



Sécurité et Canaux Auxiliaires

YU Xiyong, ZHOU Su

February 19, 2018

Table des matières

1	Chipwhisperer	4
2	SPA: simple power analysis	4
2.1	Présentation	4
2.2	Exercice 1	5
2.2.1	Identifier et expliquer la contre-mesure basic-passwdcheck.c	5
2.2.2	Généraliser l'attaque sans savoir la taille du mot de passe	6
2.3	La contre-mesure améliorée	9
3	Exercice 2	10
3.1	Rapelle d'AES128	10
3.2	Identifier des tours de AES et les différentes parties de chaque retour	12
3.3	Confirmer les hypothèses de partition de AES en utilisant l'opération NOP	13
3.4	Identifier la commande minimale pour obtenir le début du premier tour de l'AES	15
4	Exercice 3	16
4.1	Modification du programme pour enregistrer pleins de traces, correspondant au message aléatoire et vérifier si chaque chiffrement sur XMEGA sont égaux au résultat obtenu en utilisant la bibliothèque Crypto.Cipher	16
4.2	Le résultat obtenu sur un octet par la sortie de la première sbox	17
4.3	L'attaque par DPA et hamming model	17
4.3.1	DPA: Differential Power Analysis	17
4.3.2	Hamming weight power model	21
4.3.3	La théorie de l'attaque CPA (correlation power analysis) .	21
4.4	Lire le fichier CSV	22
4.5	Pearson correlation coefficient	22
4.6	Appliquer la fonction correlation coefficient sur le premier octet des hypothèses	22
4.7	Finaliser l'attaque CPA	24
5	Exercice 4	25
5.1	AES128-RSM	25
5.1.1	RSM: Rotation Sboxes masking	25
5.1.2	Pseudo Code	28
5.2	Évaluer cette contre-mesure	30
5.3	Conclusion	32
6	Exercice 5	32
6.1	IDEA	32
6.1.1	Principe	32

6.1.2	Implémentation d'IDEA pour XMEGA	32
6.1.3	Tours du chiffrement[7]	33
6.1.4	Génération des sous-clés	34
6.1.5	Transformation finale sur la dernière génération de sous-clés	36
6.1.6	Exemple pour l'exécution de l'algorithme IDEA	36
6.2	La sécurité du cryptosystème IDEA contre l'attaque CPA	38
6.3	Transformer le Timing attaque pour une attaque qui est basée sur la consommation d'énergie[5]	38
6.3.1	Attaque uniquement en message chiffré exigent des tim- ings de blocs précis	38
6.3.2	La Timings attaque pour le plaintext adaptatif choisi . . .	39
Références		41
Appendices		42
A	Position des NOPs pour AES	42
B	Modification de paramètre afin d'obtenir le début du premier tour	45
C	Sbox(input\opluskey)	45
D	Lire les fichiers	46
E	L'algorithme pour calculer Pearson correlation coefficient	47

1 Chipwhisperer

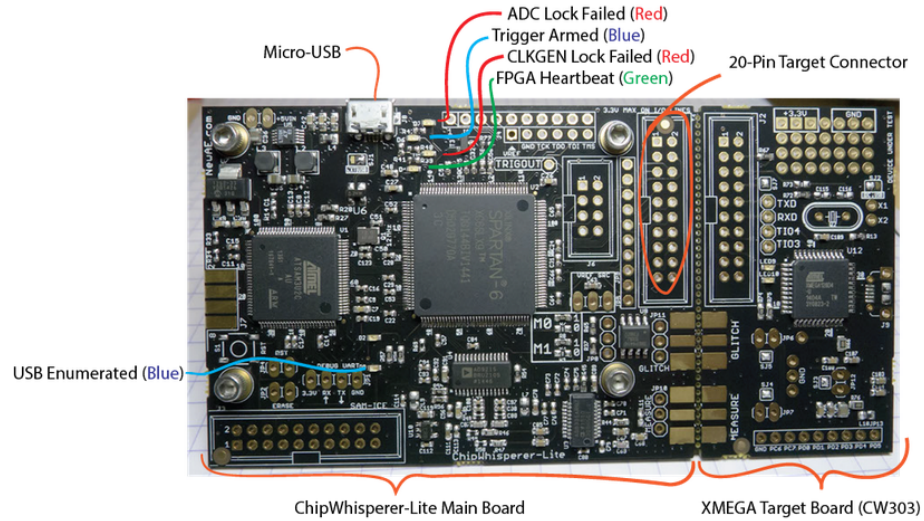


Figure 1: CW1173 ChipWhisperer-Lite [2]

Chipwhisperer-Lite est constituée de deux parties importantes: un chipwhisperer-lite main board et un target board:

- Target board est un micro-contrôleur dans lequel on peut implémenter des algorithmes. Par exemple: AES.
- La mesure des traces de consommations de ce micro-contrôleur (dépend de l'algorithmes qu'on a implémenté dans la puce) grâce au chipwhisperer-lite main board relié à un logiciel sur ordinateur avec un câble USB.

2 SPA: simple power analysis

2.1 Présentation

L'attaque par SPA consiste à exploiter des informations sur une ou très peu de traces collectées. Une trace est une mesure du courant consommé par un circuit. On peut déduire les opérations effectuées selon les traces collectées.

Voici une trace correspondant à 20 NOPS et 10 MULs

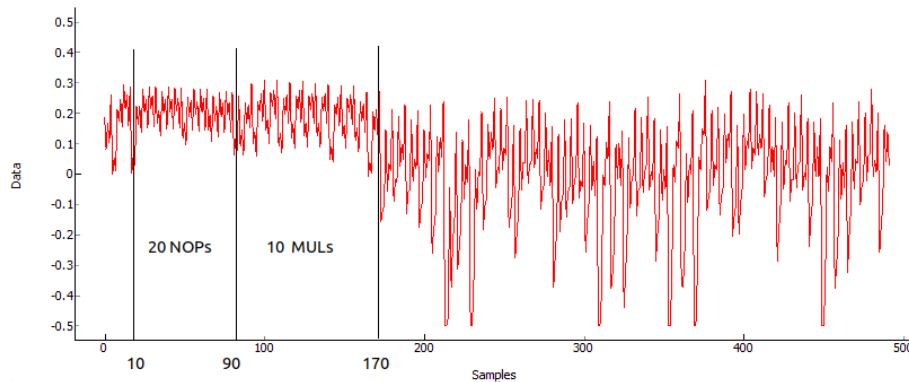


Figure 2: Trace:NOPs et MULs

- NOP:
 - Faire rien
 - 1 horloge à compléter
 - Moins de consommation de puissance
- MUL:
 - Multiplier deux 8-bits nombres
 - 2 horloges à compléter
 - Plus de consommation de puissance

On s'intéresse du sample 10 au sample 170, on remarque que 10 MULs prennent 80 samples à compléter et 20 NOPs prennent 80 samples à compléter.

2.2 Exercice 1

2.2.1 Identifier et expliquer la contre-mesure basic-passwdcheck.c

```

1  if (passbad){
2      //Stop them fancy timing attacks
3      int wait = rand();
4      for(volatile int i = 0; i < wait; i++){
5          ;
6      }
7      _delay_ms(500);
8      printf("PASSWORD FAIL\n");
9      led_error(1);
10 } else {
11     printf("Access granted, Welcome!\n");
12     led_ok(1);
13 }
14 //All done;

```

```

15     while(1);
16 }
17
18

```

La boucle "for" est principalement utilisée pour faire la contre-mesure. Si le mot de passe est faux, la fonction "rand()" nous rend un entier entre 0 et RAND_MAX stocké dans la variable "wait" et on itère wait fois dans la boucle vide. Après 500ms, il affiche PASSWORD FAIL. "led" est rouge, on entre ensuite dans "while(1)", comme la condition est toujours vérifiée, on ne sort jamais de la boucle. Si on veut encore tester le programme, il faut réinitialiser le chipwhisperer. Donc, on ne peut pas tenter consécutivement le mot de passe.

2.2.2 Généraliser l'attaque sans savoir la taille du mot de passe

Le mot de passe correct est : **h0px3**

On suppose qu'on connaît au moins un bon caractère de mot de passe dans cet exercice. Voici les mesures obtenues en entrant h (un bon caractère du mot de passe) et a (un mauvais caractère du mot de passe):

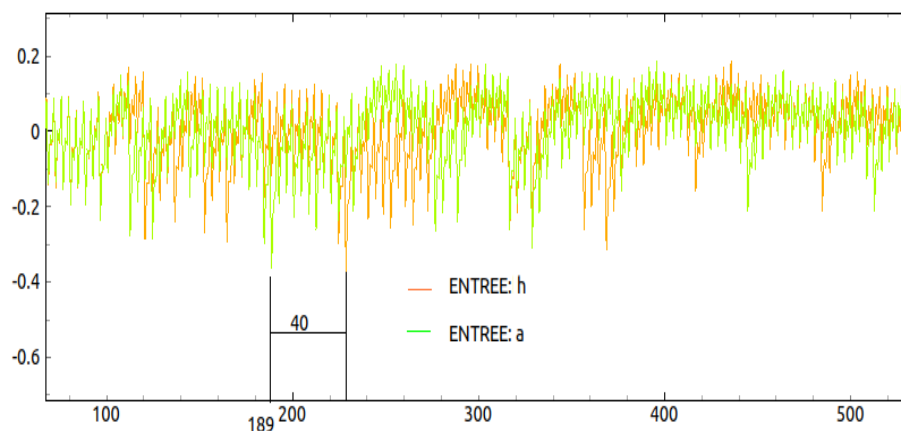


Figure 3: Comparaison entre la consommation pour h et la consommation pour a

La courbe orange correspond au premier bon caractère du mot de passe car l'entrée est h et la courbe verte correspond à aucune bonne lettre du mot de passe car l'entrée est a. Après plusieurs tentatives et le graphique au-dessus, on obtient le bon point qui correspond à l'offset de 189 échantillons, la valeur de delta correspond au décalage engendré par un mauvais caractère du mot de passe, est de 40 échantillons. On peut donc déclarer les attaques avec l'offset et la valeur de delta trouvés.

Voici l'algorithme d'attaque avec la connaissance de la taille du mot de passe:

Algorithm 1 Timing attaque: le script en python avec la connaissance de la taille du mot de passe, le mot de passe correct est : **h0px3**

Entrées: trylist = "abcdefghijklmnopqrstuvwxyz0123456789"

Sorties: password

```
1: for i in range(0,5) do
2:   print "****CHARACTER %d * * * *" % i
3:   for c in trylist do
4:     cap.setParameter(['Target Connection', 'Go Command', password +
5:       "\%c\n" % c])
6:     cap.capture1()
7:     print "Try = %c" % c
8:     print cap.scope.datapoints[189 + i*40]
9:     if cap.scope.datapoints[189 + i*40] > -0.2: then
10:      print "****CHARACTER %d = %c * * * *" % (i, c)
11:      password += c
12:      break
13:     else if c == "9" then
14:      print "****CHARACTER %d FAILED * * * *" % (i)
15:      password += "?"
16:     end if
17:     # Call to pe() causes GUI to process outstanding events
18:     # useful if you are calling API directly
19:     pe()
20:   end for
21: end for
```

L'algorithme exécute "taille" fois de boucle, chaque boucle peut trouver un bon caractère du mot de passe. Le premier bon caractère du mot de passe se situe à l'échantillon 189. La distance entre deux bons caractères est de 40 échantillons. On a bien trouvé **h0px3** avec cet algorithme.

Voici l'algorithme d'attaque sans la connaissance de la taille du mot de passe:

Algorithm 2 Timing attaque: le script en python sans la connaissance de la taille du mot de passe, le mot de passe correct est : **h0px3**

Entrées: trylist = "abcdefghijklmnopqrstuvwxyz0123456789"
Sorties: password

```

1: password = ""
2: run = 1
3: i=0
4: while run do
5:     count = 0
6:     print "****CHARACTER %d ***" % i
7:     for c in trylist do
8:         cap.setParameter(['Target Connection', 'Go Command', password +
9:             "%c\n" % c])
10:        cap.capture1()
11:        print "Try = %c" % c
12:        print cap.scope.datapoints[189 + i*40]
13:        if cap.scope.datapoints[189 + i*40] > -0.2 then
14:            print "****CHARACTER %d = %c ***" % (i, c)
15:            password += c
16:            break
17:            #never count 9
18:            #elif c == "9":
19:            #print "****CHARACTER %d FAILED ***" % (i)
20:            #password += "?"
21:        end if
22:    end for
23:    count+=1
24:    if count == len(trylist) then
25:        break
26:    end if
27:    i+=1
28:    #Call to pe() causes GUI to process outstanding
29:    #events, useful if you are calling API directly
30:    pe()
31: end while

```

On entre dans une boucle "while", on incrémente "count" par 1 lorsqu'on teste un élément du "trylist". On remet "count" à 0 si on trouve un caractère du mot de passe. On sort de la boucle "while" lorsqu'on parcourt tous les éléments du "trylist" sans avoir un bon caractère du mot de passe.

On a réussi à trouver le mot de passe: **h0px3**.

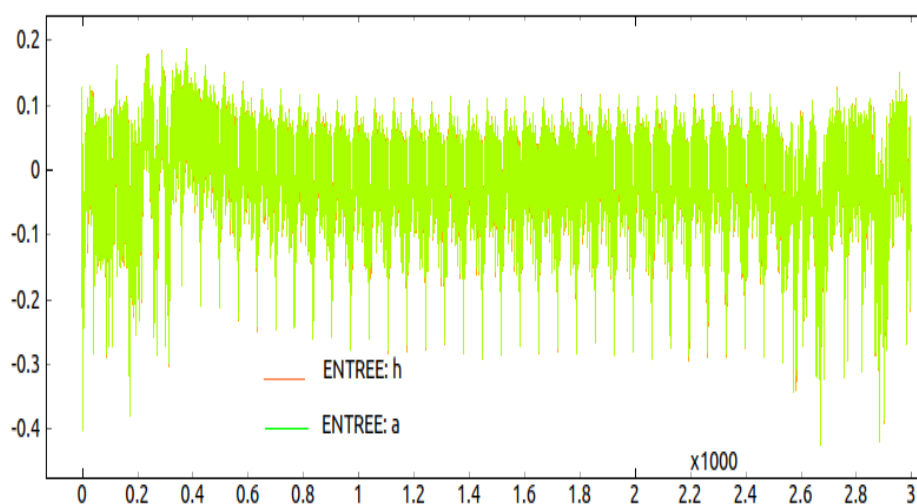
2.3 La contre-mesure améliorée

Pour déclarer les attaques qu'on a décrit précédemment. On a besoin de déterminer la valeur d'offset et la valeur de delta. Notre but est de "cacher" ces deux valeurs. Nous présentons deux contre-mesures différentes contre timing attaque.

Algorithm 3 solution 1: if et else

```
for uint8_t i = 0; i < sizeof(correc_passwd); i++) do
    wif= rand();
    welse=rand();
    if (correct_passwd[i] != passwd[i]) then
        for (volatile int i = 0; i < wif; i++) do
            ;
        end for
        passbad += 1
    else
        for volatile int i = 0; i < welse; i++) do
            ;
        end for
        passbad += 0;
    end if
end for
```

On ajoute une boucle dans "if" ensuite on ajoute aussi "else" et on ajoute "passbad +=0"; et une boucle dans "else". Notre but est d'équilibrer la consommation de puissance. On obtient le schéma suivant:

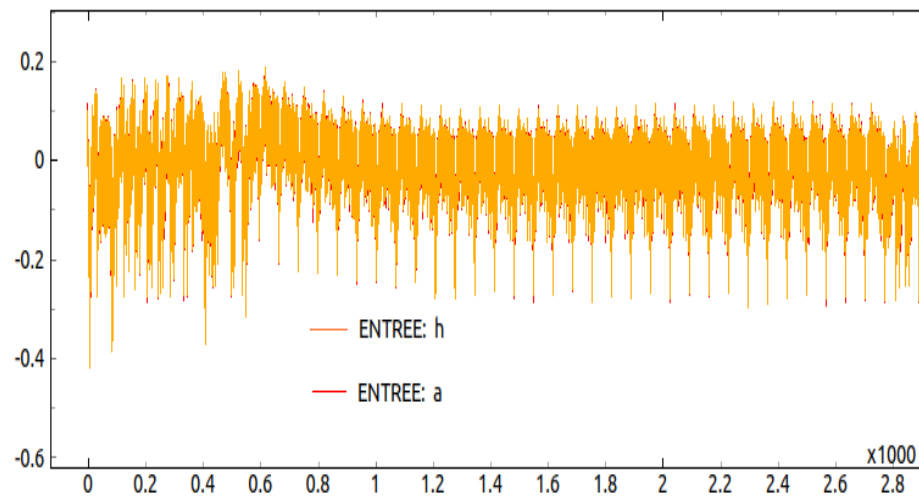


On obtient des courbes quasiment superposées:

Algorithm 4 solution 2: xor

```
for uint8_t i = 0; i < sizeof(correc_passwd); i++) do
    for (uint8_t i = 0; i < sizeof(correct_passwd); i++) do
        passbad+ = correct_passwd[i] ^ passwd[i]
    end for
end for
```

Notre but est de ne plus faire de distinction entre un bon caractère entré et un mauvais caractère entré. Autrement dire, on enlève "if" grâce à l'utilisation de "xor", on obtient un schéma suivant:



On remarque qu'on obtient des courbes superposées.

On n'arrive pas à trouver la valeur d'offset et la valeur de delta, mais on a quand même décidé de déclarer l'attaque avec les valeurs précédentes afin de vérifier si les contre-mesures résistent bien au timing attaque. On en conclut qu'ils résistent bien au timing attaque.

3 Exercice 2

3.1 Rapelle d'AES128

Voici le chiffrement D'AES128

- Un bloc de 16 octets du message clair entré est représenté sous une matrice de 4x4
- Une clé secrète Ks de 16 octets est aussi représentée comme une matrice de 4x4, xore avec le message entré appelé AddRoundKey.

- Un tour de l'AES consiste à 4 opérations. À chaque itération une nouvelle clé secrète $K(i)$ est utilisé. L'indice i est varié de 1 à 9. $K(i)$ est généré à partir de K_s .
 - SubBytes est une opération de substitution en utilisant Sbox. Elle est la seule opération non linéaire, elle joue un rôle important pour l'attaque ou la protection (surtout SubBytes au premier tour ou au dernier tour)
 - ShiftRows effectue un décalage circulaire d'un octet vers la droite pour la deuxième ligne, un décalage de deux octets vers la droite pour la troisième ligne, un décalage de trois octets vers la droite.
 - MixColumn correspond à une multiplication et réduit le modulo.
 - AddRoundKey avec une nouvelle clé.
- SubBytes
- ShiftRows
- AddRoundKey avec $K(10)$
- Un bloc de 16 octets du message chiffré

On illustre le chiffrement d'AES par l'image suivante:

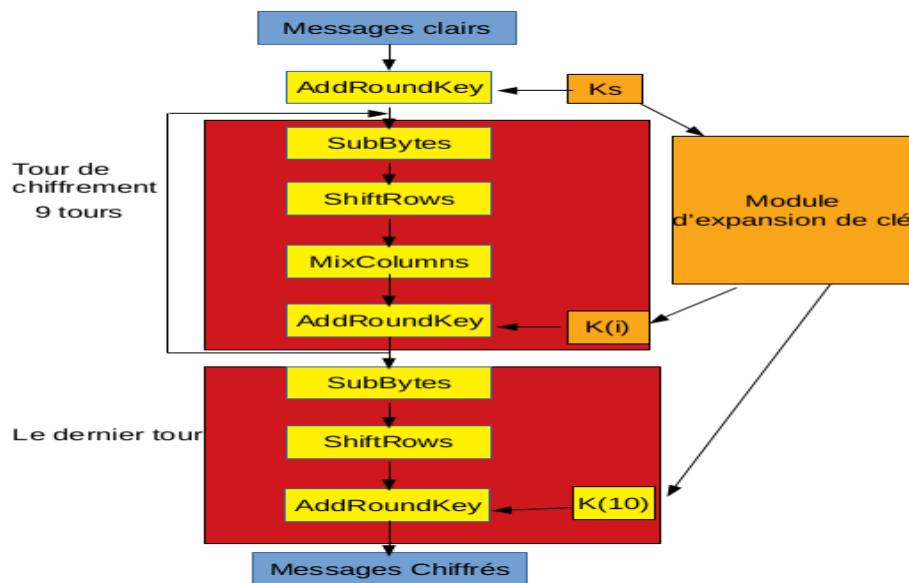


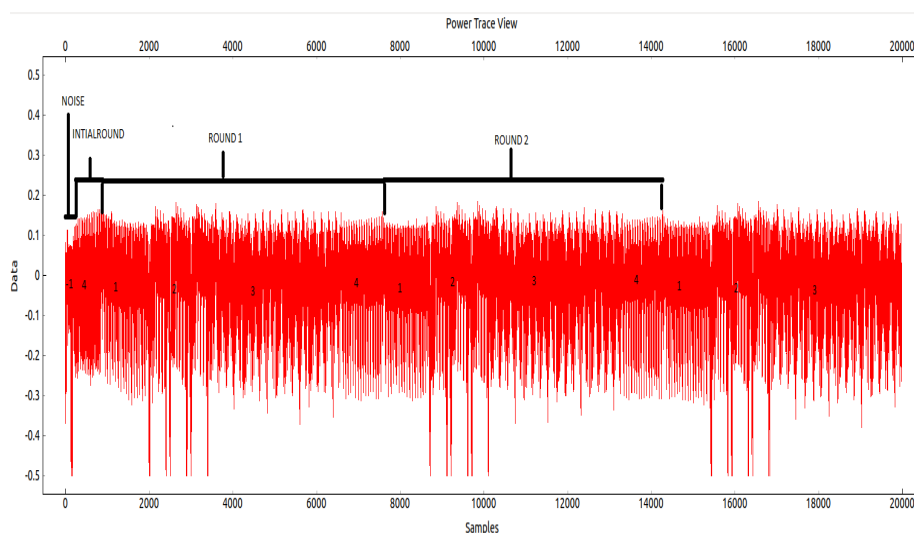
Figure 4: Principe de chiffrement AES 128 bits

3.2 Identifier des tours de AES et les différentes parties de chaque retour

AES contient 10, 12 ou 14 tours respectivement pour une clé de 128, 196 ou 256. On ne s'intéresse qu'à AES 128 ici, les premiers 9 tours contiennent 4 parties:

- Substitution d'un octet (**SubBytes**)
- Décalages des octets (**ShiftRows**)
- Le produit matriciel (**MixColumns**)
- Un XOR est appliqué entre chacun des octets de l'état et de la clé de ronde. (**AddRoundKey**)

On s'intéresse à identifier les tours et les différentes parties de chaque tour sur le schéma de consommation de puissance. On réussit à produire le schéma suivant, en contenant 2 tours:

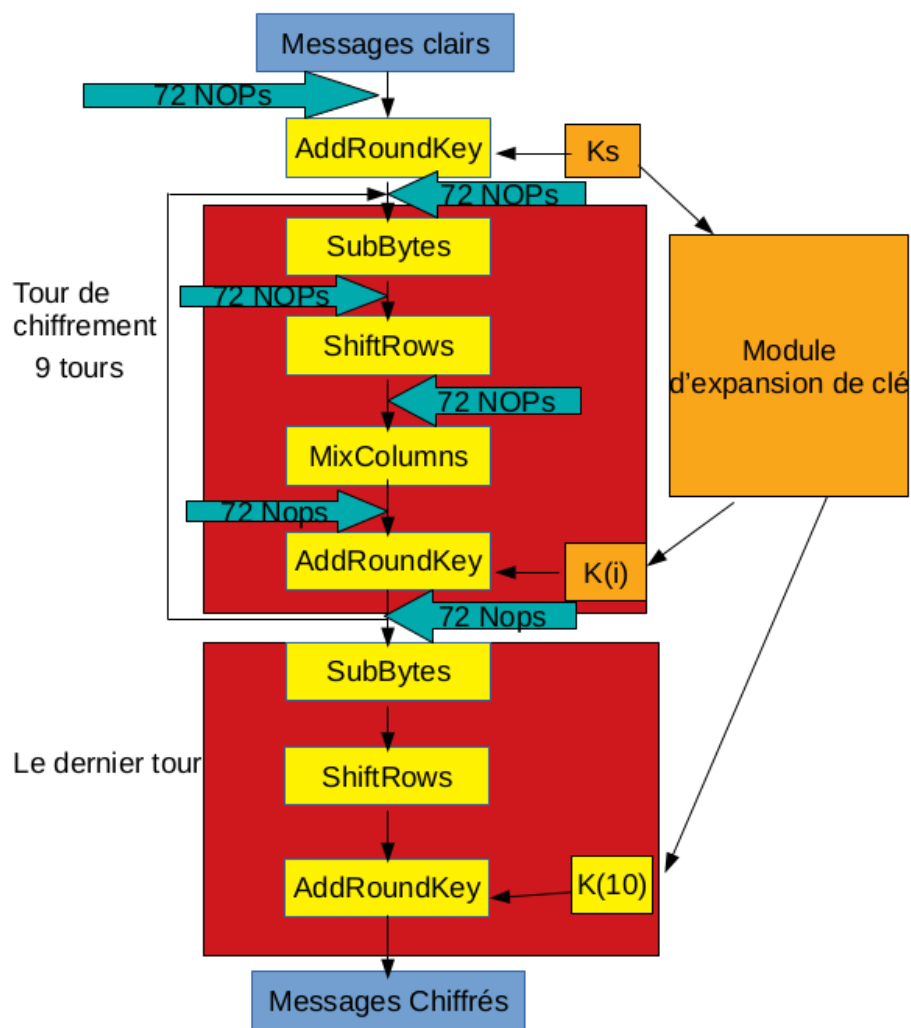


On tente d'identifier chaque partie du tour (-1,1,2,3,4 représentant la partie marquée sur le schéma)

- -1= **Bruit**
- 1 = **SubBytes**
- 2 = **ShiftRows**
- 3 = **MixColumns**
- 4 = **AddRoundKey**

3.3 Confirmer les hypothèses de partition de AES en utilisant l'opération NOP

On ouvre le fichier "aes_enc.c" situé dans "/chipwhisperer/hardware/victims/firmware/crypto/avcryptolib/aes/" pour ajouter les opérations NOP. Le fichier "aes_enc.c" contient les fonctions principales du chiffrement AES. Voici les positions où on a ajouté des NOPs pour confirmer notre hypothèse.

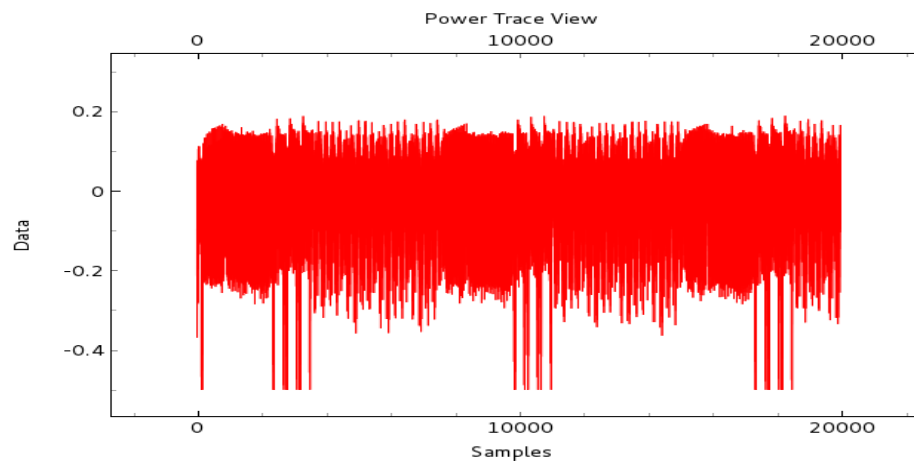


On a décidé d'utiliser 72 NOPs pour différencier chaque composants du tour (les 9 premiers tours, on s'intéresse particulièrement au premier tour). La version du code est en annexe A.

On veut voir clairement les composants des tours. On a décidé de faire une

comparaison entre le schéma de consommation de puissance sans NOPs et le schéma de consommation de puissance avec NOPs.

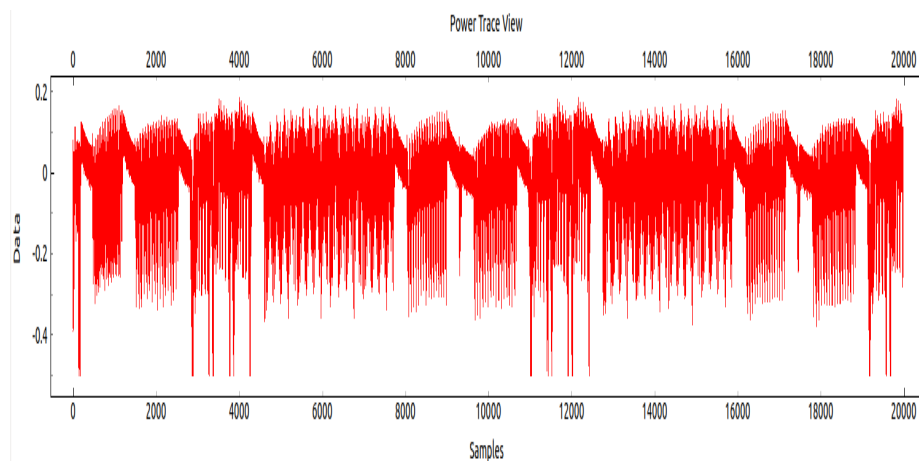
Voici le schéma sans les NOPs.



On a vraiment eu du mal à distinguer les tours et les étapes de chaque tour.

Après le relancement du chipwhisperer, on peut remarquer que chaque partie du tour sont clairement séparées sur des images obtenues à l'aide des opérations des NOPs.

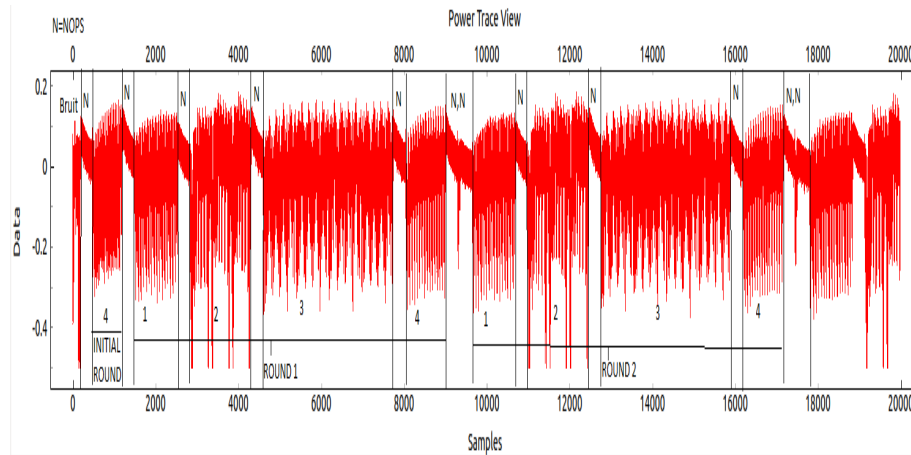
Voici le schéma obtenu:



On a comparé ces deux schémas obtenus, on a clairement vu les composants

des tours.

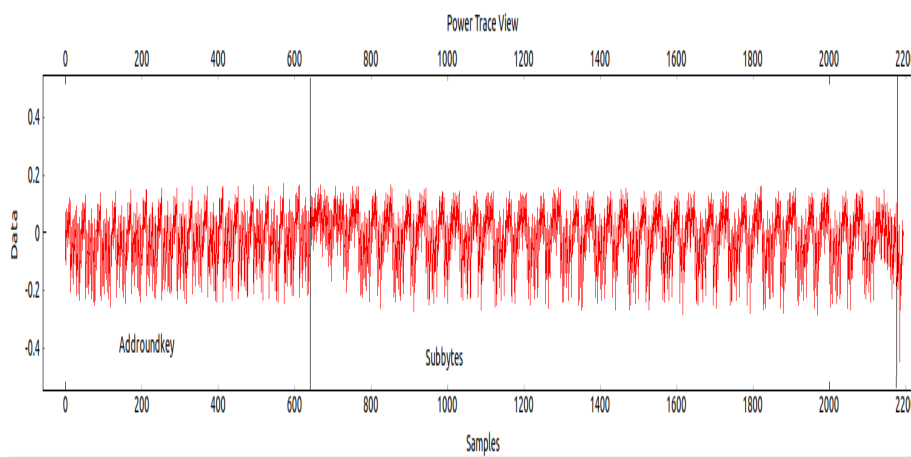
Voici les identifications des composants des tours sur le schéma obtenu:



Ce schéma vérifie bien notre hypothèse, On remarque aussi le bruit commence à 0 et se termine à 182. (N,N) fait la distinction entre les tours.

3.4 Identifier la commande minimale pour obtenir le début du premier tour de l'AES

On prend 182 offset pour enlever le bruit et on choisit 2200 échantillons correspondant au début du premier tour. Enfin, on doit changer le paramètre dans notre script: ['OpenADC', 'Trigger Setup', 'Total Samples', 2200]. Les modifications de paramètre sont en annexe B. On réussit à obtenir le début du premier tour de l'AES. Voici le schéma obtenu:

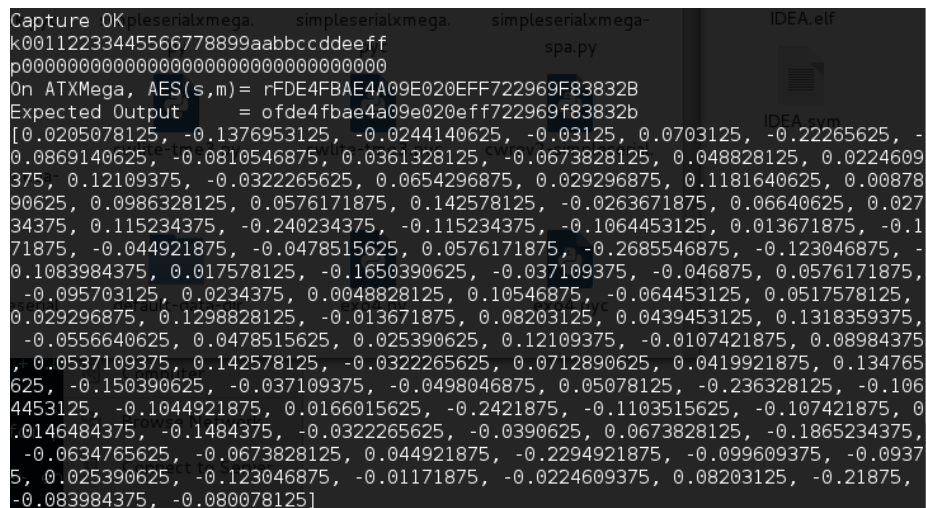


Le schéma correspond au tour initial (Addroundkey) et "Subbytes" du premier tour.

4 Exercice 3

4.1 Modification du programme pour enregistrer pleins de traces, correspondant au message aléatoire et vérifier si chaque chiffrement sur XMEGA sont égaux au résultat obtenu en utilisant la bibliothèque Crypto.Cipher

On a lancé le script python, on obtient le schéma suivant:



```
Capture OK simpleserialxmega      simpleserialxmega-      simpleserialxmega-      IDEA.elf
k00112233445566778899aabbccddeeff      spa.py
p00000000000000000000000000000000
On ATXMega, AES(s,m)= rFDE4FBAE4A09E020EFF722969F83832B
Expected Output      = ofde4fbae4a09e020eff722969f83832b
[0.0205078125, -0.1376953125, -0.0244140625, -0.03125, 0.0703125, -0.22265625, -
0.0869140625, -0.0810546875, 0.0361328125, -0.0673828125, 0.048828125, 0.0224609
375, 0.12109375, -0.0322265625, 0.0654296875, 0.029296875, 0.1181640625, 0.00878
90625, 0.0986328125, 0.0576171875, 0.142578125, -0.0263671875, 0.06640625, 0.027
34375, 0.115234375, -0.240234375, -0.115234375, -0.1064453125, 0.013671875, -0.1
71875, -0.044921875, -0.0478515625, 0.0576171875, -0.2685546875, -0.123046875, -
0.1083984375, 0.017578125, -0.1650390625, -0.037109375, -0.046875, 0.0576171875,
-0.095703125, 0.0234375, 0.0048828125, 0.10546875, -0.064453125, 0.0517578125, -
0.029296875, 0.1298828125, -0.013671875, 0.08203125, 0.0439453125, 0.1318359375,
-0.0556640625, 0.0478515625, 0.025390625, 0.12109375, -0.0107421875, 0.08984375,
-0.0537109375, 0.142578125, -0.0322265625, 0.0712890625, 0.0419921875, 0.134765
625, -0.150390625, -0.037109375, -0.0498046875, 0.05078125, -0.236328125, -0.106
4453125, -0.1044921875, 0.0166015625, -0.2421875, -0.1103515625, -0.107421875, 0
.0146484375, -0.1484375, -0.0322265625, -0.0390625, 0.0673828125, -0.1865234375,
-0.0634765625, -0.0673828125, 0.044921875, -0.2294921875, -0.099609375, -0.0937
5, 0.025390625, -0.123046875, -0.01171875, -0.0224609375, 0.08203125, -0.21875,
-0.083984375, -0.080078125]
```

On remarque qu'on obtient le même chiffrement d'AES sur XMEGA que le résultat obtenu en utilisant la bibliothèque Crypto.Cipher pour la clé "00112233445566778899aabbccddeeff" et le message clair "00000000000000000000000000000000".

On modifie le code de la manière suivante pour avoir des messages aléatoires:

```
1      # Create the null message
2      for i in range(0, 16):
3          # myaes_input [i] = 0
4          # Create random message
5          myaes_input [i] = random.randint(0,255)
```

On a aussi bien vérifié pour un message aléatoire entré, on a toujours obtenu le même résultat sur XMEGA et la fonction utilisant la bibliothèque Crypto.Cipher.

4.2 Le résultat obtenu sur un octet par la sortie de la première sbox

$$S_out = SubBytes(input \oplus sub_key)$$

input: un block de message de 8 bits

sub_key: un block de clé de 8 bits correspond à la position de *input*

s_out: Le résultat obtenu sur un octet par la sortie de Sbox

les détails de Sbox, SubBytes sont en annexe C.

On choisit un octet xore avec une sous-clé, on fait ensuite la substitution dans sbox, la fonction `sbox_xor` retourne la valeur à la sortie de premier sbox.

4.3 L'attaque par DPA et hamming model

4.3.1 DPA: Differential Power Analysis

Paul Kocher, Joshua Jaffe et Benjamin Jun ont publié une nouvelle attaque sur l'analyse différentielle de consommation[6]. Voici les étapes de cette attaque.

- On assume le modèle de fuite est le modèle de hamming.
- On choisit une étape de l'algorithme à attaquer:

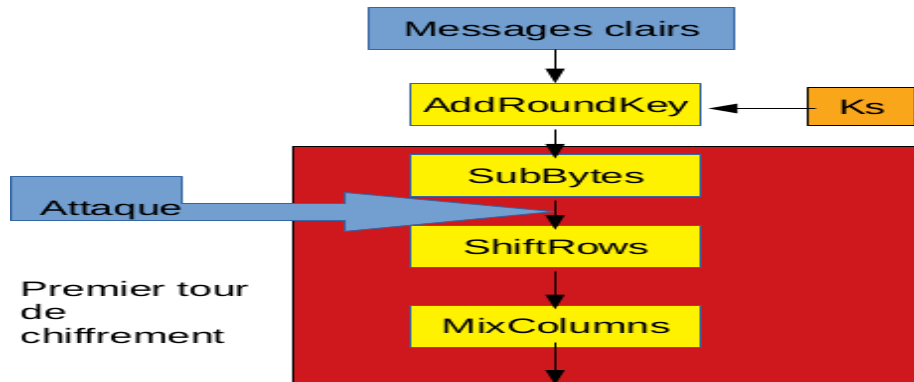


Figure 5: Instant d'attaque

- Mesurer plusieurs traces avec l'algorithme en utilisant des entrées connues et la clé secrète. On suppose qu'on ne connaît pas la clé secrète "00112233445566778899aabbccddeeff". Notre but est de la trouver.
- On suppose qu'on attaque le premier octet de clé *Ks*, le premier octet du message *i* est écrit *input_i*. L'hypothèse de clé *sub_key* avec *sub_key* ∈ [0, 255].

$$G_0[i, sub_key] = \{i | Sbox[input_i \oplus sub_key] = 0 \bmod 2\}$$

$$G_1[i, sub_key] = \{i | Sbox[input_i \oplus sub_key] = 1 \bmod 2\}$$

Pour chaque *sub_key* différent, on regroupe des traces selon la valeur de la position du bit qu'on a choisit par exemple le premier bit du premier octet.

- Pour chaque *sub_key*, on calcule la moyenne de $G_0[i, sub_key]$ et la moyenne de $G_1[i, sub_key]$

$$moyenne(G, sub_key) = \frac{1}{|G[sub_key]|} \sum_{i \in G[i, sub_key]} W_i(t)$$

- Pour chaque *sub_key*, on calcule la différence entre $G_0[i, sub_key]$ et $G_1[i, sub_key]$

$$difference(sub_key) = |moyenne(G_0, sub_key) - moyenne(G_1, sub_key)|$$

- On trouve le plus grand pic parmi tous les échantillons, le *sub_key* associé avec la trace correspond au plus grand pic est une bonne sous clé.

$$real_sub_key = argmax_{sub_key \in [0, 255]} difference(sub_key)$$

- On utilise la même méthode pour attaquer le deuxième octet de la clé secrète, on itère la boucle jusqu'à trouver toute la clé Ks.

Voici une image illustrant les étapes principales qu'on a décrit précédemment:

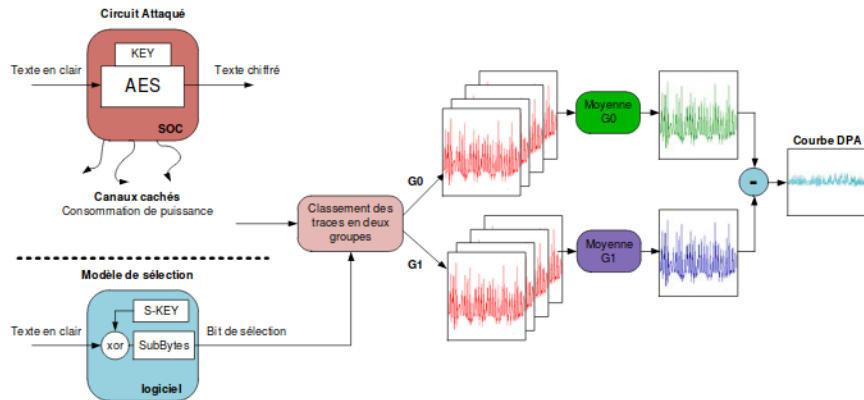


Figure 6: Principe de l'attaque DPA [3]

Voici un algorithme qu'on a implémenté.

Algorithm 5 Trouve la clé secrète par une attaque de DPA

Entrées: Traces collectées

Sorties: La clé secrète

```
sub_0={}
sub_1={}
for k in range(0,16) do
  for keyguess in range(0,256) do
    for i in range(0,nbtraces) do
      if firstsignificantBit(sbox_xor(plain_text[i][k],keyguess))==1 then
        Ajoute la trace correspondant à plain_text[i][k] dans sub_1
      else if firstsignificantBit(sbox_xor(plain_text[i][k],keyguess))==0
      then
        Ajoute la trace correspondant à plain_text[i][k] dans sub_0
      end if
    end for
    sub_0.calculeramoyenne(sub_0)
    sub_1.calculeramoyenne(sub_1)
    a[k]=max(calculatetopointdifference(sub_0,sub_1))
  end for
  Trouve max(a[k]), Identifie la bonne sous clé correspondant à cette valeur
end for
Trouve la clé entière
```

On a déclaré l'attaque au dernier octet de la clé secrète, on a obtenu le schéma suivant:

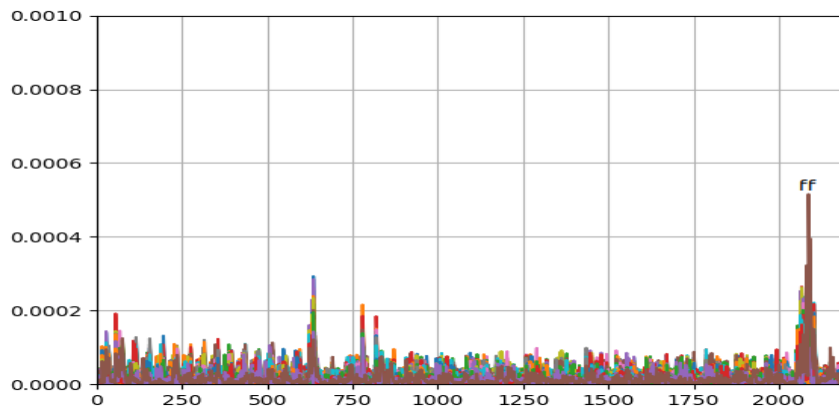


Figure 7: Diffence of mean attaque sur une sous clé

On remarque que le plus grand pic signifie que la sous clé devinée est correcte. Voici l'image obtenu lorsqu'on a attaqué la clé entière:

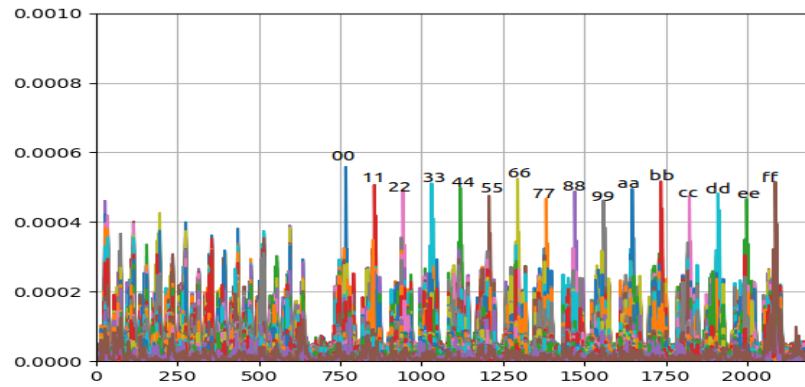
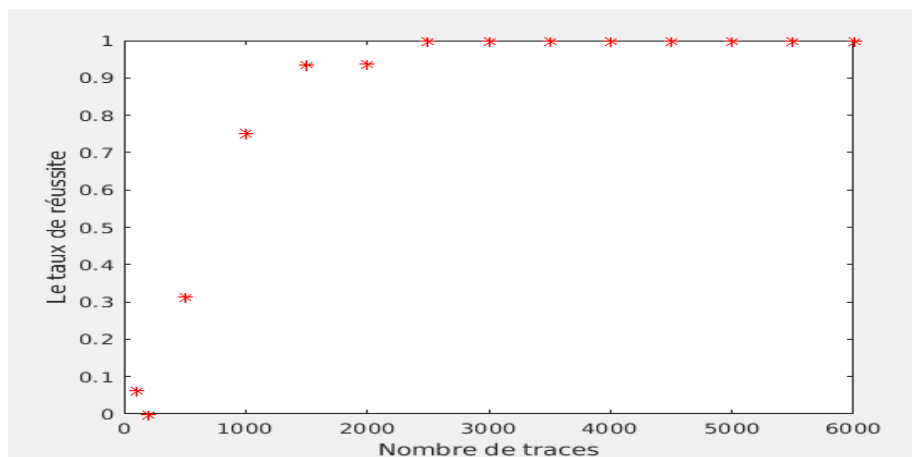


Figure 8: Diffence of mean attaque sur la clé entière

On observe que les grands pics correspondent bien à la sous clé secrète sur l'image. Voici l'affichage de la clé secrète:

```
petx@light-aspire-s5-371t:~/chipwhisperer$ python dpa.py
['\x0', '\x11', '\x22', '\x33', '\x44', '\x55', '\x66', '\x77', '\x88', '\x99',
'\xaa', '\xbb', '\xcc', '\xdd', '\xee', '\xff']
```

On a bien vérifié la clé secrète est la bonne clé secrète. On s'intéresse ensuite à la relation entre le taux de réussite à trouver la bonne clé et le nombre de traces collectées. On a donc fait les attaques en fonction du nombre de traces utilisées. Voici les résultats qu'on a obtenu:



On remarque que si on a plus de messages clairs, on a plus de chances de trouver la clé secrète. La clé secrète trouvée est fiable à partir d'un certain nombre de messages clairs.

4.3.2 Hamming weight power model

Hamming weight power model est pour produire le modèle de la consommation électrique de la devise sous une attaque. Pendant une attaque, notre but est de trouver une corrélation entre la consommation électrique modelisée et la consommation électrique actuelle.

Hamming weight power model est le nombre de 1 dans un nombre binaire. Par exemple:

$$\text{Hammingweight}(1001001)=3$$

4.3.3 La théorie de l'attaque CPA (correlation power analysis)

CPA est une attaque nous permettant de trouver la clé secrète. Voici les 4 étapes principales pour l'attaque CPA:

1. Choisir un modèle pour la consommation électrique. On utilise Hamming weight power model dans cet exemple.
2. Encrypter beaucoup de messages clairs différents. Enregistrer les traces de la consommation électrique pendant chaque chiffrement.
3. Attaquer une petite partie (sous clé) de la clé secrète :
 - (a) Essayer toutes les valeurs possibles pour la sous clé. Pour chaque valeur de sous clé et chaque trace, on utilise des messages clairs et la sous clé devinée pour calculer la consommation électrique selon notre modèle.
 - (b) Calculer "the Pearson correlation coefficient" entre la consommation électrique modelisée et la consommation électrique actuelle. Fait la même chose pour tous les points de la trace.
 - (c) Décider quelle valeur de sous clé est la mieux corrélée.
4. Assembler toutes les sous clés les mieux corrélées.

Voici une image qui illustre ce qu'on a décrit précédemment:

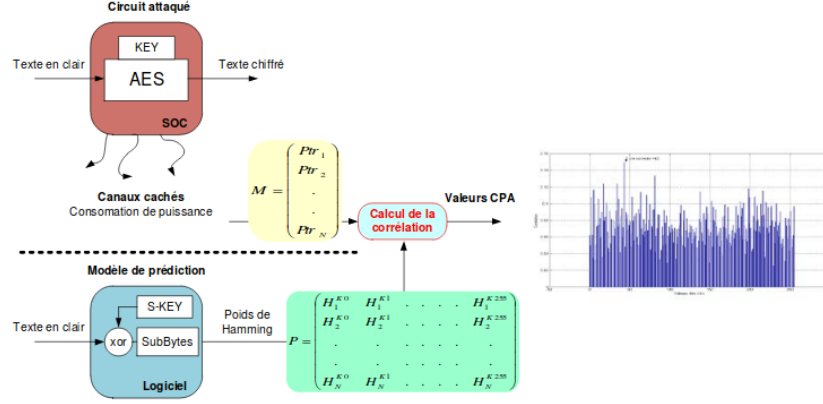


Figure 9: Principe de l'attaque par corrélation, CPA [3]

4.4 Lire le fichier CSV

L'algorithme pour lire le fichier CSV est en annexe D. Après les testes qu'on a réalisé, l'algorithme lit bien les fichiers.

4.5 Pearson correlation coefficient

Après avoir lu le fichier dans lequel les mesures prises sont, on a donc nbtrace de traces et chaque trace à T points. On utilise $t_traces_{tnum,j}$ pour présenter j-ième point de la tnum-ième trace. ($0 \leq tnum \leq nbtrace-1, 0 \leq j < T$) k est la sous clé possible qu'on va tester ($0 \leq k < 256$), $hyp_{tnum,k}$ est tnum-ième trace et l'hypothèse de sous clé est k.

$$cpaoutput_{k,j} = \frac{\sum_{tnum=0}^{nbtrace-1} [(hyp_{tnum,k} - hmean_k)(t_traces_{tnum,j} - tmean_j)]}{\sqrt{\sum_{tnum=0}^{nbtrace-1} (hyp_{tnum,k} - hmean_k)^2 \sum_{tnum=0}^{nbtrace-1} (t_traces_{tnum,j} - tmean_j)^2}}$$

on trouve le plus grand de $|cpaoutput_{k,j}|$ et on le stocke dans maxcpa.

$$maxcpa_k = \max(|cpaoutput_{k,j}|)$$

L'algorithme est placé en annexe E.

4.6 Appliquer la fonction correlation coefficient sur le premier octet des hypothèses

Maintenant, on va appliquer la fonction correlation coefficient pour tester tout le premier octet des hypothèses de clé. Et puis il faut sauvegarder les scores

dans le fichier score.csv. D'abord, on initialise les tableaux pour enregistrer les variables, ce qu'on doit utiliser: hw (Hamming Weight), maxcpa (le plus grand score), keyguess (toutes les clés devinées), tmean (la moyenne des traces), hyp (Hamming Weight total).

```

1  hw=(unsigned char *) malloc(256*sizeof(unsigned char));
2  maxcpa=(float *) malloc(256*sizeof(float));
3  keyguess=(unsigned char *) malloc(256*sizeof(unsigned char));
4
5  //tmean
6  tmean=(float *) malloc(size_trace*sizeof(float));
7  for(i=0;i<size_trace;i++){tmean[i]=0;}
8
9  //stocker hamming weight 1000*16*256
10 hyp=(unsigned char ***) malloc(size_line*sizeof(unsigned char **))
11 );
12 for(i=0;i<size_line;i++){
13     hyp[i]=(unsigned char **) malloc(size_plain*sizeof(unsigned
14     char*));
15     for(j=0;j<size_plain;j++){
16         hyp[i][j]=(unsigned char *) malloc(256*sizeof(unsigned char))
17     };
18 }

```

Ensuite, il faut appliquer la fonction correlation coefficient sur tous les messages clairs, pour trouver le score de chaque octet et les sauvegarder dans le tableau maxcpa. Enfin, on sauvegarde les scores dans le fichier scores.csv.

```

1  fprintf(fp,"key's position \t keyguess \t maxabscpa\n");
2  for(bnum=0;bnum<size_plain;bnum++){
3      for(k=0;k<256;k++){
4          hmean=0;
5          cpaoutput=0;
6          maxcpa[k]=0;
7          keyguess[k]=k;
8          //hw,calcul haming model,hamming weight
9          for(tnum=0;tnum<size_line;tnum++){
10             hyp[tnum][bnum][k]=HW(sbox_xor(t_plaintexts[tnum][bnum],
11             keyguess[k]));
12             hmean+=hyp[tnum][bnum][k];
13         }
14         //calculer hmean
15         hmean=(float)hmean/size_line;
16         //calculer tmean
17         for(j=0;j<size_trace;j++){
18             for(i=0;i<size_line;i++){
19                 tmean[j]+=t_traces[i][j];
20             }
21         }
22         for(j=0;j<size_trace;j++){
23             tmean[j]=(float)tmean[j]/size_line;
24         }
25         //get max abs j (sample'number)
26         maxcpa[k]=correlationCoefficient(hyp,t_traces,hmean,tmean,
27         ,bnum,k,size_trace,size_line);

```

```

26     fprintf(fp, "%d \t %d \t %f \n", bnum, k, maxcpa[k]);
27 }
28

```

Voici une partie du fichier scores.csv:

key's position	keyguess	maxabscpa
0	0	0.277493
0	1	0.24246
⋮	⋮	⋮
15	248	0.186106
15	249	0.173853
15	250	0.204515
15	251	0.164856
15	252	0.1703
15	253	0.162426
15	254	0.673835
15	255	0.734086

On remarque que keyguess est le bon caractère du mot de passe si sa valeur maxaabsca associée est le plus grand. Par exemple, 255 est le bon caractère pour le dernier octet du mot de passe.

4.7 Finaliser l'attaque CPA

On fait la comparaison entre les scores dans le tableau maxcpa pour trouver le plus grand score, on sauvegarde cet indice du plus grand score dans un tableau bestguess.

```

1     cpaoutput=-100;
2     //get max i(size_line:traces' number)
3     for(i=0;i<256;i++){
4         if(cpaoutput < maxcpa[i]){
5             cpaoutput=maxcpa[i];
6             bestguess[bnum]=i;
7         }
8     }
9 }
10

```

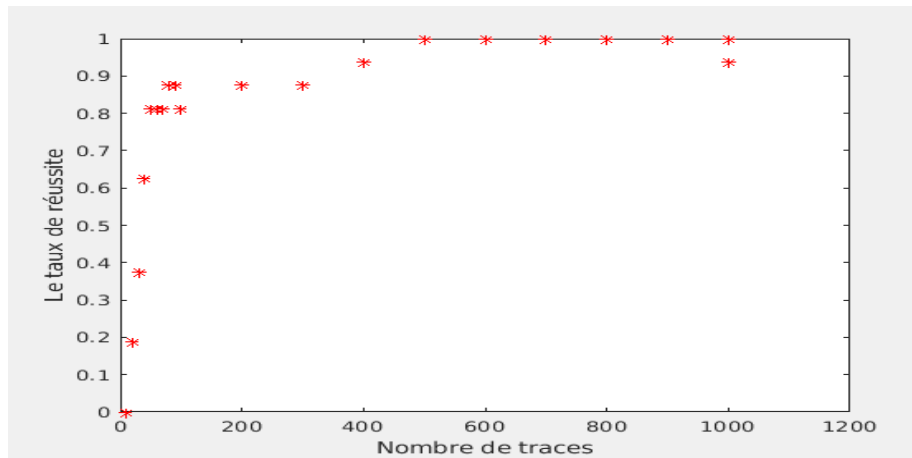
On répète ce processus 16 fois, on obtient la clé secrète suivante:

```

petx@light-aspire-s5-371t:~/chipwhisperer$ gcc cpa.c -lm
petx@light-aspire-s5-371t:~/chipwhisperer$ ./a.out
0 11 22 33 44 55 66 77 88 99 aa bb cc dd ee ff

```

En effet, on n'a pas utilisé le même data que celui qu'on utilise pour DPA. On a fait 10 traces et calculé leur moyenne, puis on a stocké la moyenne dans le fichier, on a répété plusieurs fois cette étape.



On remarque que si on a plus de messages clairs (plus de traces), on a plus de chances de trouver la clé secrète. La clé secrète trouvée est fiable à partir d'un certain nombre de messages clairs. On n'a pas trouvé la bonne clé secrète à 1000 traces, on pense que c'est à cause du fait d'avoir mal collecté les datas et on n'a pas encore assez de traces pour stabiliser le resultat. On peut par exemple calculer la moyenne de 1000 traces à la place de la moyenne de 10 traces.

5 Exercice 4

5.1 AES128-RSM

5.1.1 RSM: Rotation Sboxes masking

Le Rotation SBoxes masking a été inventé en 2012 par Nassar et al [8]. Cette méthode utilise un masque ne contenant que 16 valeurs fixes de 8 bits:

mask= 0x03, 0x0c, 0x35, 0x3a, 0x50, 0x5f, 0x66, 0x69, 0x96, 0x99, 0xa0, 0xaf,
0xc5, 0xca, 0xf3, 0xfc

Où $mask_i$ présente i-ième valeur de masque($i=0,1,...,14,15$)

Offset est un entier aléatoire qui est le premier indice du masque utilisé. Celui-ci a été tiré en début du chiffrement. On génère des nouveaux masque à la manière de Masquage(Figure 10):

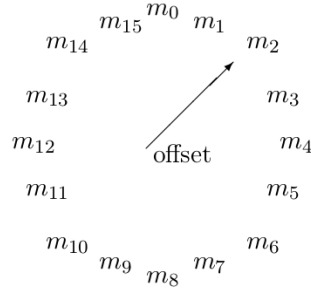


Figure 10: Masquage [9]

$mask[offset]$ est la première valeur du bloc de masque et l'offset est ensuite incrémenté modulo 16. On génère $mask_offset$ de la manière suivante:

$mask_offset =$
 $mask_{(offset+0x0)mod16}, mask_{(offset+0x1)mod16}, mask_{(offset+0x2)mod16}, mask_{(offset+0x3)mod16}$
 $mask_{(offset+0x4)mod16}, mask_{(offset+0x5)mod16}, mask_{(offset+0x6)mod16}, mask_{(offset+0x7)mod16},$
 $mask_{(offset+0x8)mod16}, mask_{(offset+0x9)mod16}, mask_{(offset+0xa)mod16}, mask_{(offset+0xb)mod16},$
 $mask_{(offset+0xc)mod16}, mask_{(offset+0xd)mod16}, mask_{(offset+0xe)mod16}, mask_{(offset+0xf)mod16}$

Voici quelques termes qu'on utilisera:

- $offsets[16]$
 Pour éviter qu'un seul offset ne soit pas trop facile à retrouver, on génère donc un tableau d'offset, c'est à dire que 16 offsets ont été générés au début du chiffrement. Chaque masque procède à son propre offset, on génère ensuite $mask_offset[...][...]$ par la manière présentée au-dessus.
- $Shuffle(shuffle0, shuffle10)$
 Shuffle n'est utilisé que pour protéger SubByte du premier tour et SubByte du dernier tour pour éviter d'attaquer au premier et au dernier tour. Shuffle ne change que l'ordre des opérations. Shuffle est permuté aléatoirement au début du chiffrement:

$$shuffle0, shuffle10 : \{0, 1, \dots, 15\} \longrightarrow \{0, 1, \dots, 15\}$$

- MSB: SB est le SubBytes original, MSB est le SubBytes masqué et MSB est défini de la manière suivante:

$$MSB_i(X) = SB(X \oplus mask_offset[r-1][...]) \oplus mask_offset[r][...]$$

avec r est le numéro de tour, [...] présente $[0,1...,15]$

Voici une image illustrant MSB:

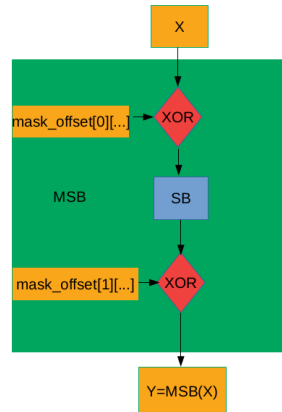


Figure 11: MSB de 1er tour: SubByted masqué de 1er tour

- Chiffrement du premier tour (voir Figure 12)

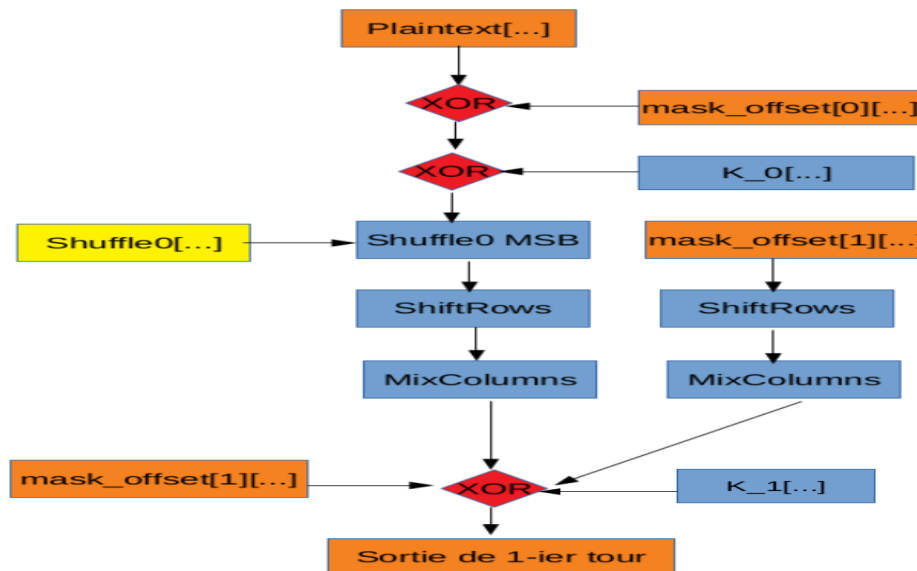


Figure 12: Chiffrement du 1er tour

- Le chiffrement du premier tour sans les masques
 - On xore plaintext[...] avec $K_0[...]$ appelé a
 $Plaintext[...] \oplus K_0[...] \rightarrow a$
 - MixColumns(ShiftRows(SubBytes(a))) appelé b
 $MixColumns(ShiftRows(SubBytes(Plaintext[...] \oplus K_0[...])) \rightarrow b$
 - b xore avec K_1
- Le chiffrement du premier tour avec les masques
 - On xore le plaintext[...] avec le $mask_offset[0][...]$ et $K_0[...]$ appelé A
 $mask_offset[0][...] \oplus Plaintext[...] \oplus K_0[...] \rightarrow A$
 - MixColumns(ShiftRows(MSB(A))) appelé B
 $MixColumns(ShiftRows(SubBytes(K_0[...] \oplus Plaintext[...]))$
 \oplus
 $MixColumns(ShiftRows(SubBytesB(mask_offset[1][...])) \rightarrow (B)$
 car MixColumns et ShiftRows sont linéaires.
 - MixColumns(ShiftRows($mask_offset[1][...]$)) appelé C
 $MixColumns(ShiftRows(mask_offset[1][...])) \rightarrow C$
 - Démasque: C xore B appelé D
 $MixColumns(ShiftRows(SubBytes(Plaintext[...] \oplus K_0[...])) \rightarrow D$
 - Remasque: D xore avec $mask_offset[1][...]$ et avec K_1

Cela prouve que les masques ne changent pas le résultat du chiffrement.

5.1.2 Pseudo Code

Algorithm 6 AES128-RSM

Entrées: Messages clairs sur 16 octets S_0, S_1, \dots, S_{15} et 11 sous-clés de 16 octets (K_0, \dots, K_{10}) obtenus par l'algorithme de key Schedule
Sorties: les messages chiffrés sur 16 octets s_0, s_1, \dots, s_{15}

```
/* Définition des masques */
mask = 0x03, 0x0c, 0x35, 0x3a, 0x50, 0x5f, 0x66, 0x69, 0x96, 0x99, 0xa0,
0xaf, 0xc5, 0xca, 0xf3, 0xfc
/* Génération de 16 offsets aléatoires sur 4 bits */
offset[0] ← rand() mod 16
...
offset[15] ← rand() mod 16
/*Génération aléatoire de 2 permutations shuffle0 et shuffle10 */
shuffle0[0...15] ← Permute([0...15])
shuffle10[0...15] ← Permute([0...15])
/*Remplir mask_offset */
for i ∈ [0, 15] do
    mask_offset[i][0]=mask[(offset[i]+[0]) mod 16]
    ...
    mask_offset[i][15]=mask[(offset[i]+[15]) mod 16]
end for
S[0...15] ← S[0...15] ⊕ mask_offset[0][0...15]
/*Tour 0 */
S ← S ⊕ K0
/*Tour 1...9*/
for r ∈ [1, 9] do
    if r == 1 then
        tmp[shuffle0[0..15]] ← SubBytes(S[shuffle0[0..15]] ⊕ mask_offset[r-1][shuffle0[0..15]]) ⊕ mask_offset[r][shuffle0[0..15]]
    else
        tmp[0..15] = SubBytes(S[0..15] ⊕ mask_offset[r-1][0..15]) ⊕ mask_offset[r][0..15]
    end if
    tmp ← Shiftrows(tmp)
    S ← MixColumns(tmp)
    mask_compensation[0...15] ← MixColumns(Shiftrows(mask_offset[r][0..15])) ⊕ mask_offset[r][0..15]
    S[0...15] ← S[0...15] ⊕ mask_compensation[0...15]
    S[0...15] ← S[0...15] ⊕ Kr[0...15]
end for
/*Le dernier tour */
tmp[shuffle10[0..15]] ← SubBytes(S[shuffle10[0..15]] ⊕ mask_offset[10][shuffle10[0..15]]) ⊕ mask_offset[10][shuffle10[0..15]]
S ← Shiftrows(S)
mask_compensation[0...15] ← Shiftrows(mask_offset[10][0..15])
/*Démasquer */
S[0...15] ← S[0...15] ⊕ mask_compensation[0...15]
S[0...15] ← S[0...15] ⊕ K10[0...15]
```

5.2 Évaluer cette contre-mesure

Avant de commencer l'évaluation de cette contre-mesure, on teste si on peut obtenir le bon résultat avec la protection.

Voici un exemple obtenu:

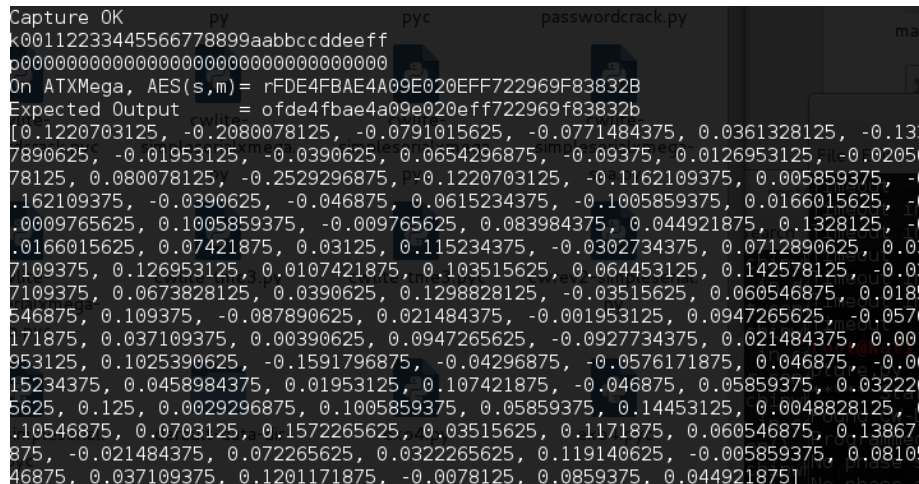


Figure 13: Le résultat après ajouter la protection utilisant la méthode RSM

On observe qu'on réussit à obtenir toujours les bons résultats. On conclue que la protection ne modifie pas le résultat du chiffrement. Il nous reste à tester si le protection résiste aux attaques (DPA, CPA) que nous avons implémenté précédemment.

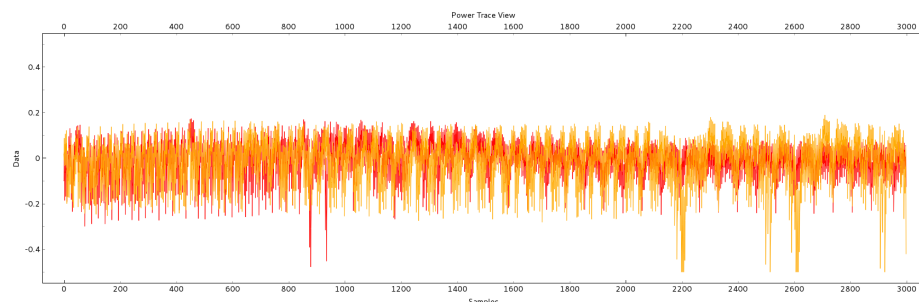


Figure 14: Comparaison entre sans protection et avec protection.

rouge: le consommation de puissance sans le protection

orange:le consommation de puissance avec le protection

On observe qu'on arrive pas à trouver des instants interessant,et le bruit joue un rôle très important. RAPELLE:0x00', '0x11', '0x22', '0x33', '0x44', '0x55',

'0x66', '0x77', '0x88', '0x99', '0xaa', '0xbb', '0xcc', '0xdd', '0xee', '0xff' est la clef correcte.

On décide de collecter 5000 traces et chaque trace est la moyenne de 10 traces. Voici des résultats obtenus:

- 200 traces
'0xda', '0x55', '0xe4', '0x6d', '0x30', '0x5b', '0x43', '0x51', '0xcd', '0x14',
'0xab', '0x81', '0xb6', '0x71', '0x7b', '0xfc'
- 500 traces
'0x88', '0x52', '0xe4', '0x4d', '0xd3', '0x89', '0xf0', '0x15', '0xe5', '0x7a',
'0xb3', '0x6d', '0x2c', '0x6e', '0x29', '0xfc'
- 800 traces
'0x92', '0xd5', '0x84', '0x3e', '0x29', '0x4f', '0x4b', '0x4f', '0x22', '0xcc',
'0x59', '0x6d', '0x6a', '0x6e', '0x29', '0x1d'
- 1000 traces
'0x17', '0xe3', '0x6f', '0x3e', '0xf1', '0x34', '0x93', '0x4f', '0x48', '0xaa',
'0xa2', '0x6d', '0x6a', '0x8c', '0x29', '0x7'
- 1500 traces
'0x12', '0x1d', '0xc6', '0xcb', '0xf1', '0x93', '0xb5', '0xed', '0x48', '0x68',
'0x48', '0xd9', '0x31', '0x8f', '0xb9', '0x87'
- 1800 traces
'0x12', '0x1d', '0x22', '0xdb', '0x3f', '0x93', '0xb5', '0xed', '0x82', '0xc4',
'0xa2', '0xd9', '0xd2', '0x8f', '0x4d', '0x87'
- 2000 traces
'0x12', '0x1d', '0x22', '0xdb', '0x3f', '0x93', '0xb5', '0xff', '0xd9', '0x6a',
'0x82', '0xd9', '0xe7', '0x8f', '0xbc', '0x87'
- 2500 traces
'0x12', '0x1d', '0xaa', '0xe7', '0x0', '0x8c', '0x77', '0x64', '0xaf', '0x6a',
'0xd4', '0xf2', '0x50', '0x8f', '0xe0', '0xd0'
- 3000 traces
'0xe', '0x43', '0x9', '0x67', '0xca', '0xe0', '0x77', '0xfb', '0x14', '0xbf',
'0x88', '0xd1', '0x5b', '0x62', '0xe0', '0x20'
- 3500 traces
'0xe', '0x8b', '0x9', '0x67', '0x29', '0xf0', '0x3b', '0x9b', '0x14', '0xbf',
'0x98', '0xf2', '0x5b', '0x62', '0xe0', '0x20'
- 4000 traces
'0x12', '0x8b', '0xe9', '0xdb', '0x0', '0x79', '0x5e', '0x67', '0x14', '0xbf',
'0x98', '0xb8', '0x84', '0x8f', '0xe0', '0xd0'

- 4500 traces
'0xe', '0xc0', '0x5a', '0xcb', '0x0', '0x62', '0x5e', '0xbf', '0xde', '0xbf',
'0x98', '0x21', '0x7f', '0x62', '0xe0', '0xd0'
- 5000 traces
'0xe', '0x8b', '0x5a', '0xcb', '0x0', '0xc0', '0x43', '0xc5', '0xb4', '0xbf',
'0x98', '0x21', '0x1b', '0x62', '0xe0', '0xd0'

On observe qu'on trouve aucune bonne sous-clé, RSM résiste bien à l'attaque CPA qu'on a implémenté dans la première partie.

5.3 Conclusion

RSM résiste aux attaques CPA grâce aux masques et surtout au shuffle. Les traitements étaient séquentiellement pour SubBytes. Shuffle rend l'ordre de leur traitement aléatoire. C'est à dire que lorsqu'on veut obtenir la consommation de puissance à la sortie de Sbox pour octet s_1 , on doit regarder à l'instant de x -ième avec $\text{shuffle0}[x]=s_1$ et shuffle0 est inconnu. Donc il est difficile de connaître la consommation de puissance à l'instant précis. En gros, il y a trop de bruit !

6 Exercice 5

6.1 IDEA

6.1.1 Principe

L'opération IDEA[4] est un chiffrement par bloc, avec le plaintext de 64 bits, elle est contrôlée par les clés de 128 bits.

Le principe de cet algorithme est d'utiliser les 3 groupes algébrique différents. Les boîtes de substitution et les recherches de table associées utilisées dans les chiffrements de bloc disponibles à ce jour ont été complètement évité.

La structure de l'algorithme a été choisie de telle sorte que, à l'exception de l'utilisation de sous-blocs de clés différentes, le processus de chiffrement est identique au processus de déchiffrement.

6.1.2 Implémentation d'IDEA pour XMEGA

1. Séparer le plaintext pour 4 parties P1, P2, P3, P4.
2. Donner P1, P2, P3, P4 comme les entrées du premier tour de l'algorithme
3. Après il obtient les 4 sorties P1', P2', P3', P4', il faut utiliser ces 4 sorties comme les entrées du deuxième tour. Totalemt, il fait 8 tours.
4. Dans le 9-ième tour, il utilise seulement les 4 dernières clés et il faut seulement Additionner et Multiplier avec ces 4 clés.

5. Les 4 sorties C1, C2, C3, C4 sont le résultat de chiffrement Multiplier(*) ne présente pas une simple multiplication, il s'agit d'une multiplication modulo $2^{16} + 1$ (i.e. multiplication modulo 65537) et Additionner(+) est une addition modulo 2^{16} (i.e. addition modulo 65536)

6.1.3 Tours du chiffrement[7]

Voir la figure 15.

1. Multiplier(*) P_1 et la première sous-clé K_1
2. Additionner(+) P_2 et la deuxième sous-clé K_2
3. Additionner(+) P_3 et la troisième sous-clé K_3
4. Multiplier(*) P_4 et la quatrième sous-clé K_4
5. XOR le résultat d'étape 1 et le résultat d'étape 3
6. XOR le résultat d'étape 2 et le résultat d'étape 4
7. Multiplier(*) le résultat d'étape 5 et la cinquième sous-clé K_5
8. Additionner(+) le résultat d'étape 6 et le résultat d'étape 7
9. Multiplier(*) le résultat de l'étape 8 par la sixième sous-clé K_6
10. Additionner(+) les résultats d'étape 7 et 9
11. XOR le résultat d'étape 1 et le résultat d'étape 9
12. XOR le résultat d'étape 3 et le résultat d'étape 9
13. XOR le résultat d'étape 2 et le résultat d'étape 10
14. XOR le résultat d'étape 4 et le résultat d'étape 10

Multiplier(*) ne présente pas une simple multiplication, il s'agit d'une multiplication modulo $2^{16} + 1$ (i.e. multiplication modulo 65537) et Additionner(+) est une addition modulo 2^{16} (i.e. addition modulo 65536)

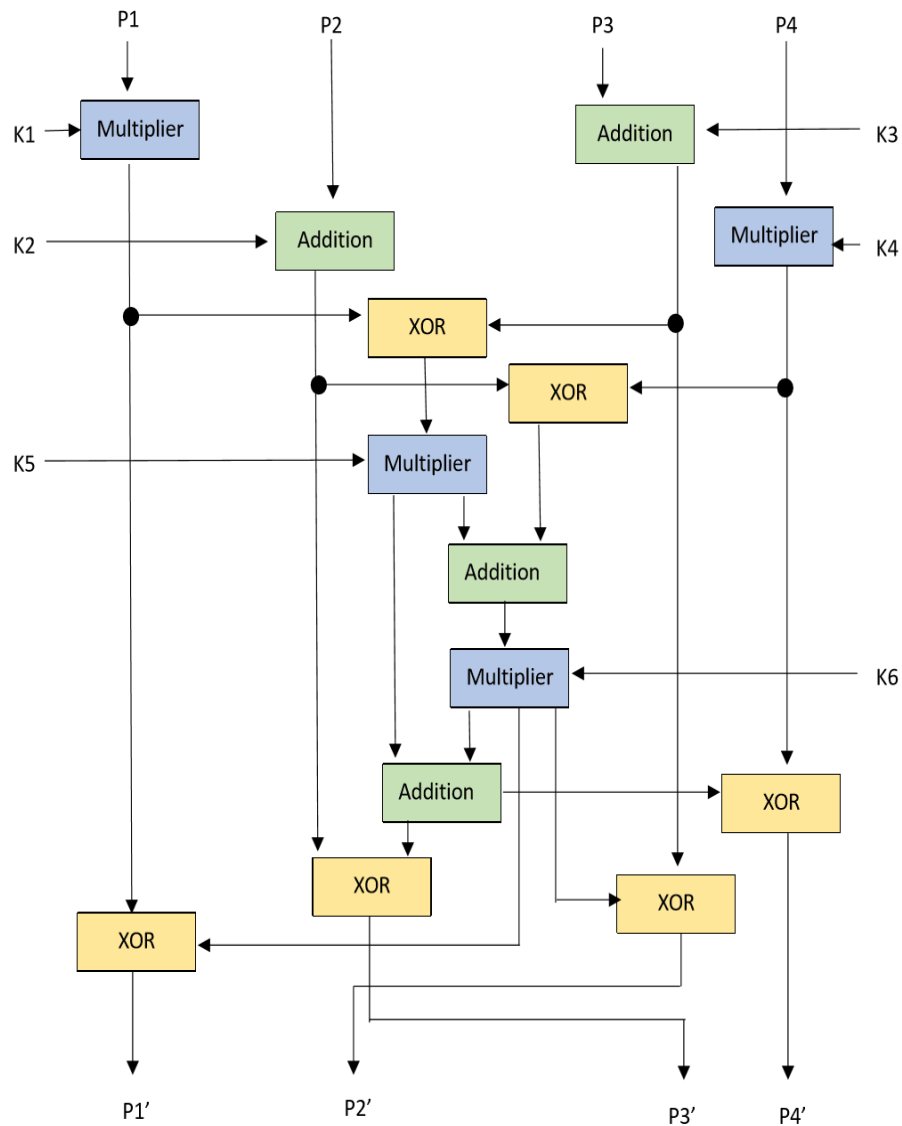


Figure 15: Un tour de IDEA

6.1.4 Génération des sous-clés

1. Les clés initiales à 128 bits, pour ici K1 ... K6 sont générées pour le premier tour. Parce que K1 ... K6 contiennent 16 bits chacune, donc nous avons 96 bits de clé pour le premier tour.
2. Donc les bits 97-128 n'est pas encore utilisé pour le premier tour. (La

figure 16)

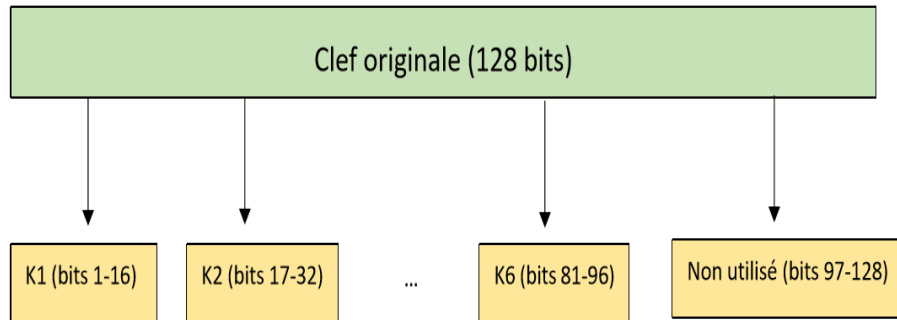


Figure 16: Génération des sous-clés pour le premier tour

Maintenant pour le deuxième tour :

1. Il faut utiliser les 97-128 bits non-utilisé dans le premier tour, donc maintenant nous devons encore générer $96 - 32 = 64$ bits de clés.
2. Maintenant nous devons décaler à gauche 25 bits de la clé originale pour obtenir les restes des bits de clé dans le deuxième tour.
3. Donc les 32 bits du premier tour combinent les 64 bits ce que nous avons décalé, nous avons la sous-clé pour le deuxième tour. Il faut aussi couper en K1 ... K6. (La figure 17)

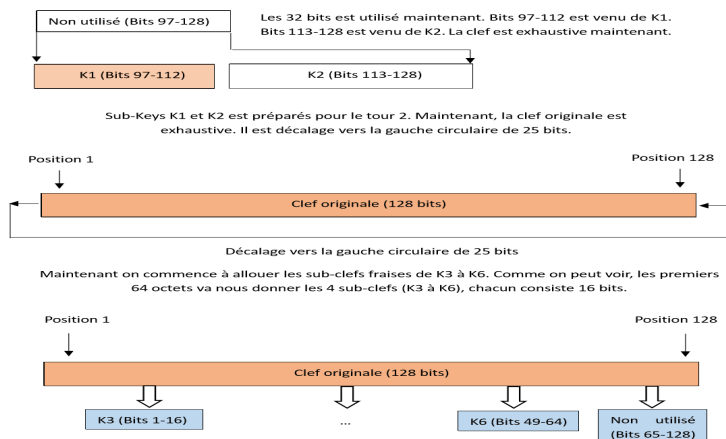


Figure 17: Décaler à gauche pour la génération du deuxième tour

Après, jusqu'à 8-ième tour, nous utilisons la même méthode du deuxième tour pour la génération des sous-clés.

6.1.5 Transformation finale sur la dernière génération de sous-clés

Pour le dernier tour, il faut juste utiliser les 4 sous-clés et nous n'utilisons pas toutes les étapes précédente, seulement les 4 premières étapes. Enfin, nous allons obtenir les sorties C1 ...C4 comme le message chiffré. (La figure 18)

1. multiplier le dernier P'_1 et la sous-clé K_1
2. additionner le dernier P'_2 et la sous-clé K_2
3. additionner le dernier P'_3 et la sous-clé K_3
4. multiplier le dernier P'_4 et la sous-clé K_4

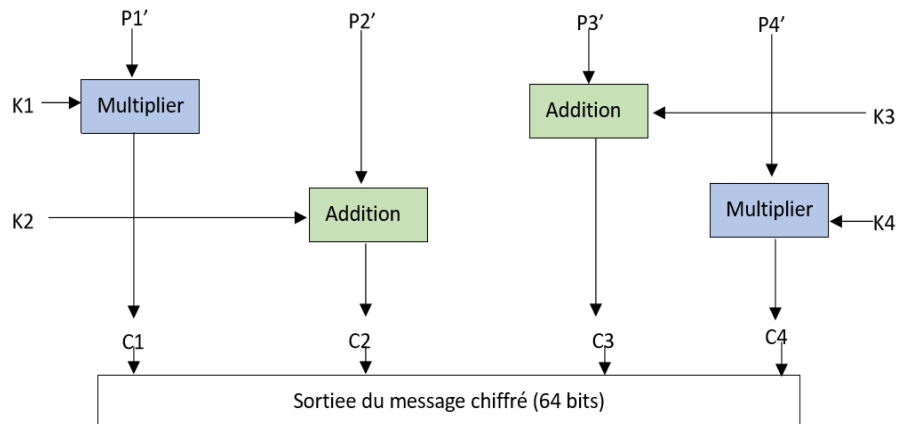


Figure 18: Sortie de la transformation pour obtenir le message chiffré

6.1.6 Exemple pour l'exécution de l'algorithme IDEA

La figure 19 est un exemple sur internet[1] avec les résultats que nous devons obtenir, nous utilisons cet exemple pour vérifier notre algorithme.

La figure 20 est notre implémentation de l'algorithme, nous l'exécutons dans le terminale.

```

setKey(006400c8012c019001f4025802bc0320)
encryptIDEA(05320a6414c819fa)
Round 1      X = 0532 0a64 14c8 19fa ; SK = 0064 00c8 012c 0190 01f4 0258
Round 2      X = 0746 1534 0c68 913c ; SK = 02bc 0320 9002 5803 2003 e804
Round 3      X = f1b7 8e88 78e2 4170 ; SK = b005 7806 4000 c801 0640 07d0
Round 4      X = ec90 b610 aa33 22ec ; SK = 0960 0af0 0c80 0190 0320 04b0
Round 5      X = e262 e986 4690 171a ; SK = a012 c015 e019 0003 2006 4009
Round 6      X = f0d8 4b29 743f 98ea ; SK = 600c 800f 2bc0 3200 0640 0c80
Round 7      X = 22a1 529b fe1f a304 ; SK = 12c0 1900 1f40 2580 000c 8019
Round 8      X = a151 f439 81d9 1462 ; SK = 0025 8032 003e 804b 0057 8064
Output       X = ecda 3ce7 3e53 a60c ; SK = 3200 4b00 6400 7d00 0000 0000
= 65be87e7a2538aed

```

Figure 19: Exemple de chiffrement IDEA trouvé sur internet

<pre> toutes les clés 64 c8 12c 190 1f4 258 2bc 320 9002 5803 2003 e804 b005 7806 4000 c801 640 7d0 960 af0 c80 190 320 4b0 a012 c015 e019 3 2006 4009 600c 800f 2bc0 3200 640 c80 12c0 1900 1f40 2580 c 8019 25 8032 3e 804b 57 8064 3200 4b00 6400 7d00 </pre>	<pre> toutes les étapes de chiffrement 532 a64 14c8 19fa 746 1534 c68 913c f1b7 8e88 78e2 4170 ec90 b610 aa33 22ec e262 e986 4690 171a f0d8 4b29 743f 98ea 22a1 529b fe1f a304 a151 f439 81d9 1462 ecda 3e53 3ce7 a60c Sortie finale 65be 87e7 a253 8aed </pre>
--	---

Figure 20: Les clés générés et les chiffrements à chaque étape de chiffrement

En conclusion, nous avons bien vérifié que notre implémentation est vraiment réussie. L'algorithme IDEA va être utiliser dans la section d'après pour bien observer l'effet de l'attaque.

6.2 La sécurité du cryptosystème IDEA contre l'attaque CPA

Nous devons collecter les traces de IDEA, Nous appliquons l'attaque CPA sur ces traces pour vérifier la sécurité de l'algorithme IDEA. Dans nos résultats, les sorties sont vraiment difficiles à être égale à la clé. Jusqu'à la fin, nous avons utilisé plusieurs traces, mais nous ne trouvons pas la clé correcte. Donc, nous concluons que la sécurité de cryptosystème IDEA est stable.

6.3 Transformer le Timing attaque pour une attaque qui est basée sur la consommation d'énergie[5]

Ce type d'attaque peut être pratique pour récupérer la clé d'une boîte inviolable qui crypte toujours sous la même clé IDEA. Le cryptanalyste n'a pas besoin de savoir quoi que ce soit au sujet du texte en clair pour cette attaque, mais doit toujours savoir exactement quand le cryptage a commencé et quand il s'est terminé. Les modes de chaînage n'ont aucun impact réel sur cette attaque.

6.3.1 Attaque uniquement en message chiffré exigent des timings de blocs précis

Entrées: Timings précis pour n chiffrement, Timings T_0, T_1, \dots, T_{n-1} et les blocs du message chiffré (C_0, \dots, C_{n-1})

Sorties: les clés K_0, K_1, \dots, K_{16}

/*Répéter les étapes ci-dessous pour les 16 bits de poids fort et la première sous-clé multiplicative de la transformation de sortie.

for $i \in \text{poids_fort}[16]$ **do**

/* Regrouper les messages chiffrés et les timings en 2^{16} sous-ensemble*/
sous-ensemble $[2^{16}] \leftarrow C$ et T en fonction des 16 bits de poids faible de la sortie

/* Tester les temps moyens de chaque groupe par rapport aux temps moyens de tous les groupes de manière à déterminer si une des moyennes (avec une probabilité élevée) est inférieure à la moyenne des autres ensembles. */

for $i \in [0, 2^{16}]$ **do**

$\text{moyen}[i] \leftarrow \text{temps_total}[i]/n$

end for

$\text{max} = \text{moyen}[0]$

for $i \in [0, 2^{16}]$ **do**

/*Si c'est le cas, les entrées de la dernière multiplication de la transformation de sortie doivent avoir été 0 pour toutes les entrées de cet ensemble.*/

if $\text{moyen}[i] < \text{max}$ **then**

$\text{max} = \text{moyen}[i]$

$\text{last_multiplication}$ de sortie = 0

Résoudre pour la dernière sous-clé multiplicative

```

    end if
  end for
  if  $max = moyen[0]$  then
    /*S'il n'y a pas de différence, alors nous avons choisi certains paramètres
    (c'est-à-dire, n) faux, ou la sous-clé est un 0.*/
    return n = faux ou sous-clé = 0
  end if
end for
/*Nous avons maintenant 32 bits de clé étendue.*/
/*Nous attaquons maintenant la deuxième sous-clé additive dans la transfor-
mation de sortie. Pour chaque valeur possible de cette sous-clé, nous regar-
dons quels chiffrements nous conduisent à une valeur nulle entrant dans la
première multiplication de la boîte MA du dernier tour. */
for  $i \in [0, 32]$  do
  if deuxième sous-clé additive (sous-ensemble[i]) = 0 then
    /*Pour l'une de ces suppositions de sous-clé, le timings moyen devrait
    être inférieur à celui de toutes les autres suppositions de sous-clé. Cela
    révèle la bonne sous-clé. S'il n'y a pas de différence, soit nous avons
    choisi certains paramètres erronés, soit la première sous-clé dans la boîte
    MA est nulle. Nous avons maintenant récupéré 48 bits de clé étendue.*/
  end if
end for
/*Nous attaquons maintenant la première sous-clé additive dans la transfor-
mation de sortie et la première sous-clé dans la boîte MA.*/
(a) Rompre les textes chiffrés et les temps en  $2^{16}$  sous-ensembles en fonction
de la valeur de la première entrée la plus à gauche dans la boîte MA.
(b) Pour chaque valeur de sous-clé possible pour la première sous-clé addi-
tive de la transformation de sortie, diviser les sous-sous-ensembles en fonctio-
https://www.sharelatex.com/project/5a87060b2951a51f4ec690c7n de la valeur
de la deuxième entrée MA-box si celle-ci était la sous-clé de droite.
(c) Pour la sous-clé de droite, chaque sous-ensemble aura un sous-sous-ensemble
qui aura une plus petite valeur que tous les autres sous-sous-ensembles de ce
sous-ensemble. Nous avons maintenant trouvé 64 bits de sous-clé.
(d) Nous choisissons maintenant trois de ces sous-sous-ensembles et nous les
utilisons pour résoudre la première sous-clé multiplicative de la boîte MA.
Nous avons maintenant trouvé 80 bits de sous-clé et pouvons brutalement
forcer les 48 restants. (Il y a aussi des façons de continuer cette attaque.)

```

6.3.2 La Timings attaque pour le plaintext adaptatif choisi

Une attaque est très similaire à celle décrite ci-dessus est possible, même si le cryptanalyste n'a pas la capacité de mesurer précisément chaque temps de cryptage. Dans ce cas, nous avons la possibilité de choisir des parties de plaintexts à envoyer via l'algorithme de chiffrement, avec la possibilité de chronométrer ces parties.

1. Choisir 2^{16} parties de plaintexts suffisamment grandes pour que l'entrée de la moyenne d'une instruction de multiplication par cryptage à zéro soit détectable dans le temps, avec une probabilité élevée. (Nous devons spécifier des nombres pour les statistiques de synchronisation ici, avant que nous puissions savoir combien de plaintexts doivent être dans chaque partie.)
Chaque plaintext est le même que X_0 , chaque partie de plaintext est la même que X_2
2. Déterminer le temps nécessaire pour que chaque partie soit chiffrée
3. La partie qui a pris le moins de temps à chiffrer indique une équation impliquant une certaine clé matérielle:
 $X_0 \odot K_0 = X_0 \oplus K_0$
4. Générer un autre ensemble de 2^{16} plaintext chiffré, cette fois en utilisant une valeur différente pour X_0 , X_0^* . Suivre les étapes ci-dessus, pour obtenir une deuxième équation:
 $X_0^* \odot K_0 = X_0 \oplus K_0$
Dans la plupart des cas, cela devrait permettre au cryptanalyste de récupérer les deux valeurs clés K_0 et K_2 (Si $Z_4 = 0$, alors cette attaque ne fonctionne pas (il n'y aura pas de différence significative dans les temps))
5. Générer un autre ensemble 2^{16} parties de plaintexts, cette fois on garde (A) constante en utilisant des valeurs fixes pour X_0 , X_2 et maintenant X_3 constante. Pour chaque partie différente, X_1 doit prendre une valeur différente.

Références

- [1] Idea calculator
. URL [https : //lpb.canb.auug.org.au/adfa/src/IDEAcalc/index.html](https://lpb.canb.auug.org.au/adfa/src/IDEAcalc/index.html).
- [2] Cw1173 chipwhisperer-lite
. URL [https : //wiki.newae.com/CW1173_ChipWhisperer - Lite](https://wiki.newae.com/CW1173_ChipWhisperer-Lite).
- [3] L. BOSSUET. Approche didactique pour l'enseignement de l'attaque dpa ciblant l'algorithme de chiffrement aes.
- [4] H.-S. Chang. International data encryption algorithm cs-627-1 fall 2004.
- [5] D. W. John Kelsey, Bruce Schneier and C. Hall. Side channel cryptanalysis of product ciphers.
- [6] P. K. J. Jun. Differential power analysis, technical report, 1998; later published in advances in cryptology – crypto 99 proceedings, lecture notes in computer science vol. 1666, m. wiener, ed., springer-verlag, 1999.
- [7] R. Kotecha. Cryptography and network security by atul karate.
- [8] M. Nassar, Y. Souissi, and S. Guilley. Rsm: A small and fast countermeasure for aes, secure against 1st and 2nd-order zero-offset scas. *Design, Automation , Test in Europe Conference , Exhibition (DATE), 2012*.
- [9] Y. C. Y. Z. H. Z. S. G. Wei Cheng, Chao Zheng and L. Sauvage. How far can we reach? breaking rsm-masked aes-128 implementation using only one trace.

Annexes

A Position des NOPs pour AES

```
1 void aes_enc_round(aes_cipher_state_t* state, const aes_roundkey_t*
    k){
2     uint8_t tmp[16], t;
3     uint8_t i;
4
5     //In fact, we have added 6*volatile for reading better
6     asm volatile(
7         "nop"          "\n\t"
8         "nop"          "\n\t"
9         "nop"          "\n\t"
10        "nop"          "\n\t"
11        "nop"          "\n\t"
12        "nop"          "\n\t"
13        "nop"          "\n\t"
14        "nop"          "\n\t"
15        "nop"          "\n\t"
16        "nop"          "\n\t"
17        "nop"          "\n\t"
18        "nop"          "\n\t"
19        "::
20    );
21    /* subBytes */
22    for(i=0; i<16; ++i){
23        tmp[i] = pgm_read_byte(aes_sbox+state->s[i]);
24    }
25
26    //In fact, we have added 6*volatile for reading better
27    asm volatile(
28        "nop"          "\n\t"
29        "nop"          "\n\t"
30        "nop"          "\n\t"
31        "nop"          "\n\t"
32        "nop"          "\n\t"
33        "nop"          "\n\t"
34        "nop"          "\n\t"
35        "nop"          "\n\t"
36        "nop"          "\n\t"
37        "nop"          "\n\t"
38        "nop"          "\n\t"
39        "nop"          "\n\t"
40        "::
41    );
42
43
44    /* shiftRows */
45    aes_shiftcol(tmp+1, 1);
46    aes_shiftcol(tmp+2, 2);
47    aes_shiftcol(tmp+3, 3);
48
49    //In fact, we have added 6*volatile for reading better
50    asm volatile(
```

```

51     "nop"          "\n\t"
52     "nop"          "\n\t"
53     "nop"          "\n\t"
54     "nop"          "\n\t"
55     "nop"          "\n\t"
56     "nop"          "\n\t"
57     "nop"          "\n\t"
58     "nop"          "\n\t"
59     "nop"          "\n\t"
60     "nop"          "\n\t"
61     "nop"          "\n\t"
62     "nop"          "\n\t"
63     ::
64 );
65
66 /* mixColumns */
67 for(i=0; i<4; ++i){
68     t = tmp[4*i+0] ^ tmp[4*i+1] ^ tmp[4*i+2] ^ tmp[4*i+3];
69     state->s[4*i+0] =
70         GF256MUL_2(tmp[4*i+0]^tmp[4*i+1])
71         ^ tmp[4*i+0]
72         ^ t;
73     state->s[4*i+1] =
74         GF256MUL_2(tmp[4*i+1]^tmp[4*i+2])
75         ^ tmp[4*i+1]
76         ^ t;
77     state->s[4*i+2] =
78         GF256MUL_2(tmp[4*i+2]^tmp[4*i+3])
79         ^ tmp[4*i+2]
80         ^ t;
81     state->s[4*i+3] =
82         GF256MUL_2(tmp[4*i+3]^tmp[4*i+0])
83         ^ tmp[4*i+3]
84         ^ t;
85 }
86
87 //In fact, we have added 6*volatile for reading better
88 asm volatile (
89     "nop"          "\n\t"
90     "nop"          "\n\t"
91     "nop"          "\n\t"
92     "nop"          "\n\t"
93     "nop"          "\n\t"
94     "nop"          "\n\t"
95     "nop"          "\n\t"
96     "nop"          "\n\t"
97     "nop"          "\n\t"
98     "nop"          "\n\t"
99     "nop"          "\n\t"
100    "nop"          "\n\t"
101    ::
102 );
103
104
105 /* addKey */
106 for(i=0; i<16; ++i){
107     state->s[i] ^= k->ks[i];

```

```

108
109 //In fact, we have added 6*volatile for reading better
110 asm volatile(
111     "nop"        "\n\t"
112     "nop"        "\n\t"
113     "nop"        "\n\t"
114     "nop"        "\n\t"
115     "nop"        "\n\t"
116     "nop"        "\n\t"
117     "nop"        "\n\t"
118     "nop"        "\n\t"
119     "nop"        "\n\t"
120     "nop"        "\n\t"
121     "nop"        "\n\t"
122     "nop"        "\n\t"
123     ::
124 );
125
126 }
127
128 void aes_encrypt_core(aes_cipher_state_t* state, const aes_genctx_t
129     * ks, uint8_t rounds){
130     uint8_t i;
131
132     //show us the difference between noise and initialround
133     //In fact, we have added 6*volatile for reading better
134     asm volatile(
135         "nop"        "\n\t"
136         "nop"        "\n\t"
137         "nop"        "\n\t"
138         "nop"        "\n\t"
139         "nop"        "\n\t"
140         "nop"        "\n\t"
141         "nop"        "\n\t"
142         "nop"        "\n\t"
143         "nop"        "\n\t"
144         "nop"        "\n\t"
145         "nop"        "\n\t"
146         ::
147     );
148
149     for(i=0; i<16; ++i){
150         state->s[i] ^= ks->key[0].ks[i];
151     }
152     i=1;
153     for(; rounds>1;--rounds){
154         aes_enc_round(state, &(ks->key[i]));
155         ++i;
156     }
157     aes_enc_lastround(state, &(ks->key[i]));
158 }
159
160

```

B Modification de paramètre afin d'obtenir le début du premier tour

```
1  #Example of using a list to set parameters. Slightly easier to
2  #copy/paste in this format
3  lstexample = [['CW Extra', 'CW Extra Settings', 'Trigger Pins',
4                'Target IO4 (Trigger Line)', True],
5                ['CW Extra', 'CW Extra Settings', 'Target IO Pins',
6                'Target IO1', 'Serial RXD'],
7                ['CW Extra', 'CW Extra Settings', 'Target IO Pins',
8                'Target IO2', 'Serial TXD'],
9                ['OpenADC', 'Clock Setup', 'CLKGEN Settings',
10               'Desired Frequency', 7370000.0],
11               ['CW Extra', 'CW Extra Settings',
12               'Target HS IO-Out', 'CLKGEN'],
13               ['OpenADC', 'Clock Setup', 'ADC Clock',
14               'Source', 'CLKGEN x4 via DCM'],
15               ['OpenADC', 'Trigger Setup',
16               'Total Samples', 2200],
17               ['OpenADC', 'Trigger Setup', 'Offset', 182],
18               ['OpenADC', 'Gain Setting', 'Setting', 45],
19               ['OpenADC', 'Trigger Setup', 'Mode', 'rising edge'],
20               #Final step: make DCMs relock in case they are lost
21               ['OpenADC', 'Clock Setup', 'ADC Clock',
22               'Reset ADC DCM', None],
23               ]
```

C Sbox(input \oplus key)

```
1  _sbox=(
2  0x63,0x7c,0x77,0x7b,0xf2,0x6b,0x6f,0xc5,0x30,0x01,0x67,0x2b,0xfe,0xd7,0xab,0x76,
3  0xca,0x82,0xc9,0x7d,0xfa,0x59,0x47,0xf0,0xad,0xd4,0xa2,0xaf,0x9c,0xa4,0x72,0xc0,
4  0xb7,0xfd,0x93,0x26,0x36,0x3f,0xf7,0xcc,0x34,0xa5,0xe5,0xf1,0x71,0xd8,0x31,0x15,
5  0x04,0xc7,0x23,0xc3,0x18,0x96,0x05,0x9a,0x07,0x12,0x80,0xe2,0xeb,0x27,0xb2,0x75,
6  0x09,0x83,0x2c,0x1a,0x1b,0x6e,0x5a,0xa0,0x52,0x3b,0xd6,0xb3,0x29,0xe3,0x2f,0x84,
7  0x53,0xd1,0x00,0xed,0x20,0xfc,0xb1,0x5b,0x6a,0xcb,0xbe,0x39,0x4a,0x4c,0x58,0xcf,
8  0xd0,0xef,0xaa,0xfb,0x43,0x4d,0x33,0x85,0x45,0xf9,0x02,0x7f,0x50,0x3c,0x9f,0xa8,
9  0x51,0xa3,0x40,0x8f,0x92,0x9d,0x38,0xf5,0xbc,0xb6,0xda,0x21,0x10,0xff,0xf3,0xd2,
10 0xcd,0x0c,0x13,0xec,0x5f,0x97,0x44,0x17,0xc4,0xa7,0x7e,0x3d,0x64,0x5d,0x19,0x73,
11 0x60,0x81,0x4f,0xdc,0x22,0x2a,0x90,0x88,0x46,0xee,0xb8,0x14,0xde,0x5e,0x0b,0xdb,
12 0xe0,0x32,0x3a,0x0a,0x49,0x06,0x24,0x5c,0xc2,0xd3,0xac,0x62,0x91,0x95,0xe4,0x79,
13 0xe7,0xc8,0x37,0x6d,0x8d,0xd5,0x4e,0xa9,0x6c,0x56,0xf4,0xea,0x65,0x7a,0xae,0x08,
```

```

14 0xba,0x78,0x25,0x2e,0x1c,0xa6,0xb4,0xc6,0xe8,0xdd,0x74,0x1f,0x4b,0xbd,0x8b,0x8a,
15 0x70,0x3e,0xb5,0x66,0x48,0x03,0xf6,0x0e,0x61,0x35,0x57,0xb9,0x86,0xc1,0x1d,0x9e,
16 0xe1,0xf8,0x98,0x11,0x69,0xd9,0x8e,0x94,0x9b,0x1e,0x87,0xe9,0xce,0x55,0x28,0xdf,
17 0x8c,0xa1,0x89,0x0d,0xbf,0xe6,0x42,0x68,0x41,0x99,0x2d,0x0f,0xb0,0x54,0xbb,0x16)
18 def sbbox(inp):
19     return _sbbox[inp]
20
21 def subbytes(inp):
22     return sbbox(inp)
23
24 def sbbox_xor(inp,key):
25     return subbytes(inp ^ key)

```

D Lire les fichiers

```

1 FILE *f=fopen("aes_traces.csv","r");
2 if (f==NULL){
3     printf("Can't open this file \n");
4     exit(EXIT_FAILURE);
5 }
6 else{
7     int size_plain=16;
8     int size_trace=2200;
9     int size_line=1001;
10    int i,j,k,line ,bnum,tnum;
11    unsigned char c , buffer [32] , *hw,*keyguess ,***hyp;
12    unsigned char **t_plaintexts;
13    float **t_traces ,**r ,*tmean ,hmean,*sum,*sum1,*sum2 ,hdiff ,*tdiff
    ,cpaoutput ,inter ,*maxcpa;
14
15    t_plaintexts=(unsigned char **) malloc(size_line*sizeof(
    unsigned char *));
16    t_traces=(float **) malloc(size_line*sizeof(float*));
17    for (i=0;i<size_line;i++){
18        t_plaintexts[i]=(unsigned char *) malloc(size_plain*sizeof(
    unsigned char ));
19        t_traces[i]=(float *) malloc(size_trace*sizeof(float ));
20    }
21
22    //line number
23    for (line=0;line<size_line;line++){
24        j=0,i=0;
25        //get one line of plaintexts
26        while(1){
27            c=fgetc(f);
28            //remove useless information(noisy)
29            if(c<0) continue;
30            if(c==' '|| c=='\n'){
31                t_plaintexts[line][j]=atoi(buffer);
32                if(c=='\n') break;
33                j++;

```

```

34     bzero ( buffer ,32 ) ;
35     i=0;
36     }
37     else {
38         buffer [ i]=c;
39         i++;}
40     }
41     j=0,i=0;
42     //get one line of the traces
43     while (1){
44         c=fgetc ( f );
45         //remove useless information (noisy)
46         if (c<0) continue;
47         if (c == ',' || c== '\n'){
48             t_traces [ line ][ j]=atof ( buffer );
49             if (c=='\n') break;
50             j++;
51             bzero ( buffer ,32 ) ;
52             i=0;
53         }
54         else {
55             buffer [ i]=c;
56             i++;}
57         }
58     }
59

```

E L'algorithme pour calculer Pearson correlation coefficient

```

1  float correlationCoefficient ( unsigned char ***hyp , float **
   t_traces , float hmean, float *tmean , int bnum, int k, int
   size_trace , int size_line){
2      int tnum,j;
3      float *sum,*sum1,*sum2,*tdiff , hdiff , inter , cpaoutput , maxcpa;
4      //tdiff
5      tdiff=(float *) malloc (size_trace*sizeof ( float ));
6      //sum
7      sum=(float *) malloc (size_trace*sizeof ( float ));
8      //sum1
9      sum1=(float *) malloc (size_trace*sizeof ( float ));
10     //sum2
11     sum2=(float *) malloc (size_trace*sizeof ( float ));
12     // initial sum,sum1,sum2 by 0
13     for (j=0;j<size_trace;j++){
14         sum [ j]=0;
15         sum1 [ j]=0;
16         sum2 [ j]=0;
17     }
18     //k is keyguess
19     for (tnum=0;tnum<size_line;tnum++){
20         hdiff=hyp [ tnum ] [ bnum ] [ k]-hmean;
21         for (j=0;j<size_trace;j++){
22             tdiff [ j]=t_traces [ tnum ] [ j]-tmean [ j ];
23             sum [ j]+=(float ) hdiff*tdiff [ j ];
24             sum1 [ j]+=(float ) hdiff*hdiff;

```

```

25         sum2[j]+=(float)tdiff[j]*tdiff[j];
26     }
27 }
28 cpaoutput=-100;
29 for(j=0;j<size_trace;j++){
30     // (EXY - EX*EY)/(sqrt(EXX - EX*EX)/sqrt(EYY - EY*EY))
31     inter=(float)fabs(sum[j])/sqrt((float)sum1[j]*sum2[j]);
32     if(cpaoutput<inter)
33         cpaoutput=inter;}
34 maxcpa=cpaoutput;
35 free(sum);
36 free(sum1);
37 free(sum2);
38 free(tdiff);
39 return maxcpa;
40 }

```