SCA - Differential power analysis of AES

Auguste WARMÉ-JANVILLE (3800191)

December 5, 2023

1 Poids de Hamming et Modèle de Consommation

Localisation des phases de l'AES et des opérations sur les sous-octets

Afin de localiser les phases de l'AES et plus particulièrement l'opération sub-bytes, on commence par "zoomer" sur les premiers milliers d'échantillons en cherchant un motif se répétant 16 fois. On trouve un motif correpondant à cette description à partir du 1500ème sample. On en déduit les valeurs suivantes :

```
scope.adc.samples = 870
scope.adc.offset = 1557
```

Les valeurs mentionnées ensuite, notamment pour les fuites de poids de Hamming, seront exprimées en considérant la valeur d'offset comme zero. On aura donc, pour une valeur `val = x`, la valeur réelle `val = x + scope.adc.offset`.
On obtient la courbe suivante en capturant dans l'intervalle décrit plus tôt.

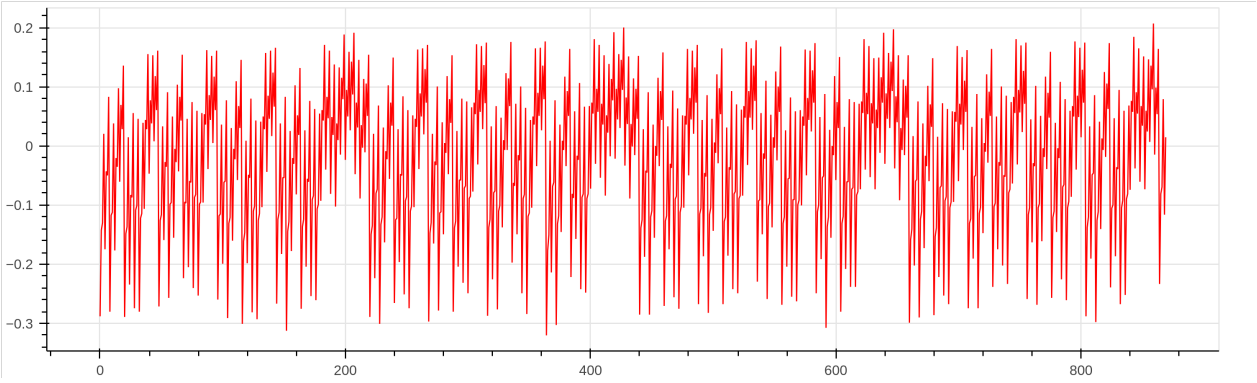


Figure 1: Power trace : sub-bytes phase

En supposant que la trace capturée contienne les 4×4 tours de boucle de calcul de la `sbox`, on définit la fonction suivante, qui pour un octet en entrée, renvoie la partie supposée de la trace correpondant au calcul de sa sbox.

```
period = 48
sb_samples = 38
subloop_samples = 10
mainloop_samples = 28

loop = 4*period + 1 * mainloop_samples

def byte_range(b) :
```

```
    q = b // 4
    r = b % 4
    s = (q*period + r*loop)
    return s, s+period
```

Les sorties de la fonction sont les suivantes :

```
byte 0 (0, 48), byte 1 (220, 268), byte 2 (440, 488), byte 3 (660, 708)
byte 4 (48, 96), byte 5 (268, 316), byte 6 (488, 536), byte 7 (708, 756)
byte 8 (96, 144), byte 9 (316, 364), byte 10 (536, 584), byte 11 (756, 804)
byte 12 (144, 192), byte 13 (364, 412), byte 14 (584, 632), byte 15 (804, 852)
```

On capture ensuite 5000 traces d'exécution de l'AES sur cet intervalle de temps, pour une même clé et des chiffrés différents.

Analyse des Traces

En utilisant le code fourni, et la fonction `byte_range` définie plus haut, on arrive à déterminer les endroits où on a une fuite poids de Hamming pour chaque octet. Il faudra parfois, élargir l'intervalle de recherche renvoyé par la fonction `byte_range`, mais la valeur renvoyée est souvent pertinente pour commencer la recherche.
En observant les courbes pour chqaue octet et en notant les samples ou on observe un dégradé net de foncé à clair de la consommation électrique en fonction du poids de Hamming supposé de la sortie de la sbox, on obtient les valeurs suivantes pour les fuites de poids de Hamming :

```
b0 : 49-50, b4 : 96-97, b8 : 144-145, b12 : 220-221
b1 : 269-270, b5 : 316-317, b9 : 364-365, b13 : 440-441
b2 : 488-489, b6 : 536-537, b10 : 584-585, b14 : 660-661
b3 : 708-709, b7 : 756-757, b11 : 804-805, b15 : 836-837
```

On voit que les valeurs coïncident partiellement avec la valeur de sortie de la fonction `byte_range`, ce qui montre que la recherche n'est pas simple et que ce que l'on regarde sur la courbe de consommation électrique ne correspond pas nécessairement à ce que l'on pense.
On obtient les courbes suivantes qui représentent la consommation moyenne en fonction du poids de Hamming pour chaque octet.

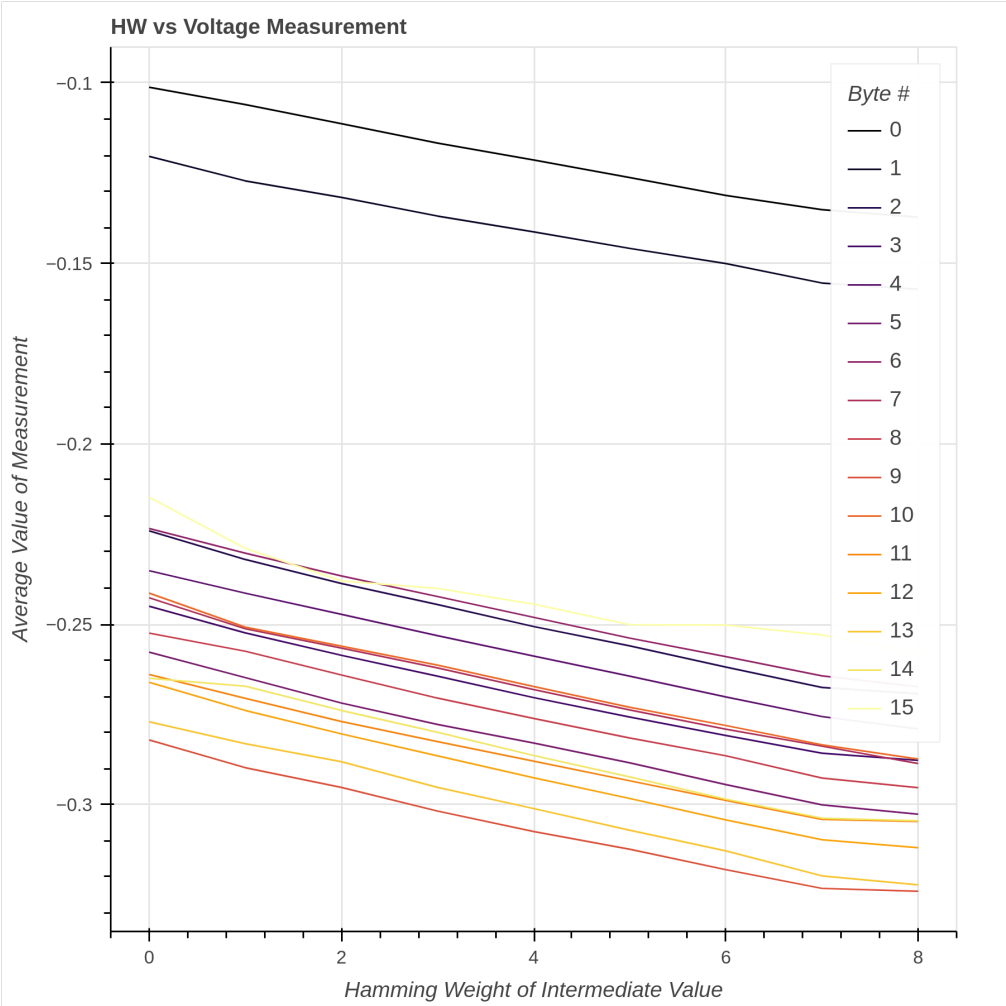


Figure 2: Hamming weight vs. mean voltage measurement

On obtient des courbes droites pour chaque octet. En supposant que c'est le seul sample où l'on a une fuite de poids de Hamming pour chaque octet, on crée le tableau suivant, qui fait correspondre chaque octet au point d'intérêt (que l'on va attaquer par la suite) :

```
hw_leak = [49, 269, 488, 708, 96, 316, 536, 756, 144, 364, 584, 804, 220, 440, 660, 836]
```

On aurait aussi pu automatiser la recherche : pour chaque octet, on cherche le point d'intérêt en calculant le HW supposé pour point de chaque courbe. Pour chaque point, on indexe ensuite toutes les trace en fonction du HW supposé, et on garde seulement le(s) point(s) qui affichent une relation affine entre le poids de Hamming et la consommation. Dans un monde parfait, cela devrait être une méthode qui fonctionne, mais cependant, les courbes ne sont pas totalement linéaires, et avec un peu de malchance, on pourrait ne pas trouver ce qu'on recherche. Mais pour palier à ce problème, il "suffit" d'augmenter le nombre d'échantillons sur lesquels on travaille.

2 Attaque DPA

DPA mono-bit

Pour cette attaque, on va classer les traces en fonction de la valeur d'un bit de sortie de la `sbox`. Pour chaque de sortie de la `sbox` et pour chaque sous octet de la clé, on essaie toutes les sous-clés possibles. Pour chaque

sous clé, on regarde la valeur de sortie du bit de sortie de la sbox que l'on veut analyser, et on classe les traces en fonction de la valeur de ce bit. Autour du point d'intérêt p_i, on regarde l'intervalle $[p_i - \alpha, p_i + \alpha]$ de chaque trace et on calcule la moyenne de toutes les traces chaque groupe, à chaque point de l'intervalle. On obtient donc deux traces "moyennes". On calcule ensuite la valeur absolue de la différence entre ces deux traces, pour chaque valeur possible de l'octet, et on fait ensuite la somme des points de la trace "différentielle". On garde ensuite en mémoire le maximum de chaque somme pour chaque bit que l'on analyse, et à la fin, on retourne l'octet de clé correspondant au plus grand maximum (parmi les 8), qui devrait être correct avec un nombre suffisant de traces. En répétant l'attaque pour tous les octets, on arrive à retrouver la clé exacte.

```
def attack_byte_sbit(bnum) :
    # best key byte guesses for each bit
    guesses = []
    # maximum difference for each bit
    diff_arr = []
    for i in range(0, 8) :
        M = 0
        best_key = 0
        for key_guess in range(256) :
            b_0 = []
            b_1 = []
            for tnum in range(0, N):
                textin = textin_array[tnum][bnum]
                sbox_out = intermediate(textin, key_guess)
                b = (sbox_out) & (0x1 << i)
                # sort the traces depending on the sbox output
                if b == 0 :
                    b_0.append(tnum)
                else :
                    b_1.append(tnum)
            # compute the range depending on the critical point we found
            ra = range(hw_leak[bnum] - alpha, hw_leak[bnum] + alpha + 1)

            mean_trace_0 = [0] * len(ra)
            mean_trace_1 = [0] * len(ra)
            # compute the mean traces for each group
            for j in range(len(ra)) :
                mean_trace_0[j] = np.mean([trace_array[tnum][ra[0]+j] for tnum in b_0])
                mean_trace_1[j] = np.mean([trace_array[tnum][ra[0]+j] for tnum in b_1])
            # compute the difference between the two mean traces
            diff = np.array(mean_trace_0) - np.array(mean_trace_1)
            sabs = np.sum(np.abs(diff))

            # store the byte that has the largest mean value difference s
            if sabs > M :
                M = sabs
                best_key = key_guess
        # append the best byte to the guess array
        diff_arr.append(M)
        # the maximum difference
        guesses.append(best_key)
    return guesses[diff_arr.index(max(diff_arr))]


def attack_key_sbit() :
    password = ""
    for bnum in range(16) :
        by = attack_byte_sbit(bnum)
        password += "{:02x}".format(by)
    return password
```

DPA multi-bit

L'attaque mise en place ici classe les traces en fonction de leur poids de Hamming. Pour chaque octet de clé, on essaie toutes les valeurs possibles. Pour chaque valeur possible de l'octet, on calcule la sortie supposée de la `sbox` en fonction de l'octet de texte clair et de la valeur de l'octet supposée, pour chaque trace. On classe

les traces en deux groupes : dans un groupe, on regroupe les traces dont le HW supposé de l'opération `sbox` est inférieur à 4, et dans l'autre les traces dont le HW est supérieur ou égal à 4. Autour du point d'intérêt p_i, on regarde l'intervalle $[p_i - \alpha, p_i + \alpha]$ de chaque trace et on calcule la moyenne de toutes les traces chaque groupe, à chaque point de l'intervalle. On obtient donc deux traces "moyennes". On calcule ensuite la valeur absolue de la différence entre ces deux traces, pour chaque valeur possible de l'octet, et on fait ensuite la somme des points de la trace "différentielle". On retourne ensuite l'octet pour lequel on a la plus grande somme parmi les 256, qui devrait être le bon octet. En répétant l'attaque pour tous les octets et avec un nombre suffisant de traces, on arrive à retrouver la clé exacte.

```
# range to consider around the critical point
alpha = 2

# number of samples to use
# the attack uses the N first samples in the sample array
N = 100 # max = 5000

def attack_byte(bnum) :
    M = 0
    best_key = 0

    for key_guess in range(256) :
        # sort traces depending on their supposed hamming weights
        # hw < 4
        hw_low = []
        # hw >= 4
        hw_high = []

        for tnum in range(0, N):
            textin = textin_array[tnum][bnum]
            # compute the supposed sbox out considering the key byte guess
            sbox_out = intermediate(textin, key_guess)
            # compute the hamming weight of the sbox out
            h = HW[sbox_out]
            # add the corresponding trace to the right list
            if h < 4 :
                hw_low.append(tnum)
            else :
                hw_high.append(tnum)
        # small range around the hamming weight leak
        ra = range(hw_leak[bnum] - alpha, hw_leak[bnum] + alpha)

        mean_trace_hw_low = [0] * len(ra)
        mean_trace_hw_high = [0] * len(ra)

        # compute the mean values of the traces belonging to the two groups
        # the result is 1 (mean) trace
        for j in range(len(ra)) :
            mean_trace_hw_low[j] = np.mean([trace_array[tnum][ra[0]+j] for tnum in hw_low])
            mean_trace_hw_high[j] = np.mean([trace_array[tnum][ra[0]+j] for tnum in hw_high
    ])
        # compute the difference between the mean values
        diff = np.array(mean_trace_hw_low) - np.array(mean_trace_hw_high)
        sabs = np.sum(np.abs(diff))

        # store the byte that has the largest mean value difference s
        if sabs > M :
            M = sabs
            best_key = key_guess
    return best_key

def attack_key_multibit() :
    password = ""
    for bnum in range(16) :
        by = attack_byte(bnum)
        password += "{:02x}".format(by)
    return password
```

3 Analyse des résultats

Comparaison de performance

Tout d'abord, on remarque qu'en théorie, l'attaque multi-bit est 8 fois plus efficace que l'attaque mono-bit, car elle détermine chaque octet en une fois au lieu de 8. Il faudrait cependant que ces deux attaques soient aussi précises pour que cette comparaison aie du sens. On exécute alors ces deux attaques sur les mêmes traces correspondant à des exécutions de l'AES avec une clé sur des textes clairs différents. On fait varier le nombre de traces utilisées de 16 à 512 par incréments de 16. Pour chaque attaque, on compte le nombre d'octets récupérés incorrects, et on prend note du temps d'exécution. On obtient le graphe suivant.

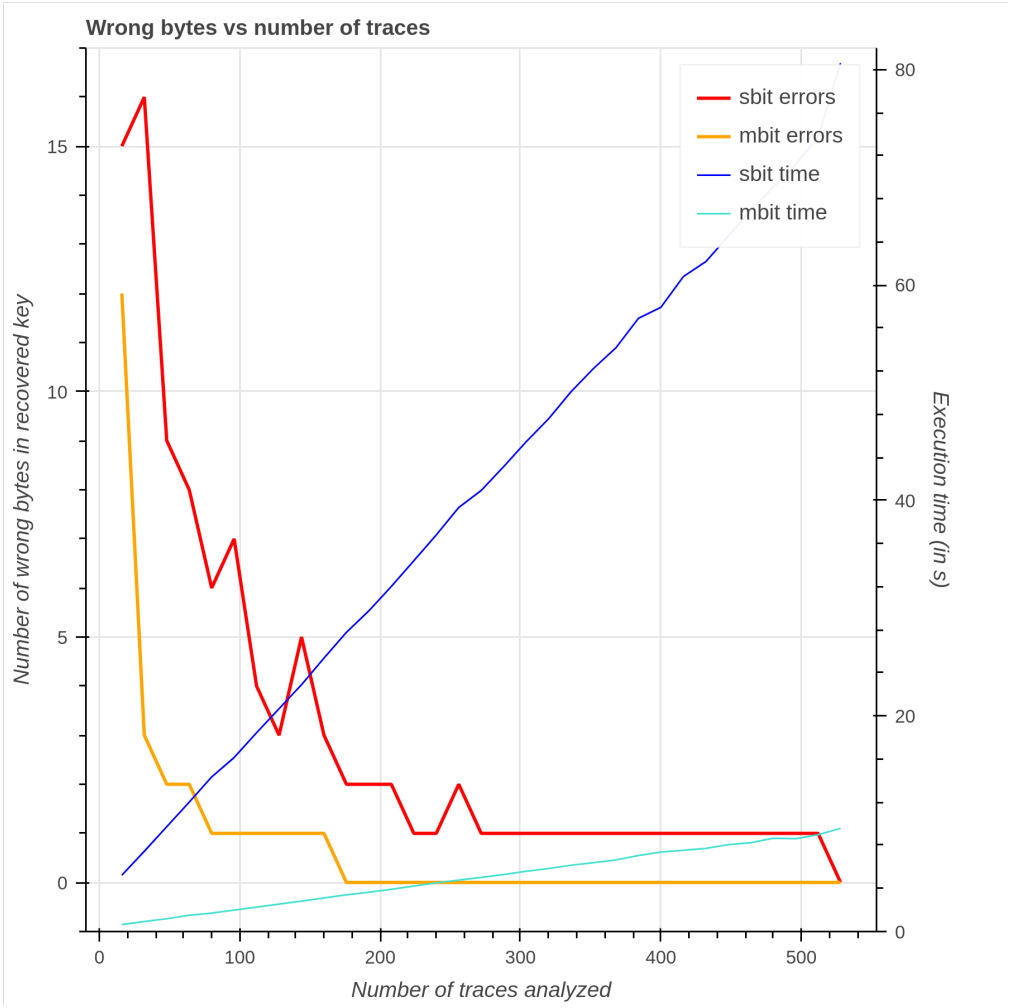
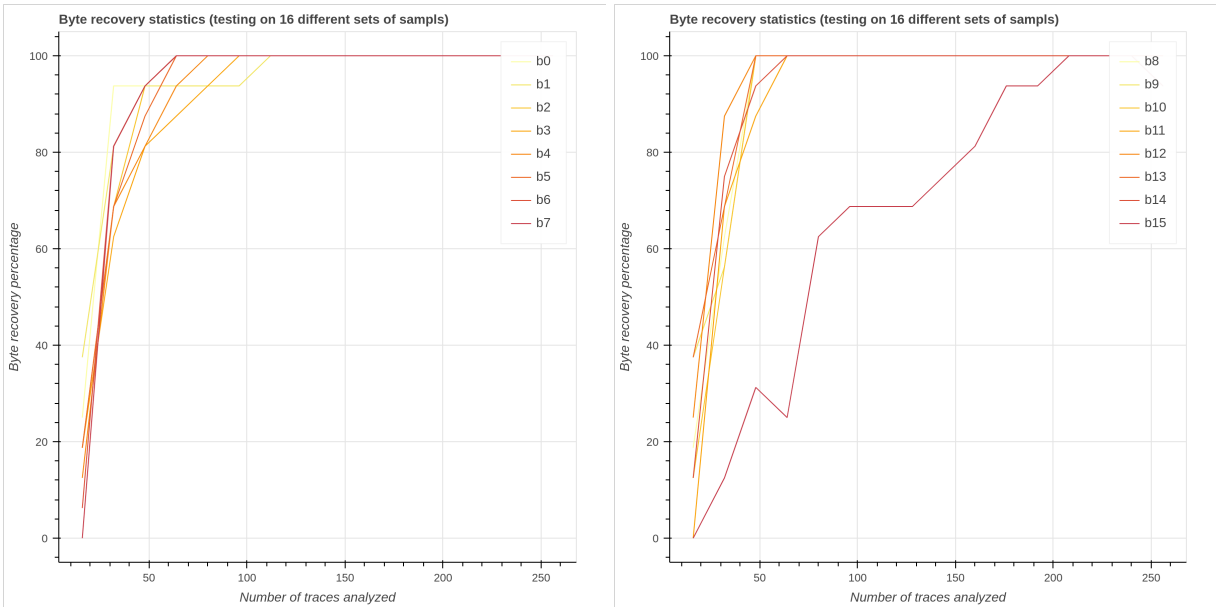


Figure 3: Single-bit vs. multi-bit efficiency comparison

On voit d'abord que l'attaque mono-bit est effectivement à peu près 8 fois plus lente que l'attaque multi-bit, et on observe que l'attaque mono-bit récupère moins d'octets corrects que l'attaque multi-bit, pour un même nombre de traces. De plus, l'attaque multi-bit récupère tous les octets corrects avec seulement 200 traces, alors que l'attaque multi-bit n'arrive toujours pas à récupérer tous les octets avec 512 traces (il reste 1 octet incorrect). Notons que l'attaque mono-bit parvient à récupérer tous les octets de cette clé en utilisant 576 traces, en environ 86 secondes.
L'attaque multi-bit est donc meilleure en tous points que l'attaque mono-bit, et parvient à casser la clé avec environ 200 samples en moins de 5 secondes !

Statistiques de succès en fonction de l'octet de clé attaqué

On veut ensuite calculer les statistiques de réussite de l'attaque pour chaque octet de la clé en fonction du nombre de traces attaquées. Pour cela, on enregistre 16 ensembles de traces capturés avec 16 clés différentes. On calculera uniquement les statistiques pour l'attaque multi-bit, l'exécution de l'attaque mono-bit étant trop lente. On obtient les deux graphiques suivants, par souci de lisibilité.



(a) (b)

Figure 4: (a) Bytes 0 to 7 (b) Bytes 8 to 15

On observe que pour tous les octets sauf l'octet en position 15, on a des statistiques de succès similaires. Seul l'octet en position 15 nécessite un nombre de traces significativement plus grand que les autres pour être récupéré à coup sûr.