Cryptologie / Cryptanalyse matérielle

TP2 : Etude approfondie : Récupération d'une clé de chiffrement AES

Rapport rédigé & produit par : **Kévin MOREAU**

Professeur responsable : **Vianney LAPOTRE**

École Nationale Supérieure d'Ingénieurs de Bretagne Sud

Apprenti - Ingénieur Expert Cyberdéfense

2ème année

Promotion

AIRBUS

Table des matières

Table des figures

1. Rappel des objectifs

Ce deuxième TP est réalisé dans le cadre du cours de cryptologie matérielle dispensée par l'Ecole Nationale Supérieure d'Ingénieurs de Bretagne Sud pour la deuxième année de formation au diplôme d'ingénieur cyberdéfense.

L'objectif de ce deuxième TP est de retrouver, à partir d'une attaque par DPA, une clé AES non-protégé.

Ce rapport traite de plusieurs documents à destination pédagogique, l'idée étant de traiter et analyser l'ensemble de ces documents mis à notre disposition pour comprendre le principe de DPA avant d'attaquer le sujet, une première partie d'étude et de compréhension alimentera le début du rapport, avant de commencer le travail dirigé dans un second temps.

2. Principe du DPA (Differential Power Analysis)

2.1. Définition et principe de base

Le premier document fourni nous permet de comprendre le principe de fonctionnement d'une étude par DPA.

En cryptanalyse de matériel cryptographique, l'analyse de consommation (en anglais, ***differential power analysis ou DPA****) est l'étude des courants et tensions entrants et sortants d'un circuit dans le but de découvrir des informations secrètes comme la clé de chiffrement. Certaines opérations, plus coûteuses, augmentent la consommation électrique du circuit, notamment par l'utilisation de plus de composants (analogiques ou logiques). Cette analyse des variations et des pics permet de tirer des informations précieuses pour le cryptanalyste.*

Wikipédia

Cette définition indique que cette analyse se base sur un phénomène de différence de puissance se situant au cœur du système électronique d'un contrôleur :

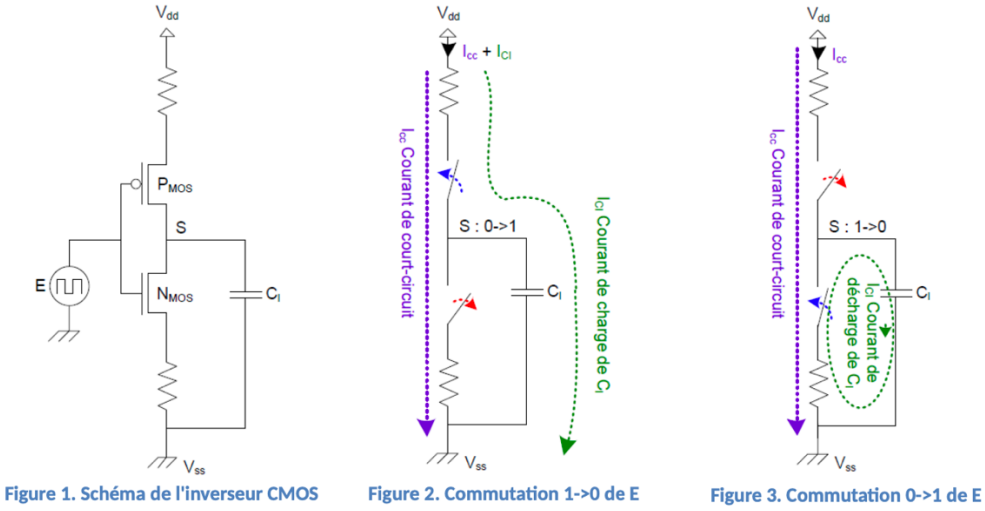

Figure 1. Schéma de l'inverseur CMOS | Figure 2. Commutation 1->0 de E | Figure 3. Commutation 0->1 de E

Figure 1 - Synthèse d'explication des CMOS

Comme on peut le voir sur les schéma ci-dessus, les courants induits par la commutation des deux transistors composant une entrée *E* provoque une puissance différente si la commutation s'effectue de $1 \rightarrow 0$ ou de $0 \rightarrow 1$. Par ailleurs, l'équation suivante permet d'affirmer que selon le type de commutation effectué en interne dans le circuit (et donc selon le type d'opérations logiques que l'on effectue, à savoir, des *ET*, des *OU*, des *NON*, etc.), la puissance dynamique varie selon ces paramètres :

$$P_{dyn} = \tau \times C_l \times V_{dd}^2 \times f$$

Ainsi, le modèle de puissance que l'on obtient est dynamique, on le note *Pdyn* et celle-ci dépend du mode de commutation et du nombre de ces derniers.

C'est cette puissance dynamique (et les tensions / courants la composant*)* qui va être visible à l'aide de matériels tels qu'on les a utilisé au premier TP (picoscope, sonde, etc.).

2.2. Analyse du signal induit

Ainsi, le sujet portant sur une étude de signal induite par le chiffrement symétrique d'une donnée en clair, voici à quoi ressemble une puissance dynamique :

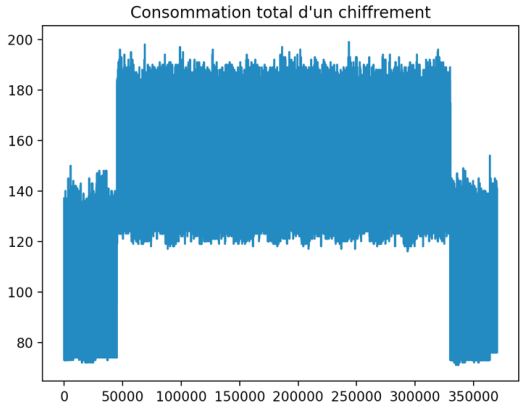

Figure 2 - Consommation totale d'un chiffrement utilisant AES

Les détails de cette courbe sont expliquées sur la *Figure 3*.
L'idée ici est de présenter la différence de puissance générée par la commutation d'entrées comme expliquées précédemment.

Ainsi, à partir de l'analyse de ce type de signal, il est possible de retrouver certains patterns correspondant à certaines types de commutations.

Avec un peu de temps et de l'expérience, il est ainsi possible de retrouver le fonctionnement d'un algorithme de chiffrement, simplement en analysant un signal.

C'est le cas ici :
Ce que l'on voit, c'est les 10 rounds de l'algorithme de chiffrement de AES (eux-mêmes composés de plusieurs opérations, comme nous allons l'expliquer plus tard).

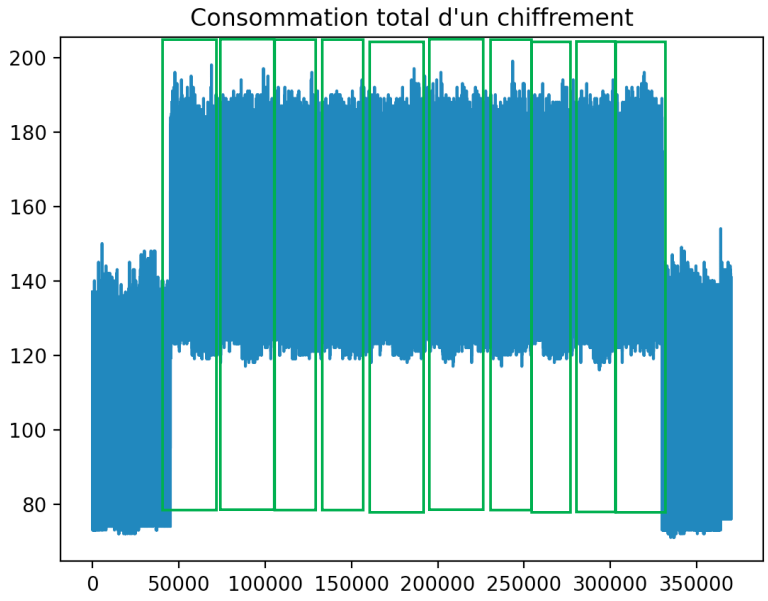

Figure 3 - Consommation totale avec décomposition des rounds AES

3. Analyse DPA

Avant de commencer, il est important de savoir que les traces qui sont la base de ce TP ne sont pas fournies dans l'archive de livraison de ce rapport (taille trop importante).

Par ailleurs, la clé que nous essayons de deviner est connue à l'avance, là voici :

00112233445566778899AABBCCDDEEFF

3.1. Réduction de l'intervalle pour accélérer l'attaque

Avant de commencer, il est nécessaire de comprendre rapidement le fonctionnement de AES ainsi que les rounds le composant.

En premier lieu, le texte en clair est *XOR* avec la clé de chiffrement utilisée. Cette opération est le *KeyWhitening* qui utilise l'opération *AddRoundKey (XOR)*.

Ensuite, 9 autres rounds identiques comportant des opérations de base :

- *SubBytes* : cette opération substitue les octets dans une matrice *SBOX.*
- *ShiftRow* : cette opération effectue un décalage ligne par ligne des octets (qui sont placés dans une matrice 4x4).
- *MixColumn* : mélange l'information entre les colonnes de la matrice 4x4.
- *AddRoundKey (XOR)* : le résultat des opérations effectués sur la matrice est *XOR* avec une expansion de la clé de chiffrement utilisée au départ.

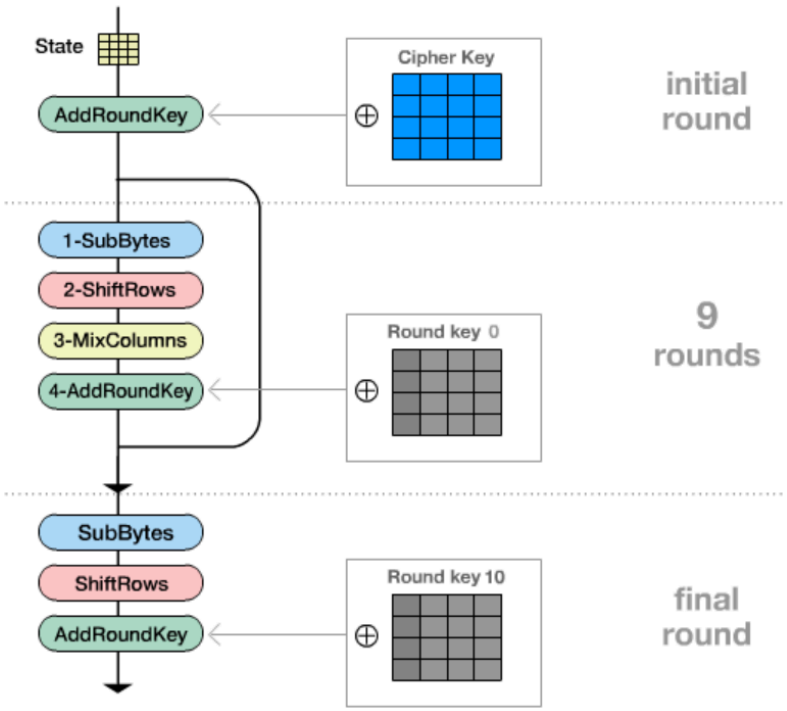

Figure 4 - Schéma détaillant le fonctionnement logique du chiffrement AES

Les figures précédentes (*Figure 2 et 3*) traitaient de la consommation totale du *FPGA* (*Field Programmable Gate Array*) implémentant AES.

A partir de ces figures, on récupère la section la plus importante qui est la première étape d'AES car elle utilise directement la clé qui nous intéresse.

Ainsi, on réduit notre intervalle correspondant à la première étape : la section se situe à l'intervalle (45000 : 75000). La modification de l'intervalle se fait directement dans le code Python :

```
offset = 40000
segmentLength = 75000
traces = O_traces[:,offset:offset+segmentLength]

plt.figure(2)
plt.plot(traces[0, :])
plt.title("Consommation total d'un chiffrement")

show()
```

Au final, la figure renvoyée ne traite que de la section correspondant au premier round d'AES :

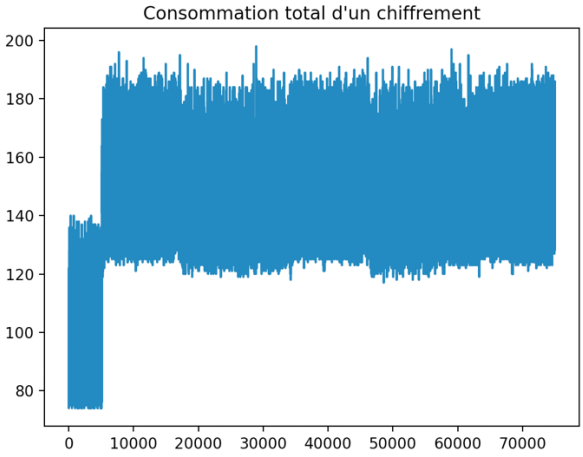

Figure 5 - Consommation du chiffrement concentré sur le round 1

Dans la prochaine partie, nous allons étudier un petit peu plus en profondeur cette section.

3.2. Création d'une matrice d'hypothèse des valeurs en sortie de fonction *SubBytes*

Une fois la section ciblée, il va falloir déterminer une *matrice d'hypothèses* contenant l'ensemble des valeurs possibles en sortie de la fonction SubBytes.

Cette *matrice* sera utilisée pour réaliser des corrélations entre les valeurs hypothèses et les traces mesurées sur la *Figure 5*.

Nous avons choisi la fonction *SubBytes* car cette fonction est non linéaire, une valeur *d'hypothèse* ressemblant fortement à une trace mesurée a donc une probabilité plus forte de correspondre.

Pour commencer, nous allons reproduire cette *matrice d'hypothèses* et attaquer uniquement le premier octet de la clé : on l'appelle la première sous-clé.

Rappel : Une clé de chiffrement AES peut être décomposée en 16 sous-clés de 8 bits (1 octet) chacune.

Ainsi, *la matrice d'hypothèse* contiendra l'intégralité des valeurs en sortie de *SBOX* en considérant le premier octet de chaque plaintext *XORé* avec toutes les valeurs de sous-clés possibles (256 valeurs car une sous-clé correspond à 1 octet = 8 bits).

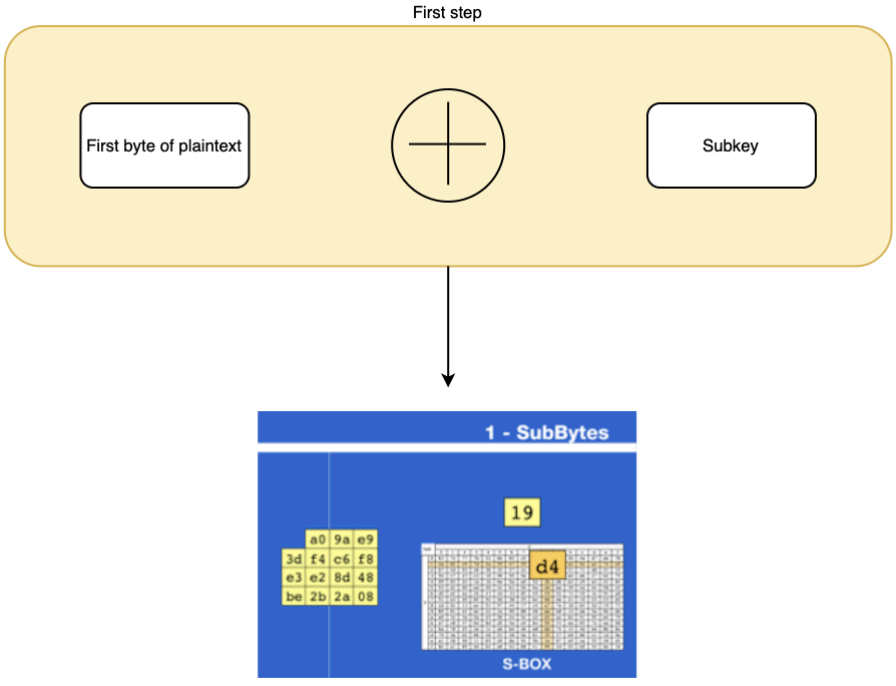

Figure 6 - Principe de fonctionnement d'un SubBytes

La matrice générée, pour le premier octet de chaque plaintext (1 octet par ligne, le premier), sera donc de l'ordre de 200x256 valeurs.

Rappel : Le fichier plaintext.txt fourni est composé de la façon suivante :

```
37 235 140 72 255 137 203 133 79 192 144 129 204 71 237 252
134 25 178 20 254 101 146 212 139 252 234 156 157 142 50 68
215 215 233 241 247 222 96 86 141 233 137 7 63 61 22 57
1 128 60 209 8 216 141 115 175 234 121 200 30 71 131 198
149 49 57 3 196 24 241 43 76 26 52 80 109 115 41 210
15 64 196 25 111 226 215 135 26 153 104 22 9 195 231 126
23 125 100 155 165 57 83 166 136 32 162 10 23 143 239 87
25 199 243 92 74 190 46 160 216 151 183 65 113 77 3 128
248 253 205 6 52 213 198 2 76 219 149 203 7 77 200 75
76 43 20 30 36 103 7 45 196 57 240 252 210 96 13 10
23 124 81 135 121 152 202 220 148 160 140 193 94 60 233 152
82 115 97 130 236 220 103 98 10 182 96 233 82 214 198 194
71 231 176 54 15 133 145 170 20 118 176 22 229 141 241 114
97 181 84 10 96 183 61 56 217 149 231 96 249 211 25 241
```

Figure 7 - Composition du plaintext

Tout au long du TP, nous allons dans un premier temps nous concentrer sur le premier octet parmi les seize de chaque ligne composant le plaintext (il y a 200 lignes dans le fichier fourni).

Concernant l'implémentation Python, elle se présente sous cette forme :

```
numberOfTraces = 200
keyCandidateStart = 0
keyCandidateStop = 255
hypothesis = zeros ((numberOfTraces,keyCandidateStop+1),uint8)
for p in range(0, numberOfTraces):
    for k in range(keyCandidateStart, keyCandidateStop+1):
        hypothesis [p,k]= plaintext[p,b]^k #key whitening
        hypothesis [p,k]= SBOX[hypothesis [p,k]]
```

Comme nous pouvons le voir, pour chaque premier octet de chaque ligne du plaintext, on remplit notre matrice avec toutes les valeurs de sous-clés possibles (de 0 à 255) suivant le protocole d'une fonction *SubBytes* à savoir :

- XOR du plaintext avec la sous clé -> Obtention d'une valeur
- Utilisation de la valeur obtenue pour récupérer la valeur indéxée dans la SBOX et la remplacer dans la matrice

Notre tableau d'hypothèse construit pour chaque valeur de sous-clé sur le premier octet de chaque ligne du plaintext, on passe à la suite.

3.3. Création des groupes initiaux

A partir de maintenant, étant donné que notre matrice d'hypothèses des 256 sous-clés potentielles est effectuée pour notre premier octet de chaque ligne du plaintext, on peut analyser la puissance dynamique pour comparer notre matrice d'hypothèse avec les traces.

Pour réaliser cette dernière, il faut former deux groupes dans lesquels nous intégrerons nos traces en fonction de notre modèle.

En fait, nous allons tracer une courbe DPA pour chaque hypothèse de sous-clés possibles (256 sous-clés = 256 courbes).

On commence par créer deux groupes contenant 256 vecteurs nuls de taille *segmentLength* :

```
mean_0 = zeros((256,segmentLength))
mean_1 = zeros((256,segmentLength))
```

Comme nous pouvons le constater, chacun de ces groupes peut contenir 256 vecteurs (1 vecteur par sous-clé).

Pour la première sous-clé donc, seul le premier vecteur de chacun des deux groupes sera utilisé.

Aussi, chaque vecteur de chaque groupe permet de stocker *segmentLength* valeurs : c'est à dire la longueur d'une trace.

3.4. Ajout des traces dans les groupes

Nos groupes étant construits, pour ajouter une trace dans un groupe, il suffit d'additionner le vecteur du groupe (initialisé précédemment) avec la trace.

Concernant l'implémentation Python, cela donne :

```
p = 0 # on considère le premier plaintext
k = 0 # on considère une sous clé = 0
mean_0[k,:]=mean_0[k,:]+traces [p,:]
```

Ici on utilise inconsciemment le premier groupe.

L'intérêt d'avoir deux groups réside dans le modèle précédemment créé :
Si le bit en sortie de SBOX est égal à 1 alors on classe la courbe hypothétique dans un groupe spécifique. Dans le cas inverse, si le bit de sortie est égal à 0, on place la courbe dans l'autre groupe :

```
p = 0 # on considère le premier plaintext
k = 0 # on considère une sous clé = 0

if (bit_get(hypothesis[p,k],1)==0):
   mean_0[k,:]=mean_0[k,:]+traces [p,:]
else:
   mean_1[k,:]=mean_1[k,:]+traces [p,:]
```

Ici, le modèle fonctionne pour le premier plaintext et la première sous clé, maintenant, il faut appliquer ce modèle pour traiter l'ensemble des premiers octets de chacune des 200 lignes composant le plaintext avec chacune des clés possibles :

```
for k in range (keyCandidateStart, keyCandidateStop+1):
        for p in range (0, 200):
            if (bit_get(hypothesis[p,k],1)==0):
                mean_0[k,:]=mean_0[k,:] + traces [p,:]
            else:
                mean_1[k,:]=mean_1[k,:] + traces [p,:]
```

3.5. Traitement de la moyenne par groupe pour chaque sous-clé et création de la courbe DPA

A chaque sous-clé traitée, on traite la moyenne dans chaque groupe pour la sous-clé en question. Ainsi, cela demande d'avoir un compteur incrémenté à chaque ajout de trace dans le groupe. Une fois le traitement effectué et les compteurs en place, on calcule la moyenne dans chaque groupe pour cette sous-clé.

A partir de là, la DPA s'appuyant sur la différence des moyennes, il suffit ensuite de faire la différence en valeur absolue des deux courbes afin d'obtenir une courbe DPA :

```
    for k in range (keyCandidateStart, keyCandidateStop+1):
        mean_0 = zeros((256,segmentLength))
        mean_1 = zeros((256,segmentLength))
        nb0 = 0
        nb1 = 0
        for p in range (0, 200):
            if (bit_get(hypothesis[p,k],1)==0):
                mean_0[k,:]=mean_0[k,:] + traces [p,:]
                nb0 += 1
            else:
                mean_1[k,:]=mean_1[k,:] + traces [p,:]
                nb1 += 1

        mean_0[k,:]=mean_0[k,:]/nb0
        mean_1[k,:]=mean_1[k,:]/nb1
        diff = abs(mean_1[k,:] - mean_0[k,:])
```

NB : Le calcul de moyenne s'effectuant sur chacune des sous-clé, il ne faut pas oublier de réinitialiser les deux groupes à 0 lorsque l'on change de valeur de sous-clé ainsi que les compteurs nb0 et nb1.

Enfin, quand une corrélation existe entre un tri de trace effectué et une hypothèse de sous-clé, un pic relativement élevé apparaît sur la courbe :

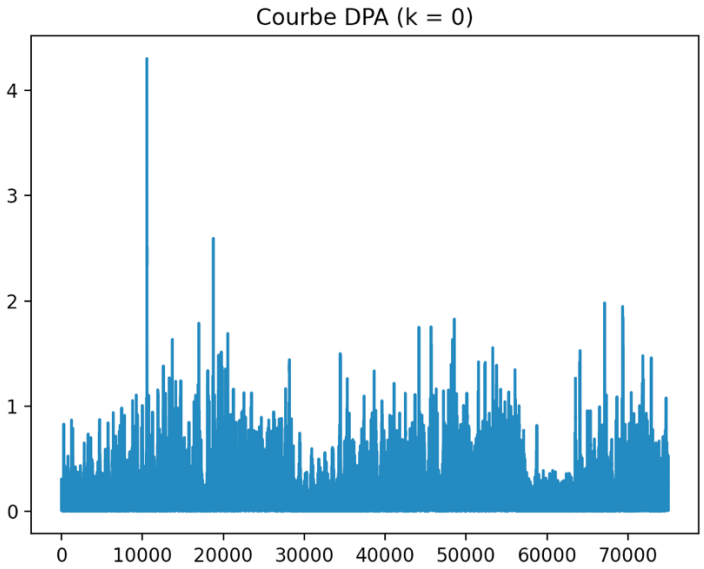

Figure 8 - Exemple effectué en montrant une DPA pour k = 0

3.6. Classement des maximums obtenus par sous-clé et détermination

Généralement, la courbe contenant le pic le plus élevé est la bonne sous-clé, mais ce n'est pas forcément vrai avec seulement 200 traces. En effet, avec un nombre d'échantillons aussi faible, il est difficile d'affirmer que la courbe avec le pic le plus élevé (par chance, ici la bonne clé est bien celle-ci) est forcément la bonne clé. Pour arriver à une telle conclusion, il faudrait un nombre d'échantillons beaucoup plus important.
Ainsi pour exploiter au mieux ces pics de donnée, il faut donc classer les sous-clés probables (ces pics) par ordre décroissant, puis afficher cette tendance.

Pour cela, on utilise la fonction *np.amax* de la librairie *numpy* pour ajouter le point maximal de la courbe diff obtenue dans un tableau vide.

```
max = np.amax(diff)
probKeys.append(max)
x = np.linspace(0,255,256)
plt.figure(4)
plt.plot(x, probKeys)
plt.title("Courbes des maximums")
show()
```

Dans le cas où k = 0, on voit bien que le point maximal de la courbe *diff* relevée se situe à environ *4.3* en ordonnée.
On répète l'opération pour toutes les sous-clés possibles (256) et on trace la courbe des maximums :

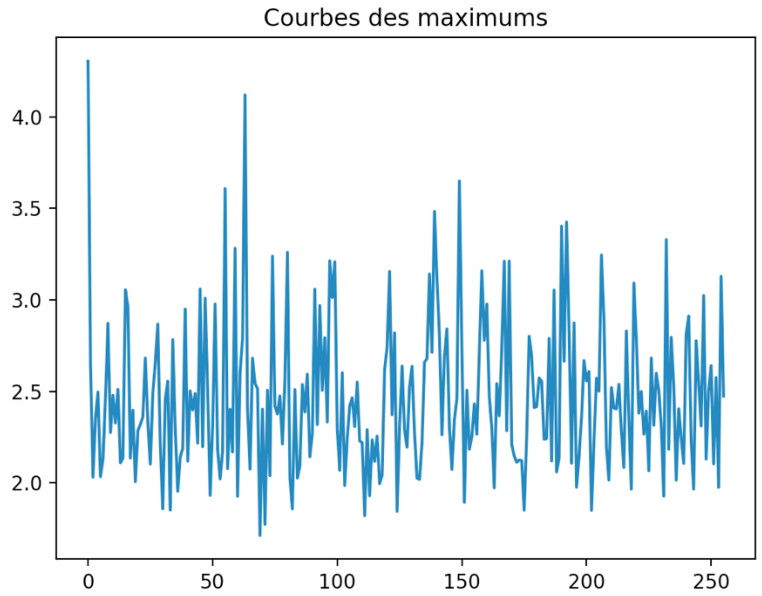

Figure 9 - Courbe des maximums pour chaque sous-clé testée

En abscisse se situe l'intégralité des valeurs de sous-clés possibles (0 à 255).

En face de chaque clé se trouve la valeur maximale de corrélation détectée pour chacun des premiers octets du plaintext. Comme on peut le constater ici, il est fort probable que la sous-clé disposée à chiffrée le contenu des premiers octets des éléments du plaintext est la sous-clé de valeur 0 (00_{16}).

4. Automatisation du processus de DPA

4.1. Limite du principe d'automatisation

Pour rappel, le principe de fonctionnement précédent ne fonctionnait que pour la détermination de la première sous-clé associée au premier octet de chaque ligne du plaintext. Pour déterminer la clé finale composée des 16 sous-clés, il faut répéter ce qui a été fait précédemment 15x de plus pour pouvoir trouver notre clé.

En introduction, il est nécessaire d'admettre que la solution proposée ici ne marche que parce que nous connaissons la valeur de la clé finale. En effet, étant donné le nombre faible d'échantillons, le principe de récupérer les valeurs maximales pour chaque courbe *diff* des sous-clés possibles est relativement peu fiable.

En effet, pour que celle-ci le devienne, il faudrait un nombre beaucoup plus conséquent d'échantillons.

Dans notre cas, avec si peu d'échantillons, la meilleure chose que l'on puisse faire consite à brute-force intelligemment les combinaisons fortement fiables (en prenant en considération, par exemple, les valeurs maximales max_{-1}, max_{-2}, celles-ci étant déterminées par leur valeur de corrélation, inférieur à *max*) au lieu de tester l'ensemble des valeurs possibles pour une clé.

Ce n'est pas ce que l'on va faire ici, on se contentera uniquement d'accorder notre confiance aveugle sur la valeur maximale repérée par la DPA, comme nous l'avons fait pour la première sous-clé, et ce, pour chaque sous-clé à tester.

Enfin, pour éviter le surplus d'apparition de courbes, l'intégralité du programme pour la première partie de ce TP (c-à-d. de la partie 1. à la partie 3.) est programmé dans le programme *main.py*

La suite de ce TP n'intègre pas l'apparition des courbes de consommation, des courbes DPA et des courbes de maximums. Elle se concentre uniquement sur la récupération des 16 sous-clés et de la restitution de la clé finale.

Ainsi, le programme a été adapté et programmé dans le programme *findKey.py* détaillé dans cette présente partie.

4.2. Implémentation

L'implémentation Python de l'automatisation se présente de la façon suivante :

Tout d'abord, la fonction de création de la matrice d'hypothèse a été revue pour pouvoir y intégrer les octets suivants d'une ligne :

```
def createHypothesisTab(numberOfTraces, keyCandidateStart, keyCandidateStop, SBOX,
byte):
    columns = 16
    rows = numberOfTraces
    plaintext = myin('plaintext.txt', columns, rows)
    hypothesis = zeros ((numberOfTraces,keyCandidateStop+1),uint8)

    k = 0 # on considère une sous clé = 0
    p = 0 # on considère le premier plaintext
    for p in range(0, numberOfTraces):
        for k in range(keyCandidateStart, keyCandidateStop+1):
            hypothesis [p,k]= plaintext[p,byte]^k #key whitening
            hypothesis [p,k]= SBOX[hypothesis [p,k]]
    return hypothesis
```

Ainsi, ces octets sont associés à un paramètre d'entrée de la fonction, que l'on va pouvoir faire varier en fonction des 15 autres octets que l'on a à traiter dans chaque ligne du plaintext.
Lors de la fonction *KeyWhitening*, on ciblera l'octet voulu sur la ligne.
Enfin, pour chaque octet, la fonction renvoie une matrice d'hypothèse différente :

```
Tableau d'hypothèses pour l'octet 0
[[ 63  54 204 ...  97 185  87]
 [ 68  23  95 ...  33 188 182]
 [ 14 246   3 ... 229 165  52]
 ...
 [ 30 155 233 ... 250 240  71]
 [142 148 105 ... 175 173 212]
 [ 43 103   1 ...  66 230 191]]
```

```
Tableau d'hypothèses pour l'octet 1
[[233 135  30 ...  71  89 250]
 [212 173 175 ... 105 148 142]
 [ 14 246   3 ... 229 165  52]
 ...
 [158  29 193 ... 147 253 183]
 [153  65  15 ... 242 197 111]
 [ 10  58  50 ...  88  76  74]]
```

```
Tableau d'hypothèses pour l'octet 2
[[100  93  25 ... 163  64 143]
 [ 55 109 231 ... 132  41 227]
 [ 30 155 233 ... 250 240  71]
 ...
 [ 77  67 133 ...  70  20 184]
 [ 29 158 134 ...  38 183 253]
 [172  98 194 ...  91  32 252]]
```

La suite est relativement triviale, on applique une boucle for pour chaque octet, ainsi que la fonction DPA renvoyant la sous-clé la plus probable pour chacun d'entre eux, et le tour est joué :

```
    #--Déchiffrement de la clé par DPA (approximation)--#
    byte = 1
    key =  []
    print("[!] Détermination des sous-clés en cours...")
    for byte in range(0,16):
        hypothesisTab = createHypothesisTab(numberOfTraces, keyCandidateStart, keyCandi-
dateStop, SBOX, byte)
        subkey = DPA(segmentLength, numberOfTraces, traces, hypothesisTab, keyCandi-
dateStart, keyCandidateStop)
        subkey = hex(subkey)
        key.append(subkey)
        print("Valeur pour la sous clé %i : %s " % (byte, subkey))

    print("Clé de chiffrement probable utilisée : ", key)
    #---------------------------------------#
```

5. Conclusion

La méthode de déchiffrement d'une clé par DPA est relativement fiable. Son efficacité est proportionnelle aux nombres d'échantillons (traces) récupérés.

En effet, dans notre cas, 200 échantillons permettent d'arriver à un résultat assez fiable si l'on en croit la sortie finale de notre programme :

```
Valeur pour la sous clé 0 : 0x0
Valeur pour la sous clé 1 : 0x11
Valeur pour la sous clé 2 : 0x22
Valeur pour la sous clé 3 : 0xc5
Valeur pour la sous clé 4 : 0x44
Valeur pour la sous clé 5 : 0x55
Valeur pour la sous clé 6 : 0x66
Valeur pour la sous clé 7 : 0x25
Valeur pour la sous clé 8 : 0xf8
Valeur pour la sous clé 9 : 0x99
Valeur pour la sous clé 10 : 0xaa
Valeur pour la sous clé 11 : 0xbb
Valeur pour la sous clé 12 : 0x9a
Valeur pour la sous clé 13 : 0x1a
Valeur pour la sous clé 14 : 0xee
Valeur pour la sous clé 15 : 0xff
Clé de chiffrement probable utilisée :  ['0x0', '0x11', '0x22', '0xc5', '0x44', '0x55', '0x66', '0x25', '0xf8', '0x99', '0xaa', '0xbb',
 '0x9a', '0x1a', '0xee', '0xff']
```

Figure 10 - Détermination finale approximative de la clé de chiffrement

Comme on peut le voir, la clé finale n'est pas correctement restituée malgré des sous-clés correctement déchiffrées.

Comme introduit dans cette partie, la fiabilité de cette solution réside dans le nombre d'échantillons. Pour arriver à nos fins, il faudrait bruteforce chacune des sous-clés en testant la combinaisons d'une plage des valeurs maximales détectées par la corrélation.

Par exemple, au lieu de tester toutes les combinaisons on pourrait tester une sous-clé en récupérant nous pas une valeur maximale, mais une plage de 2 à 5 valeurs. Ensuite, on testerait les combinaisons entre ces 5 valeurs pour chaque sous-clé. La plage de bruteforce est alors considérablement réduite pour trouver la clé finale.

En conclusion, ce TP m'a permis d'améliorer grandement ma compréhension autour du sujet de la cryptanalyse et du DPA de façon générale.

L'animation fournie dans les annexes du TP a été très enrichissante, ainsi que les explications tout au long du TP à travers les exercices dirigés.

Enfin, le fonctionnement de l'algorithme AES de façon logique est beaucoup mieux maitrisé aujourd'hui pour ma part ainsi que ma maîtrise de la librairie numpy que j'avais peu utiliser jusque-là.

Merci pour votre lecture,
Kévin MOREAU.