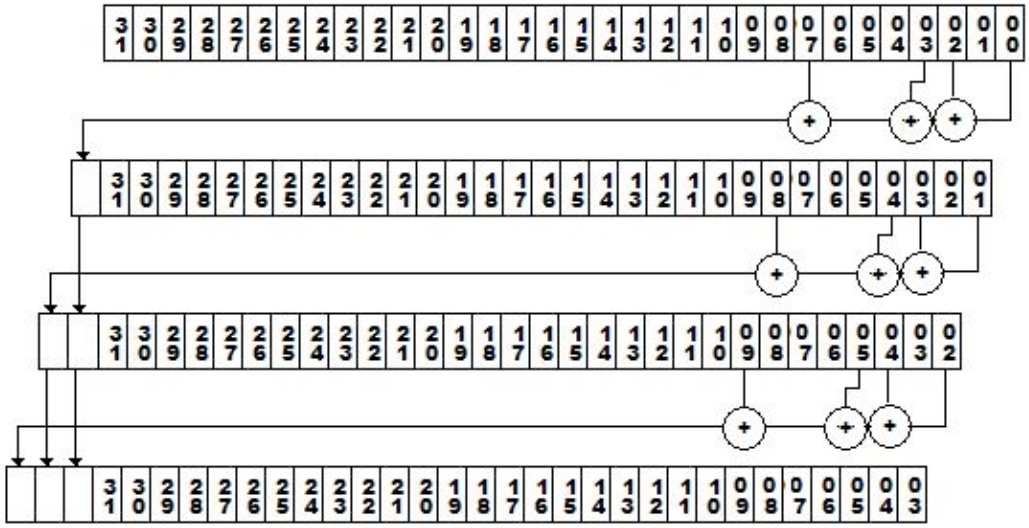
0->0x55555555	0x55555555->0xaaaaaaaa	0xaaaaaaaa->0xffffffff
1	-1	0

La meme transformation est faite a nouveau sur le resultat précedent pour avoir la valeur de decalage dec2 a partir de Trans(Trans(K)).

Une derniere transformation est alors appliqué pour avoir la nouvelle clef K utilisé pour XOR les bloques permutés.

K = Trans(Trans(Trans(K)))

Lors d'une implémentation, nous pourrions alors résumer les trois Transformation successive au modèle suivant, ce qui permettrais de tout faire d'une seul passe:



Nous voyons donc que 9bits sont concernés : b0 b1 b2 b3 b4 b5 b7 b8 b9. Ce qui nous permettras d'écrire un tableau précalculé de 512 valeurs afin de tout faire d'une seul passe.

L'algorithme de déchiffrement utilise un réseau de Feistel à 10240 tour et l'opérateur XOR.

Il y aura dans un premier temps, le choix du bloque parmi les 77 qui se fera comme cela:

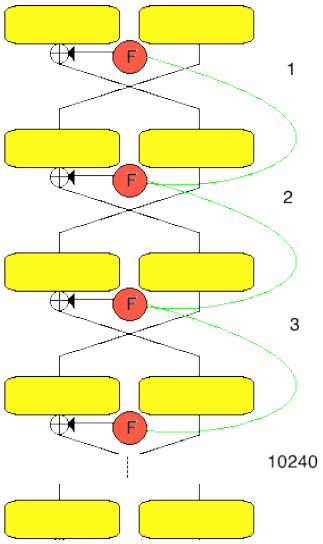
block = (77 + (oldblock + dec1)) % 77

Puis l'offset a l'intérieur d'un block est choisir de la façon suivante:

chaque bloque de 128 octet est coupé en DWORD. ce qui en forme 32.

offset = (32 + (oldoffset + dec2)) % 32

Nous avons donc une permutation qui s'effectue entre le bloque de la passe précédente et le bloque en cour, avant d'appliquer un XOR avec la clef K en cour. comme schématisé si dessous:



I1 est alors rassemblé et un MD5 est fait avec les le le code vu en I2. puis comparé a l'offset de lecture en cour sur I3 qui correspond a 6 HASH MD5 concaténé.

Si le MD5 calculé est le même que celui de I3, alors l'opération est refaite avec une nouvelle clef K qui correspond aux 2 derniers octet de l'ancienne et 2 nouveaux octets, ce qui en fait a nouveau un DWORD.

Afin de pouvoir déduire la clef, il serait possible d'implémenter le bruteforce directement en Postscript, mais il a été choisi de réimplementer en C pour gagner du temps.

Il faudra alors commencer avec une clef K faite des 2 derniers octets qui ont permi de déchiffrer I2 et I4, puis de tenter toutes les combinaisons possible entre 0x0000 et 0xffff soit 65536 possibilité maximum.

Le tableau qui permette de connaitre depuis une clef K directement Trans(Trans(Trans(K))) a été généré:

```
1.   char table[512] = {
2.     0, 1, 2, 3, 7, 6, 7, 6, 1, 0, 1, 0, 4, 5, 4, 5, 4, 5, 4, 5, 1, 0, 1, 0, 7, 6, 7, 6, 2, 3, 2, 3,
3.     6, 7, 6, 7, 3, 2, 3, 2, 5, 4, 5, 4, 0, 1, 0, 1, 0, 1, 0, 1, 5, 4, 5, 4, 3, 2, 3, 2, 6, 7, 6, 7,
4.     3, 2, 3, 2, 6, 7, 6, 7, 0, 1, 0, 1, 5, 4, 5, 4, 5, 4, 5, 4, 0, 1, 0, 1, 6, 7, 6, 7, 3, 2, 3, 2,
5.     7, 6, 7, 6, 2, 3, 2, 3, 4, 5, 4, 5, 1, 0, 1, 0, 1, 0, 1, 0, 4, 5, 4, 5, 2, 3, 2, 3, 7, 6, 7, 6,
6.     0, 1, 0, 1, 5, 4, 5, 4, 3, 2, 3, 2, 6, 7, 6, 7, 6, 7, 6, 7, 3, 2, 3, 2, 5, 4, 5, 4, 0, 1, 0, 1,
7.     4, 5, 4, 5, 1, 0, 1, 0, 7, 6, 7, 6, 2, 3, 2, 3, 2, 3, 2, 3, 7, 6, 7, 6, 1, 0, 1, 0, 4, 5, 4, 5,
8.     1, 0, 1, 0, 4, 5, 4, 5, 2, 3, 2, 3, 7, 6, 7, 6, 7, 6, 7, 6, 2, 3, 2, 3, 4, 5, 4, 5, 1, 0, 1, 0,
9.     5, 4, 5, 4, 0, 1, 0, 1, 6, 7, 6, 7, 3, 2, 3, 2, 3, 2, 3, 2, 6, 7, 6, 7, 0, 1, 0, 1, 5, 4, 5, 4,
10.    6, 7, 6, 7, 3, 2, 3, 2, 5, 4, 5, 4, 0, 1, 0, 1, 0, 1, 0, 1, 5, 4, 5, 4, 3, 2, 3, 2, 6, 7, 6, 7,
11.    2, 3, 2, 3, 7, 6, 7, 6, 1, 0, 1, 0, 4, 5, 4, 5, 4, 5, 4, 5, 1, 0, 1, 0, 7, 6, 7, 6, 2, 3, 2, 3,
12.    7, 6, 7, 6, 2, 3, 2, 3, 4, 5, 4, 5, 1, 0, 1, 0, 1, 0, 1, 0, 4, 5, 4, 5, 2, 3, 2, 3, 7, 6, 7, 6,
13.    3, 2, 3, 2, 6, 7, 6, 7, 0, 1, 0, 1, 5, 4, 5, 4, 5, 4, 5, 4, 0, 1, 0, 1, 6, 7, 6, 7, 3, 2, 3, 2,
14.    4, 5, 4, 5, 1, 0, 1, 0, 7, 6, 7, 6, 2, 3, 2, 3, 2, 3, 2, 3, 7, 6, 7, 6, 1, 0, 1, 0, 4, 5, 4, 5,
15.    0, 1, 0, 1, 5, 4, 5, 4, 3, 2, 3, 2, 6, 7, 6, 7, 6, 7, 6, 7, 3, 2, 3, 2, 5, 4, 5, 4, 0, 1, 0, 1,
16.    5, 4, 5, 4, 0, 1, 0, 1, 6, 7, 6, 7, 3, 2, 3, 2, 3, 2, 3, 2, 6, 7, 6, 7, 0, 1, 0, 1, 5, 4, 5, 4,
17.    1, 0, 1, 0, 4, 5, 4, 5, 2, 3, 2, 3, 7, 6, 7, 6, 7, 6, 7, 6, 2, 3, 2, 3, 4, 5, 4, 5, 1, 0, 1, 0,
18.  };
```

Les transformation de la clef se font de la façon suivante:

```
1.   #define keyrange(a) ((a) <= 0x55555555) ? 1 : ((a) <= 0xaaaaaaaa) ? -1 : 0
2.
3.   void roundkey(struct key_s *key)
4.   {
5.     unsigned int tmpkey = key->key;
6.     unsigned char topbits;
7.     /* get rid of b6 and bring everything on 9 bits */
8.     tmpkey = ((tmpkey & 0x3F) | ((tmpkey >> 1) & 0x1c0)) & 0x1ff;
9.     /* Get the 3 top bits */
10.    topbits = table[tmpkey];
11.    /*get the new indicator of block and offset shift*/
12.    key->a = keyrange( (key->key >> 1) | ((topbits & 1) << 31) );
13.    key->b = keyrange( (key->key >> 2) | ((topbits & 0x3) << 30));
14.    /*get the new key*/
15.    key->key = (key->key >> 3) | ((topbits & 0x7) << 29);
16.  }
```

Et finalement, l'implémentation du déchiffrement est le suivant:

```
1.   for(i = 0; i < 10240; i++){
2.       ctx->old_block = ctx->block;
3.       ctx->old_offset = ctx->offset;
4.       roundkey(ctx->key);
5.
6.       ctx->block = (nblock + (ctx->block + ctx->key->a)) % nblock;
7.       ctx->offset = ((BLOCK_SIZE / 4) + (ctx->offset + ctx->key->b)) % (BLOCK_SIZE / 4);
8.
9.       oblk = buf + (ctx->old_block * BLOCK_SIZE);
10.      blk = buf + ( ctx->block * BLOCK_SIZE);
11.
12.      memcpy(exch, blk + (ctx->offset * 4), 4);
13.      memcpy(blk + ( ctx->offset * 4), oblk + (ctx->old_offset * 4),4);
14.      memcpy(oblk + (ctx->old_offset * 4), exch,4);
15.
16.      toxor = (unsigned int *)(oblk + (ctx->old_offset * 4));
17.      *toxor ^= htonl(ctx->key->key);
18.    }
```

Pour le calcul du MD5, j'ai utilisé une implémentation de Apple présente a cette address:

http://www.opensource.apple.com/source/cups/cups-26/cups/md5.c?txt

Apres quelques minutes la clef sors de la moulinette:
bac9f7a8721fad3c9fcf271eed9abbc8

vCARD

Il est alors temps de réutiliser l'archive originale pour extraire le fichier "output.bin"

```
$gs -- script.ps bac9f7a8721fad3c9fcf271eed9abbc8
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.
```

Le fichier output.bin est donc extrait:

```
$ file output.bin
output.bin: vCard visiting card
```

```
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 s
tub_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
```

Nous voyons finalement que l'addresse email a été masqué par des mots qui commencent soit par "sys_" soit par "stub_" ce qui pourrais faire penser aux numero des syscalls.

En regardant la fin, on appercoit:

```
e  sys_ptrace
n  sys_getppid
g  sys_syslog
e  sys_ptrace
@  sys_sendmsg
s  sys_getgroups
s  sys_getgroups
t  sys_setgroups
i  sys_setuid
c  sys_sysinfo
.  sys_sendmsg
o  sys_getpgrp
r  sys_setregid
g  sys_syslog
```

L'idée d'une table de correspondance étant confirmé, aprés une recherche google, on retrouve une table ayant une correspondance similaire entre la valeur qui sera placé dns EAX au moment de la syscall.

On va alors implémenter:

```
1.   #include <stdio.h>
2.   #include <string.h>
3.
4.
5.   char * calltable[] = {
6.   /* 46 */ "sys_sendmsg",
7.   /* 48 */ "sys_shutdown",
8.   /* 49 */ "sys_bind",
9.   /* 51 */ "sys_getsockname",
10.  /* 52 */ "sys_getpeername",
11.  /* 53 */ "sys_socketpair",
12.  /* 54 */ "sys_setsockopt",
13.  /* 55 */ "sys_getsockopt",
14.  /* 56 */ "stub_clone",
15.  /* 57 */ "stub_fork",
16.  /* 64 */ "sys_semget",
17.  /* 97 */ "sys_getrlimit",
18.  /* 98 */ "sys_getrusage",
19.  /* 99 */ "sys_sysinfo",
20.  /* 100 */ "sys_times",
21.  /* 101 */ "sys_ptrace",
22.  /* 102 */ "sys_getuid",
23.  /* 103 */ "sys_syslog",
24.  /* 104 */ "sys_getgid",
25.  /* 105 */ "sys_setuid",
26.  /* 108 */ "sys_getegid",
27.  /* 110 */ "sys_getppid",
28.  /* 111 */ "sys_getpgrp",
29.  /* 115 */ "sys_getgroups",
30.  /* 114 */ "sys_setregid",
31.  /* 116 */ "sys_setgroups",
32.   0
33.  };
34.
35.  int main()
36.  {
37.    FILE *in;
38.
39.    in = fopen("final","r");
40.    unsigned char val[100];
41.    int i;
42.    while(!feof(in)){
43.      fscanf(in,"%s",val);
44.      for(i = 0; i < sizeof(calltable); i++){
45.        if(!strcmp(val,calltable[i])){
46.          printf("%c",i);
47.          break;
48.        }
49.      }
50.    }
51.
52.  }
53.
```

```
$ ./a.out
59575e0e71f1e3e9946bc307fc7a608d0b568458@challenge.sstic.org
```

Enfin ;)

Conclusion

Ce challenge était intéressant et original. La partie la plus dur a été de rédiger ce texte plein de fautes ;) Je me dis que peut être que le jeu d'instruction du processeur était connu et que j'ai pris le chemin compliqué. Peut être que la crypto faite par le postscript était connu et que j'aurai pu réutiliser une implémentation déjà présente plutôt que de refaire la roue...

Remerciements

Anaël, Eva, Anaïs, Marie, Lola, Léa, Clara, Chloé, Sarah, Juliee, Lucie, Jade, Romane, Emma, Manon, Agathe, Ambre, Lilou, Océane, Inès, Julie, Lisa, Maïwenn, Maryam, Kimberley, Louise, Mathilde, Zoé, Bienvenue, Maelys, Clémence, Emilie, Louna, Élisa, Célia, Justine, Pauline, Maéva, Alice, Laura, Noémie, Charloe, Lou, Lina, Jeanne, Lamia, Mélissa, Lana, Lily, Solene, Lena, Marion, Anna, Léonie, Margot, Mélanie, Maëlle, Nina, Marine, Alicia, Eric Alata, Fernand Lone-Sang, Vincent Nicomette du Laas-cnrs.