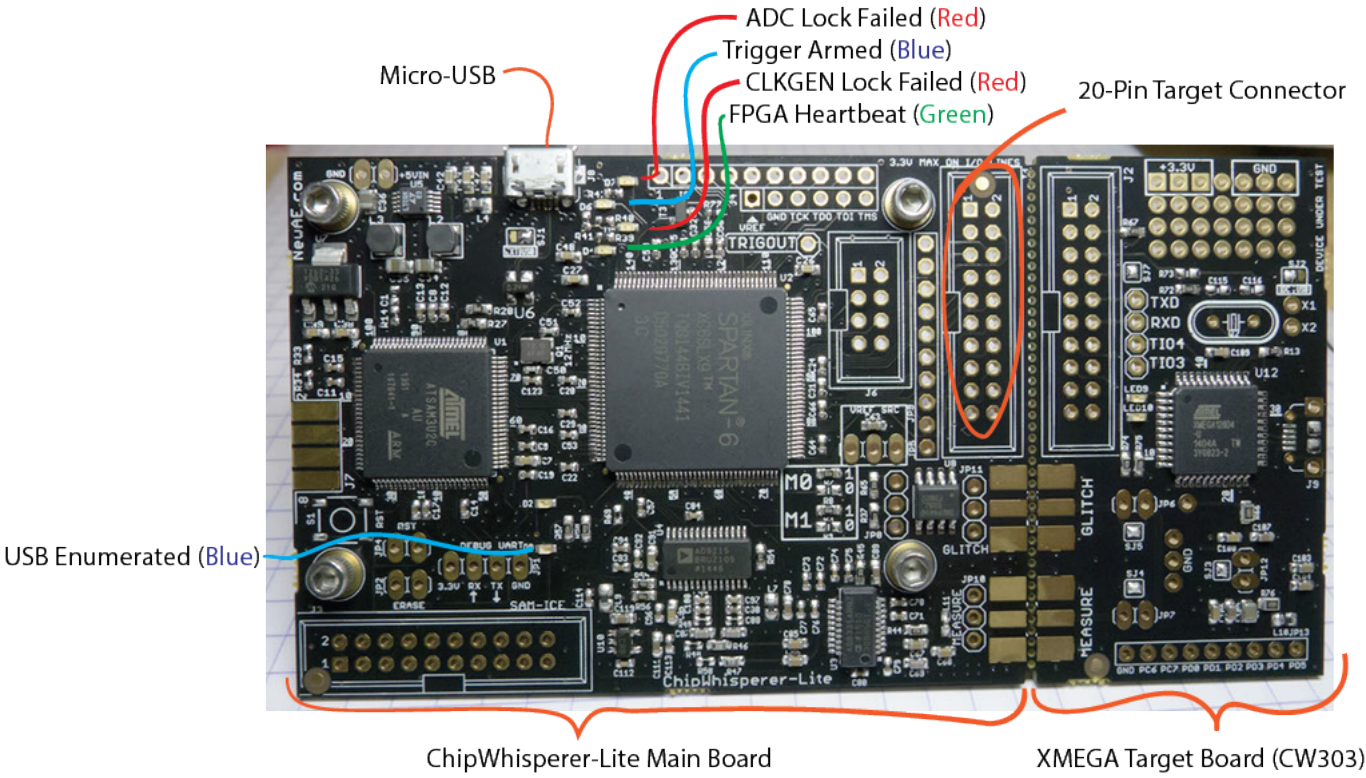
Side Channel Attack Report

MERAH Youcef, NGUYEN Phan Nguyet, YU Chenle

Les attaques par canneaux auxiliaires reposent sur des mesures de puissances du matériel afin de casser des algorithmes cryptographiques tel que AES 128. En effet les mesures physiques sont sources d'informations et peuvent ainsi révéler une part de nos secrets. Afin de comprendre leurs fonctionnements nous effectuerons des analyses à l'aide d'une ChipWhisperer-Lite Bare Board[1].



[1]La carte nue ChipWhisperer-Lite se compose de deux parties principales: un instrument de capture d'analyse de puissance multi-usage et une carte cible. La carte cible est un microcontrôleur standard sur lequel vous pouvez implémenter des algorithmes. Par exemple, si vous souhaitez évaluer une bibliothèque AES, vous pouvez programmer cette bibliothèque dans la carte cible et effectuer l'analyse de la puissance.

Sommaire

Chapter 1

SPA: Simple power analysis

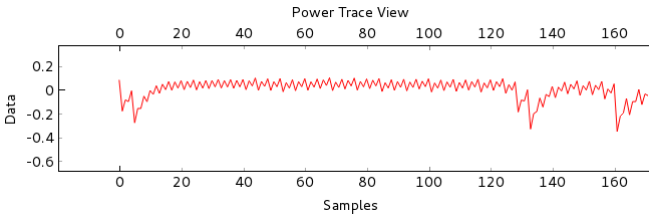
1.1 Notions de base

La forme la plus basique d'une attaque par analyse de puissance est l'attaque par SPA. Elle consiste "simplement" à examiner les traces de puissances afin d'exploiter les échelles de différence causées par des opérations.

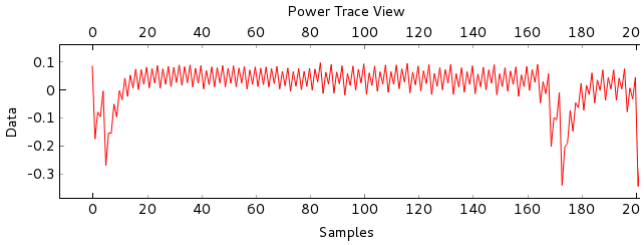
Pour l'illustrer, intéressons nous aux deux instructions suivantes :

- **mul** : Multiplie deux nombres de 8 bits nécessitant un temps de 2 cycles d'horloge
- **nop** : Ne fait rien pendant un cycle d'horloge

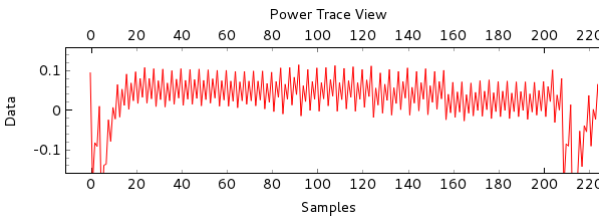
Dans un premier temps on analyse la consommation due à 10 nop suivis de 10 mul. Intuitivement on s'attend à ce que l'instruction *mul* demande plus d'énergie que le *nop*. On obtient une consommation de puissance comme suit :



Avec 20 nop et 10 mul, nous capturons la consommation qui suit:



Enfin, on fait suivre 10 nouveaux nop et nous capturons :



Nous observons que l'ajout d'instructions *nop* impacte directement les mesures de consommation de puissance.

Le temps d'horloge des instructions apparait nettement sur le graphes. Ainsi, il est possible de repérer les étapes du code source à travers des mesures physiques faites sur le cicuit intégré. Cela n'a rien d'anodin, un attanquant peut très bien se reposé sur une analyse simple des consommations de puissance afin d'obtenir des informations sur le secret. Un exemple concret est celui du timing attack

1.2 Timing Attack

Le timing attack est une attaque par SPA.

1.2.1 Échafaudage de l'attaque

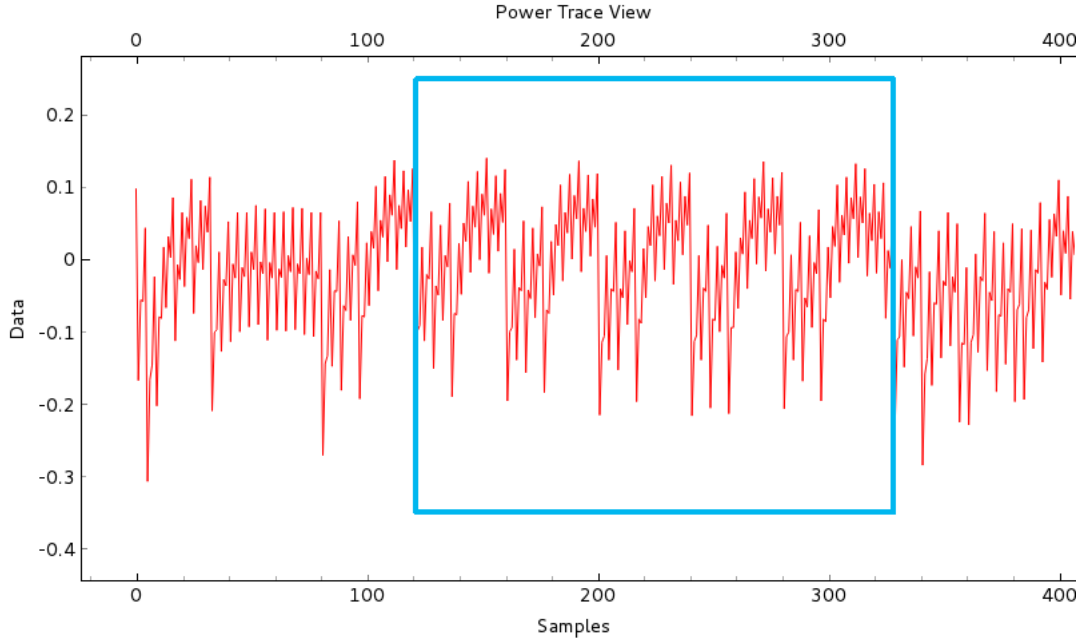
Le code suivant demande un mot de passe à l'utilisteur et vérrifie caractère par caractère l'exactitude de l'entrée. Dès qu'un caractère diffère de celui du secret en mémoire, à savoir *h0px3*, le programme envoie un message d'erreur. Le Timing-Attack consiste à jouer sur le temps de réponse afin d'obtenir des informations sur le secret. Nous verrons en quoi les mesures de consommation de puissance nous apportent ces informations.

code de base Entrée :char correct_passwd[] = "h0px3", char passwd[32]

```
1  uint8_t passbad = 0;
2
3  for(uint8_t i = 0; i < sizeof(correct_passwd); i++){
4       if (correct_passwd[i] != passwd[i]){
5          passbad = 1;
6          break;
7       }
8  }
9
10 if (passbad){
11      //Stop them fancy timing attacks
12      int wait = rand();
13
14      for(volatile int i = 0; i < wait; i++){}
15      _delay_ms(500);
16      printf("PASSWORD FAIL\n");
17      led_error(1);
18 }
19
20
21 else {
22      _delay_ms(500);
23      printf("Access granted, Welcome!\n");
24      led_ok(1);
25 }
26
27 while(1)
```

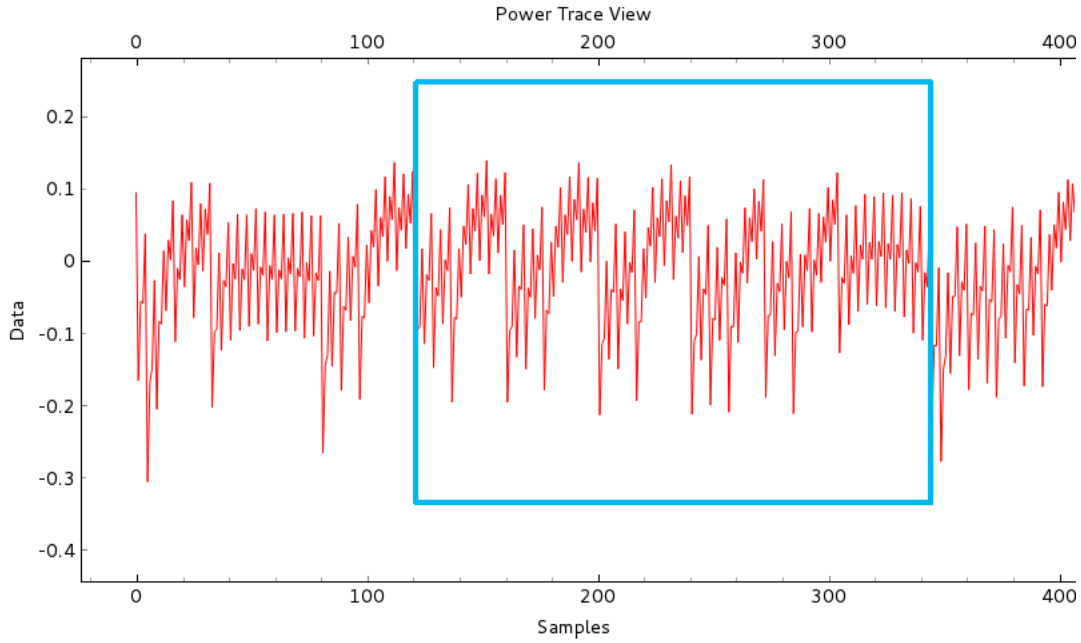
Implémentation 1. Vérification d'un mot de passe

Les mesures ci-dessous ont été obtenues après avoir entrer correctement le mot de passe.



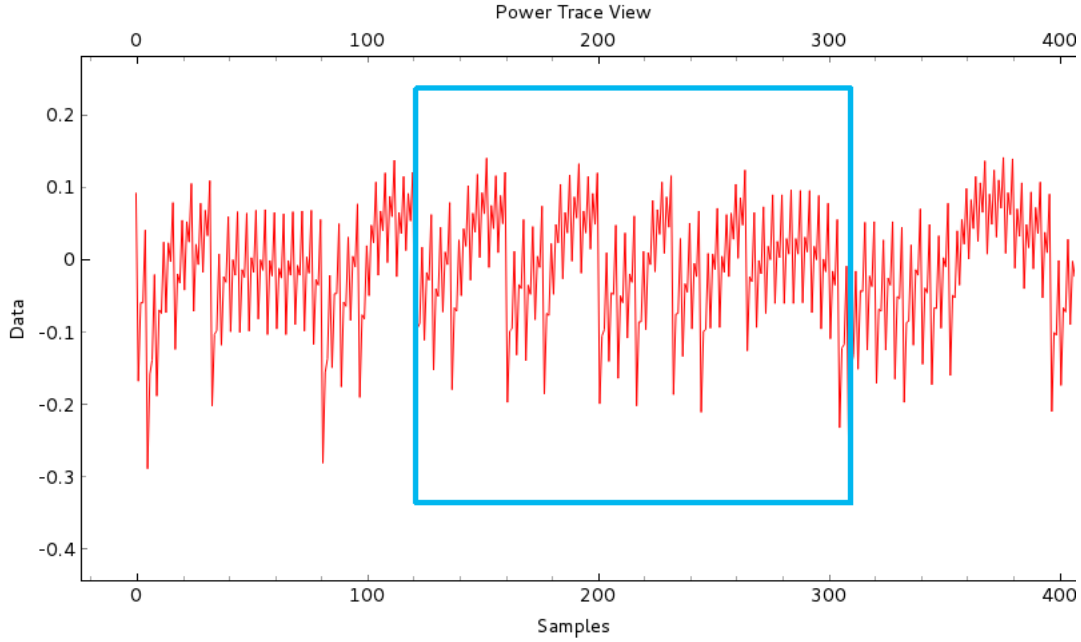
Graph 1. Mot de passe correct

Suite à un mauvais caractère en 5ème position *h0px1*, on obtient les mesures qui suivent :



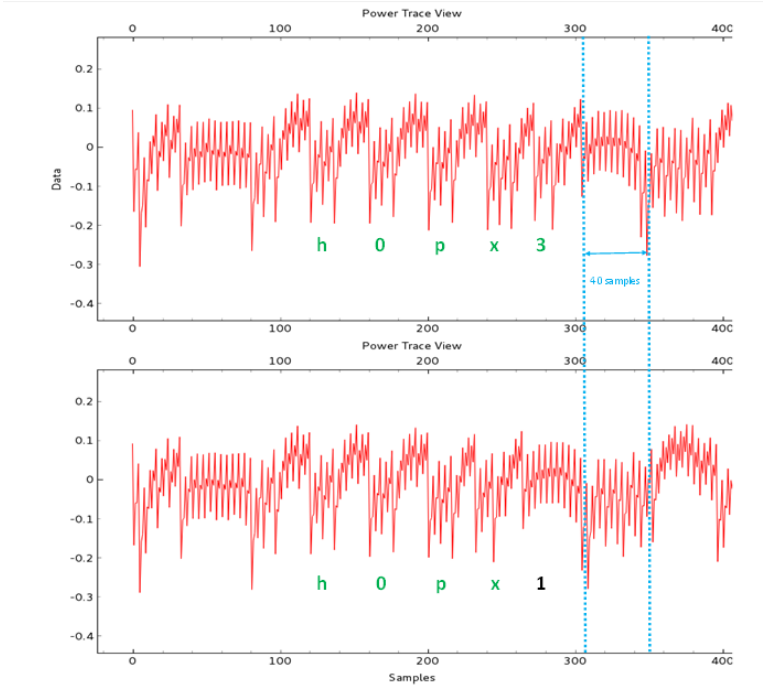
Graph 2. Mot de passe incorrect h0px1

On remarque une différence entre les deux graphes. La différence est d'autant plus flagrante lorsque l'erreur arrive en 4ème position comme le montre le graphe suivant.

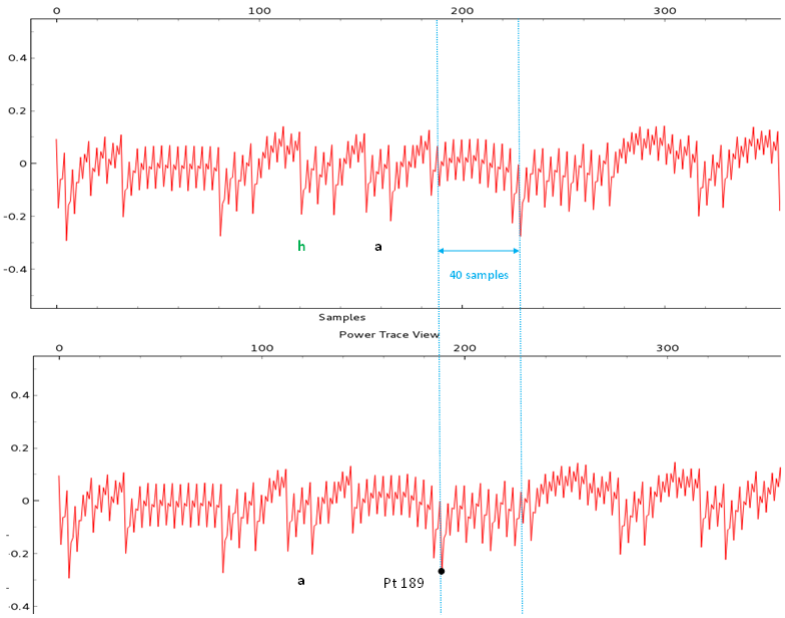


Graph 3. Mot de passe incorrect h0pa

On dicerne un changement de consommation entre les courbes : le graphe 3 semble présenté une mesure en moins. Cela est dû à la sortie prématurée de la boucle *for* (ligne 3) lors d'une erreur. A la sortie de cette boucle, les instructions ne sont plus les mêmes. Or nous avons vu que lorsque des instructions diffèrent, le consommation de puissance diffère également. C'est pourquoi la consommation de puissance qui résulte de la sortie de boucle apparaît **plus tôt** en graphe 3 qu'en graphe 2 . L'attaquant tire avantage de ce **décalage**.
Pour se faire, il se fixe judicieusement un point de repère.



Graph 4.1. Représentation décalage 1 (de haut en bas : mots de passe h0px3 et h0px1)



Graph 4.2. Représentation décalage 2 (de haut en bas : mots de passe ha et a)

On remarque qu'à chaque erreur il y a un décalage de 40 éhantillons.
Le repère doit permettre à l'attaquant de trouver la position du premier caractère erroné (i.e. la sortie prématurée du *for*). Dans notre cas, l'échantillon 189 représenté sur le graphe 4.2 semble être un bon candidat. Ce point répond à deux critères.
Premièrement, il se distingue des autres de part son pic négatif, inférieur à $-0,2Watt$. C'est une caractéristique assez distinctive. Deuxièmement, l'échantillon de ce point est réalisé après la vérification du mot de passe. Ainsi, quelque soit l'erreur donnée, sa position en sera affectée.
Le code attaquant est donc le suivant :

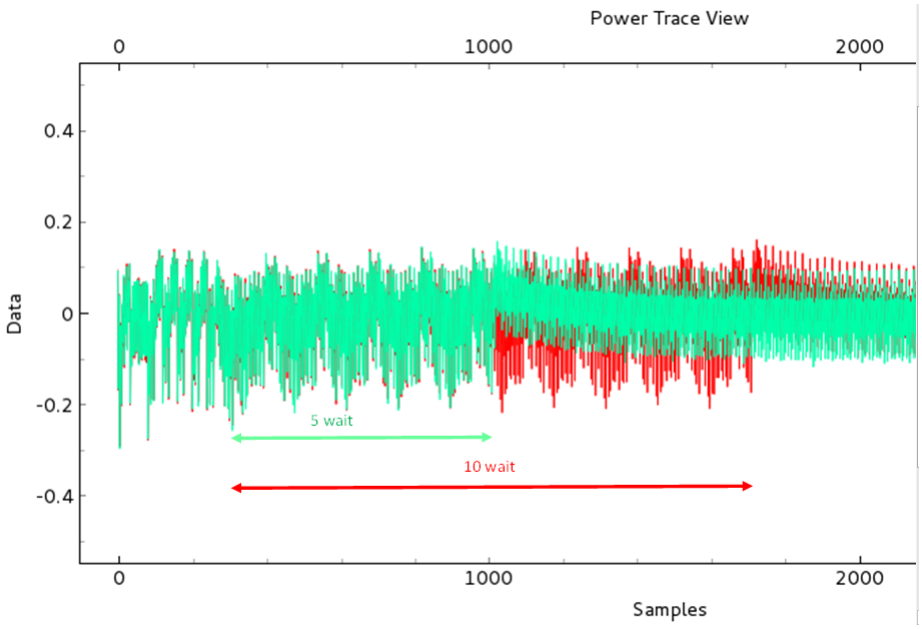
Algorithm 1 Timing-Attack

Require: $password = "", = "abcdefghijklmnopqrstuvwyx0123456789"$
Ensure: $password$
  $end \leftarrow$ **false**
  while $end ==$ **false do**
    $i \leftarrow 0$
    for c in $trylist$ **do**
      $setParameter(password \parallel c)$ {setParameter(mp) est une fonction qui permet d'essayer le mot de passe mp}
      if $scope.datapoint[189 + *i * 40] > -0.2$ **then**
        $password \leftarrow password \parallel c$ {scope.datapoint[s] permet de mesurer la puissance consommée à l'échantillon s}
        $i \leftarrow i + 1$
        break
      else if $end \leftarrow$ **true** , $c == 9$ **then**
        $password \leftarrow password \parallel ?$
        print "Non result find"
      end if
    end for
  end while

Note. *Le code a été généralisé de tel sorte qu'il puisse satisfaire une attaque sans connaissance de la taille du mot de passe*

1.2.2 Contre-mesures

Dans cette section nous verrons qu'il est, bien heureusement, possible de contrecarrer le Timing-attack. En réalité une contre-mesure était déjà présente dans le code de base. Il s'agit des lignes 10 à 18. L'intention était de ne pas entrer brutalement dans le *while(1)*, en ligne 27, lorsqu'un caractère erroné était détecté. Ainsi, lorsqu'un caractère erroné apparaissait, le programme executait une boucle dans lequel il ne faisait rien. Or nous avons vu au premier chapitre que même si le microcontrôleur ne faisait rien, il consommait de l'énergie. Le graphe 4 illustre ce phénomène.



Graph 5. En vert erreur suivie de 5 wait, en rouge même erreur suivie de 10 wait

L'idée était donc de faire croire à l'attaquant que quelque chose se faisait. Cette contre-mesure est, comme vu en section précédente, mauvaise. Puisque le programme ne s'exécute pas de la même manière lorsque le mot de passe est accepté ou non, l'attaquant peut sélectionner un point de repère.

Nous allons présenter deux contre-mesures différentes au Timing-Attack

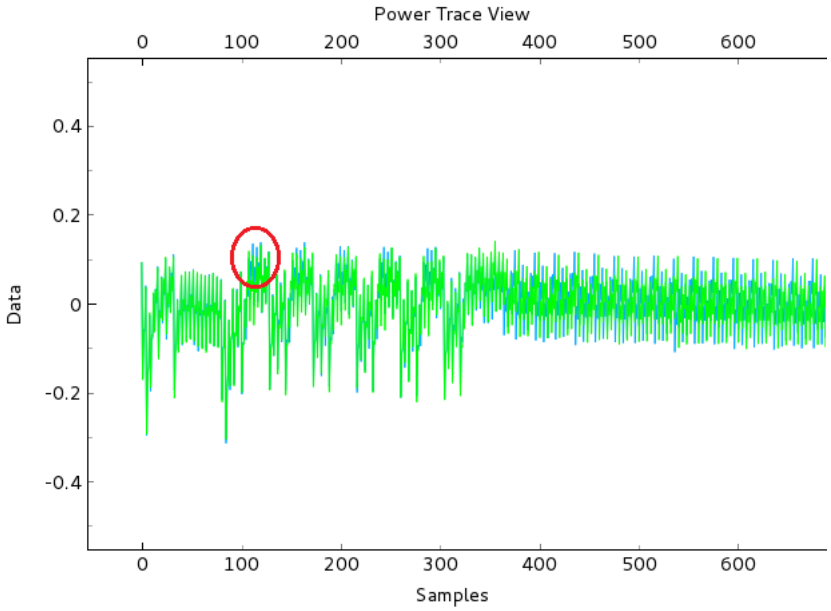
Equilibrage de charge

L'idée est la suivante. Plutôt que de ne rien faire lorsqu'un caractère est erroné, le programme devrait exécuter des instructions semblables aux instructions d'un mot de passe correct ; de sorte que les consonmmations de puissance soient équivalentes.

Algorithm 2 Equilibrage

Require: *correct_passwd* de taille t
Ensure:
  $passwd \leftarrow$ ""
  $tmp \leftarrow 0$
  for $i = 0$ to t **do**
    if $correct\_passwd[i] == passwd[i]$ **then**
      $tmp \leftarrow tmp + 1$
    else
      $tmp \leftarrow tmp + 0$
    end if
  end for
  if $tmp < t$ **then**
    print "PASSWORD FAIL"
  else
    print "Access granted, Welcom!"
  end if

Nous obtenons les graphes suivants :



Graph 6. Superposition 1 : mesures de consommation de puissance lorsqu'un message correct et incorrect sont donnés
Cette solution rend l'attaque ardue.

Checksum

L'idée est équivalente aux checksums souvent utilisés en réseau : on effectue des opérations sur les bits de notre message avec à l'esprit un résultat attendu. Si le résultat obtenu n'est

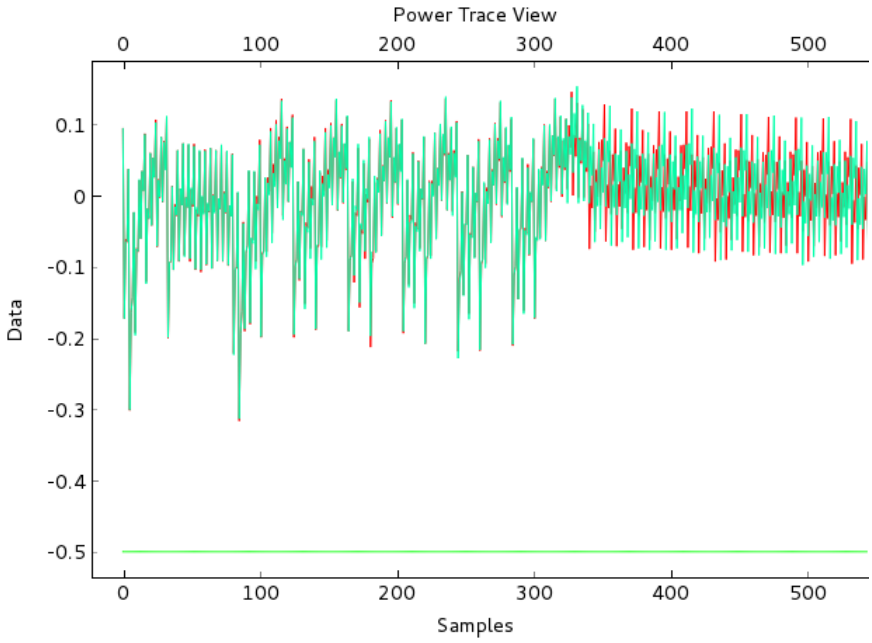
pas celui désiré, alors le message transmis est erroné.

De manière analogue, le mot de passe entré *password* est égale au message authentique *corect_password* si et seulement si le XOR des deux séquences binaires est nul. Ainsi si nous XORons, noté $\oplus$, caractère par caractère ces deux chaînes puis que nous sommons le tout, le résultat attendu est 0. Si ce n'est pas le cas, alors il existe **au moins** un caractère qui fait défaut.
Par ce procédé, il n'y a plus de distinction entre savoir si le mot de passe entré en commande est bon ou mauvais. En d'autres termes : **il n'y a plus de *if***. L'algorithme ci-dessous illustre le procédé.

Algorithm 3 Checksum

Require: *correct_passwd* de taille t
Ensure:
  $passwd \leftarrow$ ""
  $passbad \leftarrow 0$
  for $i = 0$ to t **do**
    $passbad \leftarrow passbad + correct\_passwd[i] \oplus passwd[i]$
  end for
  if $passbad > 0$ **then**
    print "PASSWORD FAIL"
  else
    print "Access granted, Welcom!"
  end if

Nous obtenons les graphes suivants :



Graph 7. Superposition 2 : En vert mot de passe valide, en rouge mot de passe invalide. Ou peut être l'inverse ?!
Cette fois-ci il y a une superposition parfaite.

Chapter 2

DPA: Differential Power Analysis

Depuis l'article de Paul Kocher en 1998, une nouvelle famille d'attaque émergea : l'attaque repose sur des mesures statistiques exploitant la correlation entre la consommation de puissance et les opérations d'un circuit du type CMOS. Nous verrons dans un premier temps en quoi le circuit CMOS révèle de l'information sur les opérations effectuées. Puis nous nous pencherons sur ce type d'attaque appliqué au le chiffrement symétique de l'AES.

2.1 Consommation d'un circuit logique CMOS

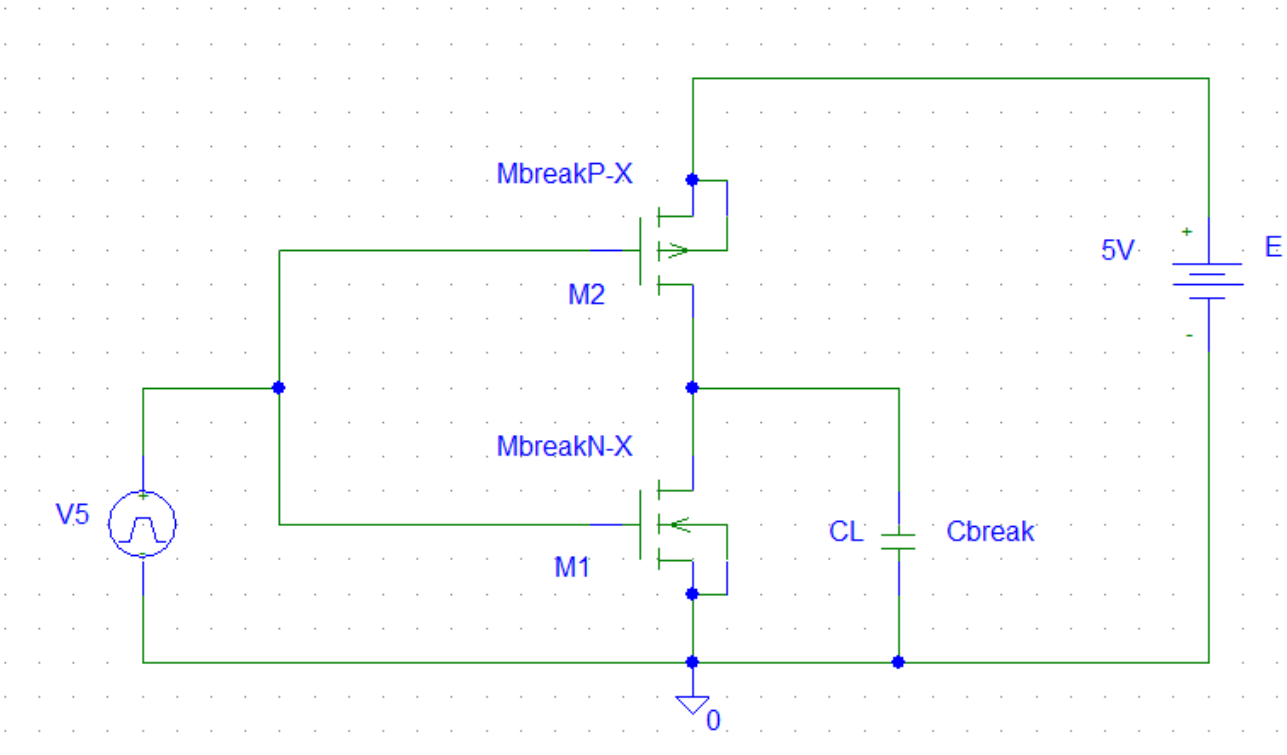
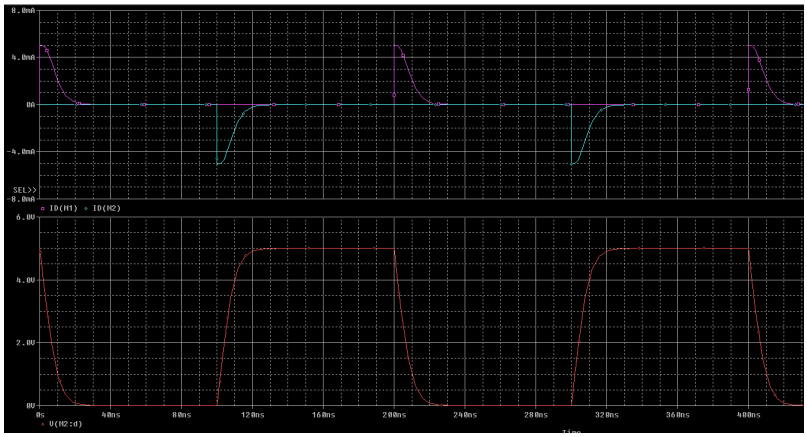


Schéma 1. Inverseur logique CMOS

Un inverseur logique est composé d'un transistor NMOS, d'un transistor PMOS, d'un générateur de tension et d'une capacité. Lorsque l'entrée passe de l'état haut à l'état

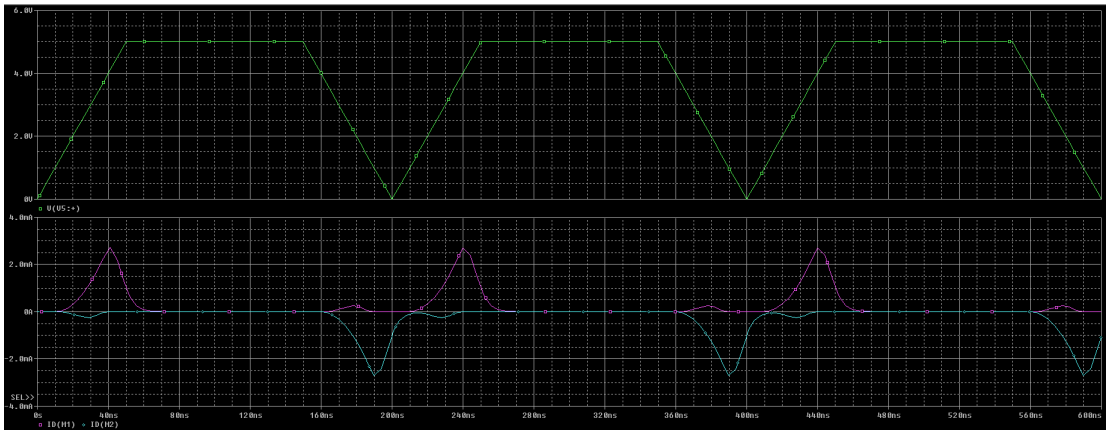
bas, le transistor PMOS conduit le courant et la capacité se charge. Le transistor NMOS quant à lui est ouvert. Cette commutation des transistors permet le passage du bit 0 à 1. Inversement, lorsque l'entrée passe de l'état bas à l'état haut, le transistor NMOS est fermé tandis que le PMOS est ouvert. Suite à cette nouvelle commutation, la capacité se décharge et le bit à 1 passe à 0.

Pour représenté ces opérations, voici une trace effectuée sur le logiciel Spice[1] lorsque l'alimentation d'entrée est périodiquement carrée :



Graph 8. Consommation en puissance du condensateur (rouge) du PMOS (bleu) et du NMOS (rose) en régime carré

En règle générale les périodes ont une allure trapezoïdale, engendrant ainsi un temps de propagation conséquant. Ainsi, lors des passage des états haut-bas et bas-haut les **transistors**(qui ont une tension de seuil respetive) **commutent en même temps** générant ainsi un courant de court-circuit.



Graph 9. Consommation en puissance du PMOS (bleu) et du NMOS (rose) en régime trapezoïdale

On en déduit que la consommation de puissance, suite à des opérations logiques, dépend gandement du nombre de commutation des transistors ! Les attaques par DPA reposent

[1]SPICE (Simulation Program with Integrated Circuit Emphasis) est un logiciel libre de simulation généraliste de circuits électroniques analogiques. Il permet la simulation au niveau du composant (résistances, condensateurs, transistors) en utilisant différents types d'analyses.

sur cette fuite d'informations. En effet, si une opération logique *op* s'effectue sur deux séquences binaires distinctes s et s' alors les consommations en puissance respectivement associées P et P' différeront aussi. La différence entre P et P' est donc due aux bits disctincts entre s et s' ; c'est à dire de **leur distance de Hamming**. Autrement dit il y a une correlation entre la consommation de puissance et la distance de Hamming entre deux séquences en sortie de *op*.

$$P - P' \backsim \delta_H(op(s), op(s'))$$

En récoltant bon nombre de traces de consommation de puissance, dans lesquelles une clef secrète intervient, il est possible de déterminer cette dernière statistiquement. Ce type d'attaque est applicable sur l'AES.

2.2 Advanced Encryption Standard : AES

2.2.1 Procédé du chiffrement

En entrée nous avons un texte clair nommé état et une clef secrète de 16 octets.

$$S = \begin{bmatrix} s_0 & s_4 & s_8 & s_{12} \\ s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \end{bmatrix}, K = \begin{bmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{bmatrix}$$

En premier lieu on procède a une extention de la clef K :

$$\begin{bmatrix} k_0 & k_4 & k_8 & k_{12} \\ k_1 & k_5 & k_9 & k_{13} \\ k_2 & k_6 & k_{10} & k_{14} \\ k_3 & k_7 & k_{11} & k_{15} \end{bmatrix} \Rightarrow [w_0, w_1, w_2, w_3] \xrightarrow{extension} [w_0, w_1, w_2, w_3, ..., w_{42}, w_{43}]$$

Le chiffrement de l'AES exécute plusieurs tours et chaque tour i utilisera la clef $K_i = [w_{4*i}, w_{4*i+1}, w_{4*i+2}, w_{4*i+3}]$

Tour initial, $i = 0$

Le tour initial consiste à une simple opération XOR entre le texte clair et la clef $K = K_0$ nommée *AddRoundKey* :

Tours intermédiaires, $i = 1, ..., 9$

Chaque tour intermédiare se découpe en plusieurs opérations successives :

- *SubBytes*
- *ShiftRows*
- *MixColumns*
- *AddRoundKey* avec la clef K_i

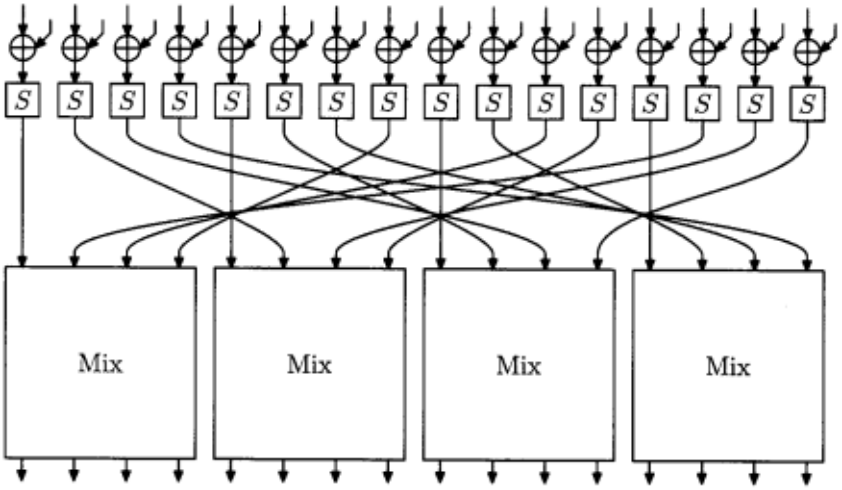


Figure 4.2: Structure of a single round of AES

Figure 1 Tour simple de l'AES [3]

Notons que seul le *SubBytes* est non-linéaire.

Tour final, $i = 10$

Le tour final est équivalent à un tour intermidaire, sauf qu'il n'effectue pas d'opération *MixColumns*.

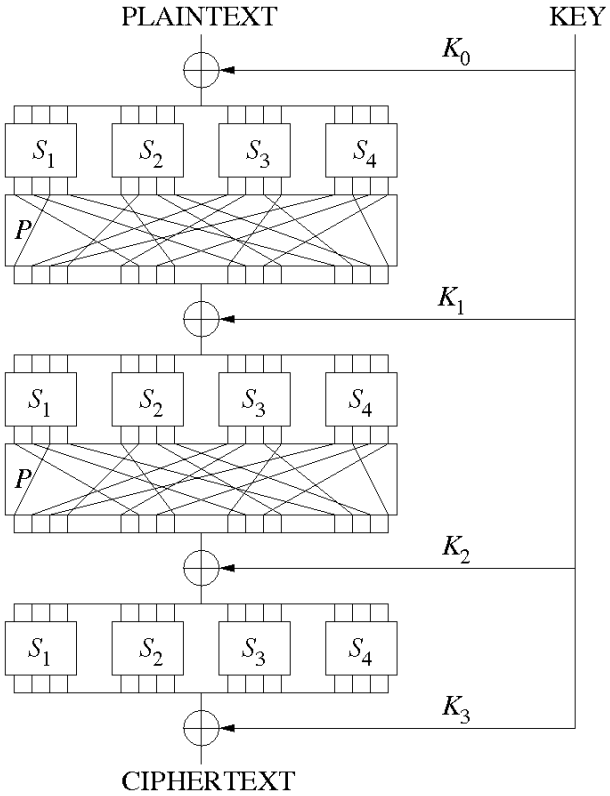


Figure 2. Procédé général de l'AES (GaborPete)

2.2.2 L'attaque DPA sur l'AES

Le procédé de l'attaque en pratique

Après avoir modifié le programme python fourni, on peut maintenant récupérer plusieurs traces avec une seule exécution du script. Comme les chiffrés obtenus de la carte ChipWhisperer et le script python sont identiques, on sait que ces chiffrements utilisent une même clé. Sachant que le circuit intégré de ChipWhisperer utilise la technologie CMOS[4], son comportement suit alors un modèle de Hamming. Nous pouvons donc procéder à une attaque DPA. Commençons par rappeler les principes de DPA comme indiquée dans l'article de Bossuet [1].

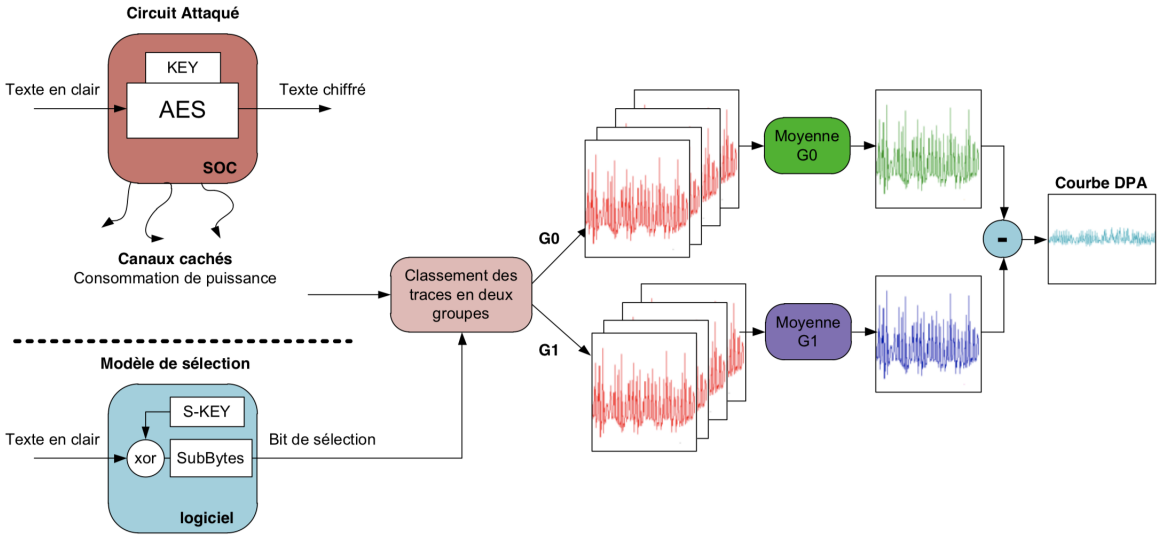


Schéma 2. Schéma représentant le principe général de l'attaque DPA

Ce qu'il faut faire en pratique [5]+[7]:

1. Réaliser un grand nombre de captures et récolter ces plaintexts ainsi que leurs consommations de puissance sur une période pendant laquelle la clé intervient (notre cible étant le premier tour de Sbox comme l'explique la section qui suit)
2. Divide-and-Conquer sur chaque octet de la clé:
   (a) Supposer une valeur, entre 0 et 255, commençons par 0
   (b) Faire AddRoundKey avec l'octet correspondant de premier plaintext (première étape d'AES) pour obtenir $s_0 \oplus k_0$
   (c) Regarder Sbox($s_0 \oplus k_0$) et classifier la première trace en fonction du premier bit en sorti de la Sbox : si c'est un 0 alors on la met dans la classe 0, dans la classe 1 sinon.
   (d) Répéter les étapes (b) à (c) pour tous les plaintexts.
   (e) Faire la différence en moyenne et récupérer le vecteur qui en résulte.
   (f) Stocker le maximum de ce vecteur comme le score de cette supposition d'octet.
   (g) Revenir en (a) avec la prochaine supposition de clef

(h) Comparer les scores de chaque hypothèse : le plus haut score est ainsi notre meilleure supposition de sous-clé pour cet octet.

Nous avons réalisé le procédé de cette attaque en C (une séquentielle et une seconde avec multi-threading). Le code en question repose sur deux tableaux de structures. La première de type *result* qui contient chaque plaintext ainsi que les datapoints correspondants. La seconde de type *class* permet la classification énoncée en point (c).

Notons le rôle central qu'a la fonction *Subbyte* dans la description du DPA appliqué à l'AES. Ceci n'est pas un hasard.

La cible de l'attaque

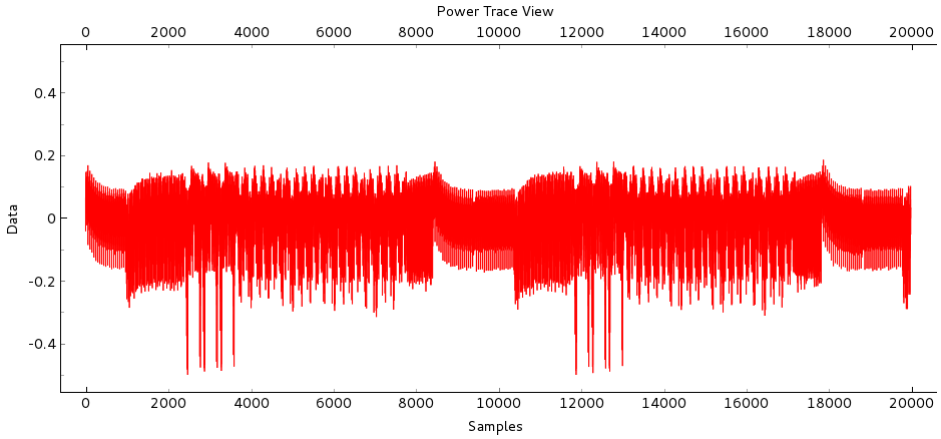
A quel moment réaliser l'attaque par DPA ? Telle est la question. On comprend tout de suite son importance puisqu'elle correspond au point 1. de ce qu'il faut faire en pratique. Nous avons vu qu'il existait une correlation entre consommation de puissance de deux séquences et la distance de Hamming de ces dernières à la sortie d'un opérateur logique.
Supposons que l'attaque soit après l'*AddRoundKey* du tour initial. On a alors
$P \rightharpoonup \delta_H(AddRoundKey(S), AddRoundKey(S'))$
Or,$\delta_H(AddRoundKey(S), AddRoundKey(S')) = \delta_H(S \oplus K_0, S' \oplus K_0) = \delta_H(S \oplus K_0 \oplus S' \oplus K_0)$
D'où,

$$P \rightharpoonup \delta_H(S \oplus S')$$

Nous voyons que la consommation en puissance ne révèlera aucune information de la clef K de par le caractère linéaire de l'opération *AddRoundKey*. Les opérations *ShiftRows* et *MixColumns* sont également linéaires et donc non exploitables. Le *SubBytes* en revanche est non-linéaire, on a :

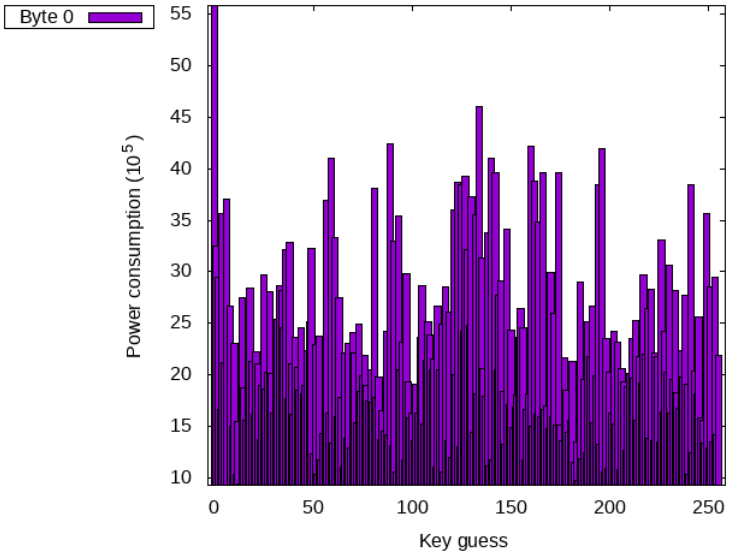
$$P \rightharpoonup \delta_H(SubBytes(S \oplus K_0), SubBytes(S' \oplus K_0))$$

Nous considérons seulement le premier *SubBytes* puisque, comme il est écrit, il dépend directement de la clef $K_0 = K$ et n'est pas soumis à l'expension de la clef.
Nous voyons donc que l'attaque DPA ne doit pas être effectuée à tâtons. Elle doit être réalisée sur des échantillons bien scpécifiques. Ainsi il faut au préalable identifier la partie qui nous intéresse. Voici un graphe qui représente deux tours d'AES séparés par des NOP.

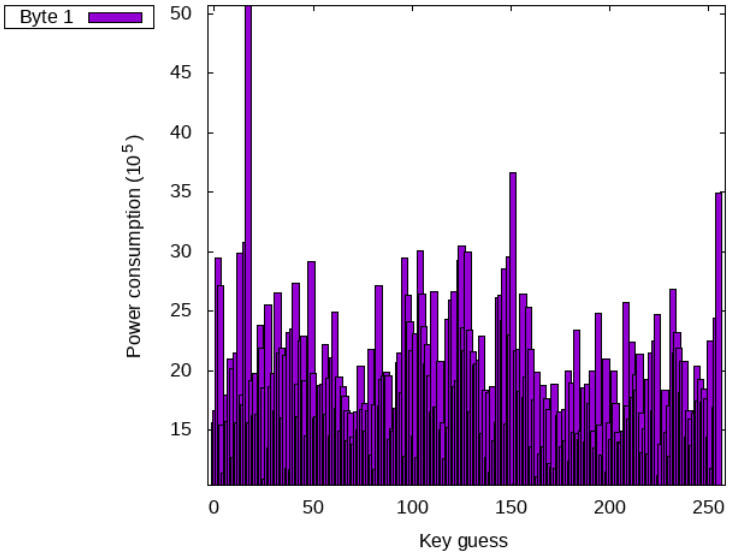


Graph 8. Résultats obtenus suite à une attaque DPA sur le premier octet

Après la réalisation de cette attaque nous avons obtenus des résultats tels que ces deux graphes qui, représentent les scores des 256 suppositions faites sur les deux premiers octets de la clé.



Graph 8. Résultats obtenus suite à une attaque DPA sur le premier octet



Graph 9. Résultats obtenus suite à une attaque DPA sur le second octet

Ces graphes ont été obtenus suite à l'utilisation de Gnuplot[2] intégré dans le code C.

On peut voir qu'avec la supposition 0x00 pour l'octet 0 et 0x11 pour l'octet 1, l'impacte de la clef sur la consommation est mise en évidence. Ce sont ainsi nos meilleures conjectures pour ces octets de la clef. En continuant de cette manière, nous construisons la clé 0x00, 0x11,...,0xff ; ce qui correspond bien à la clé initialement utilisée dans le script python.

[2]Gnuplot was originally created to allow scientists and students to visualize mathematical functions and data interactively, but has grown to support many non-interactive uses such as web scripting. It is also used as a plotting engine by third-party applications like Octave. Gnuplot has been supported and under active development since 1986.

Ce qu'il faut retenir de DPA...

Une bonne suposition de clef conduit à une bonne répartition des classes C_0 et C_1, mais la clef ayant un impact sur la consommation ***la distribution de cette répartion est mauvaise***. De ce fait, ***en moyenne les classes sont déséquilibrées en terme de consommation***. La différence en moyenne des deux classe permet d'expliciter le sample ayant été, en puissance, le plus impacté par la clef. Inversemment, avec une mauvaise conjecture de la clef, la répartition entre les deux classes est assez aléatoire et donc équilibrée. ***Une bonne distribution ne permet pas alors d'exploiter le rôle de la clef dans nos mesures***. Cette distribution est bien représenté par les *Graphes 8* et *9*.

Nous concluons donc que DPA repose sur une moyenne des mesures. Autrement dit, plus le nombre de traces augmente, plus l'attaque est, en terme de validité de la clef, efficace.

Chapter 3

CPA: Correlation Power Analysis

L'attaque CPA est une amélioriation de la DPA. Elle consiste également à exploiter le poids de Hamming.

3.0.1 Notre compréhension sur CPA

On sait que la mesure a un rapport avec le changement de bits entre deux états.
Notons:

- I : l'ensemble des plaintexts ; $n = \text{card}(I)$
- S : l'ensemble des samples ; $m = \text{card}(S)$
- $Ptr_{s,i}$: la mesure, pour un sample s donné, lors du chiffrement d'un octet du plaintext i.
- H_i^k : la distance de Hamming[1] entre l'entrée d'un octet du plaintext i de la Sbox et sa sortie avec une certaine supposition de clef k

Nous avons ainsi à notre disposition un vecteur des distances de Hamming $H^k = (H_i^k)_{0 \leq i \leq n-1}$ et un vecteur de traces : $Ptr_s = (Ptr_{s,i})_{0 \leq i \leq n-1}$ pour chaque sample $0 \leq s \leq m-1$.
Ainsi, le fort rapport entre Ptr_s et H^k peut être qualifié par le coefficient de corrélation de Pearson $\rho(Ptr_s, H^k)$.

L'idée de CPA est que la bonne hypothèse sera celle qui renvoie le plus grand coefficient parmi nos 256 conjectures d'octet de clef.
Il nous reste à déterminer le sample $\hat{\mathbf{s}}$ tel que la correlation entre Ptr_s et H^k est la plus forte. [2]. Il suffit alors de prendre $\hat{s} = \text{argmax}\{\rho(Ptr_s, H^k),\ s \in S\}$.

[1] Dnas notre implémentation nous observons le poids de Hamming en sortie de la SBox, ce qui est méthodiquement la même chose

[2] **À quoi sert le coefficient de Pearson ?**
Il permet la recherche d'absence ou de présence d'une relation linéaire entre deux variables. Il se repose sur le mesure de covariance. Sauf que cette dernière, bien qu'elle mesure la relation entre deux variables, dépend de la grandeur des variables en entrée (qui peuvent ne pas être de même grandeur). Le résultat n'est donc pas très significatif. Le coefficient de Pearson remédie à cela : plus le coefficient est proche de -1 ou de 1, plus la relation entre les deux variables est forte et inversemment. Les deux variables sont indépendantes lorsque le coefficient est nul.

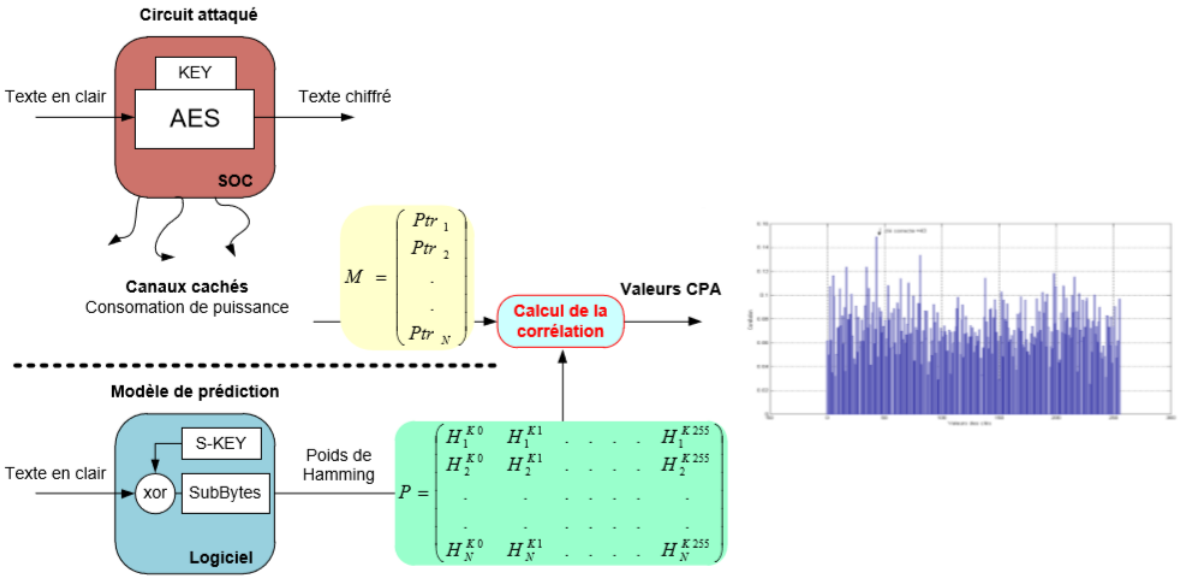


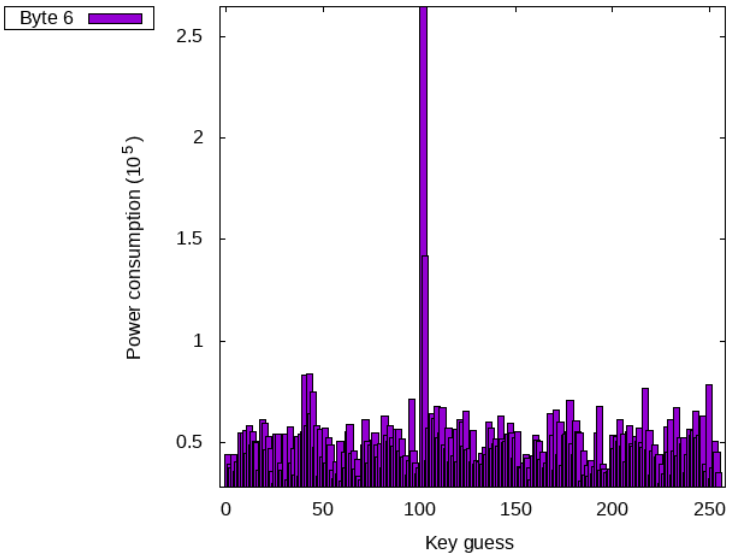
Schéma 3. Schéma représentant le principe général de l'attaque CPA

3.0.2 Notre implémentation CPA

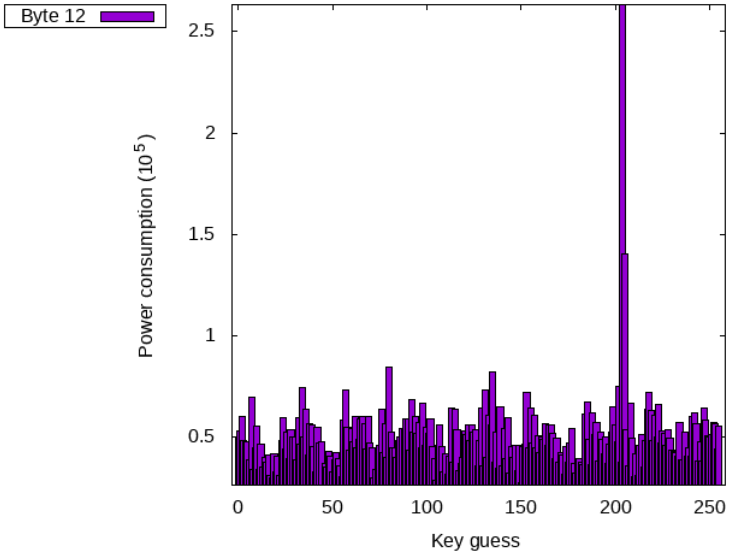
Algorithm 4 CPA-Attack

Require: Ptr_s, $s \in S$ un vecteur *cpa* de taille 256
Ensure: meilleures hypothèses
  $i, j \leftarrow 0$
  while $i < 16$ **do**
    while $j < 256$ **do**
      $H^j \leftarrow (H_0^j, H_1^j, ..., H_{n-1}^j)$
      for s from 0 to $m - 1$ **do**
        $coeffPearson[s] \leftarrow \rho(Ptr_s, H^k)$
      end for
      $\hat{s} \leftarrow \text{argmax}\{coeffPearson[s]\ ;\ s \in S\}$
      $cpa[j] \leftarrow \hat{s}$
    end while
    récupérer la meilleur hypothèse
  end while

Après avoir réaliser ce procédé nous avons retrouver la clef secrète 0x00, 0x11,...,0xff, soit la même qui a été révélée par la DPA (ce qui est rassurant).



Graph 8. Résultats obtenus suite à une attaque CPA sur le $7^{ème}$ octet



Graph 9. Résultats obtenus suite à une attaque CPA sur le $13^{ème}$ octet

Voici deux nouveaux graphes représentant les scores des 256 suppositions faites sur le $7^{ème}$ et le $13^{ème}$ octet de la clé. Il est visible que pour ces deux octects que nous présentons, la bonne hypothèse démontre la plus forte correlation avec les mesures collectées. Ceci est une démonstration pragmatique du coefficient de correlation de Pearson !

Ce qu'il faut retenir de CPA...

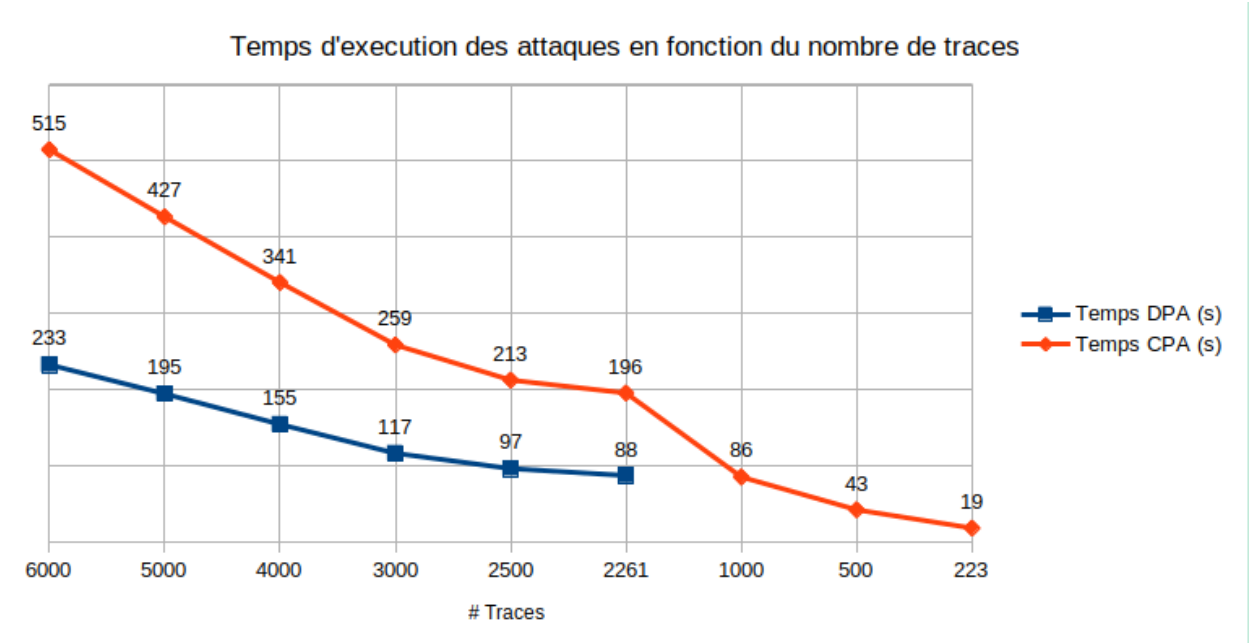
Par définition du coefficient de Pearson, le résultat collecté pour chaque octet avec la bonne conjecture est le plus grand puisque le vecteur de Hamming contenant la bonne conjecture ne peut être que très fortemment corrélé aux traces. Ainsi, contrairement à DPA, un nombre important de plaintexts n'est pas primordial pour ce type d'attaque. Certes, il faut un nombre suffisant de plaintexts pour que les résultats statistiques soient significatifs, mais **ce qui importe le plus est le nombre d'échantillons**.

Nous concluons donc que l'attaque CPA cible directement le sample qui suit la sortie de la Sbox afin de déterminer quelle conjecture est la plus corrélée avec la consommation de puissance.

Chapter 4

DPA versus CPA

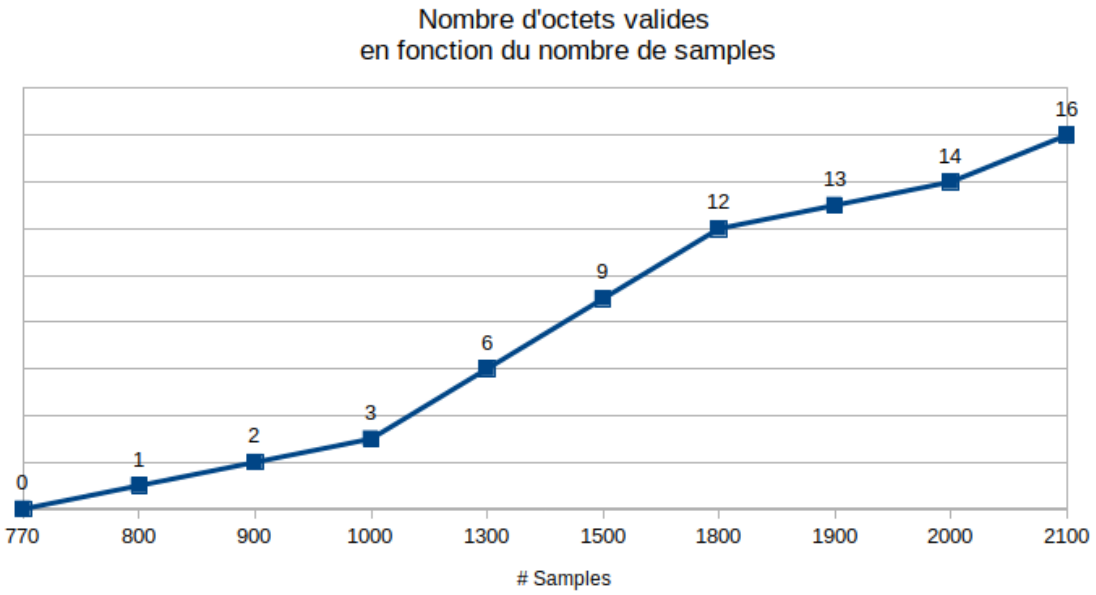
Il a été dit au début du chapitre précédent que l'attaque CPA était préférable à l'attaque DPA. **Est-ce vraiment vrai ?** Cette question nous parait légitime. En effet en effectuant nos tests d'attaques afin de récolter nos résultats, il nous a semblé que **l'attaque par CPA prenait plus de temps**. Théoriquement cela se confirme. En effet, avec DPA nous n'effectuons que des sommes au travers des moyennes ; tandis que CPA calcule des covariances nécissitant des opérations multiplicatives. De ce fait, en terme de complexité, nous aurions tendance à réaliser une attaque DPA. Sur ces interrogations nous avons décidé de confronter ces deux attaques.



Graph 10. Graphe représentant le temps de calcule des deux attaques en fonction du nombre de traces sur 2200 échantillons. Sur les points sont figurés les temps en secondes correspondant. Un point manquant signifie qu'au moins un octet de la clef produite par l'attaque est erroné

En voyant le *Graphe 10* tout se clarifie. Il est vrai que, pour un même nombre de traces, l'attaque par CPA est plus longue que l'attaque par DPA : CPA nécessite 2.2 fois plus de temps que DPA. Toutefois, cette courbe démontre que l'attaque DPA génère la totalité

de la clef secrète qu'à partir de 2261 traces. En effet, à 2260 traces la clef est erronée d'un octet sur seize. En considrant cette faiblesse de DPA nous voyons désormais qu'avec 1000 traces, alors que DPA génère une clef avec seulement 11 octets corrects sur 16, CPA casse la clef secrète en 86 secondes. Cette durée correspond au temps d'attaque DPA pour 2261 traces. Ce qui rendrait CPA deux fois plus efficace que DPA. Mais en poussant les résultats un peu plus loin, nous nous rendons compte que **seulement 223 traces suffisent à l'attaque CPA** pour retrouver la clef secrète et ce, en 19 secondes. Avec 223 traces l'attaque par CPA génère une clef en moins de 9 secondes, mais **aucun des octets généré n'est correct**.



Graph 11. Graphe représentant le nombre d'octets valides de la clef secrète trouvés suite à une attaque CPA en fonction du nombre d'échantillons pour un nombre de plaintexts fixé à 223

Ce graphe nous montre qu'il n'est pas nécessaire à CPA d'augmenter le nombre de plaintexts. Comme nous l'expliquions au chaptire précédent, CPA cible l'échantillon juste après le *Subbyte*. Nous voyons bien que plus le nombre de samples augmente, plus le nombre d'octets corrects découvert par l'attaque CPA augmente.

Synthèse

- L'attaque DPA est optiomale en $85s$ avec $\#Trace = 2261$ et $\#Samples = 2100$.
- L'attaque CPA est optiomale $9s$ avec $\#Trace = 223$ et $\#Samples = 2100$.

  $\rightarrow$ Ainsi, selon nos résultats, l'attaque par CPA a été 9.5 fois plus efficace.[a]
- CPA calcule le coefficient de correlation de Pearson tandis que DPA se repose sur des moyennes nécessitant un nombre important de traces.

[a] Si le temps nous le permet, nous chercherons à savoir si ce résultat est toujours vérifié lorsque nous réaliser les attaques en multi-threading...

Chapter 5

Contremesure pour l'AES

5.1 Échafaudage de la contre-mesure

La similarité entre la consommation et le poids de Hamming après le première SBox offre à l'attaquant, par CPA, la possibilité d'effectuer des comparaisons via des coefficients de correlations. Afin de duper l'attaquant, nous devons biaiser cette comparaison. Pour cela nous pouvons dissimuler le véritable travail de la Sbox effectué sur un authentique octet par une Sbox "faussée" (efftuée sur un octet masqué) . Autrement dit, cette nouvelle Sbox doit être construite de telle sorte qu'on puisse reconstruire le vrai chiffré à la fin d'AES. Pour cela, dans la nouvelle implémentation d'AES (noté newAES), au lieu de travailler sur le vrai octet S, nous pouvons :

- Générer aléatoirement un octet X
- Subdiviser l'octet d'origine S en 2 parties : X et $S \oplus X$
- Travailler sur $S \oplus X$
- Reconstruire le vrai résultat

Pour ne pas compliquer la procédure de reconstuction, une nouvelle SBox (noté par newSBox) est défine comme suit :

$$\text{newSBox}(y) := \text{SBox}(y \oplus X) \oplus X$$

Remarquons que lorsque y est notre octet masqué $(S \oplus X)$, cela revient à :

$$\text{newSBox}(S \oplus X) = \text{SBox}(S) \oplus X,$$

ce qui permet de reconstruire facilement la vrai chiffré. Pour cela il suffit de comparer la sortie d'un state d'un tour de l'ancienne AES avec celle de newAES
Notons:

- $S^i = S^i_0||S^i_1||...||S^i_{15}$: le state à l'entrée du tour i de l'ancienne AES où $||$ est la concaténation.
- $\hat{S}^i$: le state à l'entrée du tour i de newAES.
- sB, sR, mC, aRK sont des opérations subBytes, shiftRows, mixColumns, addRoundKey dans un tour de AES
- newsB : la nouvelle opération subBytes associée à newSBox

Si $S^1 = S_0^1||S_1^i||...||S_{15}^1$ est l'entrée du première tour de l'ancienne AES, alors $\hat{S}^1 = S_0^1 \oplus X||S_1^1 \oplus X||...||S_{15}^1 \oplus X$ est celle du première tour de newAES.

$$
\begin{aligned}
&\text{newsB}(\hat{S}^1) \\
&= \text{newsB}(S_0^1 \oplus X||S_1^1 \oplus X||...||S_{15}^1 \oplus X) \\
&= \text{newSBox}(S_0^1 \oplus X)||\ \text{newSBox}(S_1^1 \oplus X)||\ ...\ ||\ \text{newSBox}(S_{15}^1 \oplus X) \\
&= \text{SBox}(S_0^1) \oplus X||\ \text{SBox}(S_1^1) \oplus X||\ ...\ ||\ \text{SBox}(S_{15}^1) \oplus X \\
&= \text{SBox}(S_0^1)||\ \text{SBox}(S_1^1)||\ ...\ ||\ \text{SBox}(S_{15}^1) \oplus X||X||...||X \\
&= \text{sB}(S^1) \oplus X||X||...||X
\end{aligned}
$$

Comme sR, mC, aRK sont commutatives avec l'opération $\oplus\ X||X||...||X$, on en déduit que la sortie de $\hat{S}^1$ du premier tour de newAES est :

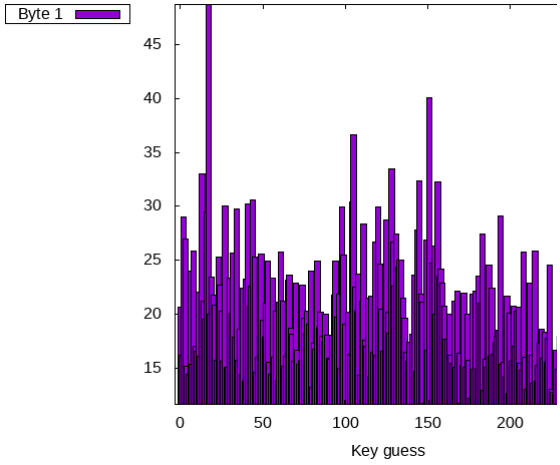
$$\text{aRK} \circ \text{mC} \circ \text{sR} \circ \text{sB}\ (S^1) \oplus X||X||...||X,$$

On obtient alors $\hat{S}^2 = S^2 \oplus X||X||...||X$. Si on reprend les arguments ci-dessus pour des $\hat{S}^i$, on en conclut que **pour reconstruire le vrai chiffré, il nous suffit de faire** $\oplus\ X||X||...||X$ **après le dernier round de newAES**.

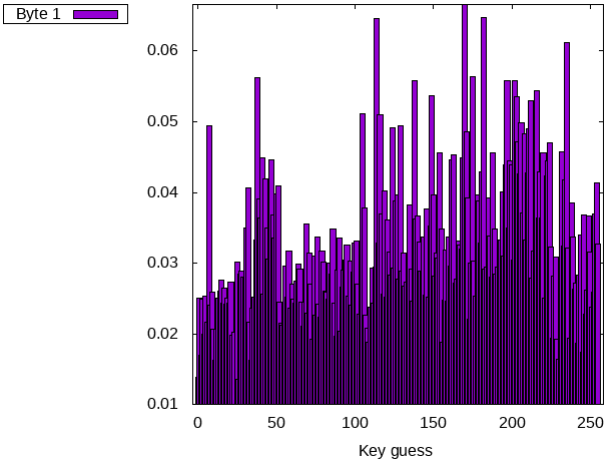
Après avoir implémenter ces modifications sur le code de base de l'AES, nous avons récolter de nouvelles traces puis nous y avons aplliquer nos attaques DPA et CPA.

5.2 Notre interprétation des résultats

5.2.1 DPA contrecarré



Grpah 12.a. Sans contre-mesure

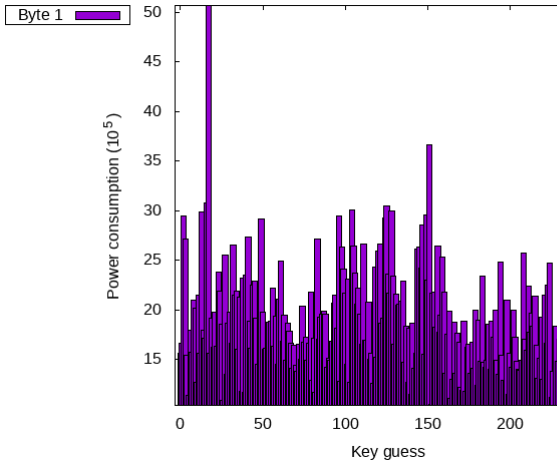


Graph 12.b. Avec Contre-mesure

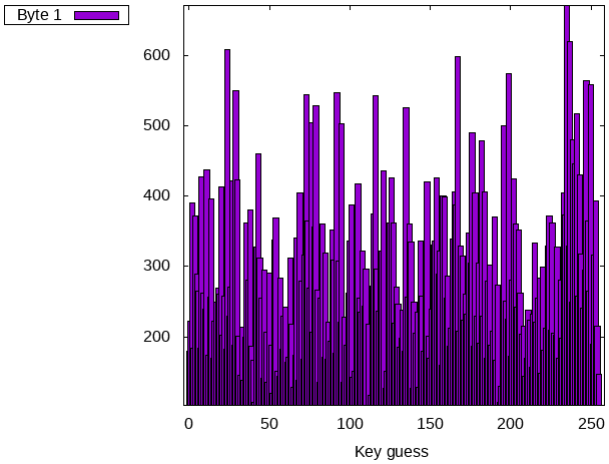
Graph 12. Résultats obtenus suite à une attaque DPA sur le $2^{\text{ème}}$ *octet avec 2261 traces et 2100 échantillons*

Rappelons notre interprétation de l'attaque : les traces sont triées en deux classes en observant le premier bit du octet à la sortie de la première SBox ; ce dernier dépend de la clef, les deux classes sont ainsi déséquilibrées. Ici le *Graphe 12.b.* nous montre un tout autre résultat. Cela montre dans un premier temps que la contre-mesure est efficace. Nous expliquons cette efficacité par le caractère aléatoire de l'entrée $S \oplus X$ de notre nouvelle Sbox *NewSbox*. Cet aléa rend la répartition plus ou moins aléatoire des traces entre les

deux classes de l'attaquant. De ce fait, nous avons une "meilleure" distribution dans les deux classes (du point de vue de la cible). Ce changement de distribution est visible lorsque nous observons les deux graphes côte à côte.
Ci-dessous, les mêmes résultats avec cette fois-ci plus de traces. Nous voyons bien que cela ne permet toujours pas à l'attaqant de retrouver la clef bien que ce nombre de trace est censé être amplement suffisant.



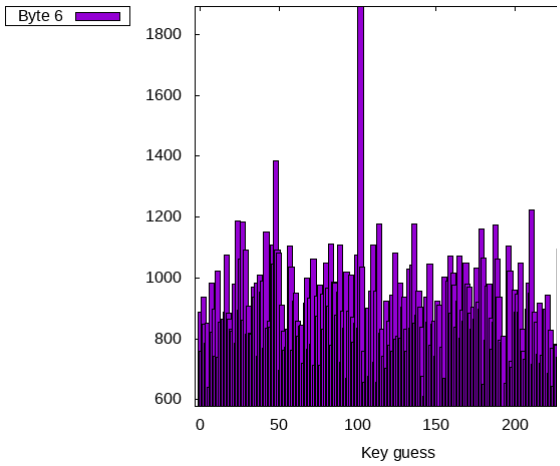
Grpah 13.a. Sans contre-mesure



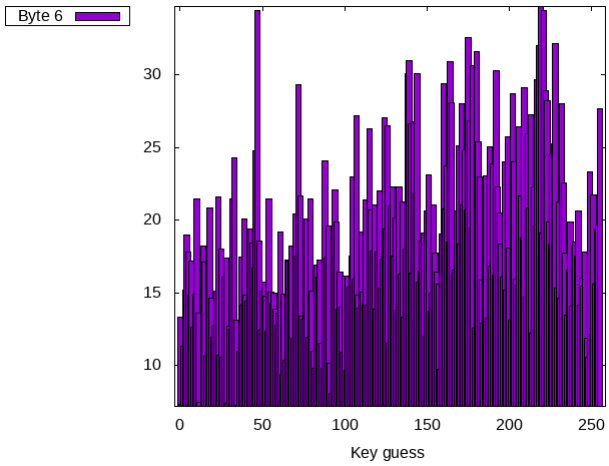
Graph 13.b. Avec Contre-mesure

Graph 13. Résultats obtenus suite à une attaque DPA sur le $2^{ème}$ octet avec 6000 traces et 2200 échantillons

5.2.2 CPA contrecarré



Grpah 14.a. Sans contre-mesure



Graph 14.b. Avec Contre-mesure

Graph 14. Résultats obtenus suite à une attaque CPA sur le $7^{ème}$ octet avec 223 traces et 2100 échantillons

Les résultats obtenus sur CPA sont encore plus flagrants. Alors que le coefficient de correlation de Pearson nous permettait de distinguer franchement la bonne supposition de clef, comme montré en *Graphe 14.a.*, ceci n'est plus le cas après l'ajout de la contre-mesure. En effet, l'attaquant pense pour trouver une correlation entre les traces et la clef, mais il ignore que les traces correspondent aux mesures faites avec l'ajout d'un aléa. De

ce fait, la correlation tend plus vers 0 quelque soit la supposition de la clef, rendant ainsi l'attaque par CPA inneficace. Ceci explique pourquoi **il y a une meilleur (de notre point de vue) distribution des coefficients de correlation**.

Chapter 6

Petite synthèse avec IDEA

6.1 IDEA : International Data Encryption Algorithm

IDEA est un algorithme de chiffrement symétique par bloc. Il se décompose en deux parties. Une premuère qui consiste en l'expansion de la clef secrète de 128 bits. Une seconde pour le chiffrement/déchiffrement d'un bloc de 64 bits. Ici, nous nous intéresserons qu'au chiffrement puisque le but de ce travail est de mettre en exergue des attaques sur la clef secrète.

6.1.1 Chiffrement

Le chiffrement se repose sur le schéma 4 qui suit. Ce procédé est réalisé en huit tours et demi avec des séquences binaires sur 16 bits. Ainsi le bloc de 64 bits en entrée est scindé en 4 mots de 16 bits. Les clefs utilisées sont des sous-clefs de 16 bits dérivées de notre clef originelle de 128 bits. Nous verrons dans la partie qui suit comment sont générées ces sous-clefs.

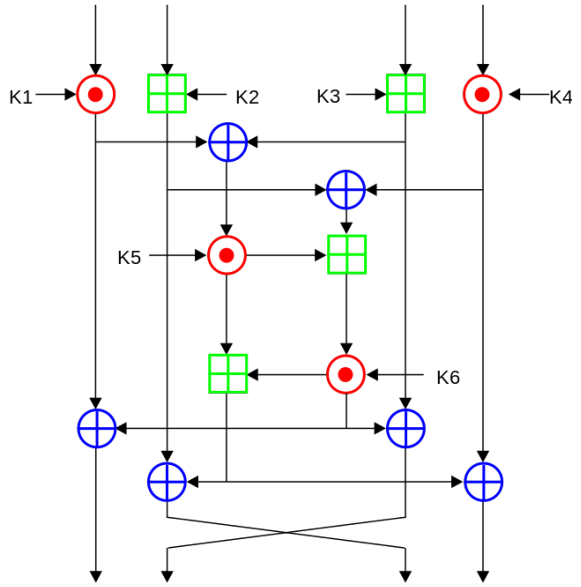


Schéma 4. Schéma représentant un tour de chiffrement d'IDEA - source : Wikipedia

Cette structure, semblable à celle de Lai-Massey, réalise trois opérations que sont les suivantes :

- $\oplus$: l'opération XOR bit à bits
- $\boxplus$: l'addition modulo 2^{16}
- $\odot$: la multiplication modulo $2^{16}+1$

Note. Apportons quelques précisions sur la multiplication modulaire. Les opérandes étant de taille 16 bits, il est surprenant de voir une telle opération nécessitant un bit de plus. Or la multiplication est réalisée dans le groupe cyclique $(\frac{\mathbb{Z}}{(2^{16}+1)\mathbb{Z}})^*$, dont le zéro n'en est pas un élément. De ce fait, le 0 en entrée est interprété comme 2^{16}, inversemment, 2^{16} en sortie est considéré comme un 0.

Ceci expliquant notre implémentation :

```
1 uint16_t multMod2 (uint16_t r1, uint16_t r2) {
2
3         int mod = (1<<16)+1;
4
5         if(!r1 && !r2) return 1;
6         else if (!r1) return mod-r2;
7              else if(!r2) return mod-r1;
8                   else return ((((uint32_t)r1*(uint32_t)r2))%mod) & 0xffff
9 }
```

Implémentation 2. Multiplication-Addition (MA)

En effet, en entrée nos éléments sont dans $\mathbb{F}_{2^{16}+1} := \frac{\mathbb{Z}}{(2^{16}+1)\mathbb{Z}}$ (puisque le 0 peut être une entrée même si par la suite il sera interprété comme 2^{16}). Or $2^{16}+1$ est premier, ainsi $\mathbb{F}_{2^{16}+1}$ est un corps. C'est donc un anneau intègre [1]. Donc si r_1 et r_2 définis en entrée de *multMod* sont tels que $r_1 * r_2 = 0$ alors soit $r_1 = 0$ (respectivement $r_2 = 0$) dans ce cas on retourne $2^{16} * r_2 \bmod (2^{16}+1)$ (respect. $2^{16} * r_1 \bmod (2^{16}+1)$), autrement dit $-r_2$ (respect. $-r_1$). Sinon $r_1 = r_2 = 0$, dans ce cas on retourne $(2^{16})^2 \bmod (2^{16}+1)$, autrement dit 1.

Schneier a présenté un algorithme qui découpent **un tour complet** du chiffrement IDEA en 14 étapes successives [6]. Notre implémentation suit cet algorithme 5.

[1] A est un anneau intègre lorsque $\forall x, y \in A, [x.y = 0] \Leftrightarrow [x = 0] \vee [y = 0]$
Soit $\mathbb{K}$ un corps. Supposons que $\mathbb{K}$ ne soit pas un anneau intègre. Choisissons arbitrairement $x, y \in \mathbb{K} - \{0\}$ tels que $x.y = 0$. Il existe $x^{-1} \in \mathbb{K} - \{0\}$ tel que $x.x^{-1} = 1$ D'où $x^{-1}.x.y = x^{-1}.0 \Rightarrow y = 0$ ce qui est absurde. On en conclut que si $\mathbb{K}$ est un corps, alors c'est forcément un anneau intègre.

Algorithm 5 Schneier algorithm

Require: subkeys (16 bits) : $Z_1, Z_2, Z_3, Z_4, Z_5, Z_6$; plaintext bloc (4*16 bits) : X_1, X_2, X_3, X_4
Ensure: the next required X_i : $t_{11}, t_{12}, t_{13}, t_{14}$

$t_1 \leftarrow X_1 \odot Z_1$
$t_2 \leftarrow X_2 \boxplus Z_2$
$t_3 \leftarrow X_3 \odot Z_3$
$t_4 \leftarrow X_4 \boxplus Z_4$
$t_5 \leftarrow t_1 \oplus t_3$
$t_6 \leftarrow t_2 \oplus t_4$
$t_7 \leftarrow t_5 \odot Z_5$
$t_8 \leftarrow t_6 \boxplus t_7$
$t_9 \leftarrow t_8 \odot Z_6$
$t_{10} \leftarrow t_7 \boxplus t_9$
$t_{11} \leftarrow t_1 \oplus t_9$
$t_{12} \leftarrow t_3 \oplus t_9$
$t_{13} \leftarrow t_2 \oplus t_{10}$
$t_{14} \leftarrow t_4 \oplus t_{10}$

Une fois les 8 tours complets achevés, un demi-tour est réalisé comme décrit en *Algorithme 6* :

Algorithm 6 Half-round

Require: $t_{11}, t_{12}, t_{13}, t_{14}$
Ensure: The ciphered block : c_1, c_2, c_3, c_4

$c_1 \leftarrow t_{11} \odot Z_1$
$c_2 \leftarrow t_{12} \boxplus Z_2$
$c_3 \leftarrow t_{13} \boxplus Z_3$
$c_4 \leftarrow t_{14} \odot Z_4$

Ainsi s'achève le procédé de chiffrement. Nous allons maintenant voir comment les sous-clefs Z_i de 16 bit ont été générées à partir de notre clef originelle K. Nous verrons après en quoi cette méthode de génération peut être exploitée lors d'une attaque de type *timing attack*.

6.1.2 Expansion de la clef

Textuellement, pour généré l'ensemble des sous-clefs il suffit de scinder notre clef K en 4 sous-séquences de 16 bits. Puis on décale notre séquence binaire K de 25 bits vers la gauche et de repositionner ces 25 bits en position de poids faibles. On réitère l'opération jusqu'à obtenir 52 sous-clefs. Seulement des contraintes matérielles font que l'expansion de la clef n'est pas aussi triviale. En effet, cette dernière est définie sur 128 bits, or nos registres ne peuvent contenir des éléments au-delà de 64 bits. Pour répondre à la génération des sous-clefs, il faut alors procédé autrement. Voici notre illustration de la méthode utilisée.

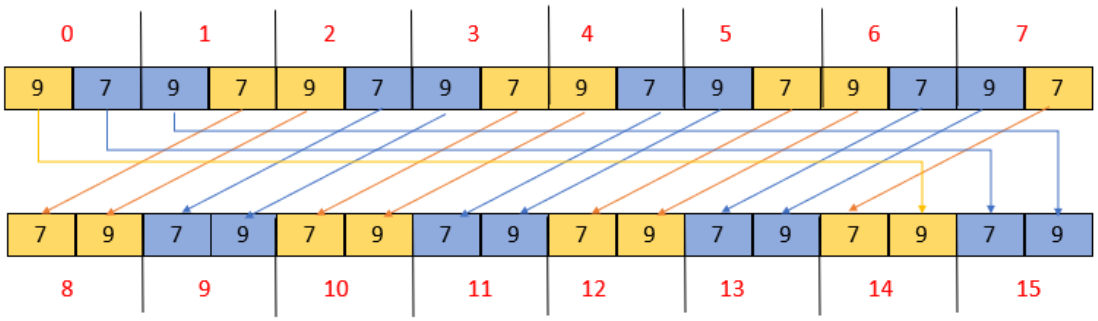


Schéma 5. Schéma représentant la méthode afin de décaler 25 bits vers la gauche de notre séquence de 128 bits représentée par un tableau de 16 bits. Les 16 bits sont vus comme deux sous-séquences : une première de 9 bits de poids forts et une seconde de 7 bits de poids faibles. Les flèches représentent le décalage subit par chaque sous-séquence pour trouver sa nouvelle place.

Nous voyons, qu'il est tout à fait possible de générer 8 sous-clefs en connaissance des 8 sous-clefs qui précèdent. Cette illustration peut être généralisée par le code qui suit.

```
1  void key_schedule(void){
2
3          int i;
4
5          for(i=0 ; i<8 ; i++) fullKey[i] = key[i];
6
7          for (i=8 ; i<52 ; i++) {
8
9          if ( i%8 == 7 ) fullKey[i] = (fullKey[i-8-7] << 9) ^ (fullKey[i-8-6] >> 7);
10         else if ( i%8 == 6 ) fullKey[i] = (fullKey[i-8+1] << 9) ^ (fullKey[i-8-6] >> 7);
11              else fullKey[i] = (fullKey[i-8+1] << 9) ^ (fullKey[i-8+2] >> 7);
12
13         }
14 }
```

Implémentation 3. Expansion de la clef

Nous obtenons ainsi la table de sous-clefs pour le chiffrement :

Tour t	Z_1^t	Z_2^t	Z_3^t	Z_4^t	Z_5^t	Z_6^t
1	K[0-15]	K[16-31]	K[32-47]	K[48-63]	K[64-79]	K[80-95]
2	K[96-111]	K[112-127]	K[25-40]	K[41-56]	K[57-72]	K[73-88]
3	K[89-104]	K[105-120]	K[121-8]	K[9-24]	K[50-65]	K[66-81]
4	K[82-97]	K[98-113]	K[114-1]	K[2-17]	K[18-33]	K[34-49]
5	K[75-90]	K[91-106]	K[107-122]	K[123-10]	K[11-26]	K[27-42]
6	K[43-58]	K[59-74]	K[100-115]	K[116-3]	K[4-19]	K[20-35]
7	K[36-51]	K[52-67]	K[68-83]	K[84-99]	K[125-12]	K[13-28]
8	K[29-44]	K[45-60]	K[61-76]	K[77-92]	K[93-108]	K[109-124]
8.5	K[22-37]	K[38-53]	K[54-69]	K[70-85]		

Table 1. Table des sous-clefs qui peut être précalculée avant toute opération

6.2 Implémentation IDEA sur ChipWhisperer

Après avoir implémenté le chiffrement IDEA, il faut maintenant l'écrire sur la carte ChipWhisperer. Il suffit d'apeler la fonction de chiffrement IDEA à la place d'AES dans le fichier *simpleserial-AES.c* afin que la carte chiffre les données avec IDEA.

```
1 trigger_high();
2 //aes_indep_enc(pt); /* encrypting the data block */
3 IDEA_enc(pt,tmp);
4 trigger_low();
```

On a remarqué que la carte ChipWhisperer risque de faire des erreurs lorsqu'elle effectue des opérations "modulo" $2^n + 1$ dans notre *Implémentation 2.* En effet, bien que les clefs soient bien générées et que le chiffrement sur les machines de la PPTI soit correct, le résultat renvoyé par le système embarqué était erroné. Pour pallier à ce problème nous nous sommes reposé sur le *Lemme* suivant tiré de **??** :

Lemme. Soient a,b deux nombre sur n bits non nuls sur $\mathbb{Z}_{2^n+1}$, alors :

$$ab \mod (2^n+1) = \begin{cases} (ab \mod 2^n) - (ab \text{ div } 2^n) \text{ si } (ab \mod 2^n) \geq (ab \text{ div } 2^n) \\ (ab \mod 2^n) - (ab \text{ div } 2^n) + 2^n + 1 \text{ sinon} \end{cases}$$

Ainsi nous nous ramenons à des opérations modulo 2^n ce qui peut être bien géré par la carte ChipWhisperer par une simple opération logique $\&fffff$. Pour exécuter cette fonction il faut que les entrée soient caster sur 32 bits. EN effet, supposons que nous les laissons sur 16 bits. Ainsi une opération ab $\&0xffff$ est de taille 32 bits et soustraire ce termes par $ab >> 16$ risque d'engendrer un bit de report qui ne tiendra pas sur les 32 bits. Il est donc nécessaire de faire ce cast.

6.3 Des attaques sur IDEA

6.3.1 L'attaque CPA

La première étape consiste, rappelons nous, à déterminer où réaliser l'attaque. Autrement dit, il faut définir notre cible : une fonction non linéaire. Le choix se restreint donc à la multiplication modulaire. Intuitivement, nous portons notre attention sur les deux première application : $X_1 \odot Z_1^t$ et $X_3 \odot Z_3^t$. Notre idée de départ était de déterminer Z_1^t et Z_3^t pour chaque tour t ; puis de reconstruire la clef d'origine à partir des sous-clefs cassées. Seulement deux problèmes se posaient : après avoir vérifier sur la *Table 1.* ci-dessus, nous nous sommes rendu compte que la clef n'étaient pas recouverte par l'ensemble des sous-clefs trouvées (nécessitant donc d'attaquer par force brute les bits manquants) ; enfin nous nous disons qu'il était vraiment dommage de réaliser les huit tours et demi pour reconstruire une clef qui était entièrement utilisée seulement dans les deux premiers tours. Ces deux problématiques nous ont poussé à regarder de plus près les multiplications de la MA. Ce qui nous dérangeait dans ces dernières étaient qu'elles dépendaient de plussieurs clefs. Nous avons essayé d'attaquer cette partie en procédant de la manière suivante :
Sur les trois clefs utilisées pour une multiplication de la MA nous en fixons 2 et testons les 2^16 possibilités de la troisième. Puis nous calculons, en fonction de nos hypothèses, les coefficients de Pearson pour chaque sample afin d'en déterminer la meilleure. Suit à cela nous réitérons ce procédé, en fixant la sous-clef trouvée, pour chaque sous-clef restante.
Après avoir ciblé la partie critique de nos traces nous n'avons pas trouvées de résultats probants. C'est pourquoi nous nous sommes ramenés au cas le plus simple énoncé

précédemment (i.e. attaquer les deux première multiplication). Nos fonctions, nommées *cpa_attack_idea* et *cpa_attack_idea2* se focalisent sur la première (à savoir, celle qui dépend de la sous-clef K_1).
Nous avons réitérer l'attaque afin de casser la sous-clef K_1 **en vain**.

Notre conclusion est la suivante :
Soit l'algorithme de chiffrement IDEA est résistante aux attaques de type CPA, soit le modèle utilisé sur l'AES **qui repose sur le poids de Hamming** est inefficace face à l'IDEA.

6.3.2 Timing attack on IDEA

Le timing attack présenté en chapitre 2 consistait en l'exploitation du temps de réponse de la vérification du mot de passe. Cette vérification se faisant caractère par caractère, un arrêt prématuré dû à un mauvais caractère rendait l'échafaudage du timing attack assez trivial. Ici l'attaque doit se reposer sur le temps de calcule des opérations. Comme opérations nous avons deux additions modulaire $\oplus$ et $\boxplus$ dont la complexité est linéaire en la taille des opérandes : $\mathcal{O}(n)$; et une multiplication modulaire $\odot$ d'une complexité quadratique en la taille des opérandes : $\mathcal{O}(n^2)$.

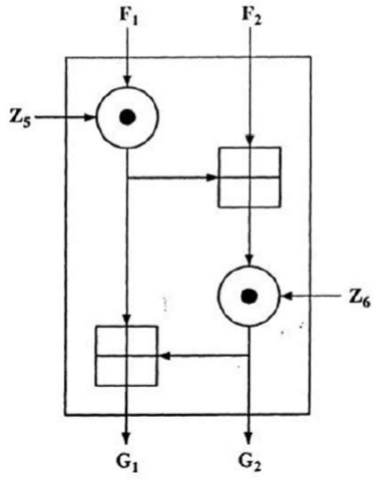


*Schéma 6. Schéma représentant l'étape **MA** du chiffrement IDEA*

Nous avons expliqué la nature de l'*Implémentation 2.* Il apparaissait que des cas particuliers explicitaient le résultat sans besoin de calcul. Autrement dit, **sous certaines conditions**, à savoir lorsque au moins l'une des opérandes était nulle, la complexité du calcul est en $\mathcal{O}(1)$. Ainsi, lorsqu'une opérande est nulle, $\odot$ retourne un résultat en $\mathcal{O}(1)$ au lieu de $\mathcal{O}(n^2)$, il va sans dire que cela doit être visible en termes de temps mais aussi **sur les courbes de consommation de puissance**.

Comme présenté dans [2] nous observerons cette invitation au timing attack au niveau du MA en supposant que Z_5 et Z_6 sont non nuls. On cherchera les valeurs X_i en entrées telles que F_1 et F_2 comme décris en *Schéma 6* soient nuls. Soient nos entrées X_2 et X_4 de l'algorithme 5 fixés.

Nous procédons donc ainsi :

1. Fixer X_1 et récupérer differentes captures des mesures pour diverses valeurs de X_3
2. Choisir le X_3 tel que les mesures montrent un temps de calcul le plus court
3. Considérer l'équation $(F_1) : X_1 \odot Z_1 = X_3 \boxplus Z_3$
4. Fixer X_1^* et récupérer differentes captures des mesures pour diverses valeurs de X_3

5. Choisir le X_3^* tel que les mesures montrent un temps de calcul le plus court
6. Considérer l'équation $(F_1^*) : X_1^* \odot Z_1 = X_3^* \boxplus Z_3$
7. Déduire des équations (F_1) et (F_1^*) les sous-clefs $Z_3 = (X_1^* \odot Z_1) \boxplus X_3^*$ avec $Z_1 = (X_3^* \boxplus X_3) \odot (X_1^* \boxplus X_1)$

On réitère la méthode pour déterminer Z_2 et Z_4. Ainsi, en se référant à la *Table 1* il est possible de récupérer au tour $t = 1$ Z_1^1, Z_2^1, Z_3^1 et Z_4^1 (i.e. $K[0-63]$). Le caractère redondent de l'expansion de la clef nous permet également de récupérer $K[96-127]$ via Z_1^2, Z_2^2. Enfin Z_1^5, Z_2^5 et Z_2^6 nous permettent de **recouvrir entièrement la clef secrète**.

Chapter 7

Conclusion

Les attaques dites par canneaux auxiliaires sont en plein essor. Il faut donc se prémunir contre ce type d'attaques. Nous avons vu que ces attaques ciblent les implémentations cryptographiques sur le matériel. Des fuites d'informations sur les clefs secrètes dues à des mesures physiques sont une aubaine pour les attaqants. Nous avons vus que les attaques peuvent être diverses. Ce travail nous montre la dichotomie entre le monde cryptographique, qui se place dans un monde idéal, et l'Univers physique qui est régit par ses propres lois. Nous devons nous y contraindre et faire appel à l'esprit inventif de l'être humain s'y acommoder.

Bibliography

[1] Lillan Bossuet. Approche didactique pour l'enseignement de l'attaque dpa ciblant l'algorithme de chiffrement aes. *Journal de l'enseignement des sciences et technologies de l'information et des systèmes*, 2012.

[2] D. Wagner J. Kelsey, B.Schneier. *Side Channel Cryptanalysis of Product Ciphers.* Counterpan Systems, U.C. at Berkeley.

[3] Bruce Schneier Neils Ferguson. *Practical Cryptography.* Wiley Publishing, April 2003.

[4] Author non mentionned. Chipwhisperer capture rev2. http://newae.com/files/chipwhisperer-flyer-full.pdf.

[5] Author non mentionned. Correlation power analysis. https://wiki.newae.com/.

[6] B Schneier. *Applied Cryptography , Second Edition.* Wiley Publishing, 1996.

[7] François-Xavier Standaert. Introduction to side-channel attacks, December 2010. https://www.researchgate.net/.