Solution du challenge SSTIC 2013

Une solution qu'elle est bien !

Auteur Guillaume Delugré
Date 3 avril 2013

Quarkslab SARL
71 – 73 avenue des Ternes
75017 Paris
France

Table des matières

1. Extraction et déchiffrement du fichier `sstic.tar.gz`

Le challenge se présente initialement sous la forme d'une trace réseau au format **`tcpdump`**.

```
$ wget http://static.sstic.org/challenge2013/dump.bin
$ md5sum dump.bin
968531851beed222851dbfd59140a395  dump.bin
$ file dump.bin
dump.bin: tcpdump capture file (little-endian) - version 2.4
          (Ethernet, capture length 65535)
```

Comme chaque année, l'objectif est de retrouver une adresse e-mail dissimulée au sein du fichier. La capture est constituée de trois flux principaux entre les machines **`192.168.1.12`** et **`192.168.1.13`** :
– Un premier échange TCP très court, contenant une indication pour le challenge ;
– Une suite de 65 *pings* de 192.168.1.13 vers 192.168.1.12 ;
– Une session FTP où 192.168.1.13 dépose un fichier **`sstic.tar.gz-chiffre`**.

L'indice fourni dans le premier flux est le suivant :

```
$ tshark -r dump.bin -z follow,tcp,ascii,0 -q
===================================================================
Follow: tcp,ascii
Filter: tcp.stream eq 0
Node 0: 192.168.1.13:57648
Node 1: 192.168.1.12:1234
546
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.

===================================================================
```

Le fichier transféré par FTP est donc chiffré en AES, avec un mode nécessitant un vecteur d'initialisation dont nous disposons. La clé de déchiffrement est passée par canal caché [1],

1. https://fr.wikipedia.org/wiki/Canal_cach%C3%A9.

c'est à dire que l'information est dissimulée à travers un autre canal de communication.

Il est ici très clair que l'information se trouve dans les 65 paquets ICMP capturés. Nous savons que le canal caché est composé de 3 bits non-temporels pour un bit temporel. En déduisant 4 bits d'informations sur 64 paquets ICMP, nous obtiendrions une potentielle clé pour un déchiffrement AES-256.

Voici ce que nous donnes un petit échantillon des paquets capturés :

```
$ tshark -r dump.bin icmp | head -8
1.030155 .13 -> .12 ICMP 98 Echo (ping) request  id=0xf132, seq=1/256, ttl=30
3.042155 .13 -> .12 ICMP 98 Echo (ping) request  id=0x0233, seq=1/256, ttl=30
5.055383 .13 -> .12 ICMP 98 Echo (ping) request  id=0x1333, seq=1/256, ttl=40
7.068428 .13 -> .12 ICMP 98 Echo (ping) request  id=0x2433, seq=1/256, ttl=30
9.082048 .13 -> .12 ICMP 98 Echo (ping) request  id=0x3533, seq=1/256, ttl=20
11.095455 .13 -> .12 ICMP 98 Echo (ping) request  id=0x4633, seq=1/256, ttl=10
12.108802 .13 -> .12 ICMP 98 Echo (ping) request  id=0x5533, seq=1/256, ttl=30
14.122642 .13 -> .12 ICMP 98 Echo (ping) request  id=0x6633, seq=1/256, ttl=30
```

Plusieurs observations peuvent être faites rapidement :

- Le temps émis entre chaque paquet vaut approximativement une ou deux secondes. Cette information est le bit temporel. La présence d'un $65^{\text{ème}}$ paquet permet ainsi de calculer 64 différences ;
- Le champ TTL a été modifié et prend 4 valeurs différentes : 10, 20, 30 et 40. Il est donc possible de le coder sur deux bits ;
- La différence des champs ID en *little-endian* entre deux paquets ICMP vaut 15 ou 17, ce qui représente un bit.

Malheureusement, la différence des champs ID pour les deux derniers paquets vaut 4.

```
$ tshark -r dump.bin icmp | tail -2
105.949627 .13 -> .12 ICMP 98 Echo (ping) request  id=0xf436, seq=1/256, ttl=30
106.952515 .13 -> .12 ICMP 98 Echo (ping) request  id=0xf836, seq=1/256, ttl=1
```

A partir de là, on peut se dire :

- soit que l'hypothèse sur les différences des ID est une fausse piste
- soit qu'on dispose quand même de 63 bits d'information, ce qui laisse seulement un bit à bruteforcer

N'étant pas capable de trouver une clé valide sous ces hypothèses, j'ai du me résoudre à chercher les 64 derniers bits ailleurs que dans le champ ID. En observant minutieusement chaque champ des paquets ICMP, on remarque que le dernier octet du *checksum* IP ne prend que deux valeurs :

```
$ tshark -r dump.bin -Tfields -e ip.checksum icmp | head -8
0xd93d
0xd93d
0xcf3b
0xd93b
```

```
0xe33d
0xed3b
0xd93d
0xd93d

$ tshark -r dump.bin -Tfields -e ip.checksum icmp | tail -1
0xf63f
```

Les valeurs des champs ayant été clairement manipulées, et le dernier paquet prenant une valeur différente, nous pouvons bien en déduire 64 bits d'information. Nous avons donc en résumé : 64 bits de temps, 64 bits de *checksum* et 128 bits de TTL.

Disposant de tous les bits, il suffit de construire et tester toutes les clés possibles. Nous construisons chaque clé par bloc de 4 bits en modifiant :
- l'ordre des champs (temps, *checksum*, TTL). Ceci donne 6 possibilités ;
- l'encodage des champs. Pour TTL par exemple, on testera d'abord 10=00b, 20=01b, 30=10b, 40=11b, puis 10=01b, 20=10b, 30=11b, 40=00b, et ainsi de suite. Il y a deux possibilités pour le bit de temps et le *checksum*, et 24 pour le TTL ;
- l'ordre dans lequel nous ajoutons les blocs à la clé (en partant du début ou de la fin).

Le fichier échangé par FTP est en *base64*, on commence donc par le décoder. Pour chaque clé on essaiera ensuite de déchiffrer le premier bloc de 128 bits dans différents modes avec l'IV fourni en indice, jusqu'à ce que les premiers octets du clair correspondent à un en-tête *gzip* (`1F 8B 08`).

Le parcours de toutes les clés est très rapide et conduit à déchiffrer une archive **`.tar.gz`** valide.

```
$ ./bruteforce_aes.rb dump.bin sstic.tar.gz-bin
Found key: dd8cf2d52e69aafb734e3acd0e4a69e83ed93bc4870ecd0d5b6faad86a63ae94

$ file sstic.tar.gz
sstic.tar.gz: gzip compressed data, was "archive.tar", from Unix,
last modified: Mon Mar 18 12:24:37 2013

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

```ruby
#!/usr/bin/env ruby

require 'openssl'
require 'digest/md5'

PCAP = ARGV.shift
ENCRYPTED_DATA = File.read(ARGV.shift)
PLAIN_DIGEST = "61c9392f617290642f9a12499de6b688"
IV = ["76C128D46A6C4B15B43016904BE176AC"].pack('H*')

times, ttls, csums = [], [], []
first_frame = true
%x{tshark -r "#{PCAP}" -Tfields -e frame.time_delta -e ip.checksum -e ip.ttl icmp}
  .each_line do |line|

  strtime, strcsum, strttl = line.split

  time = strtime.to_f
  csum = strcsum.hex
  ttl = strttl.to_i

  times.push(time - 1) unless first_frame
  ttls.push((ttl / 10) - 1) unless ttl == 1
  csums.push({0x3b => 0, 0x3d => 1}[csum & 0xff]) unless csum & 0xff == 0x3f

  first_frame = false
end

# construct a key
def make_key(times, ids, ttls, time_perm, id_perm, ttl_perm, order = 0)
  pos = 0
  byte = 0
  key = ""

  64.times do
    time, id, ttl = times.shift, ids.shift, ttls.shift
    bits = [ [:time, time_perm[time]], [:id, id_perm[id]], [:ttl, ttl_perm[ttl]] ]

    nibble = 0
    nibble_pos = 0
    bits.permutation.to_a[order].each do |field, b|
      nibble |= (b << nibble_pos)
      nibble_pos +=
        case field
          when :ttl then 2
          when :id, :time then 1
        end
    end
    byte |= (nibble << pos)

```

```ruby
    pos += 4
    if pos == 8
      key = byte.chr + key
      pos = byte = 0
    end
  end

  key
end

# reverse nibbles in a key
def reverse_key(key)
  key.unpack('C*')
     .reverse.map! {|byte| (byte >> 4) | ((byte << 4) & 0xf0) }
     .pack('C*')
end

def try_decrypt(data, key)
  %w{ cbc cfb ctr }.each do |mode|
    alg = "aes-#{key.length << 3}-#{mode}"
    aes = OpenSSL::Cipher::Cipher.new(alg).decrypt
    aes.iv, aes.key = IV, key
    aes.padding = 0

    decrypted = aes.update(data[0,16]) + aes.final
    if decrypted[0,3] == "\x1f\x8b\x08"
      aes = OpenSSL::Cipher::Cipher.new(alg).decrypt
      aes.iv, aes.key = IV, key

      plain = aes.update(data) + aes.final
      if Digest::MD5.hexdigest(plain) == PLAIN_DIGEST
        puts "Found key: #{key.unpack('H*')[0]}"
        File.write("sstic.tar.gz", plain)
        exit
      end
    end
  end
end

(0..1).to_a.permutation.each do |time_perm|
  (0..1).to_a.permutation.each do |id_perm|
    (0..3).to_a.permutation.each do |ttl_perm|
      6.times do |order|
        key = make_key(times.dup, csums.dup, ttls.dup,
                       time_perm, id_perm, ttl_perm, order)
        try_decrypt(ENCRYPTED_DATA, key)
        try_decrypt(ENCRYPTED_DATA, reverse_key(key))
      end
    end
  end
end
```

2. L'épreuve du circuit électronique

Suite à la décompression de l'archive, nous obtenons un répertoire **archive** composé des fichiers suivants :
– **data**, un *blob* binaire ;
– **smp.py**, un simple tableau de 231 octets en Python ;
– **s.ngr**, un fichier représentant un circuit électronique dans un format propriétaire de Xilinx ;
– **decrypt.py**, un script Python faisant appel au circuit pour déchiffrer le fichier **data**.

Cette épreuve était à première vue assez décourageante. Et pour cause : l'installeur de la suite logicielle Xilinx pèse 6 Go (26 Go pendant l'installation et 16 Go une fois installés), le circuit peut seulement être visualisé et le plus gros composant du circuit mettait près de 10 minutes à s'afficher chez moi... La seule possibilité d'exportation étant l'impression dans un fichier PDF.

Pourtant cette épreuve est beaucoup plus impressionante que difficile, et il est possible d'en venir à bout assez rapidement en analysant posément chaque partie du circuit, sans connaissances préalables en électronique.

Le script **decrypt.py** attend en argument une clé de 16 octets. Le fichier **data** est découpé en blocs de 224 octets et est envoyé avec sa taille, la clé, et le tableau **smp** au FPGA. A chaque itération, le résultat est récupéré et concaténé. Si le MD5 des données finales est valide, le résultat est écrit dans un fichier et nous pouvons passer au niveau suivant.

```
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()
```

2.1 Analyse du circuit

Il faut tout d'abord comprendre comment ce circuit a été généré, car il n'a vraisemblablement pas été écrit "à la main". Le circuit proposé résulte de la synthèse d'un code écrit dans un langage de plus haut-niveau, tel que VHDL [1] ou Verilog [2], à destination d'un FPGA. Le code HDL est converti dans un circuit numérique composé de primitives supportées par le FPGA, avant d'être plus tard compilé dans un *bitstream*. Le circuit fourni

1. https://fr.wikipedia.org/wiki/VHDL
2. https://fr.wikipedia.org/wiki/Verilog

est donc plus ou moins l'équivalent d'une représentation intermédiaire dans le monde des compilateurs logiciels.

Il n'est pas ici question de manipuler des tensions ou des intensités, mais uniquement des opérations logiques sur des branches véhiculant des 0 et des 1. Aucune connaissance particulière en électronique n'est donc requise (à part savoir comment marche un fil). Il suffit de connaître trois types de composants pour analyser le circuit :

- **les portes logiques** effectuent une simple opération sur les entrées comme `AND`, `NOT`, ou `OR`
- **les multiplexeurs** choisissent un signal d'entrée en fonction d'un signal de sélection. Ils correspondent à une structure *if else* dans un langage de plus haut-niveau.
- **les bascules** sont des registres qui stockent un signal d'entrée durant plusieurs coups d'horloges. Ils correspondent à des variables et permettent de créer des états.

Le dernier point à garder en tête est que les entrées-sorties entre composants ne se font pas séquentiellement mais en un seul temps. Le système est dans un seul état par coup d'horloge. La présence visuelle de boucle dans le circuit n'implique pas une logique de boucle au sens séquentiel du terme.

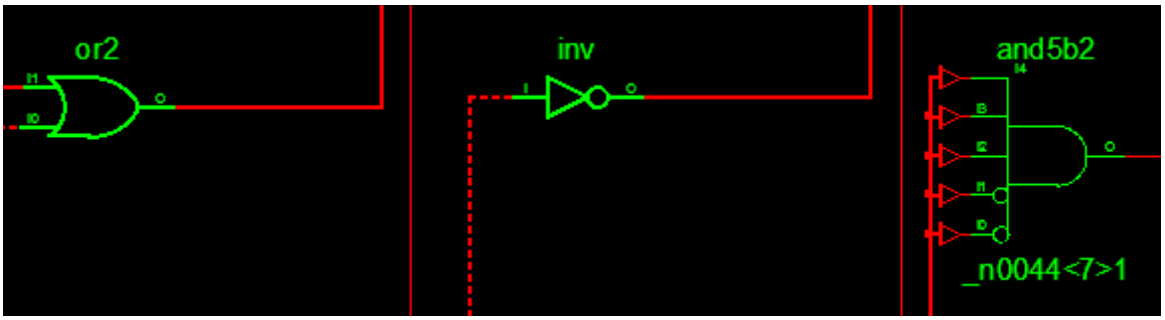


FIGURE 2.1 – Quelques portes logiques

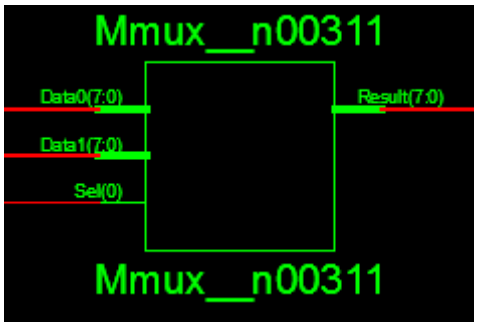


FIGURE 2.2 – Un multiplexeur crée une condition *if*

L'agencement et le nom des composants du circuit nous indique déjà qu'il s'agit probablement d'un processeur : le composant `x_r` peut lire et écrire dans 8 bascules en fonction d'un signal d'entrée sur 3 bits, et est directement relié au composant `x_ip` dont le nom suggère qu'il puisse d'agir d'un pointeur d'instruction.

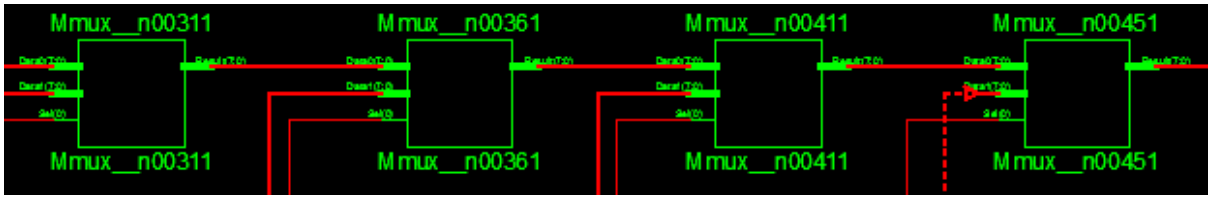


FIGURE 2.3 – Des multiplexeurs en chaîne forment des *else if*

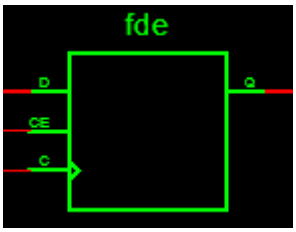
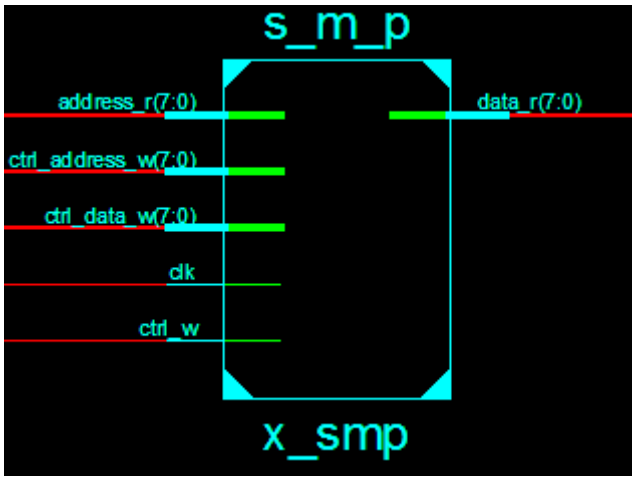


FIGURE 2.4 – Un *flip-flop* enregistre une valeur sur plusieurs cycles



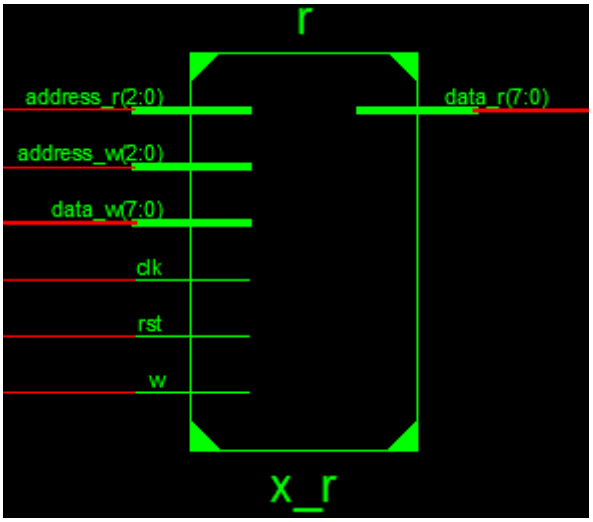
2.1.1 Le composant `x_smp`

Ce composant est très simple à analyser, malgré sa taille une fois déroulé. Le composant contient 256 bascules de 1 octet. La sortie est un multiplexeur qui choisit le contenu de l'une des bascules en fonction d'un signal d'entrée de 8 bits nommé *address_r*. Si le bit d'entrée *ctrl_w* est à 1, la valeur *ctrl_data_w* est inscrite dans la bascule dont le numéro correspond au signal de 8 bits *ctrl_address_w*.

Ce composant est donc un espace mémoire de 256 octets. Les 231 octets du tableau **`smp`** sont inscrits dans cette zone mémoire à chaque itération du script.

L'adresse de lecture en entrée est directement cablée sur la sortie du composant **`x_ip`**, et la sortie *data* est passée à la moulinette dans tout le reste du circuit. En supposant qu'il s'agisse bien d'un processeur, il semble logique que cette zone mémoire servent à contenir du code et que le tableau dans **`smp.py`** soit le programme à exécuter.

2.1.2 Le composant `x_r`

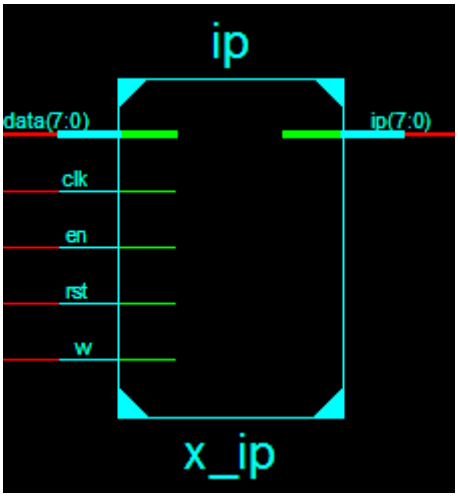


De la même façon que **`x_smp`**, ce composant permet de lire et écrire dans 8 bascules de 1 octet. La séléction de la bascule d'entrée se fait en choisissant les 3 bits de poids faible de l'instruction en sortie de **`x_smp`**.

La valeur du registre choisie est écrasée par la valeur d'entrée *data_w* si le bit *w* est à 1.

Or **`w = smp.data[7..3] AND 10110`** et la valeur *data_w* est cablée sur la sortie du composant **`x_accu`**. Nous pouvons donc en déduire le code :

```
opcode = smp.data[7..3]
reg = smp.data[2..0]
if ( opcode == 10110 )
  registers[reg] = x_accu.data  // mov <reg>, accu
```



2.1.3 Le composant `x_ip`

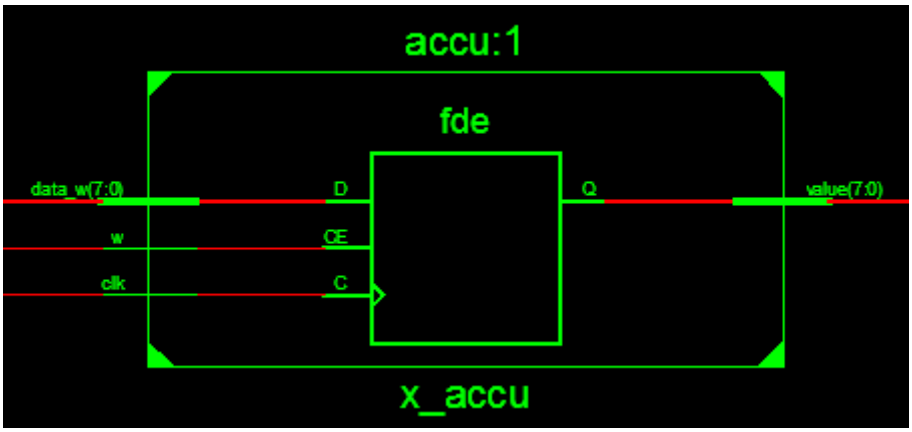
`x_ip` contient une bascule de 8 bits. Si *finished* vaut 1, l'état n'est pas modifié. Si le signal w est à 1, le contenu de la bascule est mis à la valeur d'entrée *data*, sinon il est incrémenté de 1. Etant donné que ce composant contient le pointeur d'instruction, nous sommes en présence d'un saut si le bit w est à 1.

Nous avons la relation : `w = NOR(x_accu.value) AND (x_smp.data[7..3] == 10111)`

Le pointeur d'instruction est modifié si les 5 bits de poids fort de l'instruction courante valent 10111, et si la valeur en sortie du composant **`x_accu`** vaut 0. La valeur du signal d'entrée *data* est quant à elle directement liée à la sortie du composant **`x_r`**.

Nous sommes donc en présence d'une instruction de type saut conditionnel sur registre.

2.1.4 Le composant `x_accu`



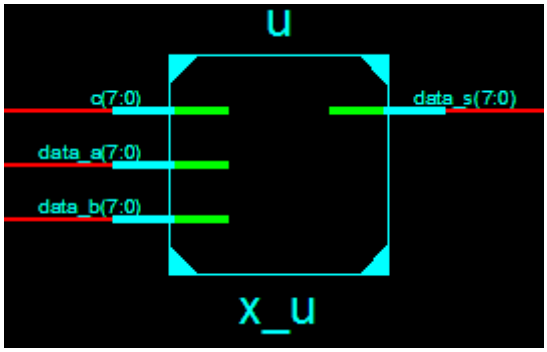
`x_accu` contient un simple registre de 8 bits dont la valeur dépend de 3 multiplexeurs chaînés en entrée.

En analysant les signaux de sélection des multiplexeurs, nous pouvons en déduire le pseudo-code suivant :

```
if ( smp.data[7] == 0 )
  accu = smp.data[6..0]  // mov accu, <imm7>
else if ( smp.data == 11010000 )
  accu = smd.data
else {
  opcode = smp.data[7..3]
  if ( opcode == 10101 )
    accu = x_r.data    // mov accu, <reg>
  else
    accu = x_u.data
}
```

Pour savoir comment est mise à jour la valeur de **x_accu**, il nous faut donc comprendre les signaux de sortie de **x_u** et **x_smd**.

2.1.5 Le composant x_u



x_u prend en entrée l'instruction courante, le registre sélectionné par l'instruction et la valeur de **x_accu**. En fonction de l'opcode de l'instruction courante, différentes opérations sont effectuées sur les entrées. La sortie est ensuite cablée en direction du registre **x_accu**.

Le code correspondant à ce composant est le suivant :

```
opcode = smd.data[7..3]
if ( opcode == 10001 )
  accu &= x_r.data   // and accu, <reg>
else if ( opcode == 10010 )
  accu |= x_r.data   // or accu, <reg>
else if ( opcode == 10100 )
  accu = ~accu       // not accu
else if ( opcode == 11100 )
  accu = accu        // nop
else if ( opcode == 11011 )
  accu = ???
```

Dans le cas où l'opcode de l'instruction courante vaut 11011, le signal sélectionné est directement relié à la masse du circuit et ne dépend pas des entrées. Il s'avère que cette instruction est néanmoins utilisée dans le code de **smp**. Elle fut donc déterminée plus loin

par intuition à partir du reste du code. Il n'est pas clair cependant si cette erreur a été introduite volontairement ou provient d'un bug d'affichage de Xilinx ISE.

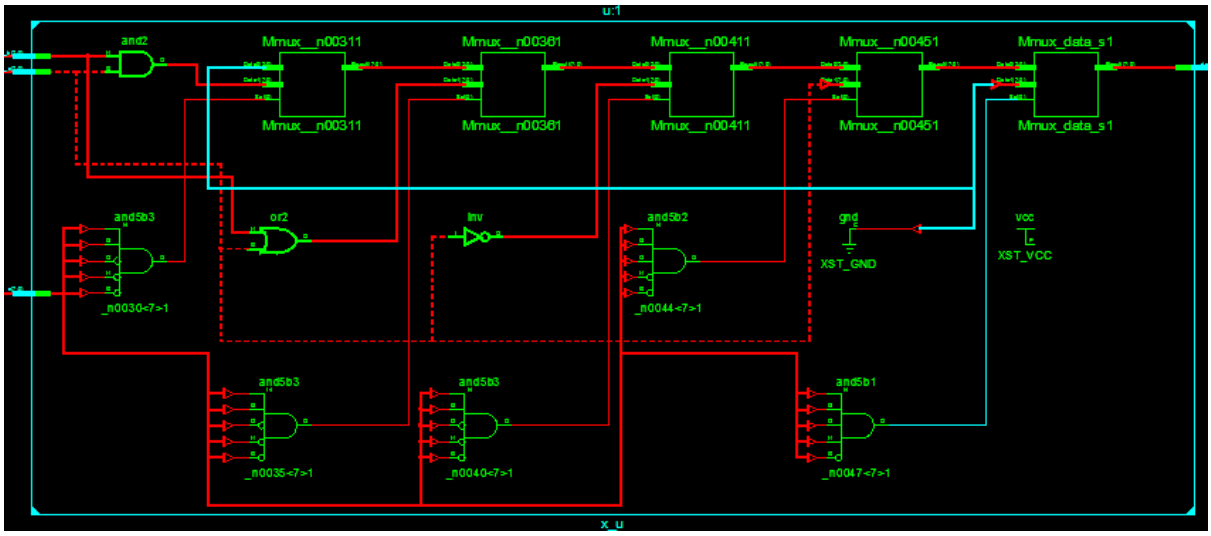
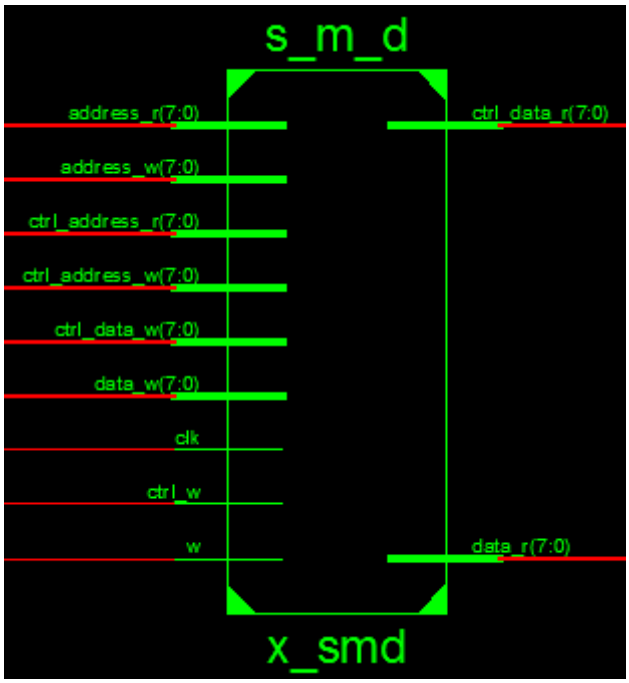


FIGURE 2.5 – Erreur volontaire ou bug de Xilinx ?

2.1.6 Le composant `x_smd`



Malgré sa complexité apparente, **`x_smd`** est une simple zone mémoire de 256 octets au même titre que **`x_smp`**. Il possède deux niveaux d'accès à la mémoire (dont l'un est préfixé par *ctrl*) ce qui multiplie par deux le nombre de portes et de liaisons nécessaires par rapport à **`x_smp`**. Le script Python inscrit dans **`x_smd`** les 16 octets de la clé utilisateur et les 224 octets du fichier **`data`** via les broches *ctrl*. Le processeur accède à la mémoire de **`smd`** via les autres broches.

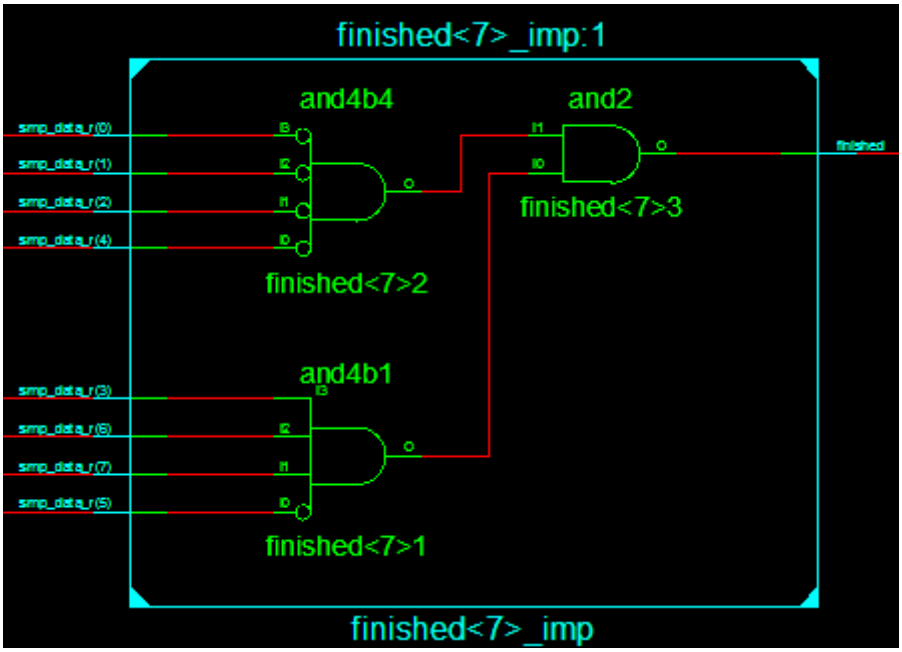
L'adresse de lecture *address_r* est cablée sur le registre **x_accu**, nous pouvons donc en déduire que l'instruction 11010000 mentionnée dans la section **x_accu** correspond à : **mov accu, smd[accu]**.

Concernant les accès en écriture, le signal *address_w* est cablé sur la valeur de **x_r**, *data_w* est cablé sur **x_accu** et le bit *w* suit la relation : **w = x_smp.data[7..3] == 11000**.

Nous avons donc le code suivant :

```
opcode = smp.data[7..3]
if ( opcode == 11000 )
  smd[x_r.data] = accu    // mov smd[<reg>], accu
```

2.1.7 Le composant `finished`



Lorsque ce composant émet la valeur 1, la valeur du pointeur d'instruction n'est plus modifiée et le script Python récupère le contenu de **x_smd**.

finished prend en entrée l'instruction courante et stoppe l'exécution du programme si celle-ci vaut 11001000.

2.2 Désassemblage et analyse du programme

Nous avons donc un processeur possédant les caractéristiques suivantes :
- 8 registres généraux de 1 octet, que l'on nommera **r0**, ..., **r7**
- 1 registre temporaire de 1 octet **accu**
- 1 pointeur d'instruction **ip**
- un espace 8 bits **smp** référencé par **ip**

– un espace 8 bits **smd** adressable par le code

2.2.1 Désassembleur

A partir de l'analyse des composants faite plus haut, il est possible d'écrire un petit désassembleur. Le processeur ne dispose en tout que de 12 instructions.

	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
`end`	1	1	0	0	1	0	0	0
`mov accu, smd[accu]`	1	1	0	1	0	0	0	0
`mov accu, imm7`	0	Imm7						
`mov accu, reg`	1	0	1	0	1	reg		
`mov reg, accu`	1	0	1	1	0	reg		
`mov smd[reg], accu`	1	1	0	0	0	reg		
`jz reg`	1	0	1	1	1	reg		
`and accu, reg`	1	0	0	0	1	reg		
`or accu, reg`	1	0	0	1	0	reg		
`not accu`	1	0	1	0	0	X	X	X
`??? accu`	1	1	0	1	1	X	X	X
`nop`	1	1	1	0	0	X	X	X

Le processeur a été implémenté dans Metasm [3] ce qui permet de visualiser le graphe du programme.

3. Un grand merci à Alex pour le coup de main ;)

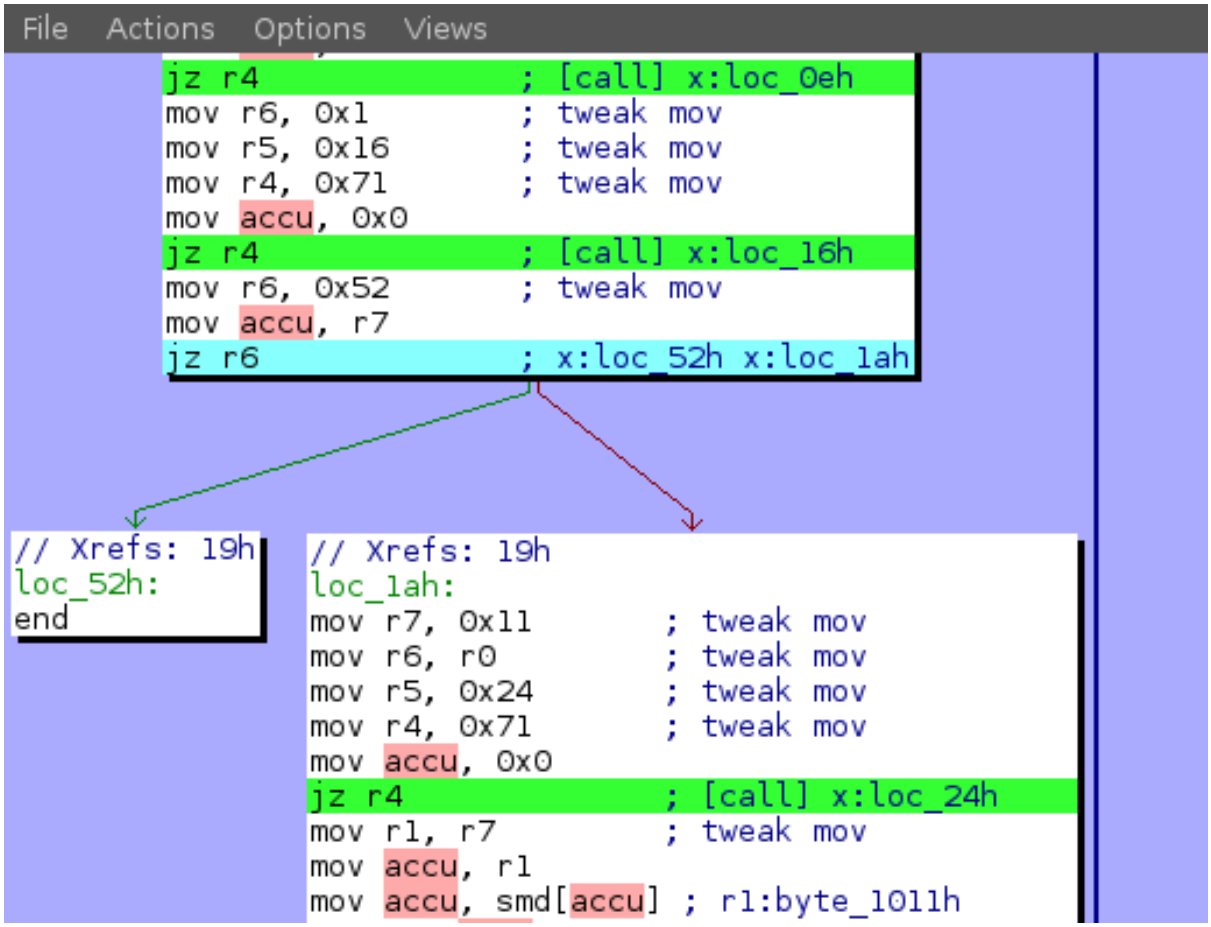


FIGURE 2.6 – Vue du programme dans l'interface de Metasm

Motifs de code

Le code du programme contient 231 instructions. Certains motifs de code sont très rapidement identifiables et simplifiables.

```
mov accu, imm7
mov reg, accu    ; mov reg, imm7

mov accu, reg1
mov reg2, accu   ; mov reg2, reg1

mov reg, imm7
mov accu, 0x0
jz reg           ; jmp imm7
```

Une observation de certains sauts nous indique ce qui ressemble à un appel de sous-routine :

```
mov r7, <arg2>
mov r6, <arg1>
mov r5, <return_address> ; adresse suivant le call
mov r4, <addr>
mov accu, 0x0
jz r4                    ; call addr
; r7 utilisé comme une valeur de retour
```

Correction des adresses de saut

La boucle principale du programme fait appel à deux sous-routines en 0x53 et 0x71. Le flot du code en 0x71 est cependant très difficile à lire et semble incohérent.

De plus certaines parties du code du programme sont simplement inaccessibles : les adresses de sauts étant codées sur 7 bits dans les instructions **`mov`**, le programme ne sautera jamais sur les instructions aux adresses supérieures à 0x7F. Or le code présent à ces adresses semble valide, quelque chose cloche donc quelque part.

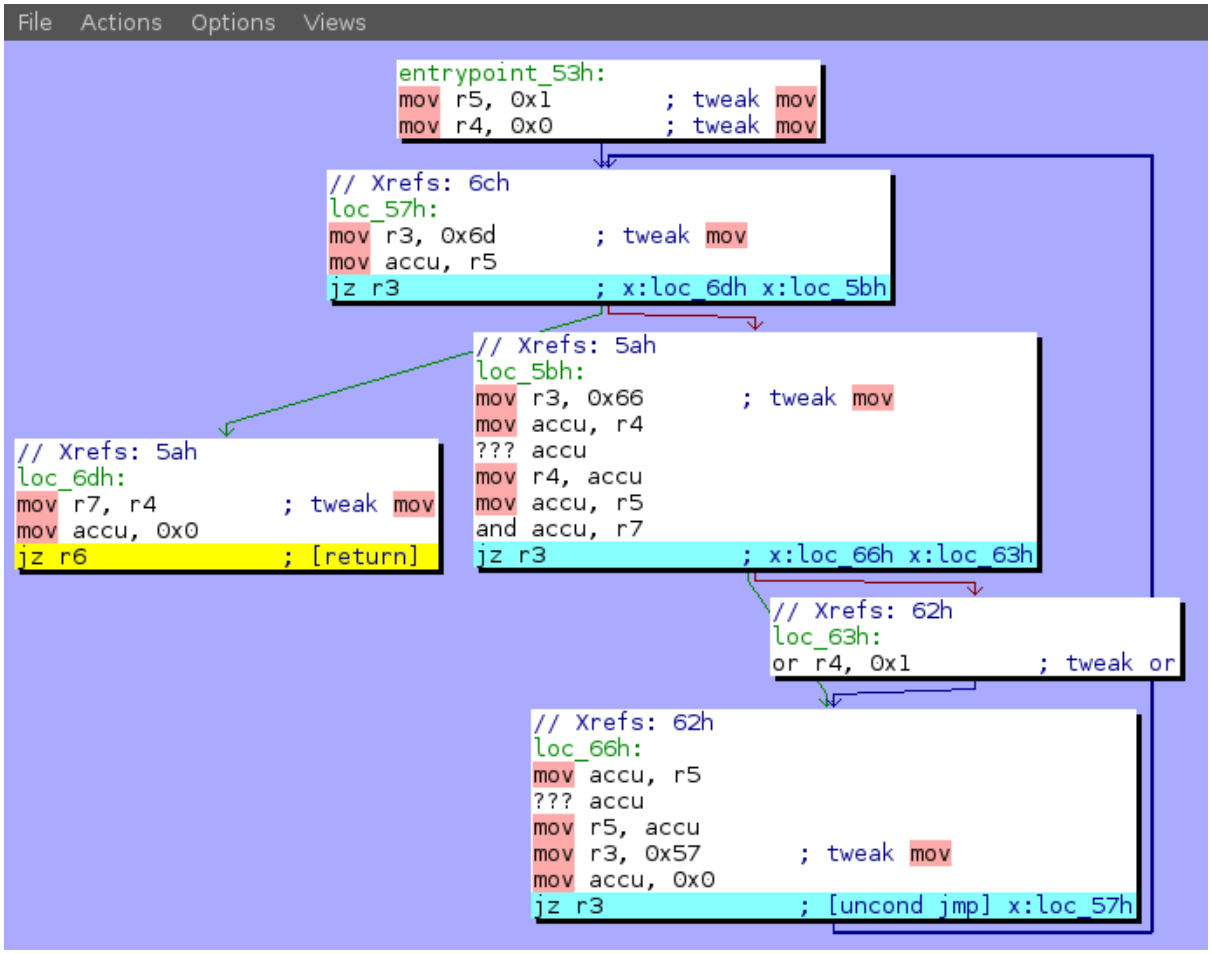
Le code semble beaucoup plus cohérent en ajoutant 0x80 à certaines adresses de sauts étranges à partir de 0x71. Les adresses de sauts ont donc été corrigées en appliquant un *or* 0x80 pour tous les sauts présents entre 0x71 et 0xE0.

Je n'ai cependant pas pu trouver la trace d'un tel comportement dans le circuit. C'est du vaudou, mais ça marche.

Identification de l'instruction inconnue

Un dernier problème dans le code reste à régler : l'une des instructions du circuit n'était pas valide. Celle-ci peut être devinée par observation de la routine en 0x53.

Le code assembleur peut se traduire dans le pseudo-code suivant :


File Actions Options Views
entrypoint_53h:
mov r5, 0x1 ; tweak mov
mov r4, 0x0 ; tweak mov
// Xrefs: 6ch
loc_57h:
mov r3, 0x6d ; tweak mov
mov accu, r5
jz r3 ; x:loc_6dh x:loc_5bh
// Xrefs: 5ah
loc_5bh:
mov r3, 0x66 ; tweak mov
mov accu, r4
??? accu
mov r4, accu
mov accu, r5
and accu, r7
jz r3 ; x:loc_66h x:loc_63h
// Xrefs: 5ah
loc_6dh:
mov r7, r4 ; tweak mov
mov accu, 0x0
jz r6 ; [return]
// Xrefs: 62h
loc_63h:
or r4, 0x1 ; tweak or
// Xrefs: 62h
loc_66h:
mov accu, r5
??? accu
mov r5, accu
mov r3, 0x57 ; tweak mov
mov accu, 0x0
jz r3 ; [uncond jmp] x:loc_57h

```
func_0x53(x)
{
  r4 = 0
  r5 = 1
  while ( r5 )
  {
    r4 = ???(r4s)
    r4 |= 1 if ( x & r5 )
    r5 = ???(r5)
  }
  return r4
}
```

On remarque que si l'opération **???** décale le registre **accu** d'un bit vers la gauche, la fonction en 0x53 inverse les bits de l'octet passé en premier argument. On considère donc que la fonction 0x53 est une fonction *bitswap* et que l'instruction manquante est un **shl1**.

2.2.2 Émulation

Cette étape n'est pas indispensable mais permet de déboguer et d'instrumenter le code du programme en mode pas-à-pas. Connaissant toutes les instructions, il est très facile d'implémenter un émulateur en quelques lignes de code.

L'émulation de la sous-routine en 0x71 nous indique très vite qu'il s'agit d'une addition entre deux octets.

```
====================================
IP: 0x7d; ACCU: 0x2c
r0: 0x00, r1: 0x00, r2: 0x10, r3: 0x0f
r4: 0x63, r5: 0x0e, r6: 0xff, r7: 0xe0
====================================
7d: mov r4, accu
7e: mov accu, r7
7f: and accu, r2
80: jz r4
81: mov accu, 0x16
```

2.2.3 Analyse du programme

Après simplification, la boucle principale du programme ressemble au pseudo-code suivant :

```
   mov r0, 0x0
   mov r7, smd[0x10]
   mov r6, not r0
   call add                          # r7 = len(smd) - 1

   mov r6, 0x1
```

```
   call add                              # r6 = 1

loop:
   end if (r7 == 0)
   mov r7, 0x11
   mov r6, r0
   call add                              # r7 = 17 + r0

   mov r1, r7
   mov r6, smd[r1]
   mov r6, (smd[0xf & r0] or r6)         # r6 = smd[17 + r0] | smd[r0 % 16]
   mov r7, smd[r1]
   mov r7, (smd[0xf & r0] and r7)        # r7 = smd[17 + r0] & smd[r0 % 16]
   mov r7, (~r7 and r6)                  # r7 = smd[17 + r0] ^ smd[r0 % 16]
   call bitswap

   mov smd[r1], r7
   mov r6, 0x1
   mov r7, r0
   call add

   mov r0, r7                            # r0++
   jmp loop
```

La taille du bloc de données est lue à l'octet 17. Les données sont *xorée* en boucle avec la clé contenue dans les 16 premiers octets, et les bits de chaque octet sont réécrits à l'envers.

2.3 Déchiffrement du fichier *data*

L'algorithme est un simple **XOR** avec une clé de 16 octets appliquée en boucle. Les fréquences d'apparition des octets du clair sont donc conservées tous les 16 octets. Une simple énumération des octets chiffrés montrent que les octets du clair ne prennent qu'une soixantaine de valeurs différentes.

En faisant l'hypothèse que le clair est un texte encodé en *base64*, il devient trivial de bruteforcer la clé.

```ruby
#!/usr/bin/env ruby

KEYSIZE = 16

def bitswap(x)
  y = 0
  a = 1
  until a == 0x100
    y <<= 1
    y |= 1 unless (x & a).zero?
    a <<= 1
```

```
  end
  y
end

def is_base64?(c)
  ("A".."Z") === c or
  ("a".."z") === c or
  ("0".."9") === c or
  %w{+ / =}.include?(c) or c == "\n"
end

cipher = File.read("data").unpack('C*')

key = []
KEYSIZE.times do |n|
  sample = []
  (cipher.size / KEYSIZE).times do |l|
    sample.push cipher[n + l * KEYSIZE]
  end

  256.times do |k|
    key.push(k) and break if sample.all? {|c| is_base64?(bitswap(c ^ k).chr)}
  end
end

if key.size == KEYSIZE
  key_stream = key.cycle
  print cipher.map{|c| bitswap(c ^ key_stream.next)}.pack('C*')
else
  fail "Key not found"
end
```

```
$ ./decrypt_data.rb > atad
$ md5sum atad
6c0708b3cf6e32cbae4236bdea062979  atad

$ base64 -d atad > level3.ps
```

Il en résulte un fichier PostScript qui nous amène au niveau suivant.

3. Le *crackme* en PostScript

Le fichier déchiffré par le FPGA s'avère être un crackme écrit en PostScript. PostScript est un vieux langage d'Adobe né dans les années 80, avant l'apparition de PDF. Il est aujourd'hui toujours utilisé par de nombreuses imprimantes et pour décrire des polices de caractères.

L'une des particularités de PostScript est sa syntaxe postfixée qui le rend très pénible à lire pour un humain. Il ne s'agit pas d'un simple format descriptif, mais d'un vrai langage dynamique interprêté par une machine virtuelle à pile. PostScript possède des structures de boucles, de conditions, et peut même accéder au système de fichiers.

L'implémentation libre la plus répandue est GhostScript. Je n'ai pas été en mesure de trouver un débogueur assez complet pour émuler le script de cette épreuve, il a donc fallu s'y prendre à la main.

3.1 Analyse du programme

Pour commencer, quelques modifications sur le fichier s'imposent. La première, c'est d'indenter le code. La deuxième c'est de réactiver l'affichage des erreurs du script en commentant les lignes suivantes :

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

```
main
%clear
quit
```

Afin de comprendre l'état du programme, il est nécessaire de pouvoir inspecter les valeurs sur la pile. Le contenu de la totalité de la pile peut être affiché grâce à la commande native **`pstack`**. Cette commande devient malheureusement trop verbeuse lorsqu'elle est appelée dans une boucle, j'ai donc défini quelques petites commandes supplémentaires pour afficher seulement le haut de la pile :

```
% Affiche une string sur stdout
/puts { (%stdout)(w) file exch writestring } bind def
/debugstr 10000 string def

% Convertit le haut de la pile en string et l'affiche sur stdout.
/pp { dup debugstr cvs puts (\n) puts } bind def

% Affiche le type de l'élément en haut de la pile.
/ptype { dup type debugstr cvs puts (\n) puts pop } bind def
```

Nous pouvons dès lors nous attaquer au script :

```
$ gs -q -dNODISPLAY -- level3.ps
no key provided
usage: gs -- script.ps key
```

Le programme attend une clé en argument. Le code nous indique que la clé en question fait 16 octets et doit être passée sous la forme de 32 caractères hexadécimaux :

```
/main
{ mark shellarguments
  % if argc == 1
  { counttomark 1 eq
    % la clé est constituée de 32 caractères hexadécimaux
    { dup length exch /ReusableStreamDecode filter exch 2 idiv string
      readhexstring pop dup length 16 eq
      { ... }
      if
    }
    if
  }
  if
}
bind def
```

Le fichier embarque aussi 4 flux nommés **I1**, **I2**, **I3** et **I4** encodés en hexadécimal au début du fichier. Ces flux contiennent des données binaires sur lesquelles le script travaille avec la clé passée en argument.

Le flux **I1** puis le flux **I2** sont d'abord décodés. S'en suit une boucle un peu tordue et l'appel à une instruction **exec**. L'instruction **exec** permet d'évaluer une string sur la pile comme du code. Nous pouvons donc en déduire que l'un des flux est déchiffré et évalué comme du PostScript.

3.2 Déchiffrement de I2

```
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
    %
    % soudain... un xor
    %
    get xor 3 1 roll
  }
  for
  [...]
}
for
[...]
cvx exec
```

La boucle ne semble pas faire grand chose de très intéressant à part un **xor**.

Il suffit de poser un **pp** après le **xor** et avant le **exec** pour constater que les données *xorées* sont celles exécutées quelques lignes plus bas. Le premier argument du **xor** correspond aux octets du flux **I2** décodé. Si nous affichons le deuxième argument et passons une clé au programme, voici ce que nous pouvons observer :

```
$ gs -q -dNODISPLAY -- level3.ps 1234567890abcdeffedcba0987654321
86
120
86
120
147
224
129
180
...
```

Or nous avons :
- **86 = 0x56**
- **120 = 0x78**
- **147 = (0x56 ^ I2[0])**
- **224 = (0x78 ^ I2[1])**
- **129 = (0x56 ^ I2[2])**
- **180 = (0x78 ^ I2[3])**
- **...**

Le clair est donc chiffré avec une clé de 4 octets, composé des octets 3 et 4 de la clé répétés deux fois. Le déchiffrement s'applique en *xorant* un bloc de 4 octets avec cette clé pour le premier bloc, et avec le précédent bloc en clair pour les suivants.

On déroule le chiffrement à l'envers en *xorant* chaque bloc avec tous les précédents pour obtenir un chiffré *xoré* avec une unique clé. On bruteforce ensuite 2 fois un octet de clé pour obtenir un texte PostScript valide.

```ruby
#!/usr/bin/env ruby

def is_ascii?(c); (" ".."~") === c or c == "\n" end

x = 0
cipher = File.read(ARGV[0])
             .unpack('L*')
             .map {|w| x = x ^ w }
             .pack('L*').unpack('C*')

key = [[],[]]
2.times do |n|
  sample = []
  (cipher.size / 2).times do |l|
    sample.push cipher[n + l * 2]
  end
```

```
  256.times do |k|
    key[n].push(k) if sample.all? {|c| is_ascii?((c ^ k).chr)}
  end
end

key[0].each do |k0|
  key[1].each do |k1|
    key_stream = [k0, k1].cycle
    plain = cipher.map{|c| c ^ key_stream.next}.pack('C*')

    # On cherche une instruction PostScript parmi tous les clairs en ASCII.
    if plain =~ / exch /
      puts "Found key: #{(k0.chr + k1.chr).unpack('H*')[0]}"
      print plain
    end
  end
end
```

```
$./decrypt_stream.rb I2.bin
Found key: f7a8
```

Le script déchiffré est exécuté via **exec**. Le script définit seulement une nouvelle fonction nommée **calc**. Les constantes impliquées dans le code nous indique qu'il s'agit d'une fonction MD5.

3.3 Déchiffrement de I4

Le code de **I4** est déchiffré et exécuté exactement de la même façon que le code de **I2**, mais en utilisant les 2 premiers octets de la clé passée en argument.

```
$ ./decrypt_stream.rb I4.bin
Found key: bac9
```

En appelant le script avec ce début de clé, un nouveau message apparaît :

```
$ gs -q -dNODISPLAY -- level3.ps `ruby -e 'puts "bac9f7a8" + "00" * 12'`
Key is invalid. Exiting ...
```

Le code de **I4** utilise les 12 octets de clé restants pour déchiffrer le contenu de **I1**. En cas de succès, le fichier déchiffré est écrit dans **output.bin**.

3.4 Cassage de I1 en boite noire

L'algorithme utilisé ici ne semble pas standard et effectue 6 itérations en prenant à chaque fois 4 octets de la clé passée en argument. A chaque itération **i**, l'algorithme utilise les octets **i + 2** à **i + 6** de la clé pour déchiffrer **I1** en mémoire.

A la fin de chaque itération, le contenu des données déchiffrées est haché en MD5 grâce à la fonction **calc** de **I2**, et le *hash* est comparé avec l'un des 6 *hashs* stockés dans **I3**. Si les *hashs* ne correspondent pas lors d'une passe, le programme s'arrête avec le message *Key is invalid*.

```
% Calcul du MD5 et comparaison avec le prochain hash de I3
pop calc I3 16 string readstring pop
ne
{
  0 1 1073741823 {pop} for % sleep() de plusieurs secondes
  (Key is invalid. Exiting ...\n) error
  flush
  quit
}
if
```

Comme nous possédons les 4 premiers octets de la clé, il suffit de bruteforcer 16 bits de clé pour chaque itération de l'algorithme jusqu'à ce que les 6 *hashs* MD5 soient corrects.

Pour cela, il y a deux possibilités :
- soit on retranscrit tout l'algorithme depuis le PostScript vers un langage où il pourra être cassé efficacement comme le C ;
- soit on bruteforce l'algorithme en boite noire dans l'interpréteur, sans chercher à comprendre le PostScript, et on conserve ainsi sa santé mentale.

J'ai opté pour la deuxième solution. Bien que le calcul soit plus long (à cause de la lenteur des opérations en PostScript), le bruteforce est beaucoup plus rapide à implémenter et à mettre en place, puisqu'il n'est pratiquement pas nécessaire d'analyser le code.

3.4.1 Bruteforce des 12 octets restants de la clé

Le flux **I4** déchiffré a été extrait dans un fichier à part et modifié pour être appelé directement par **gs** avec la clé en argument.

Au début de chaque itération, je sauvegarde la pile du programme dans un fichier **last_stack.ps**. La boucle principale de **I4** a été supprimée pour que l'état de la pile soit rechargé depuis ce fichier à chaque exécution. De cette façon, chaque itération peut être cassée en temps constant, sans avoir a réexécuter les itérations précédentes.

```
shellarguments pop /ReusableStreamDecode filter 16 string readhexstring
(last_stack.ps) (r) file dup resetfile
65000 string readstring pop
exch pop
cvx exec
```

Le calcul du MD5 et la comparaison avec **I3** ont aussi été supprimés du programme. À la place, on affiche le contenu du tampon à passer dans MD5 sur *stdout*. Il est ainsi possible de récupérer les données en dehors de **gs** et de calculer le MD5 plus rapidement.

```
  pop pp
  quit
```

Pour chaque itération, l'espace de 16 bits à casser est divisé et le calcul est effectué en parallèle sur plusieurs coeurs grâce au script suivant :

```
#!/usr/bin/env ruby

require 'digest/md5'

md5 = File.read "I3.bin", encoding:"binary"

KNOWN_PREFIX = 'bac9f7a8'
ITER = 6 - (32 - KNOWN_PREFIX.size) / 4
N = 16
N.times do |n|
  fork {
    (65536 / N).times do |k|
      base = n * (65536 / N)
      key = KNOWN_PREFIX + (base + k).to_s(16).rjust(4,'0')
      out = %x{gs -dNODISPLAY -q -- i4_single_iter.ps #{key.ljust(32,'0')}}

      if Digest::MD5.digest(out.chomp) == md5[16*ITER,16]
        puts "Found key block #{ITER}: #{base + k}"
        exit
      end
    end
  }
end
```

Le cassage de 16 bits de clé a pris entre 20 et 30 minutes sur un processeur Intel *Core i7* à 8 coeurs. Les 12 octets ont été totalement cassés en moins de 3 heures de calcul.

Connaissant la clé complète, il ne reste plus qu'à exécuter le script original avec cette clé, et à récupérer **I1** déchiffré dans le fichier **output.bin**.

```
$ gs -q -dNODISPLAY -- level3.ps bac9f7a8721fad3c9fcf271eed9abbc8
$  file output.bin
output.bin: vCard visiting card
```

4. Reconstruction de l'adresse e-mail finale

Le challenge est pratiquement terminé à cette étape. Le fichier **`output.bin`** généré s'avère être le carnet d'adresses que nous recherchons.

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

L'adresse que nous recherchons pour valider le challenge n'est pas en clair, mais se présente comme une suite d'appels système Linux.

```
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
```

Il suffit de prendre le numéro de chaque appel système sur un système Linux *amd64* et de concaténer les octets pour obtenir l'adresse recherchée.

```ruby
#!/usr/bin/env ruby

SYSCALL_H = File.read("/usr/include/asm/unistd_64.h")

def syscall_to_num(name)
  SYSCALL_H.each_line do |line|
    return $1.to_i if line =~ /^#define #{name} (\d+)$/
  end

  fail "Cannot find syscall #{name}"
end

puts %w{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}
    .map {|sc| syscall_to_num(sc.gsub(/^sys_|stub_/, '__NR_')) }
    .pack('C*')
```

L'adresse e-mail finale apparait alors :

```
$ ./syscall_to_mail.rb
59575e0e71f1e3e9946bc307fc7a608d0b568458@challenge.sstic.org
```

5. Le mot de la fin

J'ai particulièrement apprécié la partie analyse du CPU sous forme de circuit électronique. C'était une épreuve originale et abordable même pour des non-spécialistes en électronique.

Merci à mes collègues de bureau pour leurs conseils et la relecture de cette solution.

Et bien sûr merci aux concepteurs du challenge !

A. Sources

A.1 emul.rb

```
#!/usr/bin/env ruby

class SSTICEmulator
  class SMD
    attr_reader :addr
    def initialize(addr); @addr = addr end
    def self.[](addr); self.new(addr) end

    def to_s
      "smd[#{Instruction.dump_arg(addr)}]"
    end
  end

  class Instruction
    attr_reader :addr, :opcode, :arg1, :arg2

    def self.dump_arg(arg)
      case arg
        when SMD then arg.to_s
        when Symbol then arg.to_s
        when Fixnum then "0x#{arg.to_s(16).rjust(2, '0')}"
      end
    end

    def r(n)
      "r#{n}".to_sym
    end

    def disass
      "#{@addr.to_s(16).rjust(2,'0')}: #{disass_insn}"
    end

    def disass_insn
      str = @opcode.dup
      str << " #{Instruction.dump_arg(@arg1)}" if @arg1
      str << ", #{Instruction.dump_arg(@arg2)}" if @arg2

      str
    end

    def initialize(addr, insn)
      @addr = addr
      opcode, reg = (insn >> 3), (insn & 7)
```

```
      if insn == 0xc8
        @opcode = "end"
      elsif (insn >> 7) == 0
        @opcode, @arg1, @arg2 = "mov", :accu, insn
      elsif insn == 0b11010000
        @opcode, @arg1, @arg2 = "mov", :accu, SMD[:accu]
      else
        case opcode
        when 0b10111
          @opcode, @arg1 = "jz", r(reg)
        when 0b10110
          @opcode, @arg1, @arg2 = "mov", r(reg), :accu
        when 0b10101
          @opcode, @arg1, @arg2 = "mov", :accu, r(reg)
        when 0b10001
          @opcode, @arg1, @arg2 = "and", :accu, r(reg)
        when 0b10010
          @opcode, @arg1, @arg2 = "or", :accu, r(reg)
        when 0b10100
          @opcode, @arg1 = "not", :accu
        when 0b11100
          @opcode = "nop"
        when 0b11011
          @opcode, @arg1 = "shl1", :accu
        when 0b11000
          @opcode, @arg1, @arg2 = "mov", SMD[r(reg)], :accu
        else
          @opcode = "unknown"
        end
      end
    end
  end

  attr_reader :halted
  attr_reader :ip, :registers, :cycles
  attr_reader :data

  def initialize(code, data, eip = 0)
    @registers = Hash.new(0)
    @ip = eip
    @cycles = 0
    @code = []
    @data = data
    @halted = false
    addr = 0
    code.each do |byte|
      @code.push(Instruction.new(addr, byte))
      addr += 1
    end
  end
```

```
def to_s
  regs = Array.new(8) { |i| "r#{i}".to_sym }

  "=" * 38 + "\n" +
  "IP: 0x#{@ip.to_s(16).rjust(2, '0')}; ACCU: 0x#{@registers[:accu].to_s(16).rjust(2,'0')
  regs[0..3].map { |reg|
    "#{reg.to_s}: 0x#{@registers[reg].to_s(16).rjust(2,'0')}"
  }.join(", ") + "\n" +
  regs[4..7].map { |reg|
    "#{reg.to_s}: 0x#{@registers[reg].to_s(16).rjust(2,'0')}"
  }.join(", ") + "\n" +
  "=" * 38 + "\n" +

  @code[@ip, 5].map {|insn| insn.disass}.join("\n") + "\n\n"
end

def disass
  @code.each do |insn|
    puts insn.disass
  end
end

def step
  insn = @code[@ip]
  @cycles += 1

  case insn.opcode
  when "jz"
    if @registers[:accu] == 0
      # voodoo !
      if (0x71...0xe0) === @ip
        @ip = @registers[insn.arg1] | 0x80
      else
        @ip = @registers[insn.arg1]
      end
      return
    end
  when "or"
    @registers[insn.arg1] |= @registers[insn.arg2]
  when "and"
    @registers[insn.arg1] &= @registers[insn.arg2]
  when "not"
    @registers[insn.arg1] = (~@registers[insn.arg1]) & 0xff
  when "shl1"
    @registers[insn.arg1] = (@registers[insn.arg1] << 1) & 0xff
  when "end"
    @halted = true
    return
  when "mov"
    if insn.arg1.is_a? SMD
      @data[@registers[insn.arg1.addr],1] = @registers[insn.arg2].chr
```

```
      elsif insn.arg2.is_a? SMD
        @registers[insn.arg1] = @data[@registers[insn.arg2.addr],1].ord
      elsif insn.arg2.is_a? Fixnum
        @registers[insn.arg1] = insn.arg2
      else
        @registers[insn.arg1] = @registers[insn.arg2]
      end
    end

    @ip += 1
  end

  def run; step until @halted end
  def run_until(&cond); step until @halted or cond[self] end
end

smp = File.read(ARGV[0], encoding:"binary").unpack('C*')
smd = File.read("data", encoding:"binary")
key = ARGV[1]

emul = SSTICEmulator.new(smp, key + 224.chr + smd[0,224])
puts emul
while not emul.halted and gets
  emul.step
  puts emul
end
```

A.2 fpga/opcode.rb

```
require 'fpga/main'

module Metasm

  class Fpga

    def addop(name, bin, *args)
      o = Opcode.new(name)

      o.bin = bin << 3
      o.args.concat(args & @fields_mask.keys)
      (args & @valid_props).each { |p| o.props[p] = true }

      (args & @fields_mask.keys).each { |f|
        o.fields[f] = [@fields_mask[f], @fields_shift[f]]
      }

      @opcode_list << o
    end
```

```
    def init
      @opcode_list = []
      @fields_mask = {
        :reg => 0b111, :smd => 0b111, :accu => 0x0, :imm => 0b1111111 }

      @fields_shift = {
        :reg => 0x0, :smd => 0x0, :accu => 0x0, :imm => 0b0 }

      @valid_props = [:setip, :stopexec]

      addop 'mov', 0b00000, :accu, :imm
      addop 'jz',  0b10111, :reg, :setip, :stopexec
      addop 'mov', 0b10110, :reg, :accu
      addop 'mov', 0b10101, :accu, :reg
      addop 'and', 0b10001, :accu, :reg
      addop 'or',  0b10010, :accu, :reg
      addop 'not', 0b10100, :accu
      addop 'nop', 0b11100
      addop 'shl1', 0b11011, :accu
      addop 'mov', 0b11010, :accu, :smd
      addop 'mov', 0b11000, :smd , :accu
      addop 'end', 0b11001,  :stopexec
    end


  end

end
```

A.3 fpga/decode.rb

```
require 'fpga/opcode'
require 'metasm/decode'

module Metasm
  class Fpga
    def build_opcode_bin_mask(op)
      op.bin_mask = 0
      op.args.each { |f|
        op.bin_mask |= @fields_mask[f] << @fields_shift[f]
      }
      op.bin_mask = 0xff ^ op.bin_mask
    end

    def build_bin_lookaside
      lookaside = Hash.new { |h,k| h[k] = []}
      opcode_list.each { |op|
        next if not op.bin.kind_of? Integer
```

```
      build_opcode_bin_mask op
      lookaside[op.bin >> 3] << op
    }
    lookaside
  end

  def decode_bin(edata)
    edata.decode_imm(:u8, @endianness)
  end

  def decode_findopcode(edata)
    return if edata.ptr >= edata.data.length

    di = DecodedInstruction.new(self)
    val = edata.decode_imm("u#{@instrlength}".to_sym, @endianness)
    edata.ptr -= @instrlength/8

    val = 0 if (val >> 7) == 0
    op = @bin_lookaside[val >> 3].select{|opcode| (val & opcode.bin_mask) == opcode.bin}

    if op == nil or op.size != 1
      raise "Die with your boots on !"
    else
      op = op.first
    end

    di if di.opcode = op
  end

  def decode_instr_op(edata, di)
    before_ptr = edata.ptr
    op = di.opcode
    di.instruction.opname = op.name
    val = edata.decode_imm("u#{@instrlength}".to_sym, @endianness)

    field_val = lambda{ |f|
      r = (val >> @fields_shift[f]) & @fields_mask[f]

      case f
      when :reg
        r = GPR.new(r&0x7)

      when :imm
        r = Imm.new(r)

      when :accu
        r = Acc.new()

      when :smd
        r = Memref.new(di.opcode.bin == 0b11010000 ? Acc.new() :  GPR.new(r&0x7))
      end
```

```
    r
  }

  op.args.each { |a|
    di.instruction.args << field_val[a]
  }

  di.bin_length += edata.ptr - before_ptr
  di
end

def disassembler_default_func
  df = DecodedFunction.new
  df.backtrace_binding = {}
  8.times{|i| df.backtrace_binding["r#{i}".to_sym] = Expression::Unknown}
  df.backtracked_for = []
  df.btfor_callback = lambda { |dasm, btfor, funcaddr, calladdr|
    if funcaddr != :default
      btfor
    elsif di = dasm.decoded[calladdr] and di.opcode.props[:saveip]
      btfor
    else []
    end
  }
  df
end

def backtrace_binding
  @backtrace_binding ||= init_backtrace_binding
end

def init_backtrace_binding
  @backtrace_binding ||= {}

  opcode_list.map { |ol| ol.name }.uniq.each { |op|
    binding = case op
    when 'mov' ; lambda { |di, a0, a1|  ret = {  a0 => a1  }}
    when 'not' ; lambda { |di, a0| ret = {  a0 => Expression[a0, :^, 0xff]  }}
    when 'and' ; lambda { |di, a0, a1| ret = {  a0 => Expression[a0, :&, a1]  }}
    when 'or' ; lambda { |di, a0, a1| ret = {  a0 => Expression[a0, :|, a1]  }}
    else  {}
    end
    @backtrace_binding[op] ||= binding if binding
  }

  @backtrace_binding
end

def get_backtrace_binding(di)
  a = di.instruction.args.map { |arg|
```

```
        case arg
        when GPR, Acc, Imm; arg.symbolic
        when Memref; arg.symbolic(di.address, arg.sz)
        else arg
        end
      }

      if (binding = backtrace_binding[di.opcode.basename]) != {}
        bd = binding[di, *a]
      else
        puts "unhandled instruction to backtrace: #{di}" if $VERBOSE
        {:incomplete_binding => Expression[1]}
      end
    end

    def get_xrefs_x(d, di)
      return [] if not di.opcode.props[:setip]

      args = [case di.instruction.opname
        when /jz/
        di.instruction.args.last
        else di.instruction.args.last
        end]

      ddi = d.di_at(di.address-1)

      if ddi.instruction.to_s == "mov accu, 0x0"
        args = []

        ddi = d.di_at(di.address-2)
        if ddi.instruction.to_s == "mov r7, accu"
          di.add_comment("[return]")

        elsif (dddi = d.di_at(di.address-3))

          d2 = d.di_at(di.address-2)
          d4 = d.di_at(di.address-4)
          d6 = d.di_at(di.address-6)

          if (d2 and d4 and d6)
            regs = [d2, d4, d6].map{|di| di.instruction.args.first.i}

            if (regs == [4, 5, 6]) or (regs == [5, 6, 7]) or  (regs == [4, 5, 7])
              di.add_comment("[call]")
              args = [di.next_addr]
            else
              di.add_comment("[uncond jmp]")
              args = [di.instruction.args.last]
            end

          end
```

```
        end

      else
        args << di.next_addr
      end

      args.map{|a|
        [Expression[
          case a
          when Reg; a.symbolic
          else a
          end]]}
    end

    def backtrace_is_function_return(expr, di=nil)
      false
    end

    def delay_slot(di=nil)
      (di.opcode.props.has_key? :delay_slot) ? 1 : 0
    end

  end

end
```

A.4 fpga/main.rb

```
#    This file is part of Metasm, the Ruby assembly manipulation suite
#    Copyright (C) 2006-2010 Yoann GUILLOT
#
#    Licence is LGPL, see LICENCE in the top-level directory

require 'fpga/main'

module Metasm

  class Fpga < CPU

    def initialize(e = :little)
      super()
      @endianness = :little
      @size = 8
      @instrlength = 8
    end

    class Reg
      include Renderable
```

```
  def ==(o)
    o.class == self.class and (not respond_to?(:i) or o.i == i)
  end
end

# general purpose reg
class GPR < Reg
  attr_accessor :i

  def initialize(i); @i = i end
  Sym = (0..7).map { |i| "r#{i}".to_sym }

  def symbolic
    Sym[@i]
  end

  def render
    ["r#@i"]
  end

end

class Acc < Reg
  attr_accessor :i

  def initialize(); @i = 0 end
  Sym = [:accu]

  def symbolic
    Sym[@i]
  end

  def render
    ["accu"]
  end
end

class Imm
  include Renderable

  attr_accessor :val

  def initialize(val)
    @val  = val
  end

  def symbolic
    Expression[val].reduce_rec
  end
```

```
      def render
        ["0x#{@val.to_s(16)}"]
      end
    end

    class Memref
      attr_accessor :sz, :base

      def initialize(reg)
        @base  = reg
        @sz =  1
      end

      def symbolic(orig, sz)
        b = @base
        b = b.symbolic if b.kind_of? Reg
        Indirection[ e = Expression[[b, :+, 0x1000]].reduce, sz, orig]
      end

      include Renderable

      def render
        rep = ["smd[", @base, ']']
      end

    end

    def init_opcode_list
      init
    end

  end
end
```