

Solution du Challenge SSTIC 2013

Auteur : Michaël Sanchez
Date : 30/04/2013

Sommaire

1.	Introduction	4
2.	Cache-cache	4
2.1.	Découverte du challenge	4
2.2.	À la recherche du canal caché	6
2.3.	Déchiffrement de l'archive	7
3.	En toute logique.....	8
3.1.	Analyse du fichier decrypt.py	8
3.2.	Analyse du fichier s.ngr.....	8
3.3.	Analyse du bloc x_u	11
3.4.	Analyse du bloc x_accu	12
3.5.	Analyse du bloc x_r	14
3.6.	Analyse du bloc finished	16
3.7.	Analyse du bloc x_ip	16
3.8.	Analyse du bloc x_smd.....	18
3.9.	Synthèse de l'analyse du schéma logique.....	18
3.10.	Analyse du fichier smp.py et cryptanalyse	19
4.	Polski Mówię.....	21
4.1.	Compréhension du programme principal.....	21
4.2.	Cryptanalyse de I2 et I4	22
4.3.	Analyse de I2.....	24
4.4.	Analyse de I4.....	25
4.5.	Reconstruction de la clé finale.....	25
5.	Récupération de l'email de soumission	26
6.	Conclusion.....	27
7.	Annexes et codes sources.....	28
7.1.	search_key.py	28
7.2.	extrait de smp.py	30
7.3.	<i>decrypt.py</i>	31
7.4.	disassemble_smp.py.....	32
7.5.	dis_smp.txt.....	32
7.6.	decrypt_data.py.....	35
7.7.	script.ps.....	36
7.8.	fonction principale de scripts.ps indentée et commentée	37
7.9.	decodeI2I4.py.....	39
7.10.	fonction I4 indentée et commentée	40
7.11.	descript.py	42
7.12.	decodevcard.py.....	44

Table des illustrations

FIGURE 1: BLOC S DU CIRCUIT LOGIQUE.....	9
FIGURE 2: VUE DETAILLEE DU CIRCUIT LOGIQUE.....	10
FIGURE 3: BLOC X_U.....	11
FIGURE 4: BLOC X_ACCU.....	12
FIGURE 5: ALIMENTATION DU BLOC X_ACCU.....	12
FIGURE 6: BLOC PWR3.....	13
FIGURE 7: BLOC ACCU_W_IMP.....	14
FIGURE 8: BLOC X_R.....	14
FIGURE 9: BLOC X_R.....	15
FIGURE 10: BLOC FINISHED.....	16
FIGURE 11: BLOC X_IP.....	16
FIGURE 12: PORTES AVANT ENTREE W DANS X_IP.....	17
FIGURE 13: BLOC X_SMD.....	18

1. Introduction

Le présent document décrit une solution au challenge SSTIC 2013. Les étapes d'analyse, les outils utilisés ainsi que les codes sources programmés sont présentés et permettent de comprendre comment l'adresse email de soumission a été retrouvée.

Cette présentation part du postulat que le lecteur détient des connaissances de bases en cryptographie, rétro-ingénierie, circuit logique et programmation. Ces notions ne seront donc pas présentées en détail.

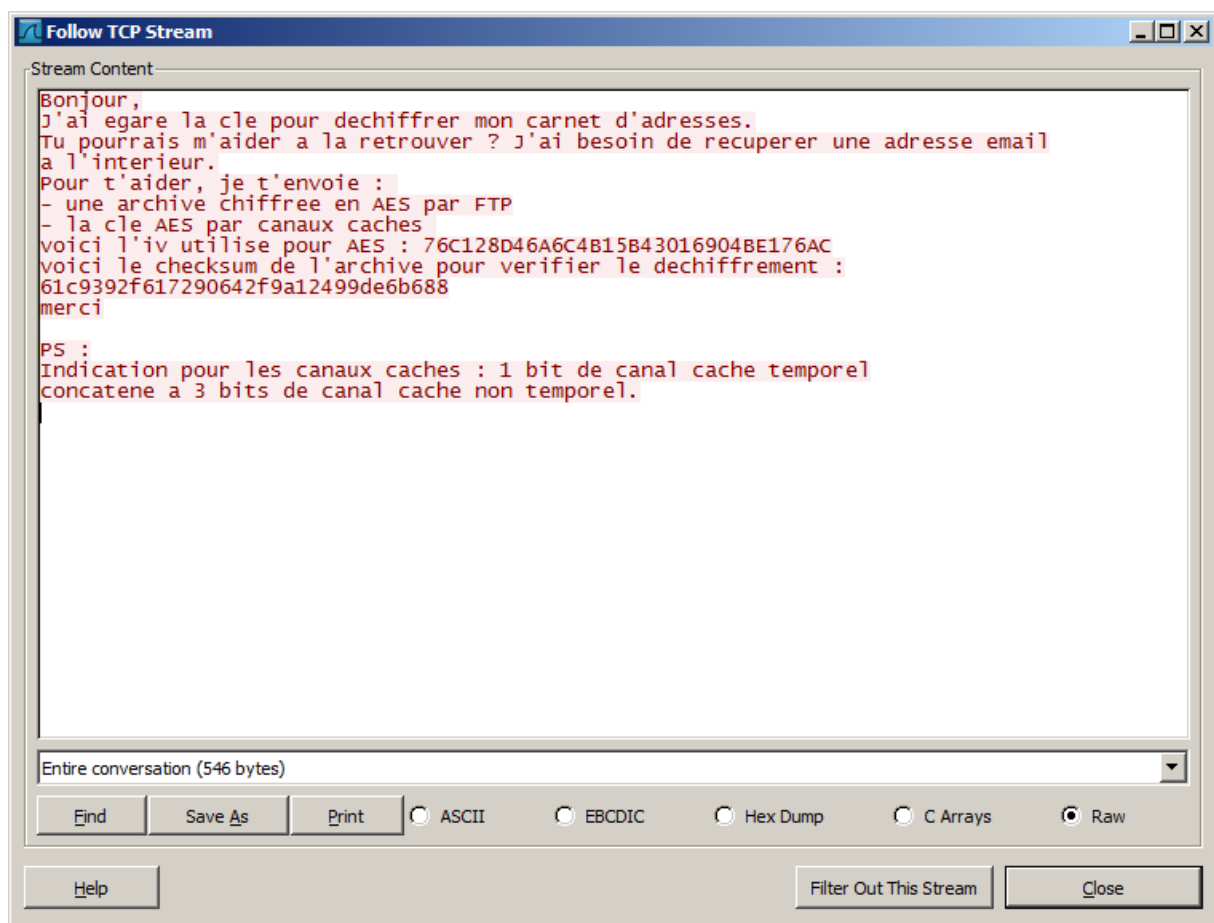
2. Cache-cache

2.1. Découverte du challenge

L'objectif du challenge SSTIC est de découvrir une adresse email au format ...@challenge.sstic.org dans une trace réseau mise à disposition à l'adresse <http://static.sstic.org/challenge2013/dump.bin>.

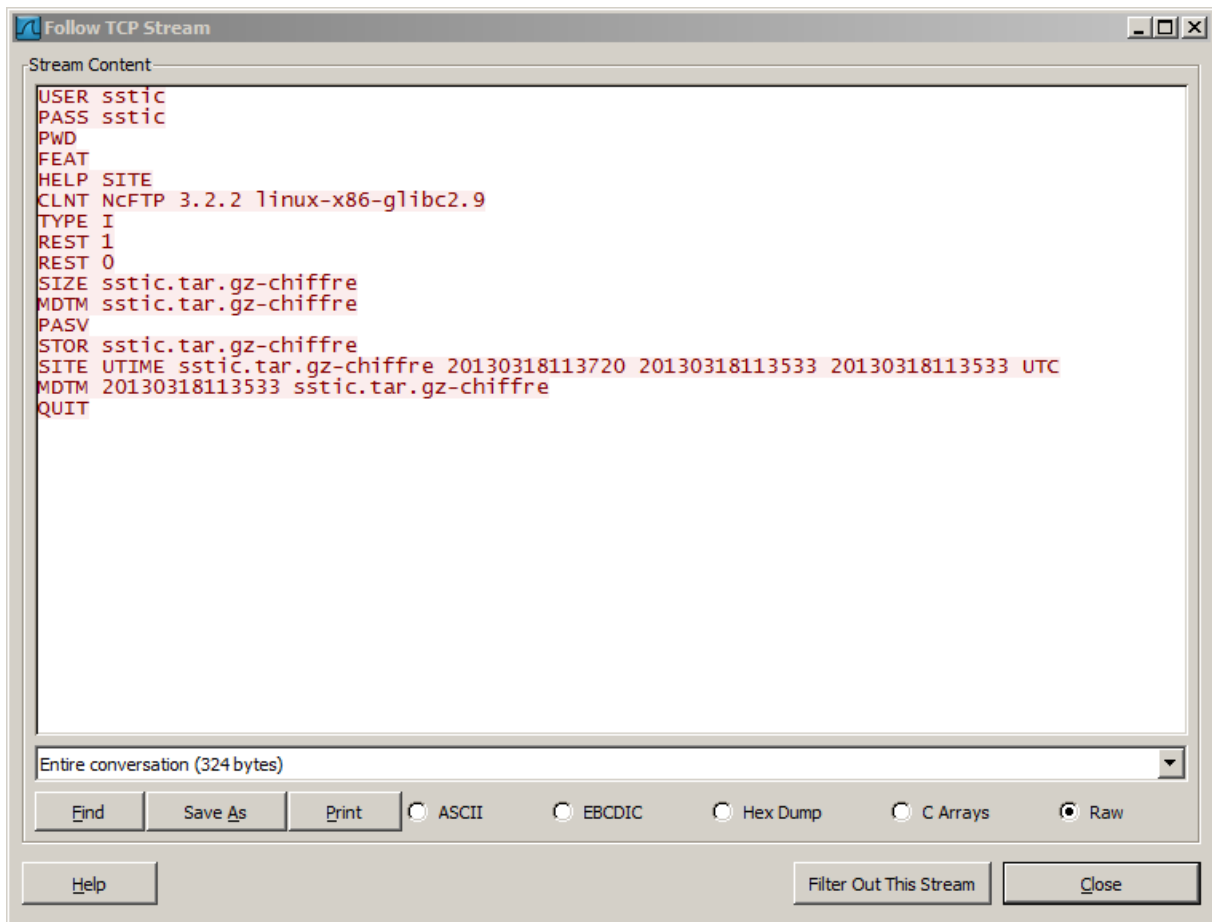
Une fois le fichier téléchargé et son empreinte md5 vérifiée, on utilise le logiciel Wireshark (<https://www.wireshark.org/>) pour l'analyser. On découvre une trace réseau de 228 paquets.

Le 3^{ème} paquet semble contenir les fragments d'une conversation. On choisit alors d'afficher le flux TCP de ce paquet :



A la lecture de la conversation, on comprend qu'une archive chiffrée en AES a été transmise par FTP et que la clé symétrique utilisée a été transmise par un canal caché qui reste à découvrir.

En continuant l'analyse de la trame réseau, on constate une succession de paquets ICMP (paquet 6 à 70, soit un total de 65 paquets) puis un échange de fichier par FTP.

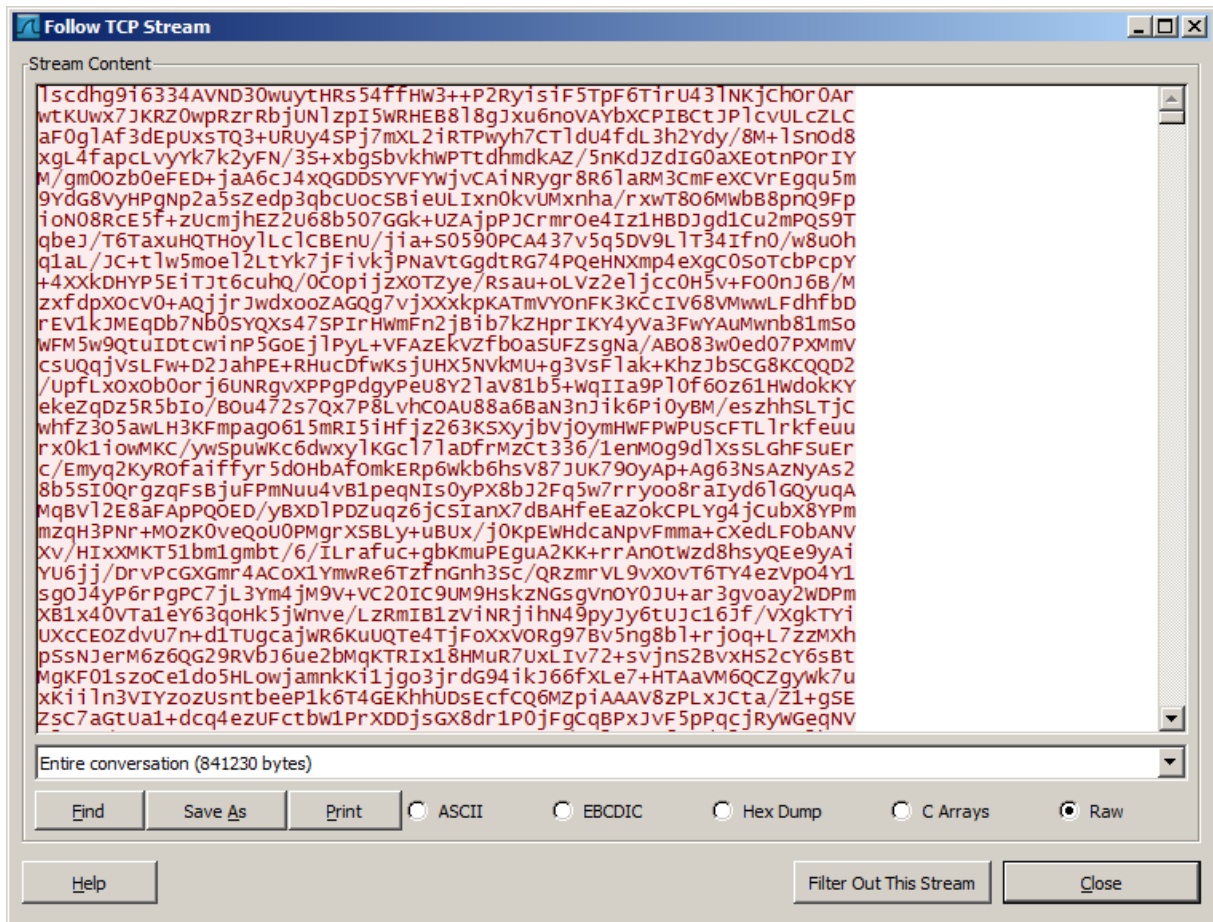


The screenshot shows a window titled "Follow TCP Stream" with a "Stream Content" pane. The pane displays the following FTP session output:

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

Below the stream content, there is a dropdown menu showing "Entire conversation (324 bytes)". At the bottom of the window, there are several buttons: "Find", "Save As", "Print", and a set of radio buttons for "ASCII", "EBCDIC", "Hex Dump", "C Arrays", and "Raw" (which is selected). There are also "Help", "Filter Out This Stream", and "Close" buttons.

Le fichier échangé, nommé sstic.tar.gz-chiffre, est transmis à partir du 93^{ème} paquet. On procède alors à l'enregistrement de ce fichier de 841230 octets.



On constate dès à présent que le fichier est encodé en base64. Nous avons l'archive chiffrée, reste à découvrir le canal caché.

2.2. À la recherche du canal caché

Nous recherchons le canal caché qui a été utilisé pour transmettre la clé AES. Cette dernière peut être de longueur 128, 192 ou 256 bits. D'après les indications transmises lors de la conversation, la clé est constituée de la concaténation d'un bit de canal caché temporel et de 3 bits de canal caché non temporels, ce qui donne un total de 4 bits par transmission du canal caché.

La succession des 65 paquets ICMP étant suspicieuse, on devine facilement que le canal caché est inclus dans ces paquets. De plus, en extrayant 4 bits de 64 paquets, nous obtiendrions 256 bits, ce qui correspondrait à la taille de la clé recherchée.

Le bit temporel est probablement lié à l'heure d'envoi des paquets ICMP. En considérant les écarts de temps entre les différents paquets, on s'aperçoit rapidement que ce dernier est de 1 ou 2 secondes entre les 65 paquets un à un. On suppose avoir trouvé le premier bit en faisant la correspondance suivante :

différence de 1 seconde => bit à 0

différence de 2 secondes => bit à 1

Comme il est impossible de savoir si cette correspondance est juste, nous testerons également la correspondance opposée lors de la phase de déchiffrement.

Pour trouver les bits non temporels du canal caché, nous analysons les champs de la couche IP à la recherche de valeurs suspectes.

La 1^{ère} valeur suspecte qui apparaît dans la couche IP des 64 premiers paquets ICMP est celle du champ TTL qui vaut 10, 20, 30 ou 40. Cela représente 4 valeurs qui pourraient être représentées sur 2 bits (00, 01, 10, 11).

La 2^{ème} valeur suspecte est celle du champ *Differentiated services* (ancien champ Type Of Service) dont la valeur est soit 2 soit 4. Comme pour le bit temporel, nous transcoderons ces valeurs sur un bit.

Les différentes combinaisons de ces 4 bits vont nous donner un ensemble de clés que nous testerons pour déchiffrer l'archive transmise par FTP. Ces clés seront par ailleurs utilisées avec les modes de déchiffrement CBC, CFB et OFB car un vecteur d'initialisation IV a été transmis lors de la conversation ce qui laisse supposer l'usage de l'un de ces modes.

Avant de générer toutes les clés possibles, nous calculons le nombre maximal de déchiffrement à réaliser afin de savoir si le bruteforce est concevable en un temps acceptable. Le nombre de déchiffrement maximum est :

- 2 (combinaisons du bit temporel)
- x 2 (combinaisons du bit du champ DiffServ)
- x 24 (combinaisons des 2 bits du champ TTL)
- x 2 (combinaisons pour la concaténation du bit temporel et des 3 bits non temporels :
time_bit|diffserv_bit|ttl_bits ou *time_bit|ttl_bits|diffserv_bit*)
- x 3 (modes de chiffrement CBC, CFB, OFB)
- = **576 tests de déchiffrement à réaliser au maximum**

2.3. Déchiffrement de l'archive

Le code python *search_key.py* (cf. annexe 7.1) permet de générer l'ensemble des clés candidates et de les tester selon les 3 modes de déchiffrement.

La clé est rapidement trouvée et affichée :

```
Decryption done with key
dd8cf2d52e69aafb734e3acd0e4a69e83ed93bc4870ecd0d5b6faad86a63ae94 in mode
CBC!!
```

Le fichier *sstic.tar.gz* est alors disponible. On vérifie qu'il s'agit bien d'une archive gzip.

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

Nous décompressons l'archive et obtenons un dossier nommé *archive* avec 4 fichiers.

```
$ ls archive
data decrypt.py s.ngr smp.py
```

3. En toute logique

3.1. Analyse du fichier decrypt.py

Nous commençons par découvrir les 4 fichiers précédemment obtenus.

```
$ file *  
data:          data  
decrypt.py:    Python script, ASCII text executable  
s.ngr:         data  
smp.py:        ASCII text
```

Le fichier *smp.py* contient seulement la définition d'un tableau de 231 entiers définis en hexadécimal (cf. annexe 7.2)

Le fichier *data* semble être un fichier de données binaires brutes. Le programme *strings* sur ce fichier ne donne aucune information intéressante.

Le fichier *decrypt.py* est un script Python dont le code source est présenté en annexe 7.3. A la lecture de ce script, nous comprenons que :

- le script prend un seul paramètre en entrée ;
- le paramètre à saisir est une clé de 16 octets au format hexadécimal ;
- le fichier *data* est lu successivement par bloc de 224 octets ;
- le bloc de données lu ainsi que la clé saisie en paramètre sont transmis à un device initialisé avec le fichier *sp.ngr* (NB : le fichier présent dans l'archive décompressée s'appelle *s.ngr* et non pas *sp.ngr*. Nous supposons qu'une coquille s'est glissée lors du développement du code et que le développeur aura mal recetté son application☺) ;
- le tableau d'entiers du fichier *smp.py* est également transmis au device ;
- un résultat est retourné par le device pour chaque bloc de données transmis ;
- les différents résultats retournés par le device sont concaténés en un seul bloc de données et une empreinte md5 est calculée ;
- si l'empreinte md5 du bloc de données est égale à *6c0708b3cf6e32cbae4236bdea062979* alors le bloc de données est « base64 décodé » et le résultat est écrit dans un fichier nommé *atad*.

En résumé, cette étape du challenge va consister à trouver une clé de 16 octets qui nous permettra de générer un fichier encodé en base64 dont l'empreinte md5 sera *6c0708b3cf6e32cbae4236bdea062979*.

Nous passons alors à l'analyse du fichier *s.ngr*.

3.2. Analyse du fichier s.ngr

Nous commençons par regarder l'entête du fichier.

```
$ strings s.ngr | head -n 3  
XILINX-XDB 0.1 STUB 0.1 ASCII  
XILINX-XDM V1.6e  
$754=~2?3&bdah!jamcwe*Tbkaoyoek SRVJG+EOIEFNBJK  
cicoh`h`m&zycn!fmqn,twid'hihy#xgd548+mijm&oj`lzn/Sg`l`td`l%X_YGL.BJBHICIOLE%  
hdlbceoeff+uthk&cf|a!
```

Par une recherche Google sur les mots clés « XILINX ngr », nous comprenons rapidement que le fichier *s.ngr* est un fichier représentant un circuit logique (cf. [premier résultat Google](#)). Le format NGR est un format propriétaire de la société Xilinx, fabricant de FPGA, qui peut être ouvert avec le logiciel *ISE Project Navigator* de la suite *ISE Design Suite*¹ de la même société. Nous téléchargeons et installons cette suite logicielle.

Nous ouvrons le fichier *s.ngr* en commençant par une représentation haut niveau du schéma.

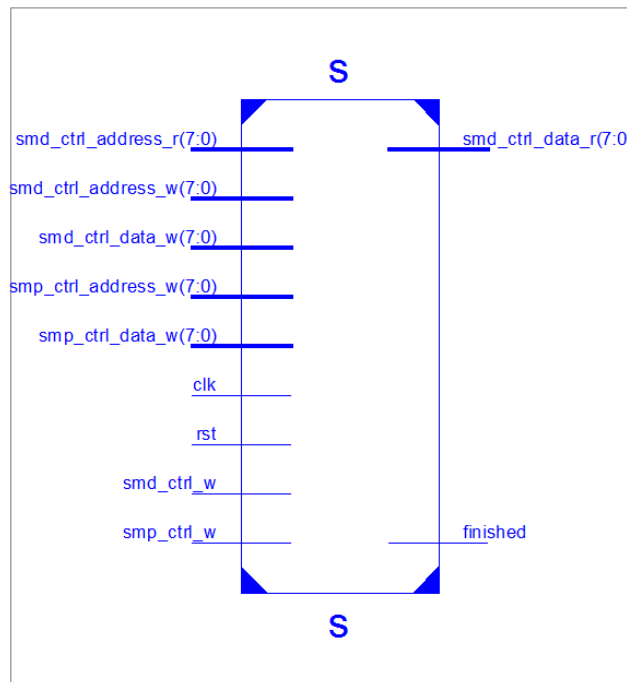


Figure 1: bloc S du circuit logique

A l'aide de la terminologie des entrées/sorties du bloc, nous comprenons que *smd* (bloc de données) et *smp* (octets du fichier *smp.py*) sont utilisés pour retourner un résultat nommé *data*. Une sortie nommée *finished* est également présente.

En double cliquant sur le bloc S, nous obtenons les détails de sa composition.

NB : le schéma étant complexe, il est illisible dans sa vue à 100%. Les blocs analysés ultérieurement seront présentés de nouveau.

¹ Téléchargeable gratuitement à l'adresse

<http://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/design-tools.html>

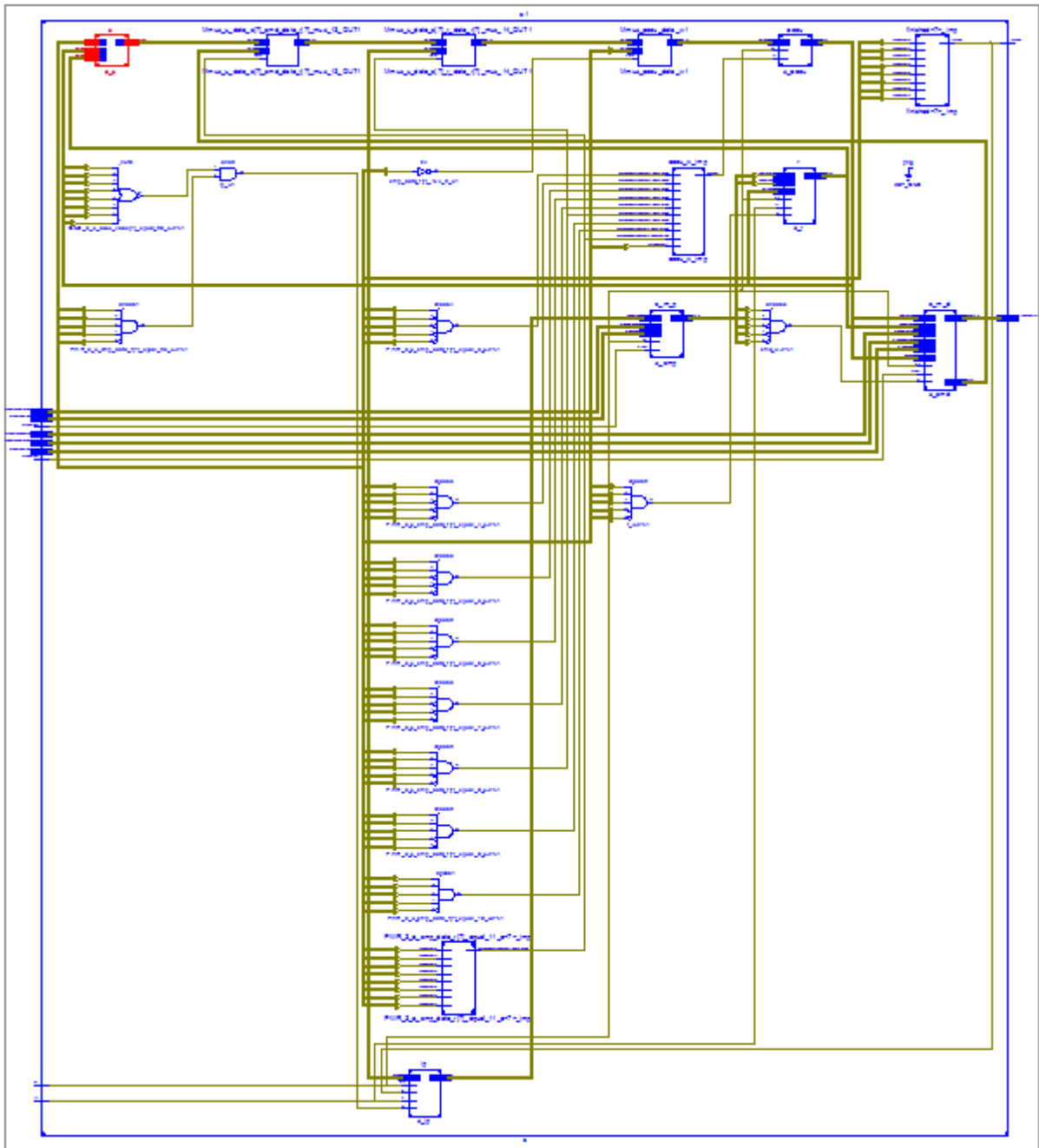


Figure 2: vue détaillée du circuit logique

En zoomant sur le schéma, nous identifions un certain nombre de blocs à analyser :

- x_u
- x_smp qui est notamment alimenté par les valeurs smp fournies en entrée
- x_smd qui est notamment alimenté par les valeurs smd fournies en entrée
- x_accu dont le nom explicite semble laisser penser qu'il s'agit d'un accumulateur
- finished dont la sortie est utilisée comme sortie du bloc S
- x_r
- accu_w_imp
- x_ip
- PWR_3_o_smp_data_r[7]_equal_11_o<7>_imp, renommé ci-dessous PWR3

En plus de ces blocs, nous dénombrons :

- 3 multiplexeurs
- 12 portes logiques ET (AND)
- 1 porte logique NON-OU (NOR)
- 1 porte logique NON

A l'aide du logiciel ISE qui met en évidence les connexions entre les différents éléments, nous identifions les liens suivants :

- la sortie du bloc x_smp alimente toutes les portes logiques AND, le bloc x_r, le bloc x_u et le bloc finished.
- la sortie du bloc x_r alimente le bloc x_smd, le bloc x_u, le bloc x_ip et un des multiplexeurs.
- la sortie de x_accu alimente les blocs x_smd, x_r et x_u.
- le bloc x_u est alimenté par les sorties des blocs x_smp, x_accu et x_r
- le bloc x_smd est alimenté par les sorties des blocs x_accu et x_r ainsi que par les données *smd* fournies en entrée.

Pour aller plus loin dans la compréhension de ce circuit, nous commençons à analyser les différents blocs.

3.3. Analyse du bloc x_u

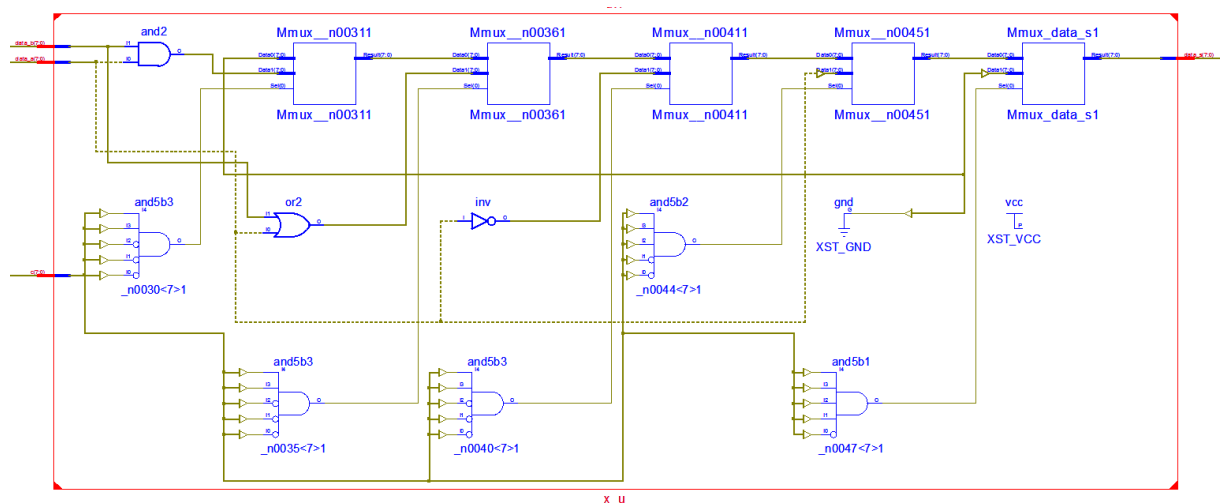


Figure 3: bloc x_u

Le bloc x_u prend 3 octets en entrée (a, b, c) et retourne un octet s. La valeur s retournée dépend de la valeur de certains bits de c (les 5 bits de poids les plus forts). En analysant les différentes portes logiques, nous élaborons la table suivante :

c[7]	c[6]	c[5]	c[4]	c[3]	s
1	0	0	0	1	a&b
1	0	0	1	0	a b
1	0	1	0	0	~a
1	1	1	0	0	a 0x80
1	1	0	1	1	a << 1

A noter que pour les autres valeurs de c, la sortie s vaut 0.

En revenant au schéma global, nous comprenons que les entrées de x_u sont :

- a => sortie de x_accu (soit accu_value)
- b => sortie de x_r (soit r_data_r)
- c => sortie de x_smp (soit smp_data_r)

Nous passons alors à l'analyse du bloc x_accu pour comprendre son fonctionnement.

3.4. Analyse du bloc x_accu

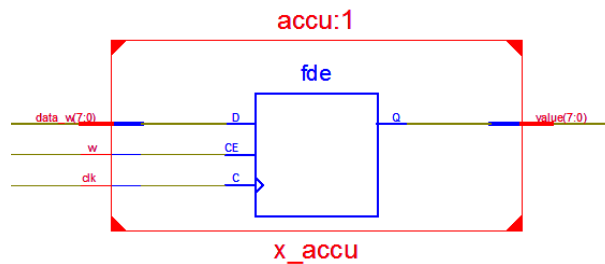


Figure 4: bloc x_accu

Comme nous le supposions initialement, le bloc x_accu est une simple mémoire dans laquelle un octet est écrit à chaque cycle d'horloge. Nous nous intéressons donc à l'alimentation de ce bloc.

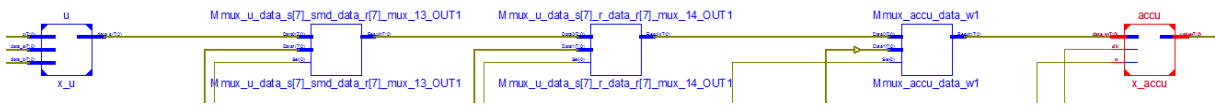


Figure 5: alimentation du bloc x_accu

Le bloc x_accu est précédé par les 3 multiplexeurs qui vont permettre de choisir la valeur inscrite dans l'accumulateur selon la valeur de leur sélecteur respectif. Pour plus de compréhension, nous renommons les multiplexeurs M1, M2, M3 de gauche à droite selon leur position dans le schéma. Par l'analyse des entrées des multiplexeurs, nous obtenons :

	M1	M2	M3
Entrée 1	u_data (sortie de x_u)	sortie M1	sortie M2
Entrée 2	smd_data (sortie de x_smd)	r_data (sortie de x_r)	0x7f & smp_data
Sélecteur	PWR3_data (sortie de PWR3)	smp_data[3:7]& 0x15	~smp_data[7]

où s[a:b] représente les bits de la position a à la position b comprise de l'octet s.

L'analyse de PWR3 est simple et nous apprend que la valeur retournée est un bit valant 1 si et seulement si smp_data vaut 0xd0.

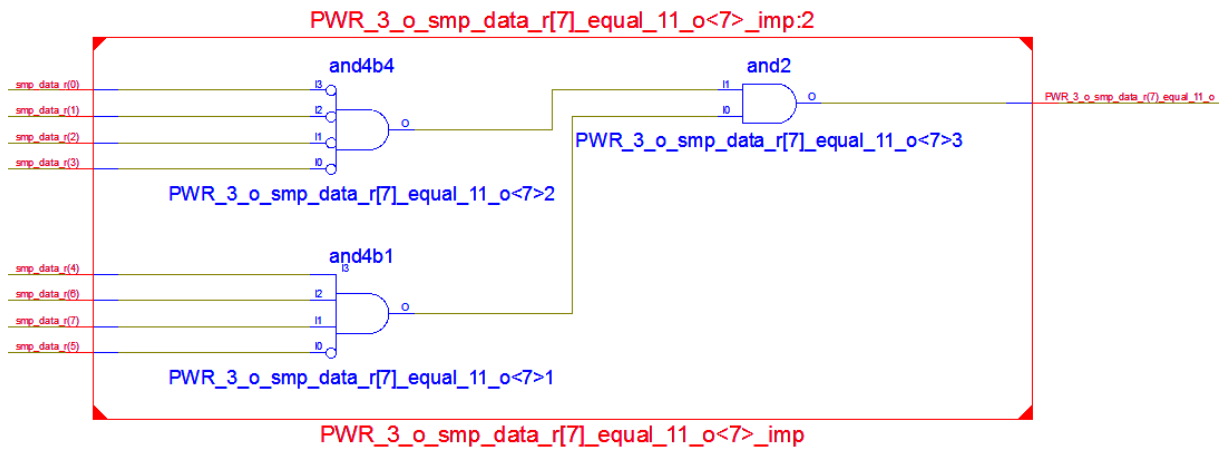


Figure 6: bloc PWR3

Le tableau précédent devient donc :

	M1	M2	M3
Entrée 1	u_data (sortie de x_u)	sortie M1	sortie M2
Entrée 2	smd_data (sortie de x_smd)	r_data (sortie de x_r)	0x7f & smp_data[0:6]
Sélecteur	smp_data & 0xd0	smp_data[3:7]& 0x15	~smp_data[7]

Le sélecteur des 3 multiplexeurs utilise donc la valeur de smp_data (sortie de x_smp) pour choisir quelle sera la valeur mémorisée dans l'accumulateur. Comme pour l'analyse du bloc x_u, nous réalisons donc la table suivante qui, à une valeur de smp_data, enregistre une valeur de sortie s dans l'accumulateur.

smp_data [7]	smp_data [6]	smp_data [5]	smp_data [4]	smp_data [3]	s
1	1	0	1	0	smd_data
1	0	1	0	1	r_data
0	x	x	x	x	0x7f&smp_data[0:6]

avec x pouvant prendre la valeur 0 ou 1 de manière indépendante

L'enregistrement de la valeur s ne peut cependant avoir lieu si et seulement si l'entrée w du bloc x_accu est à 1. Cette dernière correspond à la sortie du bloc accu_w_imp ce qui nous pousse à analyser ce bloc.

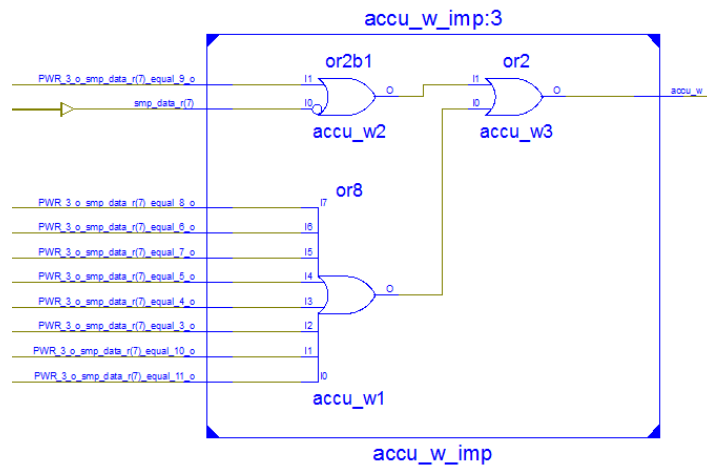


Figure 7: bloc accu_w_imp

Le bit de sortie vaut 1 dès que l'une des entrées vaut 1 ou que smp_data[7] vaut 0. En regardant les entrées, nous constatons rapidement qu'il s'agit des sorties de la plupart des portes logiques AND et que leur valeur retournée vaut 1 dès que smp_data vaut une des valeurs présentées dans les précédentes tables logiques. L'écriture dans l'accumulateur a donc bien lieu avec les valeurs de la sortie s présentée précédemment.

3.5. Analyse du bloc x_r

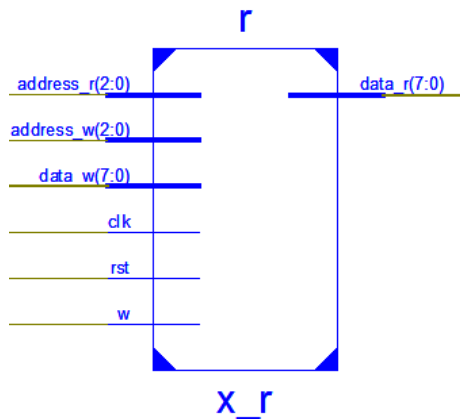


Figure 8: bloc x_r

Le bloc prend en entrée les 3 bits de poids faible de smp_data, la sortie accu_value de l'accumulateur ainsi qu'un bit w permettant d'écrire dans les mémoires du bloc. La vue détaillée de x_r nous montre qu'il y a 8 mémoires et que selon la valeur des 3 bits de smp_data fournies en entrée, l'une des mémoires est sélectionnée en tant que valeur de sortie. Nous comprenons donc que les 3 bits de smp_data définissent un numéro de mémoire et que, selon la valeur de w, cette mémoire est lue ou écrite avec la valeur de sortie de l'accumulateur.

La porte logique manipulant le bit w retourne la valeur 1 lorsque smp_data[3:7] = 0b10110.

A la lumière de cette analyse, nous comprenons que le bloc x_r représente 8 registres et que les 3 derniers bits de smp_data sont utilisés pour identifier un numéro de registre lors des opérations de

lecture ou d'écriture. L'opération d'écriture est déclenchée lorsque les 5 bits de poids fort de `smp_data` valent 0b10110.

Ainsi, la table précédente est complétée par :

<code>smp_data</code> [7]	<code>smp_data</code> [6]	<code>smp_data</code> [5]	<code>smp_data</code> [4]	<code>smp_data</code> [3]	sortie
1	0	1	1	0	$R_n = A$

où R_n est le n-ème registre et n est représenté par les 3 bits `smp_data` [0:2]

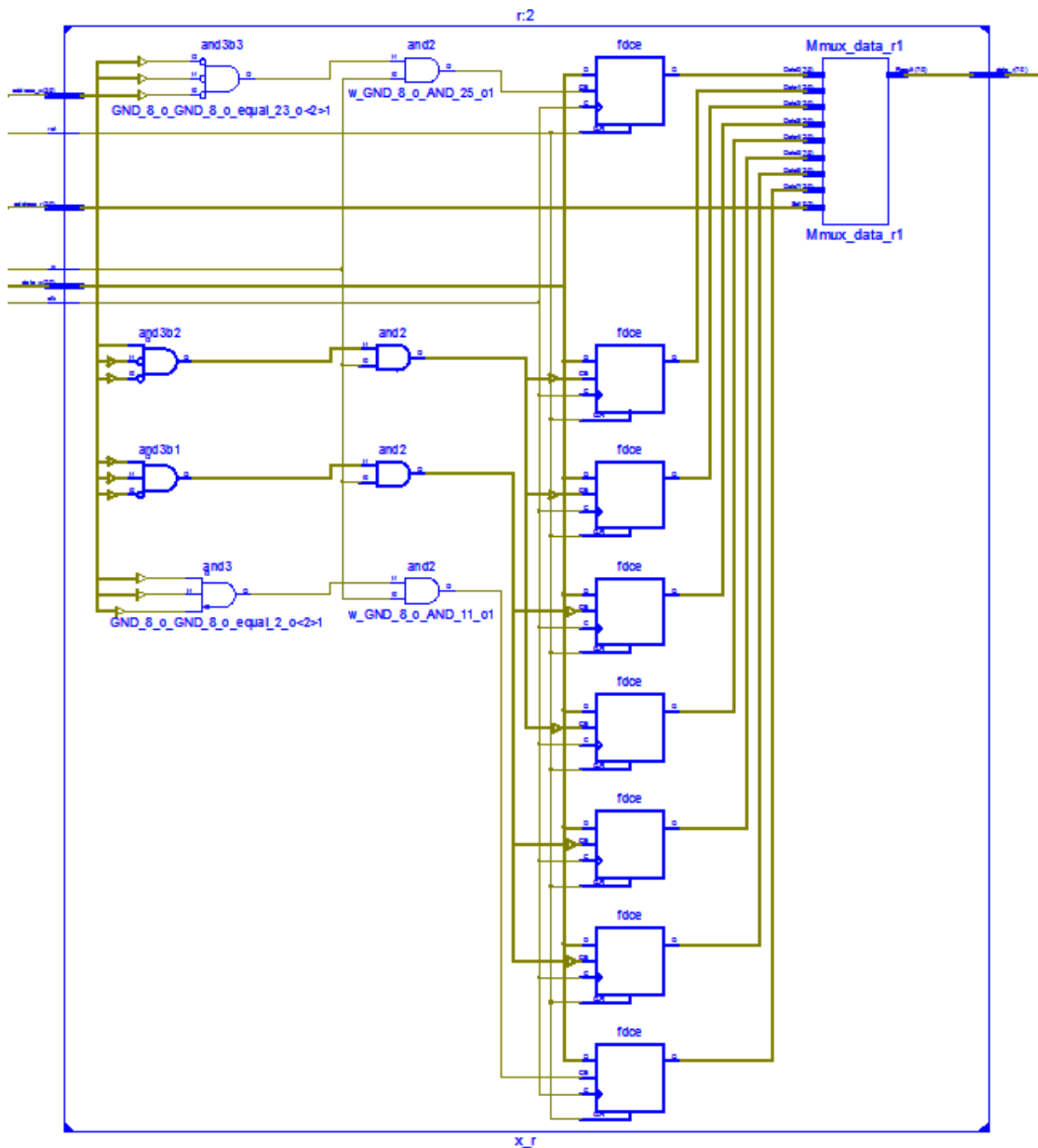


Figure 9: bloc `x_r`

3.6. Analyse du bloc finished

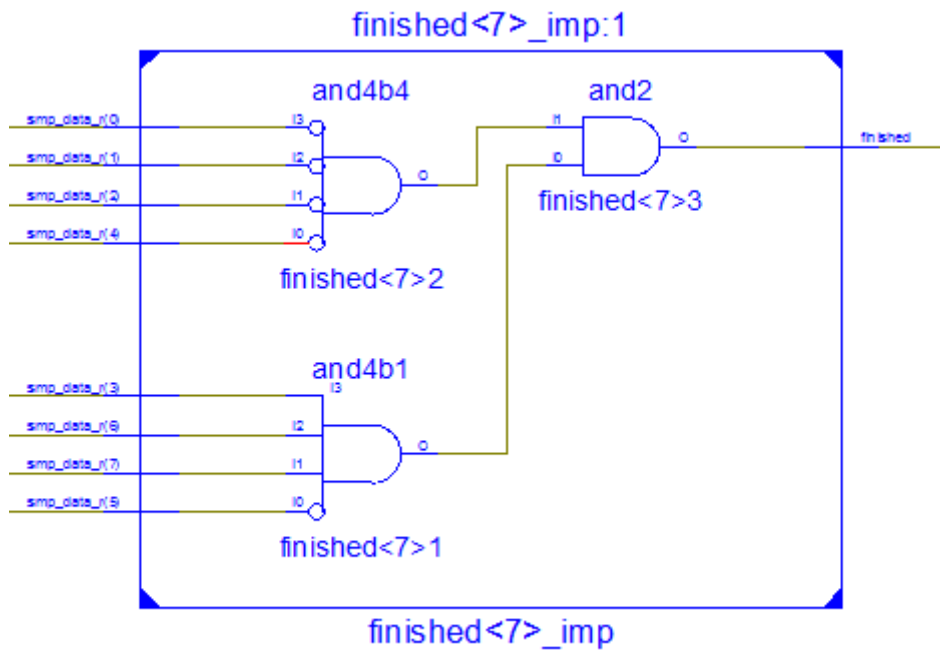


Figure 10: bloc finished

Le bloc finished est très simple à analyser. La valeur 1 est renvoyée si et seulement si simp_data vaut 0b11001000 soit 0xc8.

La sortie du bloc finished est utilisée en sortie du bloc principal s mais aussi en entrée du bloc x_ip (entrée en).

3.7. Analyse du bloc x_ip

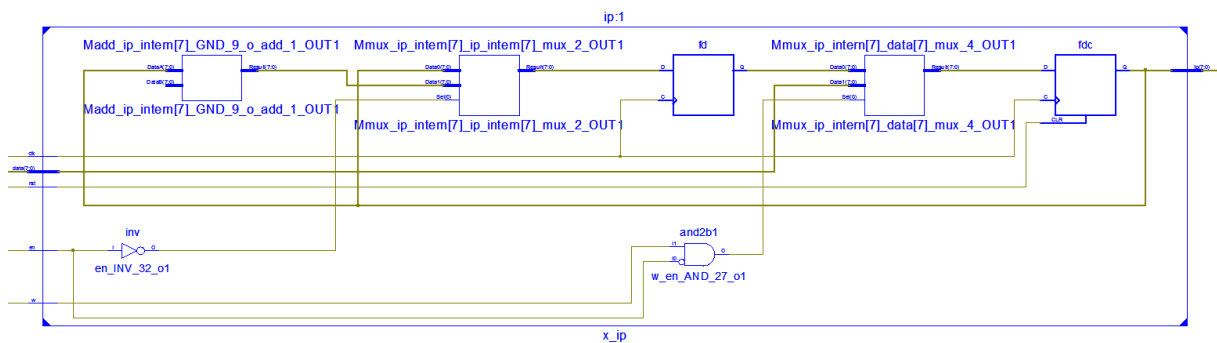


Figure 11: bloc x_ip

Avant tout analyse, compte tenu que le bloc x_r se réfère à des registres, nous supposons que le bloc x_ip va être utilisé en tant que pointeur d'instruction qui sera lu ou écrit.

Lors de l'analyse, nous constatons que le bloc `x_ip` contient une mémoire située après 2 multiplexeurs. La valeur de sortie de ce bloc va donc dépendre des valeurs des entrées `en` et `w`.

w	en	sortie
0	0	ip
0	1	ip
1	1	ip
1	0	data

Comme nous l'avons vu précédemment, l'entrée `en` correspond à la sortie `finished`.

L'entrée `w`, elle, correspond à la sortie du schéma suivant :

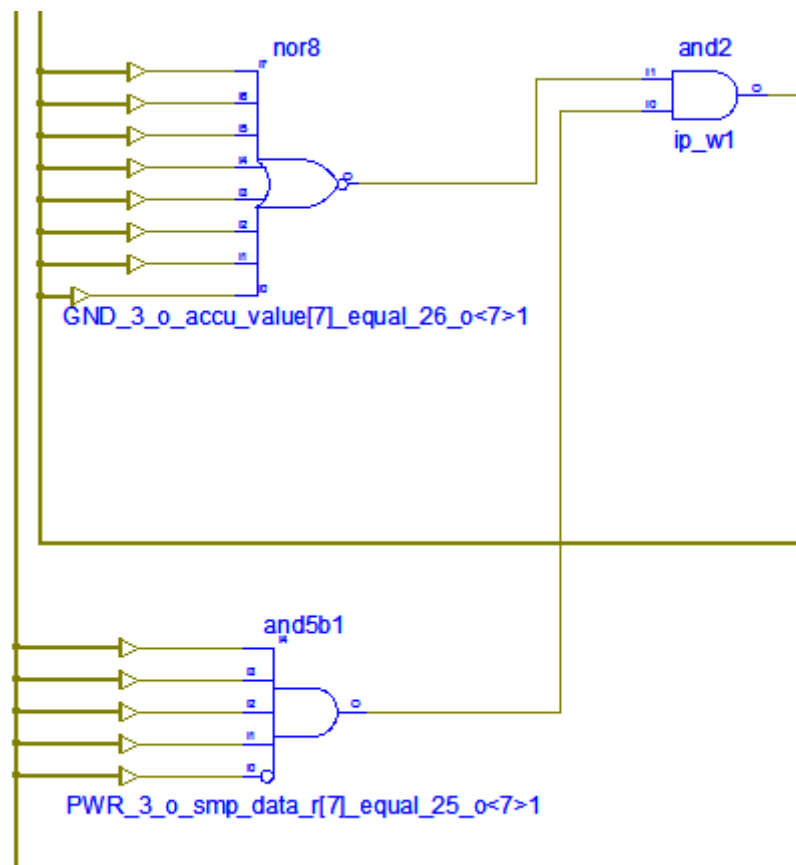


Figure 12: portes avant entrée `w` dans `x_ip`

En regardant les entrées des portes AND et NOR, nous comprenons que la sortie de ce schéma vaudra 1 si les 2 conditions suivantes sont réunies :

- `accu_value = 0`
- `smp_data[3:7] = 0b10111`

Sous ces conditions, la valeur `data` correspondant à la sortie du bloc `x_r` (soit le registre de numéro `smp_data[0:2]`) sera retournée. Cette sortie est alors utilisée en entrée du bloc `x_smp`.

Cette analyse nous permet de comprendre que nous sommes face à un saut conditionnel permettant de modifier le cours d'exécution du schéma logique.

En synthétisant l'ensemble des éléments découverts (registres, saut conditionnel, pointeur d'instruction, accumulateur), nous commençons à comprendre que le schéma logique que nous analysons décrit un programme dont les instructions sont codées au sein de smp. Les données que traitent ce programme sont quant à elle stockées dans smd. Cette intuition sera confirmée par l'analyse du bloc x_smd.

3.8. Analyse du bloc x_smd

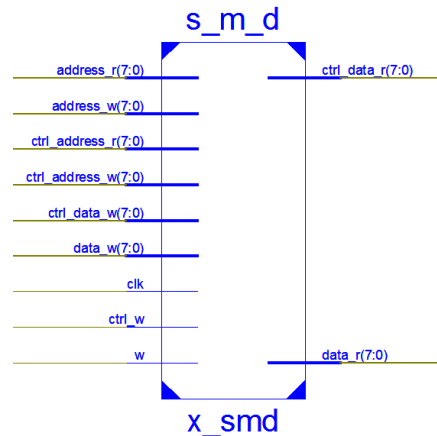


Figure 13: bloc x_smd

Lorsque l'on regarde le bloc x_smd en détail, on constate que sa structure est semblable à celle du bloc x_smp. Ce bloc est constitué de 256 blocs mémoires d'un octet qui vont être appelés en écriture ou en lecture selon la valeur des entrées address_r, address_w, data_w et w.

L'écriture est déclenchée si et seulement si l'entrée w vaut 1, c'est-à-dire lorsque smp_data[3:7] vaut 0b11000. Dans ce cas-là, la valeur de l'accumulateur est écrite à l'adresse R_n où R_n est la valeur du n-ème registre et n vaut smp_data[0:2] (représentation des 3 bits de poids faible de smp_data).

Lorsque l'entrée w vaut 0 (=opération de lecture), la valeur de l'octet à l'adresse accu_value est retournée et est utilisée en entrée du multiplexeur M1 décrit précédemment (cf. analyse du bloc x_accu).

3.9. Synthèse de l'analyse du schéma logique

Après avoir analysé l'ensemble des blocs du schéma logique, nous avons compris que ce dernier décrit :

- un jeu d'instructions codé sur les 5 bits de poids fort de chaque octet de smp ;
- 8 registres dont l'identifiant est codé sur les 3 bits de poids faible de chaque octet de smp ;
- un accumulateur utilisé pour stocker différentes valeurs (registre, données, calcul, etc.)
- une mémoire de 256 octets

A l'aide des précédentes analyses sur les valeurs de `smp_data`, nous sommes alors en mesure d'écrire le jeu d'instructions comme suit :

smp_data						instruction
bit_7	bit_6	bit_5	bit_4	bit_3	bit_0 à bit_2 = n	
0	x	x	x	x	x	A = 0x7f & smp_data
1	0	0	0	1	R _n	A = A & R _n
1	0	0	1	0	R _n	A = A R _n
1	0	1	0	0	x	A = ~A
1	0	1	0	1	R _n	A = R _n
1	0	1	1	0	R _n	R _n = A
1	0	1	1	1	R _n	jz R _n
1	1	0	0	0	R _n	[R _n] = A
1	1	0	0	1	000	finished = 1
1	1	0	1	0	000	A = [A]
1	1	0	1	1	x	A = A << 1
1	1	1	0	0	x	A = 0x80 A
pour toutes les autres valeurs de smp_data						A = 0

où :

- x représente une valeur quelconque non prise en compte
- [K] représente la valeur de la mémoire à l'adresse K

Nous utilisons alors ce jeu d'instructions pour « désassembler » le fichier `smp.py` à l'aide du script python `disassemble_smp.py` présenté en annexe 7.4. Le résultat du code obtenu est alors lu et simplifié pour obtenir le code présenté en annexe 7.5.

3.10. Analyse du fichier `smp.py` et cryptanalyse

Pour rappel, la compréhension du script `decrypt.py` nous avait appris que le bloc de données suivant était transmis au device :

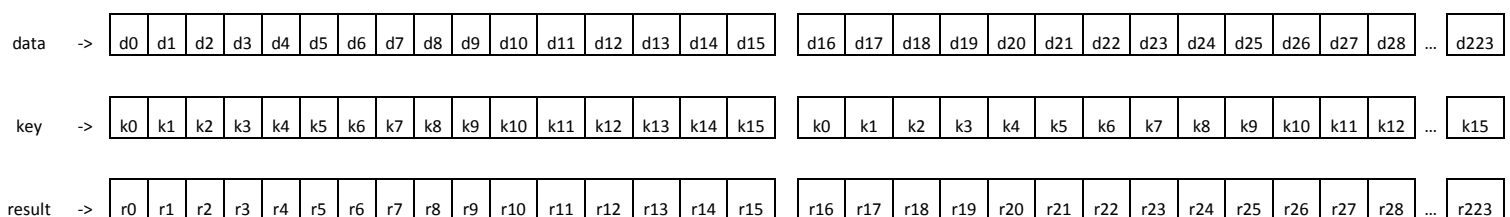
clé de 16 octets fournie en argument || taille T du bloc de données || bloc de données du fichier data

Le bloc de donnée extrait du fichier `data` (224 octets maximum) est en réalité traité par le programme `smp` dont le code est présenté en annexe 7.5.

La lecture détaillée de ce code nous permet de comprendre l'opération réalisée par le programme `smp`. Pour chaque octet de donnée :

- une opération XOR est réalisée avec l'octet de la clé en position `pos_octet_de_donnée%16`
- le résultat de l'opération XOR est inversé de gauche à droite (fonction nommée `reverse` ci-après)

Schématiquement cela peut être représenté ainsi :



avec $r_i = \text{reverse}(d_i \wedge k_{i\%16})$ pour i allant de 0 à la taille T du bloc de données.

Il reste donc à trouver la clé K qui permettra de déchiffrer le fichier de données *data*.

Comme nous l'avons précisé en 3.1, la clé à trouver doit permettre de générer un fichier encodé en base64 dont l'empreinte md5 est *6c0708b3cf6e32cbae4236bdea062979*. Nous en déduisons donc que les octets du fichier résultat sont limités à l'alphabet des caractères autorisés en base64, c'est-à-dire *abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+/\n*, soit seulement 66 caractères. Cela limite donc les valeurs que peut prendre chaque octet de la clé K (seulement 66 valeurs possibles pour chaque octet au lieu de 256).

De plus, comme la clé K est réutilisée plusieurs fois sur des blocs de données différents, nous pouvons, pour chaque octet de la clé, restreindre le nombre de candidats potentiels. Par exemple, k_0 doit satisfaire le système d'équation suivant :

$$r_{16*i} = \text{reverse}(d_{16*i} \wedge k_0) \text{ pour } i \text{ allant de } 0 \text{ à } T/16$$

r_{16*i} appartient à l'alphabet des caractères autorisés en base64 pour i allant de 0 à $T/16$

Nous pouvons alors bruteforcer les valeurs de chaque octet de la clé et trouver la valeur de la clé K en nous appuyant sur la vérification md5 du résultat obtenu. Le script python *decrypt_data.py* de l'annexe 7.6 réalise cette cryptanalyse.

```
$ python decrypt_data.py
Key found ! - 67c13b3d68f71a63a635c4cb78b685a4
Decrypting data file...
md5(s): 6c0708b3cf6e32cbae4236bdea062979
Decryption done in file data.clear
```

Nous obtenons alors le fichier *data.clear*.

```
$ file data.clear
data.clear: ASCII text, with very long lines
```

4. Polski Mówię

En ouvrant le fichier *data.clear* dans un éditeur de texte, nous nous apercevons que le fichier contient du texte ressemblant à du code source. Une rapide recherche Google sur les mots clés *ReusableStreamDecode* ou *SubFileDecode* nous permet de découvrir qu'il s'agit de code source PostScript. Nous apprenons alors par la page [Wikipedia sur Postscript](#) que ce langage est interprété, s'appuie sur une pile et des dictionnaires et utilise la notation polonaise inversée, ce qui ne va pas faciliter la compréhension du code source. Une version partiellement tronquée du code source est présentée en annexe 7.7.

Nous renommons le fichier *data.clear* en *script.ps* d'après la fin du code source (cf. *usage: gs -- script.ps key*).

Nous cherchons alors à exécuter le script.

```
$ gs -- script.ps
GPL Ghostscript 9.07 (2013-02-14)
Copyright (C) 2012 Artifex Software, Inc. All rights reserved.
This software comes with NO WARRANTY: see the file PUBLIC for details.
no key provided
usage: gs -- script.ps key
```

Une nouvelle fois, une clé va devoir être trouvée pour pouvoir valider ce niveau. Nous commençons alors l'analyse du fichier *script.ps*.

4.1. Compréhension du programme principal

A la lecture du code source de *script.ps*, nous identifions 4 blocs de données encodées en hexadécimal : I1, I2, I3, I4. I1 est le plus gros bloc et I3 le plus petit.

Ces 4 blocs de données sont suivis du programme principal où 2 fonctions sont définies : *error* et *main*. La fonction *main* est alors appelée.

Pour faciliter la lecture et réussir à comprendre le script :

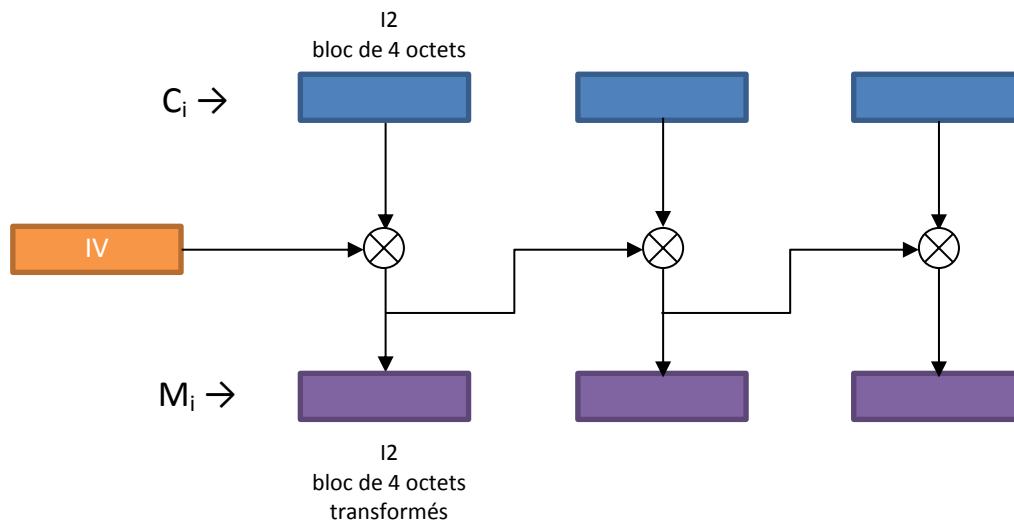
- nous réindentons le code source en cherchant à revenir à la ligne après chaque fonction postscript de base (index, pop, string, dup, getinterval, etc.) ;
- nous exécutons le script pas à pas en insérant « *dup ==* » aux endroits où l'on cherche à connaître l'état de la pile. Cette fonction permet de connaître le dernier élément de la pile en le dupliquant, le dépilant et en l'affichant ;
- nous multiplions l'usage de l'opérande « *==* » lorsque nous cherchons à connaître plusieurs éléments de la pile. « *== == ==* » nous donnera par exemple les 3 derniers éléments de la pile.

Etant donné la taille du programme principal, nous considérons qu'une lecture de code et un debug manuel sont envisageables et nous ne cherchons donc pas à coder un debugger complet.

Notre analyse nous permet alors de comprendre les éléments suivants (cf. annexe 7.8 pour les commentaires et l'analyse complète) :

- la longueur de la clé doit être de 16 octets (noté k0k1k2...k15 par la suite) et saisie en hexadécimal

- le bloc I2 est découpé en blocs de 4 octets qui sont xorés de manière chaînée selon le schéma suivant :



- le vecteur d'initialisation utilisé est $k_2k_3k_2k_3$ où k_2 et k_3 sont respectivement les 3^{ème} et 4^{ème} octets de la clé fournie en paramètre ;
- le bloc de données obtenu correspond à du code postscript car ce dernier est ensuite exécuté ;
- la même opération est réalisée sur le bloc de données I4 avec comme vecteur d'initialisation $k_0k_1k_0k_1$ où k_0 et k_1 sont respectivement les 1^{er} et 2^{ème} octets de la clé fournie en paramètre.

Il est désormais nécessaire de retrouver les blocs I2 et I4 déchiffrés en cryptanalysant les 4 premiers octets de la clé.

4.2. Cryptanalyse de I2 et I4

Afin de retrouver les 4 premiers octets de la clé, nous pourrions procéder de la même manière que pour le déchiffrement du fichier *data* de l'étape précédente. Par bruteforce, sachant que le résultat obtenu doit être du code lisible, nous pourrions limiter le nombre de clés candidates qui nous permettent d'obtenir un caractère dans l'alphabet autorisé par le langage PostScript. Mais il existe une méthode plus efficace.

Le système de chiffrement par bloc présenté ci-dessus se modélise ainsi :

$$M_i = C_i \wedge M_{i-1} \text{ pour } i \text{ allant de } 0 \text{ à } (\text{taille de } I)/4-1$$

$$M_{-1} = IV$$

Cette équation est équivalente à :

$$M_n = IV \wedge C_0 \wedge C_1 \wedge C_2 \wedge \dots \wedge C_n \text{ pour } n \text{ compris entre } 0 \text{ et } (\text{taille de } I)/4-1$$

Si $S_n = C_0 \wedge C_1 \wedge C_2 \wedge \dots \wedge C_n$ pour n compris entre 0 et $(\text{taille de } I)/4-1$ alors l'équation précédente devient :

$$M_n = IV \wedge S_n \text{ pour } n \text{ compris entre } 0 \text{ et } (\text{taille de } I)/4-1$$

Soit :

$$M_n \wedge IV = S_n \text{ pour } n \text{ compris entre } 0 \text{ et } (\text{taille de } I)/4-1$$

Cela revient donc à cryptanalyser un chiffrement xor classique où la clé K est répétée.

On va résoudre le système d'équations en calculant l'ensemble des S_n et en procédant à une analyse statistique des caractères obtenus selon leur position dans S (position paire ou position impaire). En effet, l'IV étant composé de 2 octets répétés, les positions paires de S seront toujours xorées avec le premier octet de l'IV et les positions impaires de S avec le second octet de l'IV.

De plus, la longueur de l'IV étant petite, la répartition statistique des caractères du langage PostScript présent dans le code M est conservée dans le texte chiffré S. Nous supposons alors que l'octet le plus fréquent pour les octets en position paire de S et l'octet le plus fréquent pour les octets en position impaire de S représente le caractère espace (0x20 en ASCII), caractère le plus fréquent dans un code source PostScript non indenté.

Le script python *decodeI2I4.py* (cf. annexe 7.9) réalise ces opérations et permet de retrouver l'IV utilisé pour les blocs I2 et I4. Nous obtenons donc les 4 premiers octets de la clé recherchée.

```
$ python decodeI2I4.py
```

```
k0k1 - I4 en clair
key: bac9
0 0 0 0 2 2 16 4 sub { 6 index exch 4 getinterval 10240 { 0 0 1 3 { 3 -1
roll dup 4 1 roll exch get exch 8 bitshift add } for exch pop dup -2
bitshift exch dup -3 bitshift 1 index -7 bitshift xor exch dup 4 1 roll xor
xor 1 and 31 bitshift exch -1 bitshift or 4 string exch 3 -1 0 {3 copy exch
255 and put pop -8 bitshift} for pop dup <55555555> le {1} { dup <aaaaaaaa>
le {-1} {0} ifelse } ifelse 4 -1 roll add 5 index length add 5 index length
mod 3 1 roll 0 0 1 3 { 3 -1 roll dup 4 1 roll exch get exch 8 bitshift add
} for exch pop dup -2 bitshift exch dup -3 bitshift 1 index -7 bitshift xor
exch dup 4 1 roll xor xor 1 and 31 bitshift exch -1 bitshift or 4 string
exch 3 -1 0 {3 copy exch 255 and put pop -8 bitshift} for pop dup
<55555555> le {1} { dup <aaaaaaaa> le {-1} {0} ifelse } ifelse 3 -1 roll
add 5 index 0 get length 4 idiv add 5 index 0 get length 4 idiv mod exch 0
0 1 3 { 3 -1 roll dup 4 1 roll exch get exch 8 bitshift add } for exch pop
dup -2 bitshift exch dup -3 bitshift 1 index -7 bitshift xor exch dup 4 1
roll xor xor 1 and 31 bitshift exch -1 bitshift or 4 string exch 3 -1 0 {3
copy exch 255 and put pop -8 bitshift} for pop 6 -2 roll 2 copy 8 2 roll
get 4 index 4 mul 7 index 5 index get 4 index 4 mul 4 5 copy dup 4 1 roll
getinterval 4 1 roll getinterval exch dup length string 0 3 -1 roll { 3
copy put pop 1 add } forall pop exch 3 -1 roll pop 4 -2 roll 3 -1 roll
putinterval putinterval 5 index 5 index get 4 index 4 mul 4 getinterval 1
index 0 0 1 1 {pop 2 index length} for exch 1 sub { 3 copy exch length
getinterval 2 index mark 3 1 roll 0 1 3 -1 roll dup length 1 sub exch 4 1
roll { dup 3 2 roll dup 5 1 roll exch get 3 1 roll exch dup 5 1 roll exch
get xor 3 1 roll } for pop pop ] dup length string 0 3 -1 roll { 3 -1 roll
dup 4 1 roll exch 2 index exch put 1 add } forall pop 4 -1 roll dup 5 1
roll 3 1 roll dup 4 1 roll putinterval exch pop } for pop pop 5 1 roll 2
copy 7 -3 roll pop pop } repeat pop 4 index 0 1 index { length add } forall
string 0 3 2 roll { 3 copy putinterval length add } forall pop calc I3 16
string readstring pop ne {0 1 1073741823 {pop} for (Key is invalid. Exiting
...\n) error flush quit } if } for pop pop pop pop (output.bin) (w) file
exch 1 index resetfile {1 index exch writestring} forall closefile
%%%

```

k2k3 - I2 en clair

key: f7a8

```
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#0FFFFFFFF 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
%
```

Les 4 premiers octets de la clé sont donc *bac9f7a8*. Nous obtenons également le code source des blocs I2 et I4.

4.3. Analyse de I2

Au début du code source du bloc I2, un tableau avec de nombreuses valeurs représentées en octal est initialisé. Une recherche Google sur ces valeurs nous apprend qu'il s'agit de valeurs utilisées par l'algorithme md5. Nous supposons alors que les auteurs du challenge n'ont pas été trop sadiques et que cette fonction I2 représente bien une implémentation en PostScript de l'algorithme md5. Notre hypothèse sera confirmée ultérieurement avec l'analyse de I4.

4.4. Analyse de I4

De même que pour la fonction principale de *script.ps*, nous indentons le code source de I4 et l'exécutons pas à pas en affichant l'état de la pile lorsque cela est nécessaire. Le code source indenté et commenté est présenté en annexe 7.10.

D'après l'analyse du code source, nous comprenons que la fonction I4 réalise les opérations suivantes sur le bloc I1 qui est vu comme un tableau de 77 chaînes de 128 caractères :

- la clé fournie en paramètre de *script.ps* est lue par bloc de 4 octets à partir du 2^{ème} octet et avec un décalage successif de 2 octets. 6 sous-clés sont ainsi utilisées : k2k3k4k5 puis k4k5k6k7 puis k6k7k8k9 puis k8k9k10k11 puis k10k11k12k13 et enfin k12k13k14k15 ;
- pour les 6 sous-clés, les actions suivantes sont réalisées 10240 fois :
 - o définition d'une position initiale,
 - o dérivation de la sous-clé une première fois,
 - o comparaison de la clé dérivée obtenue aux valeurs 0x55555555 et 0xaaaaaaaa pour obtenir un premier offset de -1, 0 ou 1,
 - o nouvelle dérivation de la clé dérivée et nouveau calcul d'offset,
 - o les 2 offsets sont appliqués sur la position initiale pour définir une position cible dans le tableau I1,
 - o nouvelle dérivation de la dernière clé dérivée,
 - o permutation des 4 octets en position initiale avec les 4 octets en position cible,
 - o opération de xor entre la dernière clé dérivée et la valeur en position initiale (ancienne valeur en position cible),
 - o définition de la position cible comme la nouvelle position initiale et usage de la clé dérivée comme nouvelle sous-clé ;
- à la fin des 10240 répétitions, une empreinte md5 de I1 est calculée ;
- l'empreinte est comparée à l'empreinte correspondante lue dans I3, I3 stockant 6 empreintes md5 consécutives ;
- si les 2 empreintes sont identiques, l'algorithme se poursuit avec la sous-clé suivante ; sinon l'algorithme se termine après avoir effectué une boucle d'1073741823 itérations inutiles.
- lorsque toutes les sous-clés ont été parcourues, la table I1 finale est écrite dans le fichier *output.bin*.

4.5. Reconstruction de la clé finale

La reconstruction de la clé finale nécessite de retrouver les 12 octets manquants. Ces derniers vont pouvoir être retrouvés 2 par 2 à l'aide des 6 sous-clés précédemment évoquées. En effet, la première sous-clé est k2k3k4k5. Or la cryptanalyse de I2 (cf. ci-dessus), nous a permis de retrouver k2 et k3, respectivement 0xf7 et 0xa8. Il ne reste donc plus qu'à découvrir les 2 octets k4 et k5 ce qui représente un total de seulement 65536 valeurs possibles. Nous pouvons donc bruteforcer cet espace de valeurs afin d'obtenir la sous-clé qui donnera l'empreinte md5 attendue.

Par raisonnement récursif, une fois k4 et k5 connu, nous pourrions attaquer la sous-clé suivante et découvrir les valeurs de k6 et k7. Ainsi de suite.

Par cette attaque en bruteforce, nous aurons à tester, au maximum, $6 * 65536 = 393216$ valeurs possibles.

Nous décidons de réimplémenter la fonction I4 (cf. algorithme décrit précédemment) et lançons notre attaque par bruteforce. Le code de cette attaque est présenté en annexe 7.11.

Avec une vitesse de bruteforce d'environ 20 clés par seconde, il faut un peu moins de 3h pour retrouver la clé finale complète : `bac9f7a8721fad3c9fcf271eed9abbc8`

NB : pour de meilleures performances, les personnes impatientes recoderont le programme de bruteforce en C.

Nous obtenons alors le fichier `output.bin` qui n'est autre qu'une vCard.

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

Nous nous empressons alors de l'ouvrir en espérant y trouver l'email de soumission du challenge.

5. Récupération de l'email de soumission

Nous ouvrons le fichier `output.bin` avec un éditeur de texte et trouvons rapidement un contact qui semble intéressant :

```
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
```

Les auteurs du challenge auront voulu faire durer le plaisir jusqu'au bout. L'adresse email est encodée avec des noms d'appels systèmes Linux.

Une recherche Google de la chaîne « `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`» nous retourne un premier lien très intéressant vers le site http://stuff.mit.edu/afs/sipb/contrib/linux/arch/x86/syscalls/syscall_64.tbl

Nous retrouvons ainsi le nombre associé à chaque appel système. Nous recopions donc cette table et écrivons un script python, `decodevcard.py` (cf. annexe 7.12), qui décode la suite d'appels système de la vCard et affiche les caractères ASCII associés.

```
$ python decodevcard.py  
59575e0e71f1e3e9946bc307fc7a608d0b568458@challenge.sstic.org
```

Nous obtenons finalement l'adresse email de soumission du challenge.

6. Conclusion

Le challenge SSTIC a une nouvelle fois tenu toutes ces promesses. La difficulté pour résoudre les différentes étapes, excepté la dernière, était croissante et permettait donc de faire la première étape du challenge sans se décourager. Le niveau global du challenge était assez difficile et nécessitait de solides compétences dans divers domaines (programmation, rétro-ingénierie, analyse de code, etc.) ainsi qu'une solide motivation pour ne pas se laisser décourager par le schéma logique ou la notation polonaise inversée de PostScript.

Ce challenge a été l'occasion pour moi d'en apprendre davantage sur les technologies FPGA, la représentation logique d'un processeur simpliste ainsi que sur le langage PostScript.

Je tiens à remercier :

- les auteurs du challenge pour la qualité de l'épreuve proposée
- Pierre Petit pour les échanges très intéressants que nous avons eu ensemble lors de ce challenge

7. Annexes et codes sources

7.1. search_key.py

```
import base64, hashlib
from scapy.all import *
from Crypto.Cipher import AES

def create_key(ti, to, ttl):
    #time | tos | ttl
    key1 = ''.join([hex((ti[i] << 3) + (to[i] << 2) + ttl[i])[2] for i in
range(len(ti))])
    key1 = '0' * (64 - len(key1)) + key1

    #time | ttl | tos
    key2 = ''.join([hex((ti[i] << 3) + (ttl[i] << 1) + to[i])[2] for i in
range(len(ti))])
    key2 = '0' * (64 - len(key2)) + key2
    return [key1, key2]

def decrypt(ciph, key, mode):
    chainmode = {"CBC" : AES.MODE_CBC,
                 "ECB" : AES.MODE_ECB,
                 "CFB" : AES.MODE_CFB,
                 "CTR" : AES.MODE_CTR,
                 "OFB" : AES.MODE_OFB}

    IV = "76C128D46A6C4B15B43016904BE176AC"
    obj = AES.new(key.decode('hex'), chainmode[mode], IV.decode('hex'))
    clear = obj.decrypt(ciph)
    padding = ord(clear[-1])
    clear = clear[:-padding]
    if check_md5(clear):
        print "Decryption done with key %s in mode %s!!" % (key, mode)
        outfile = open("./sstic.tar.gz", 'w')
        outfile.write(clear)
        outfile.close()

def check_md5(s):
    return hashlib.md5(s).hexdigest() == "61c9392f617290642f9a12499de6b688"

def extract_bits(p):
    t = [int(round(p[i+1].time-p[i].time,0))-1 for i in range(5,69)]
    u = [(p[i][IP].tos-2)/2 for i in range(5,69)]
    return (t, u)

def extract_ttl(p, m):
    tab = dict(zip([10, 20, 30, 40], m))
    return [tab[p[i][IP].ttl] for i in range(5,69)]

p = rdpcap("./dump.bin")
time_bits, tos_bits = extract_bits(p)

keys = []
for b1 in [time_bits, [b^1 for b in time_bits]]:
    for b2 in [tos_bits, [b^1 for b in tos_bits]]:
        for b3 in [extract_ttl(p, m) for m in itertools.permutations([0, 1,
2, 3], 4)]:
            keys.extend(create_key(b1, b2, b3))

ciph = base64.b64decode(open("./sstic.tar.gz-chiffre", 'r').read())
```

```
for k in keys:
    if len(k) == 64:
        decrypt(ciph, k, "CBC")
        decrypt(ciph, k, "CFB")
        decrypt(ciph, k, "OFB")
```

7.2. extrait de smp.py

```
smp = [0x00,  
0xb0,  
0x10,  
0xd0,  
0xb7,  
0xa8,  
0xa0,  
0xb6,  
0x0e,  
0xb5,  
0x71,  
0xb4,  
0x00,  
0xbc,  
0x01,  
0xb6,  
0x16,  
0xb5,  
0x71,  
[...]  
[...]  
[...]  
0xb1,  
0x76,  
0xb4,  
0x00,  
0xbc,  
0xab,  
0xb7,  
0x00,  
0xbd,]
```

7.3. decrypt.py

```
#!/usr/bin/python

import sys
import base64
import md5

import dev
import smp

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

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

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

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

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

7.4. disassemble_smp.py

```
import smp

for o in smp.smp:
    if (o >> 7) == 0:
        print "A = 0x%x" % (o & 0x7f)
    elif (o >> 3) == 0x11:
        print "A = A & R%s" % (o & 7)
    elif (o >> 3) == 0x12:
        print "A = A | R%s" % (o & 7)
    elif (o >> 3) == 0x14:
        print "A = ~A"
    elif (o >> 3) == 0x15:
        print "A = R%s" % (o & 7)
    elif (o >> 3) == 0x16:
        print "R%s = A" % (o & 7)
    elif (o >> 3) == 0x17:
        print "jz R%s" % (o & 7)
    elif (o >> 3) == 0x18:
        print "[R%s] = A" % (o & 7)
    elif o == 0xc8:
        print "finished (return)"
    elif o == 0xd0:
        print "A = [A]"
    elif (o >> 3) == 0x1b:
        print "A = A << 1"
    elif (o >> 3) == 0x1c:
        print "A = 0x80 | A"
    else:
        print "A = 0"
```

7.5. dis_smp.txt

```
label_00:
    R0 = 0x00
label_02:
    R7 = [0x10]
    R6 = ~R0
    R5 = 0x0e
    jmp label_71

label_0d:
    R6 = 0x01
    R5 = 0x16
    jmp label_71

label_16:
    jz R7, label_52
    R7 = 0x11
    R6 = R0
    R5 = 0x24
    jmp label_71

label_24:
    R1 = R7
    R6 = [R1]
    R6 = [R0 & 0x0f] | R6
    R7 = [R1]
    R7 = [R0 & 0x0f] & R7
    R7 = ~R7 & R6
```



```

R6 = 0x40
jmp label_53

label_40:
[R1] = R7
R6 = 0x01
R7 = R0
R5 = 0x4c
jmp label_71

label_4c:
R0 = R7
jmp label_02

label_52:
finish

label_53:
R5 = 0x01
R4 = 0x00
label_57:
jz R5, label_6d
R4 = R4 << 1
jz R5 & R7, label_66
R4 = R4 | 0x01
label_66:
R5 = R5 << 1
jmp label_57
label_6d:
R7 = R4
jmp (R6)

label_71:
R1 = 0x00
R3 = 0x00
R2 = 0x01
label_76:
jz R2, label_e3
jz R2 & R7, label_ac
jz R2 & R6, label_96
jz R1, label_8f
R3 = R3 | R2
label_8f:
R1 = R2
jmp label_d9

label_96:
jmp label_a2

label_9b:
R1 = R2
jmp label_d9

label_a2:
R3 = R3 | R2
R1 = 0x00
jmp label_d9

label_ac:
jz R2 & R6, label_c8
jz R1, label_be

```

```
R1 = R2
jmp label_d9
label_be:
R3 = R3 | R2
R1 = 0x00
jmp label_d9

label_c8:
jz R1, label_d7
R3 = R3 | R2
R1 = 0x00
jmp label_d9

label_d7:
R1 = 0x00
label_d9:
R2 = R2 << 1
R1 = R1 << 1
jmp label_76

label_e3:
R7 = R3
jmp (R5)
```

7.6. decrypt_data.py

```
import sys, base64, hashlib

def revbits(x):
    return int(bin(x)[2:].zfill(8)[::-1], 2)

def compute_keypart(data, pos):
    dict_b64 =
    "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789+/\n"
    keys = range(256)

    j = pos
    while len(keys) > 1 and j < len(data):
        r_data = sum(1<<(7-i) for i in range(8) if ord(data[j])>>i&1)
        for k in keys:
            if chr(k ^ r_data) not in dict_b64:
                keys.remove(k)
        j += 16
    if len(keys) != 1:
        print "Warning : several candidates for position %d - %s" % (pos,
keys)
    return "%0.2x" % keys[0]

def decrypt(data, key):
    output = open("data.clear", "wb")
    k = key.decode('hex')
    s = ''
    for j in range(len(data)):
        atad = sum(1<<(7-i) for i in range(8) if ord(data[j])>>i&1)
        s += chr(ord(k[j%16]) ^ atad)
    print "md5(s): ", hashlib.md5(s).hexdigest()
    output.write(base64.b64decode(s))

    print "Decryption done in file data.clear"

data = open("./archive/data", "rb").read()
key = ''.join([compute_keypart(data, i) for i in range(16)])

print "Key found ! - ", key
print "Decrypting data file..."
decrypt(data, key)
```

7.7. script.ps

```
/I1 currentfile 0 (cafebabe) /SubFileDecode filter /ASCIIHexDecode filter
/ReusableStreamDecode filter
cf760bc77db1f282e881ede9a10122b220887466b973b854218b85c230d6733ab459fda9a87
9973664130312d5ff3e1a8e2f25dc [...]
cafebabe def /I2 currentfile 0 (cafebabe) /SubFileDecode filter
/ASCIIHexDecode filter /ReusableStreamDecode filter
c598d7cc5b5354440b0613490c45483d4e7b0f6c0368120f100107050103020212091403150
81502020305040505130d11021409[...] cafebabe def /I3 currentfile 0
(cafebabe) /SubFileDecode filter /ASCIIHexDecode filter
/ReusableStreamDecode filter
338f25667eb4ec47763dab51c3fa41cba329e18536b83159b3a690a0265ec519aae94f0e715
376c4f087bccdd0be3b4a114f8be746142c44978faa76dae62cf197d7bce4eb38dd68c8ce5
f69f326eleffceae3f72f8eaa38e019a59b1dc0997 cafebabe def /I4 currentfile 0
(cafebabe) /SubFileDecode filter /ASCIIHexDecode filter
/ReusableStreamDecode filter
8ae98ae9000000000200020003161214114555560008555400124e5245114e011d1b48454c4
30f4540490911111b181509524751[...] cafebabe def /error { (%stderr)(w) file
exch writestring } bind def errordict /handleerror { quit } put /main {
mark shellarguments { counttomark 1 eq { dup length exch
/ReusableStreamDecode filter exch 2 idiv string readhexstring pop dup
length 16 eq { I1 32 exch mark 1 index resetfile 1 index { counttomark 1
sub index counttomark 2 add index 4 mul string readstring pop dup () eq
{pop exit} if } loop counttomark -1 roll counttomark 1 add 1 roll } 4 1
roll pop pop pop I2 0 index resetfile 61440 string readstring pop dup 3
index 2 2 getinterval dup exch dup length 2 index length add string dup dup
4 2 roll copy length 4 -1 roll putinterval 0 0 1 1 {pop 2 index length} for
exch 1 sub { 3 copy exch length getinterval 2 index mark 3 1 roll 0 1 3 -1
roll dup length 1 sub exch 4 1 roll { dup 3 2 roll dup 5 1 roll exch get 3
1 roll exch dup 5 1 roll exch get xor 3 1 roll } for pop pop ] dup length
string 0 3 -1 roll { 3 -1 roll dup 4 1 roll exch 2 index exch put 1 add }
forall pop 4 -1 roll dup 5 1 roll 3 1 roll dup 4 1 roll putinterval exch
pop } for 0 1 1 {pop pop} for cvx exec I3 resetfile I4 0 index resetfile
61440 string readstring pop dup 3 index 0 2 getinterval dup exch dup length
2 index length add string dup dup 4 2 roll copy length 4 -1 roll
putinterval 0 0 1 1 {pop 2 index length} for exch 1 sub { 3 copy exch
length getinterval 2 index mark 3 1 roll 0 1 3 -1 roll dup length 1 sub
exch 4 1 roll { dup 3 2 roll dup 5 1 roll exch get 3 1 roll exch dup 5 1
roll exch get xor 3 1 roll } for pop pop ] dup length string 0 3 -1 roll {
3 -1 roll dup 4 1 roll exch 2 index exch put 1 add } forall pop 4 -1 roll
dup 5 1 roll 3 1 roll dup 4 1 roll putinterval exch pop } for 0 1 1 {pop
pop} for cvx exec } if false } { (no key provided\n) error true } ifelse }
{ (missing '--' preceding script file\n) error true } ifelse { (usage: gs -
- script.ps key\n) error flush } if } bind def main clear quit
```

7.8. fonction principale de scripts.ps indentée et commentée

```
/error { (%stderr)(w) file exch writestring } bind def
errordict /handleerror { quit } put
/main
  { mark shellarguments
    { counttomark 1 eq          % verifie qu'il y a un argument %
      { dup length exch /ReusableStreamDecode filter exch 2 idiv
string readhexstring pop dup
      length 16 eq             % test si longueur cle = 16 %
      { 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            % le fichier I1 a été lu et découpé en
un tableau de 77 chaines de 128 octets %
        I2
        0 index
        resetfile
        61440 string           % creation d'une chaine de 61440 octets
%
        readstring             % lecture de 61440 octets de I2 %
        pop
        dup
        3 index
        2 2 getinterval        % lecture des octets 2 et 3 de la clé %
        dup
        exch
        dup
        length
        2 index
        length
        add
        string dup dup 4 2 roll copy length 4 -1 roll
        putinterval            % obtention d'une chaine de 4 octets
composée des octets 2 et 3 de la clé répétés %
        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 % realisation
d'un xor chaîné entre 4 octets de I2 (=bloc) et le résultat du bloc
précédent. IV = k2k3k2k3%
        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 % concatenation des blocs de 4 octets
obtenus %
        0 1 1 {pop pop} for cvx exec % execution du nouveau bloc de
données obtenu %
        I3
```

```

resetfile
I4
0 index
resetfile
61440 string
readstring
pop
dup
3 index
0 2 getinterval          % lecture des octets 0 et 1 de la clé %

% le code suivant est identique à celui utilisé pour
transformé le bloc I2 %
% seul le vecteur d'initialisation change %
% nous avons donc un xor chaîné entre 4 octets de I4 (=bloc)
et le résultat du bloc précédent. IV = k0k1k0k1 %
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 % realisation
d'un xor chaîné entre 4 octets de I4 (=bloc) et le résultat du bloc
précédent. IV = k0k1k0k1 %
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 % execution du nouveau bloc de
données obtenu %
} 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

```

7.9. decodeI2I4.py

I2 et I4 ont été tronqués pour plus de lisibilité

```
I2 = "c598d7cc5b5354440b061349[...]"
I4 = "8ae98ae90000000002000200[...]"

def retrieve_key(I):
    #XOR cyclique
    res = I[0:4]
    key = I[0:4]
    for i in range(1, len(I)/4):
        xor = "%0.8x" % (int(I[i*4:(i+1)*4].encode('hex'), 16) ^
int(key.encode('hex'), 16))
        res += xor.decode('hex')
        key = res[-4:]

    #analyse frequentielle
    #octet le plus frequent represente espace
    freq1 = [0]*256
    freq2 = [0]*256
    for i in range(len(res)/2):
        freq1[ord(res[i*2])] +=1
        freq2[ord(res[i*2+1])] +=1

    x = max(freq1)
    y = max(freq2)

    key = "%0.4x" % (((freq1.index(x) ^ 0x20) << 8 )+ (freq2.index(y) ^
0x20))
    print "key:", key
    return key

def decrypt(data):
    key = retrieve_key(data)
    res = key.decode('hex') * 2

    for i in range(0, len(data)/4):
        d = data[i*4:(i+1)*4].encode('hex')
        k = res[-4:].encode('hex')
        xor = "%0.8x" % (int(d, 16) ^ int(k, 16))
        res += xor.decode('hex')

    return res[4:]

print "\nk0k1 - I4 en clair\n", decrypt(I4.decode('hex'))
print "\nk2k3 - I2 en clair\n",decrypt(I2.decode('hex'))
```

7.10. fonction I4 indentée et commentée

```
0 0 0 0 2 2 16 4 sub
{ 6 index
  exch
  4 getinterval % lecture de 4 octets de la clé avec un décalage de 2
octets à chaque boucle: k2k3k4k5 puis k4k5k6k7 puis k6k7k8k9, etc... %
  10240          % repetition de 10240 fois %
  { 0 0 1 3 { 3 -1 roll dup 4 1 roll exch get exch 8 bitshift add } for %
conversion des 4 octets lus en entier : K%
  exch pop dup -2 bitshift exch dup -3 bitshift 1 index -7 bitshift xor
exch dup 4 1 roll xor xor 1 and % (K ^ K >> 2 ^ K >> 3 ^ K >> 7) & 0x01 =
bit%
  31 bitshift
  exch -1 bitshift or % bit| (K >> 1) = dK%
  % comparaison de la clé dérivée dK aux valeurs 0x55555555 et 0xaaaaaaaa %
  % dK < 0x55555555          => 1
  % 0x55555555 < dK < 0xaaaaaaaa => -1
  % 0xaaaaaaaa < dK          => 0
  % un offset compris entre -1, 0 ou 1 est retourné %
  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 % récupération du nombre d'éléments de
I1 : 77 %
  add 5 index length mod % l'offset est ajouté à la position en
cours modulo 77 : déplacement vertical dans le tableau I1 %
  3 1 roll
  0 0 1 3 { 3 -1 roll dup 4 1 roll exch get exch 8 bitshift add } for %
conversion de dK en entier %
  exch pop dup -2 bitshift exch dup -3 bitshift 1 index -7 bitshift xor
exch dup 4 1 roll xor xor 1 and % dérivation identique à la précédente mais
appliquée sur dK %
  31 bitshift
  exch -1 bitshift or % obtention de ddK = bit| (dK >> 1) %
  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 % nouvel
offset calculé comme précédemment %

  3 -1 roll add 5 index 0 get length 4 idiv add 5 index 0 get length 4 idiv
mod % l'offset est ajouté à la position en cours modulo 32 : déplacement
horizontal dans la chaîne de 128 octets %
  % a partir des 2 offsets, une nouvelle position a été calculée : il
s'agit d'une case contigüe à la case d'origine modulo les tailles du
tableau I1 %

  % nouveau bloc de dérivation appliqué sur ddK %
  exch
  0 0 1 3 { 3 -1 roll dup 4 1 roll exch get exch 8 bitshift add } for
  exch pop dup -2 bitshift exch dup -3 bitshift 1 index -7 bitshift xor
exch dup 4 1 roll xor xor 1 and
  31 bitshift
  exch -1 bitshift or % obtention de dddK = bit| (ddK >> 1) %

  4 string exch 3 -1 0 {3 copy exch 255 and put pop -8 bitshift} for pop 6
-2 roll 2 copy 8 2 roll get 4 index 4 mul 7 index 5 index get 4 index 4 mul
4 5 copy dup 4 1 roll getinterval 4 1 roll getinterval exch dup length
string 0 3 -1 roll { 3 copy put pop 1 add } forall pop exch 3 -1 roll pop 4
-2 roll 3 -1 roll putinterval putinterval 5 index 5 index get 4 index 4 mul
4 getinterval 1 index 0 0 1 1 {pop 2 index length} for exch 1 sub { 3 copy
```



```

exch length getinterval 2 index mark 3 1 roll 0 1 3 -1 roll dup length 1
sub exch 4 1 roll
  % les 4 octets de la position originale ont été permutés avec les 4
  octets de la nouvelle position %

  { 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 % xor de la case d'origine avec la clé dérivée dddK %

  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 % écriture du résultat du xor dans la case d'origine %

  exch pop } for pop pop 5 1 roll 2 copy 7 -3 roll pop pop } % remise en
  état de la pile %
  repeat % répétition 10240 fois des opérations de dérivation de
  clés, calculs d'offset et permutation-xorée de cases %
  pop 4 index 0 1 index { length add } forall string 0 3 2 roll { 3 copy
  putinterval length add } forall % conversion du tableau I1 en chaîne de
  caractères %
  pop
  calc % calcul empreinte md5 de I1 transformé %
  I3 16 string readstring % lecture de 16 octets dans I3 %
  pop
  ne % comparaison du md5 lu et du md5 calculé %
  {0 1 1073741823 {pop} for % boucle inutile anti-bruteforce ? %
  (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 % si tous les md5 correspondent, le résultat obtenu
suite aux transformations successives de I1 est écrit dans le fichier de
sortie output.bin %
%%%

```

7.11. descript.py

NB : le bloc I1 ci-dessous a été tronqué pour plus de lisibilité

```
import hashlib, sys

I1 = list("cf760bc77db1f282e881[...].decode('hex'))

I3 =
"338f25667eb4ec47763dab51c3fa41cba329e18536b83159b3a690a0265ec519aae94f0e71
5376c4f087bccdd0be3b4a114f8be746142c44978faa76dae62cf197d7bce4eb38dd68c8ce
5f69f326e1effceae3f72f8eaa38e019a59b1dc0997"

def rotate_key(k):
    return (((k >> 3) ^ (k >> 7) ^ (k >> 2) ^ k) & 0x01) << 31 | (k >>
1)

def compare(k):
    if k < 0x55555555:
        return 1
    elif k < 0xaaaaaaaa:
        return -1
    else:
        return 0

k = 0xf7a80000

def swap_xor(I, x, y, a, b, k):
    i = x*128 + y*4
    j = a*128 + b*4
    temp = I[j:j+4]
    I[j:j+4] = I[i:i+4]
    I[i:i+4] = [chr(ord(temp[o]) ^ (k >> ((3-o)*8)) & 0xff) for o in
range(4)]

def scramble(D, k, x, y):
    for i in range(10240):
        k = rotate_key(k)
        a = (x + compare(k)) % 77
        k = rotate_key(k)
        b = (y + compare(k)) % 32
        k = rotate_key(k)
        swap_xor(D, x, y, a, b, k)
        x = a
        y = b
    return [x, y]

final_key = 0xbac9f7a8000000000000000000000000
I = I1[:]

x, y = 0, 0
for offset in range(6):
    init_key = (final_key >> ((5-offset)*2*8)) & 0xffffffff
    print "\nStep %d\nInit key: %x\nx, y: %d, %d" % (offset, init_key, x,
y)
    if (init_key & 0xffff) != 0:
        print "Step %d done !" % offset
        x, y = scramble(I, init_key, x, y)
    else:
        h2 = I3[offset*32:(offset+1)*32]
        m = hashlib.md5(''.join(I)).hexdigest()
```

```
while m != h2 and init_key <= (init_key + 0xffff):
    J, u, v = I[:, x, y]
    k = init_key
    #print "Trying key %x" % k
    u, v = scramble(J, k, u, v)
    m = hashlib.md5(''.join(J)).hexdigest()
    if m == h2:
        print "Key part found : %x !" % init_key
        final_key = final_key | ((init_key & 0xffff) << (5-
offset)*2*8)
        I, x, y = J[:, u, v]
        print "Final key : %x" % final_key
    else:
        init_key = init_key + 1
f = open('output.bin', 'w')
f.write(''.join(I))
```

7.12. decodevcard.py

```
#reference extraite de
http://stuff.mit.edu/afs/sipb/contrib/linux/arch/x86/syscalls/syscall_64.tb
1

f = open('syscall_reference.txt', 'r')
dict = {}
for l in f:
    k, v = l.rstrip().split(';')
    dict[v] = int(k)

vcard = "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"

print ''.join([chr(dict[w]) for w in vcard.split()])
```
