Analyse et Développement Logiciel
Semestre 2

Gaétan BOIS–BAUMANN
Anaïs ESPICIER
Noah STAPLE

Encadrant : **Nicolas Aragon**

Master Cryptis
Université de Limoges

5 mai 2025

Table des matières

1 Introduction **2**

2 À propos du ChipWhisperer **2**

3 Attaques par canaux auxiliaires **2**
3.1 Attaques SPA sur RSA . . . 3
3.2 Attaques DPA sur AES . . . 4
3.3 Attaques CPA sur AES . . . 6
3.3.1 Attaque CPA sur firmware AES . . . 7
3.3.2 Attaque CPA sur AES 32 bits . . . 8
3.4 Attaques en conditions réelles . . . 9

4 Analyse d'un bootloader AES-256 **13**
4.1 Dans le cas du bootloader AES-256 . . . 13
4.2 Schéma . . . 13
4.3 Réponses possibles après envoi du firmware . . . 14
4.4 Informations sur le processus de déchiffrement . . . 14
4.5 Attaque sur le vecteur d'initialisation . . . 18
4.5.1 Principe de l'attaque . . . 19
4.6 Déchiffrement du firmware . . . 19
4.7 Conclusion . . . 20

5 Deep Learning-based Side-channel Analysis **21**
5.1 Travail réalisé . . . 21
5.2 Rappel des termes techniques pour le DL . . . 22
5.3 ASCAD . . . 22
5.4 Caractéristiques de la base de données . . . 23
5.4.1 ASCADv1 . . . 23
5.4.2 ASCADv2 . . . 23
5.5 Problèmes rencontrés . . . 24
5.6 Modèles proposés et cas de figure étudié . . . 25
5.6.1 Perceptron Multicouche (MLP) . . . 26
5.6.2 CNN . . . 29
5.6.3 Comparaison des différents modèles . . . 29

6 Conclusion **30**

Acronyms **32**

Glossaire **32**

Attaques par canaux auxiliaires et contre-mesures pour les chiffrements AES et RSA

Keywords : Attaques par canaux auxiliaires, AES, RSA, Attaques par analyse de consommation électrique, Chipwhisperer, Hiding, Masking, SPA, DPA, CPA, Rétro-ingénierie AES

1 Introduction

Ce rapport fait suite à l'état de l'art des attaques par canaux auxiliaires et des contre-mesures existantes réalisé lors du premier semestre. Nous cherchons ici à mettre en avant les implémentations réalisées lors de ce semestre tout en abordant également leurs fondements théoriques.

Nous commencerons tout d'abord par étudier et analyser les différentes attaques par canaux auxiliaires réalisées sur AES et RSA et pour lesquelles nous nous sommes basés principalement sur les labs au format Jupyter Notebooks proposés par Newaetech comme introduction à la technologie Chipwhisperer. En particulier, nous avons effectué l'analyse d'un bootloader AES-256 à l'aide de ces labs. Enfin, nous terminerons avec les attaques basées sur le deep learning.

2 À propos du ChipWhisperer

Une grande majorité des exercices, tests et labs ont été suivis à l'aide de la carte programmable **ChipWhisperer**. C'est l'outil le plus adapté à notre projet ADL. Le **ChipWhisperer** a été créé par Colin O'Flynn lors de ses recherches. dans la confection de sa thèse portant sur la sécurité des systèmes embarqués. Ses recherches ont permis d'apporter un standard de qualité dans l'analyse d'algorithmes. Le **ChipWhisperer** a rendu la recherche sur ce sujet beaucoup plus accessible en permettant aux chercheurs de travailler sans avoir à construire leur propre système. Le **ChipWhisperer** est aussi open-source, ce qui signifie qu'il est communautaire et que n'importe qui peut créer sa propre version.

Il faut noter que le **ChipWhisperer** existe en plusieurs versions, nos tests ont été effectués sur la version `ChipWhisperer-Lite XMEGA`.

3 Attaques par canaux auxiliaires

Dans cette section, nous aborderons en détail les attaques SPA, DPA et CPA sur AES et RSA ainsi que leurs fondements théoriques. Pour l'implémentation des attaques, nous nous sommes appuyés sur les sections suivantes des labs Chipwhisperer :

— SCA101 - Introduction, DPA & CPA sur AES, Chipwhisperer Analyzer, attaques CPA en pratique
— SCA201 - CPA sur AES 32 bits, CPA sur AES Hardware, attaque et reverse engineering sur AES-256 Bootloader

3.1 Attaques SPA sur RSA

Pour rappel, les attaques par ***Simple Power Analysis*** consistent à observer et analyser directement la trace de consommation électrique d'un appareil pendant qu'il effectue des opérations cryptographiques, dans le but de révéler des informations sur un ou plusieurs secrets de l'algorithme utilisé.

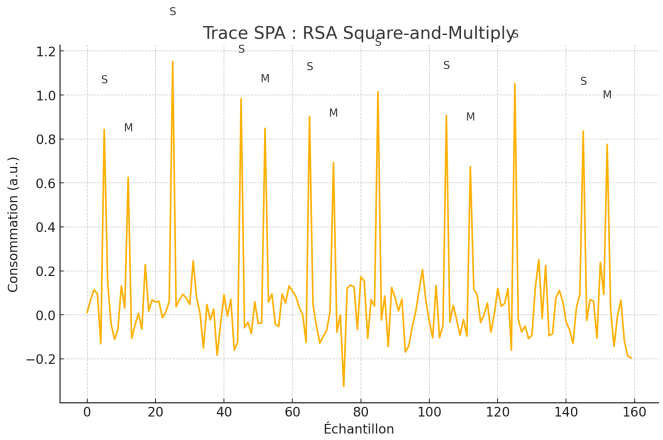


FIGURE 1 – Reconstitution de l'exposant secret RSA par attaque SPA

Nous avons essayé de réaliser une SPA sur RSA en utilisant Chipwhisperer sans utiliser les Jupyter labs. Cependant, nous n'avons pas été capables d'implémenter une version atomique de RSA, résistant à cela.

```
uint8_t square_and_multiply(uint8_t cmd, uint8_t scmd, uint8_t len, uint8_t *buf) {
  // Start actual measurement
  trigger_high();
  // temp
  volatile uint8_t x = 3;
  volatile uint8_t d[4] = {1, 1, 0, 0};
  volatile uint8_t n = 15;

  volatile uint8_t r0 = 1;
  volatile uint8_t r1 = x;
  volatile int i = 0;

  while (i < 4) { // Len(intToBinary(d=12))
    r0 = (r0 * r0) % n;
    if (d[i] == 1) {
      r0 = (r0 * r1) % n;
    }
    i++;
  }
  // Stop measurement
  trigger_low();

  // Send debug info
  uint8_t debug_buf[1] = { scmd };
  simpleserial_put('d', 1, debug_buf);

  // Return result
```

```
  uint8_t buff[1] = { r0 };
  simpleserial_put('r', 1, buff);

  //return r;
  return 0;
}
```

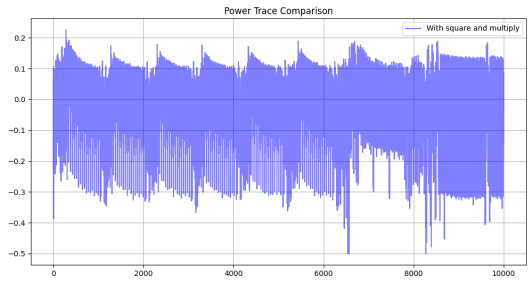


FIGURE 2 – Analyse de courant Square And Multiply (1100)

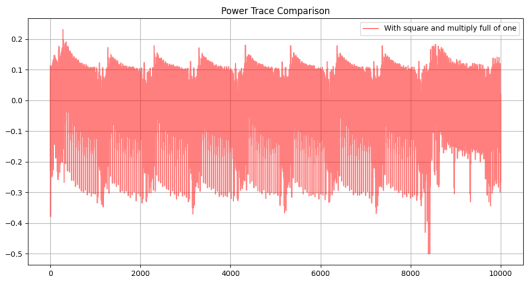


FIGURE 3 – Analyse de courant Square And Multiply (1111)

Nous pouvons observer qu'il y a 2 opérations de plus entre la figure de droite et celle de gauche. Cela est dû au nombre de 1 que contient le message dans la deuxième trace *(il y en a deux de plus)*.

Cela illustre le principe de base d'une attaque SPA.

3.2 Attaques DPA sur AES

Les attaques par **Analyse différentielle de la consommation d'énergie** (*Differential Power Analysis*) exploitent la corrélation entre la consommation d'énergie des dispositifs cryptographiques et les données : en effet, ceux-ci consomment plus ou moins d'énergie selon les opérations et les données traitées. Ainsi, en mesurant précisément la consommation électrique pendant de nombreuses opérations (ce qui nécessite donc un grand nombre de traces), puis en corrélant ces mesures à des hypothèses sur les clés (en utilisant la corrélation de Pearson par exemple), on peut retrouver petit à petit la clé secrète.

Nous avons réalisé dans les lab 3_1 : *Large Hamming Weigth Swings* et 3_2 : *Recovering AES Key from a Single Bit of Data* une attaque DPA classique sur AES en nous basant sur la variation du poids de Hamming (nombre de bits à 1) des données manipulées, puisqu'un poids de Hamming élevé consomme plus de puissance électrique qu'un poids de Hamming faible.

L'attaque se déroule en 4 étapes :

1. **Capture des traces** de consommation en forçant le premier octet du texte clair à `0x00` ou `0xFF` : cela permet de *mettre en évidence les variations de consommation*.

```
for i in trange(N, desc='Capturing traces'):
    scope.arm()
    if text[0] & 0x01:
        text[0] = 0xFF
    else:
        text[0] = 0x00
```

```
    target.simpleserial_write('p', text)
```

2. **Séparation des traces** en deux listes `one_list` et `zero_list` et selon les entrées (`0x00` ou `0xFF`)
3. **Débruitage des données** : la moyenne des traces de chaque groupe est calculée afin d'obtenir un signal plus clair en éliminant les valeurs aberrantes pour obtenir une trace équilibrée et représentative de l'ensemble des données

```
one_avg: float = np.mean(one_list, axis=0)
zero_avg: float = np.mean(zero_list, axis=0)
```

4. **Observation du résultat** : La différence entre `one_avg` et `zero_avg` met en évidence les variations de consommation dues aux différences de données puisque si l'on trace le graphe correspondant, un pic clair au début de la trace indique que l'opération manipulant le premier octet de l'entrée AES *génère une différence de consommation exploitable*.

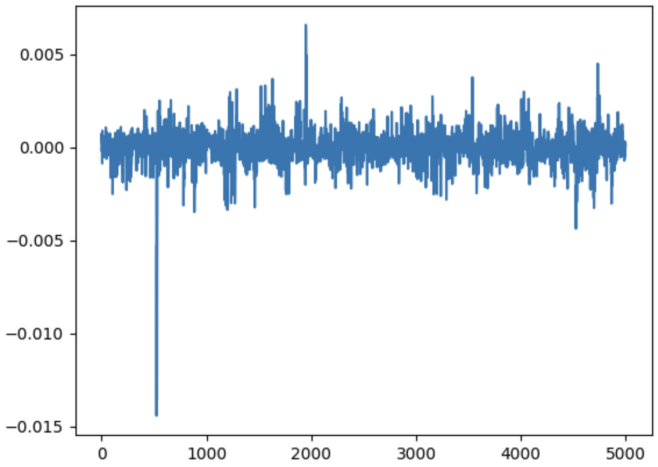


FIGURE 4 – Attaque DPA sur AES : pic visible au début de la trace

Maintenant, nous allons utiliser ce que nous venons d'apprendre sur la **corrélation entre les données manipulées et la consommation électrique** pour réaliser l'attaque sur AES à proprement parler.

On rappelle les bases du fonctionnement d'AES : on fait un `XOR` entre les données en entrée et la clé secrète, puis le résultat est passé au travers d'une `S-BOX`. On va supposer dans notre cas que c'est à la sortie de la S-BOX qu'a lieu la fuite de données. Grâce à notre attaque DPA, nous allons pouvoir reconstituer l'entièreté de la clé secrète simplement grâce à cette fuite d'informations.

Voici le déroulement de l'attaque, qui suit le schéma classique d'une attaque DPA :

— Pour **chaque octet** de la clé, on génère des **fuites hypothétiques** en essayant toutes les valeurs de clé possibles (`0x00` à `0xFF`)
— Pour un bit donné du résultat S-BOX, on réalise une **analyse de puissance** en séparant les traces en **deux groupes** et en calculant la moyenne de chaque groupe
— On effectue la différence entre la moyenne de chaque groupe
— Puis on cherche la **plus grande différence pour cette hypothèse de clé** puisque, plus la séparation est grande, plus il est probable que la clé soit correcte

— On retient l'hypothèse de clé avec la plus grande différence comme **valeur de clé secrète**

```
bytes_founded: list = []

for bytes_to_attack in range(16): #For each key byte
    mean_diffs = np.zeros(256)
    for key_bytes_guess_value in range(256): #For all possible bytes values
        one_list = []
        zero_list = []

        for trace_index in range(numtraces):
            input_bytes = textin_array[trace_index][bytes_to_attack]

            #Generates a hypothetical leakage : aes_internal(a,b) simulates the XOR and then the S - BOX for a  and b as inputs
            hypothetical_leakage = aes_internal(key_bytes_guess_value, input_bytes)

            #Splits the traces in two groups depending on the leakage bit value
            if hypothetical_leakage & 0x01:
                one_list.append(trace_array[trace_index])
            else:
                zero_list.append(trace_array[trace_index])

        zero_avg = np.mean(zero_list, axis=0)
        one_avg = np.mean(one_list, axis=0)


        max_diff_value = np.max(abs(one_avg - zero_avg))
        mean_diffs[key_bytes_guess_value] = max_diff_value

    sorted_list = np.argsort(mean_diffs)[::-1]
    #Print top 5 only

    bytes_founded.append(sorted_list[0])
    print(f":information: Bit {bytes_to_attack + 1}:  ({hex(sorted_list[0])}): {mean_diffs[sorted_list[0]]} => Difference with the second: {mean_diffs[sorted_list[0]] - mean_diffs[sorted_list[1]]} ({hex(sorted_list[1])}: {mean_diffs[sorted_list[1]]})")
```

3.3 Attaques CPA sur AES

Pour rappel, les attaques CPA, pour **Correlation Power Analysis**, exploite la consommation électrique d'un système pendant l'exécution d'un algorithme cryptographique avec comme objectif de retrouver une partie ou la totalité de la clé secrète, en analysant les fuites physiques. L'attaque se déroule en 4 étapes majeures :

1. **Capturer de nombreuses traces** de consommation pendant l'exécution du chiffrement (ex : AES), avec des textes clairs *connus*.
2. Faire des **hypothèses sur la clé** (ou les sous-clés) et **modéliser la consommation attendue** : modèle de fuite = poids de Hamming, distance de Hamming, etc...
3. **Comparer** la consommation prédite et réelle en calculant une **corrélation statistique** (ex : Coefficient de corrélation de Pearson).

4. La **meilleure corrélation** révèle la bonne hypothèse de clé.

Attention, à la différence des attaques DPA où l'analyse porte sur la **différence moyenne** entre deux groupes de traces, les attaques CPA reposent sur l'analyse de la **corrélation entre consommation réelle et prédite**.

3.3.1 Attaque CPA sur firmware AES

L'attaque réalisée ici est classique et utilise le poids de Hamming comme modèle de fuite, ce qui est très performant dans le cas d'AES. Elle se base sur le lab SCA101 lab 4_2 : CPA on Firmware Implementation of AES.

On commence par capturer 50 traces à partir de **textes clairs connus aléatoires**.

On modélise ensuite la consommation comme le **poids de Hamming** de la sortie `S-BOX`.

```python
sbox = [...]  # S-Box AES standard

def aes_internal(inputdata, key):
    return sbox[inputdata ^ key]

def calc_hamming_weight(n):
    return bin(n).count("1")

HW = [calc_hamming_weight(i) for i in range(256)]
```

L'attaque CPA repose sur le calcul du **coefficient de corrélation linéaire de Pearson**, dont la formule est la suivante :

$$r = \frac{cov(X,Y)}{\sigma_X \cdot \sigma_Y}$$

où X = données de consommation mesurées et Y = hypothèse de consommation théorique (poids de Hamming), avec cov la covariance associée aux données X et σ_X l'écart-type associé à X. De plus, on rappelle les formules suivantes :

$$cov(X,Y) = \sum_{n=1}^{N}[(Y_n - \overline{Y})(X_n - \overline{X})]$$

$$\sigma_X = \sqrt{\sum_{n=1}^{N}(X_n - \overline{X})^2}$$

où $\overline{X}$ correspond à la moyenne de X et $\overline{Y}$ à la moyenne de Y. On implémente donc ce calcul en Python en définissant les fonctions correspondant aux formules ci-dessus (et en prêtant attention aux axes de calcul) :

```python
def mean(X): # X_bar
    current_sum = np.sum(X, axis=0)
    return current_sum/len(X) # np.mean(X, axis=0) # Built-in way to do it

def std_dev(X, X_bar): # sigma_X
    return np.sqrt(np.sum(np.power(X - X_bar, 2), axis=0))

def cov(X, X_bar, Y, Y_bar): # covariance
    return np.sum((Y - Y_bar)*(X - X_bar), axis=0)
```

Puis on implémente l'attaque CPA en elle-même en suivant son déroulé classique et en utilisant les fonctions définies précédemment pour les calculs mathématiques :

```
t_bar = mean(trace_array)
o_t = std_dev(trace_array, t_bar)

cparefs = [0] * 16 # key byte guess correlations
bestguess = [0] * 16 # key byte guesses

for bnum in range(16): #for each key byte
    maxcpa = [0] * 256
    for kguess in range(0, 256): #for each possible byte value
        #calculates the hypothetical leakage for this value
        hws = np.array([[HW[aes_internal(textin[bnum],kguess)] for textin in
    textin_array]]).transpose()
        hws_bar = np.mean(hws)
        covar = cov(trace_array, t_bar, hws, hws_bar)
        cpaoutput = covar / (o_t * std_dev(hws, hws_bar))#correlation calculation
        maxcpa[kguess] = max(abs(cpaoutput))

    cparefs[bnum] = max(maxcpa)
    bestguess[bnum] = np.argmax(maxcpa) #key byte value = one with the best correlation
```

Cette attaque illustre bien la vulnérabilité d'une implémentation firmware d'AES sans contre-mesure puisque la clé entière est retrouvée à l'aide de seulement 50 traces ! Il est même possible de retrouver la clé avec encore moins de traces. Il suffit de modifier légèrement le code précédent afin qu'il teste l'attaque pour un nombre croissant de traces en s'arrêtant lorsque l'attaque réussit, i.e lorsque tous les octets de la clé sont retrouvés :

```
if (bestguess == key).all():
    key_bytes_guessed_for_i_traces[i-1] = len(key)
    return i
```

```
❌ Key not find with 32 trace(s). 15 bytes of the key found.
  0%|          | 0/16 [00:00<?, ?it/s]
✅ The minimum number of traces needed to recover the key is: 33
```

FIGURE 5 – Nombre de traces nécessaires pour une attaque CPA réussie

3.3.2 Attaque CPA sur AES 32 bits

L'attaque réalisée dans cette section est similaire à celle réalisée précédemment à la différence qu'elle est maintenant **optimisée pour les processeurs 32 bits**.

Sur une plateforme 32 bits, manipuler un seul octet à la fois (comme dans une implémentation "classique" 8 bits) sous-utilise la taille des registres. On peut donc gagner en vitesse en traitant quatre octets simultanément. On fusionne donc les opérations suivantes réalisées à chaque round (sauf le dernier) d'AES :

— SubBytes
— ShiftRows
— MixColumns

— AddRoundKey

en quatre tables de 256 entrées de 32 bits appelées **T-Tables** (T_0, T_1, T_2, T_3).

Ainsi, pour exécuter rapidement un round, au lieu d'appeler séparément les 4 opérations listées ci-dessus, on fait pour chaque mot de 32 bits :

```
#Pour la colonne j
d0 = T0[a0,j+0] ^ T1[a1,j+1] ^ T2[a2,j+2] ^ T3[a3,j+3];
rk = roundKey[j];
out = d0 ^ rk;
```

Soit 16 accès aux T-Tables + 16 `XOR` 32 bits par round, au lieu de nombreuses opérations 8 bits et déplacements mémoire.

La bibliothèque Chipwhisperer adaptée à ce format optimisé s'appelle *MBEDTLS*. Le modèle de fuite utilisé pour cette attaque est aussi légèrement différent de celui de la section précédente. En effet, nous continuons d'utiliser le poids de Hamming mais plutôt que de prendre seulement le poids de la S-Box, nous prenons cette fois le poids de Hamming de l'opération suivante qui est réalisée à chaque accès aux T-Tables :

$$h = hw[sbox[a]] + hw[sbox[a]] + hw[2 \times sbox[a]] + hw[3 \times sbox[a]]$$

```
# Pour forcer l'utilisation de MBEDTLS
SCOPETYPE = 'OPENADC'
PLATFORM = 'CWLITEARM'
VERSION = 'HARDWARE'
SS_VER = 'SS_VER_1_1'
CRYPTO_TARGET = 'MBEDTLS'
```

```
import chipwhisperer.analyzer as cwa

# run the CPA attack on the S-Box output
leak_model = cwa.leakage_models.sbox_output
attack = cwa.cpa(project, leak_model)
results = attack.run(cwa.get_jupyter_callback(attack))
```

3.4 Attaques en conditions réelles

En situation réelle, les traces de consommation capturées par le chipwhisperer ne sont pas parfaitement alignées comme celles utilisées ci-dessous et cela implique nécessairement une forte diminution de la corrélation entre les traces, or il s'agit de la base de nos attaques DPA et CPA.

En effet, l'attaque DPA compare des milliers de traces en alignant un évènement critique (ex : `S-BOX` d'AES) et observe des fuites subtiles, donc le jitter empêcherait complètement les effets statistiques (différences de consommation) de ressortir. De même, l'attaque CPA compare chaque point de trace entre elles donc si les opérations cryptographiques d'AES ne sont pas alignées, la corrélation est très faible et il est impossible de retrouver la clé.

Cela peut être dû à des **perturbations naturelles** qui peuvent introduire du décalage (***jitter***), comme par exemple :

— des **variations dans les horloges** du système attaqué ou de l'oscilloscope,

— des **temps de traitement aléatoires** : Si l'appareil ciblé utilise des interruptions, du multitâche ou des périphériques (comme UART), des délais peuvent être introduits de manière imprévisible,

— des **imprécisions du déclenchement** (*trigger*) : Le signal qui indique ≪ commence à enregistrer ≫ n'est jamais parfaitement aligné au cycle près.

Le résultat est que d'une trace à l'autre, l'opération critique pour notre attaque (dans notre cas, l'application de la S-BOX dont on observe la sortie) ne commence pas exactement au même moment pour toutes les traces. Cependant, le jitter est souvent introduit de manière intentionnelle comme **contre-mesure** afin de protéger le système contre les attaques par canaux auxiliaires présentées précédemment. Par exemple, les méthodes suivantes permettent de casser l'alignement entre les traces de consommation :

— **Ajout d'une boucle** for qui tourne un nombre variable de fois avant AES (SCA201 ¿ Lab 1_1A - Resynchronizing Traces with Sum of Absolute Difference)
— Insertion de **délais aléatoires** programmés avant ou pendant le chiffrement

Comemnt remédier à ces pertubations afin de pouvoir réaliser nos attaques malgré tout ? Les principales méthodes existantes sont :

— **Sum of Absolute Differences (SAD)** : glisser une portion d'une trace de référence sur les autres traces et calculer la somme des différences absolues
— **Cross-Correlation** : glisser une trace par rapport à une autre et calculer leur corrélation
— **Pattern Matching** (Matching sur des caractéristiques spécifiques) : détecter une signature spécifique dans la trace (par exemple la montée de courant quand l'AES démarre)
— **Machine Learning** (alignement automatique) : entraîner un modèle pour reconnaître des parties caractéristiques des traces malgré le bruit et les déplacements, par exemple le *Dynamic Time Warping* (DTW)

Sum of Absolute Differences

Détaillons la méthode SAD : **Somme des Différences Absolues** présentée dans le lab 1_1A - Resynchronizing Traces with Sum of Absolute Difference. Cette méthode consiste à trouver **l'alignement optimal** entre deux signaux en se basant sur les différences entre ceux-ci. Elle se déroule en 3 étapes décrites ci-après :

1. Choisir une petite portion représentative d'une trace bien choisie comme référence

   La trace choisie doit être relativement unique, pour que le motif soit assez reconnaissable pour que l'alignement se fasse au bon endroit. De plus, le décalage maximal que nous pouvons obtenir est limité par la proximité de la portion de référence par rapport aux bords de la trace de consommation, elle doit donc être relativement éloignée des bords.

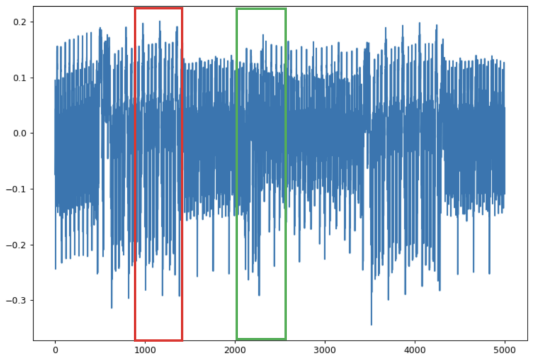


FIGURE 6 – en vert : bon choix de référence, en rouge : choix non judicieux

2. Faire glisser cette référence sur une autre trace

```python
def get_sad_plot(ref_trace, target_trace):
    ref_len = len(ref_trace)
    sads = []
    for i in range(0, len(target_trace) - ref_len):
        sads.append(np.sum(abs(ref_trace-target_trace[i:i+ref_len])))
    return sads

cw.plot(get_sad_plot(ref_trace, proj.waves[1]))
```

3. À chaque décalage (offset), on calcule la somme des différences absolues entre la référence et la portion correspondante de la trace cible (voir formule ci-dessous) : le meilleur alignement est celui où la **SAD est minimale** !

$$SAD(offset) = \sum_{i=0}^{L-1} |ref[i] - target[i + offset]|$$

où L = longueur de la fenêtre de référence, $ref[i]$ = i-ème point de la référence, $target[i+offset]$ = i-ème point de la trace décalée.

On calcule la SAD pour tous les décalages possibles, et on choisit celui qui donne la plus petite SAD.

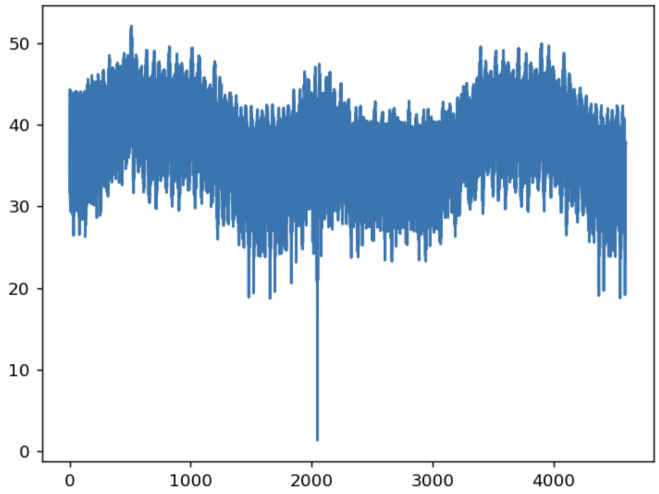


FIGURE 7 – Calcul de la SAD pour chaque point entre la trace et la référence : point minimum clairement visible

```python
def calculate_trace_offset(ref_trace, orig_offset, target_trace, threshold):
    ref_len = len(ref_trace)
    for i in range(0, len(target_trace) - ref_len):
        SAD_SUM = np.sum(abs(ref_trace - target_trace[i:i+ref_len]))
        if SAD_SUM < threshold:
            return i - orig_offset
```

Dynamic Time Warping

Abordons maintenant une méthode bien plus efficace que la SAD comme contre-mesure au jitter afin de performer nos attaques. En effet, la méthode SAD permet de corriger des petits décalages constants, mais dans le cas du lab SCA201 1_1B : *Resynchronizing Traces with Dynamic Time Warp*, l'appareil introduit du jitter pendant le chiffrement AES lui-même, ce qui produit des traces très perturbées difficiles à réaligner.

Essayons de réaliser une attaque CPA directe avec LASCAR (*Lightweight and Scalable Side Channel Analysis in Python*, une bibliothèque Python dédiée à l'analyse de traces de consommation pour les SCA) :

```python
from lascar import *
cw_container = CWContainer(proj, proj.textins)
cpa_engines = [CpaEngine("cpa_%02d" % i, sbox_HW_gen(i), range(256)) for i in range(16)]
session = Session(cw_container, engines=cpa_engines).run(batch_size=50)
```

Sans surprise, la clé n'est pas retrouvée. Pourquoi cela ? Tout simplement parce que la méthode SAD ne décale qu'une trace dans son ensemble, alors que dans notre cas le jitter est variable à l'intérieur même de l'exécution AES. Autrement dit, il nous faut une méthode qui permet de réaligner des traces **point par point**.

Le Dynamic Time Warping est un algorithme classique de traitement du signal qui permet d'aligner deux signaux similaires mais déformés dans le temps, ce que nous avons simulé ici en introduisant du

jitter aléatoire dans les traces de consommation. Il est utilisé en reconnaissance vocale, bio-informatique, et dans notre cas en analyse de consommation.

Le principe est le suivant : Construire un **chemin d'alignement optimal** entre deux signaux, le critère d'optimalité étant la minimisation de la **distance cumulée** entre points correspondants. Cependant, contrairement à la méthode SAD, DTW autorise des **compressions/étirements temporels locaux**.

ChipWhisperer propose un préprocesseur basé directement sur DTW (ou sa version accélérée ***FasterDTW***). En effet, DTW est un algorithme performant mais très lent (complexité quadratique), surtout si le radius, paramètre essentiel définissant à quel point on autorise l'algorithme à s'écarter du chemin direct, est élevé. En pratique, on a donc tendance à utiliser FasterDTW qui est une approximation efficace de DTW.

```
import chipwhisperer.analyzer as cwa

resync_traces = cwa.preprocessing.ResyncDTW(proj)
% j'ai du enlever deux lignes ici car ca compilait pas sinon
resync_analyzer = resync_traces.preprocess()
```

Le code ci-dessus crée un nouvel ensemble de traces réalignées, adapté à une attaque CPA !

4 Analyse d'un bootloader AES-256

Cette section reprend les explications et montre les solutions que nous avons apportés pour le jupyter notebook `Lab 3_1B - Reverse Engineering on the AES256 Bootloader`. Ce document montre comment il est possible d'utiliser les attaques par canaux auxiliaires afin de faire la rétro-ingénierie d'un bootloader utilisant AES.

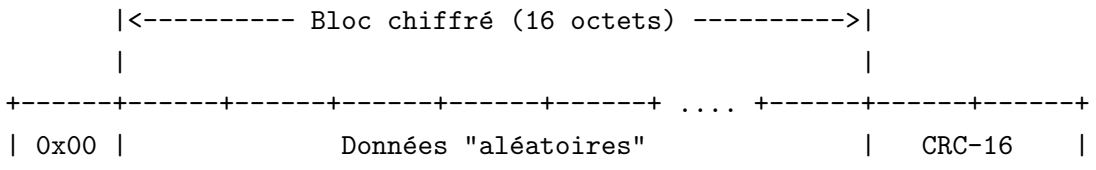
Un bootloader est un petit morceau de code qui permet de pouvoir mettre à jour le programme *(firmware)* qui est en exécution sur un microcontrôleur. Certains fabricants chiffrent ce firmware pour que seulement des firmware "**vérifiés**" soient acceptés par le bootloader.

4.1 Dans le cas du bootloader AES-256

Nous possédons quelques information préliminaires concernant ce bootloader

- Vitesse de communication : 38400bps
- À l'envoi, le premier octet est toujours un zéro
- Sur le BUS USB il y a 16 octets qui ont l'air aléatoire *(c'est sûrement une partie chiffrée)*
- Chaque message contient deux octets de cyclic redundancy check (CRC)
- Pas de répétition dans le message chiffré

4.2 Schéma



```
+------+------+------+------+------+------+ .... +------+------+------+
```

4.3 Réponses possibles après envoi du firmware

— `0xA4` → `Mauvais CRC`
— `0xA1` → `Succès`

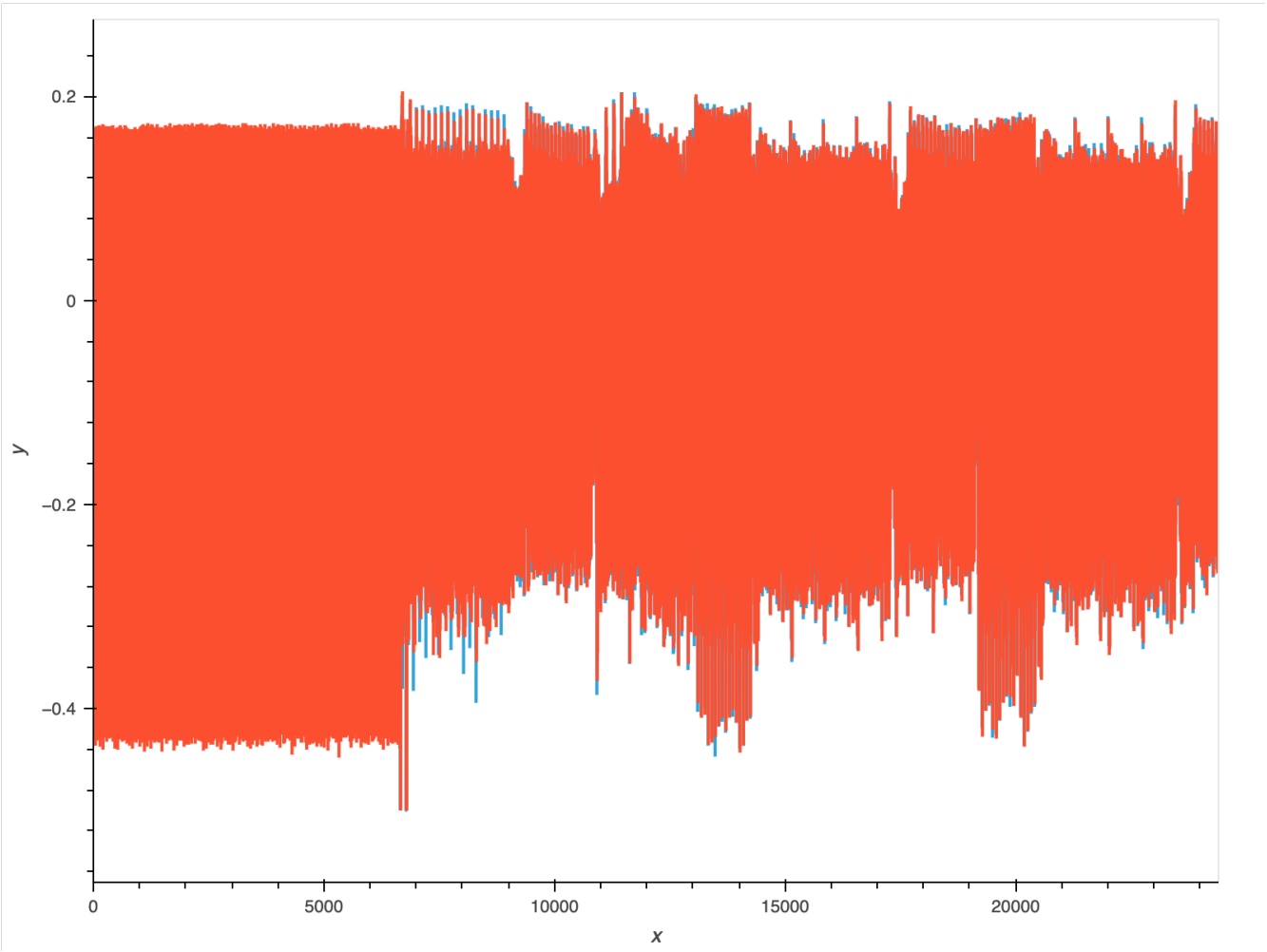


FIGURE 8 – Trace de courant : Processus de déchiffrement

4.4 Informations sur le processus de déchiffrement

— Les données sont **directement déchiffrées**, il n'y a **pas de pre-traitement des données** ;
— Les opérations se répètent plusieurs fois ;
  — On émet comme **hypothèse** qu'il s'agit d'AES-128 et que ses opérations sont probablement des **rondes** d'AES.
— On ne voit pas tout le processus de déchiffrement ;
— Il y a sûrement du jitter
  — Il faut donc **resynchroniser** les traces.

En changeant le **decimate** *(4.4)* à 10, on obtient :

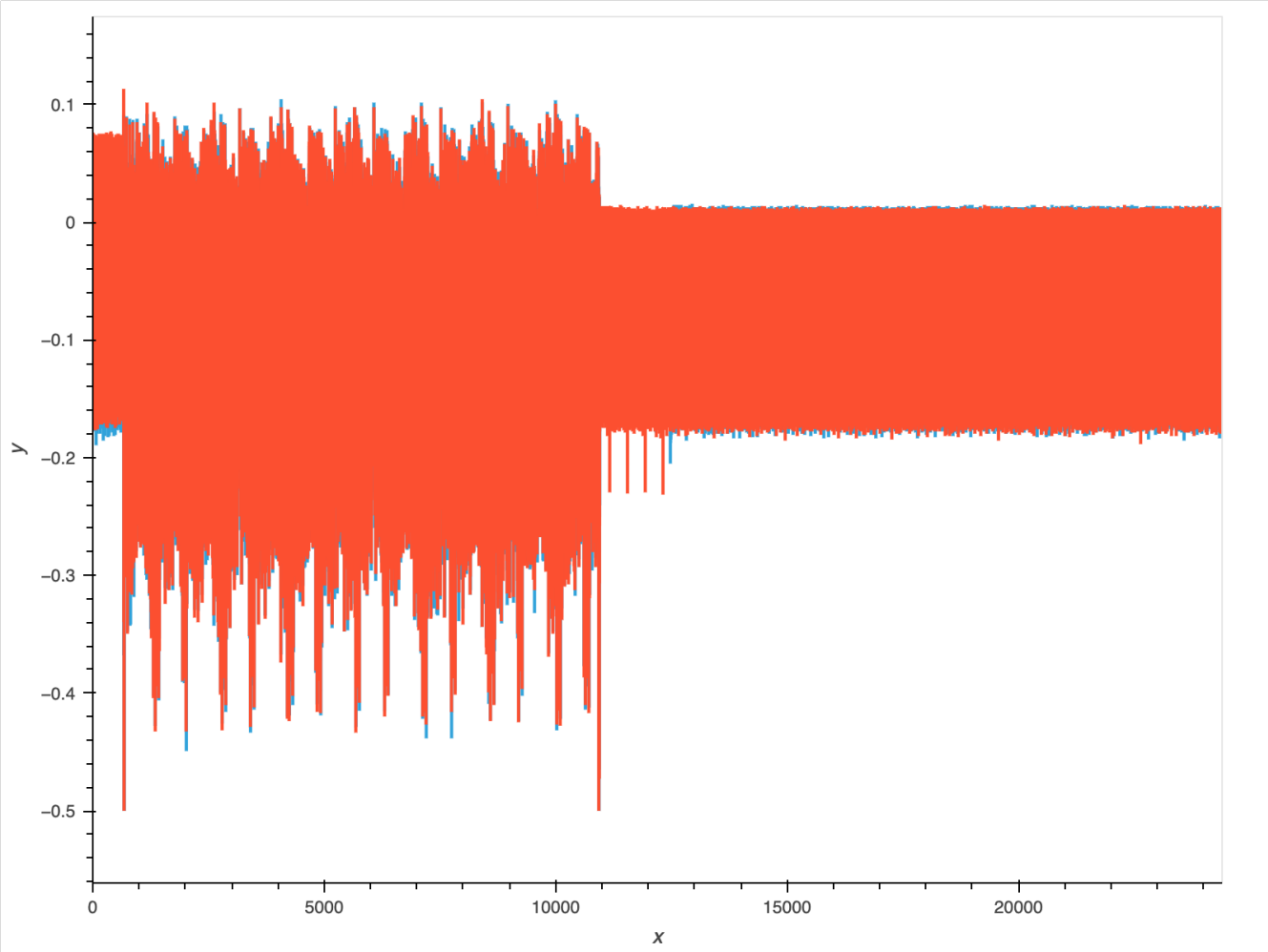


FIGURE 9 – Trace de courant : Processus de déchiffrement *(decimate* 10*)*

Information [1]

Decimate contrôle à quelle vitesse le chipwhisperer va enregistrer des valeurs basés sur la vitesse de l'horloge.

Par exemple, pour une valeur **decimate** de 10, le chipwhisperer va prendre des mesures à $1/10$ème de la vitesse de l'horloge. C'est très utile pour mieux visualiser quand les opérations sont longues.

```
scope.adc.decimate = 10 # try to get the full encryption in a single trace, then set
    back to 1
```

On peut voir sur la figure 9 13 **rondes** de **MixColumns** ce qui indique que nous ne sommes pas sur du **AES-128** mais **AES-256**. Ce qui signifie que notre hypothèse faite précédemment *(4.4)* est fausse.

Cela ne change pas grand-chose pour notre attaque, on devra simplement répéter notre attaque **CPA** deux fois : Une fois pour avoir la première moitié de la clé, une autre pour la seconde moitié.

Nous avons remarqué que le texte chiffré ne se répète pas : Cela veut dire que le bootloader utilise une version **avancée** du chiffrement **AES-256**. Cela a plusieurs implications :

— Notre attaque CPA pourrait ne pas être efficace.

— Nous devons savoir quel type est utilisé, si c'est un mode comme *GCM*, *CTR* ou *CBC*.

On a remarqué qu'au démarrage de l'appareil, aucune opération de chiffrement n'est faite. Si le bootloader utilisait une version GCM par exemple, elle aurait eu besoin de faire des opérations au démarrage comme le calcul du **counter**. Cela implique que la version **AES-256** utilisée est probablement assez basique, on va **supposer** pour le moment qu'une attaque simple pourrait fonctionner.

Après avoir supprimé le jitter en utilisant l'algorithme de **SAD** *(3.4)*, nous avons lancé notre **attaque par corrélation**.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
PGE=	215	174	248	99	117	36	67	141	87	225	84	247	217	149	132	182
0	EA 0.746	79 0.700	79 0.585	20 0.756	C8 0.634	71 0.627	44 0.583	7D 0.552	46 0.621	62 0.639	5F 0.602	51 0.654	85 0.592	C1 0.645	3B 0.558	CB 0.670
1	32 0.463	5D 0.457	8C 0.459	D1 0.479	A2 0.485	B3 0.501	42 0.461	A6 0.495	6A 0.464	4B 0.486	88 0.466	DF 0.485	9C 0.480	EE 0.507	CF 0.461	93 0.499
2	4D 0.457	C7 0.455	6D 0.445	F9 0.466	B6 0.479	D8 0.493	05 0.445	BA 0.472	61 0.462	8B 0.466	C0 0.463	26 0.475	14 0.468	5B 0.457	A0 0.442	D0 0.478
3	33 0.452	A8 0.423	B9 0.442	B3 0.458	7A 0.464	A9 0.486	18 0.439	D3 0.464	08 0.439	6D 0.457	BF 0.459	7E 0.449	8E 0.449	CA 0.442	29 0.442	F5 0.461
4	4B 0.451	34 0.421	F2 0.441	5E 0.456	B2 0.454	6E 0.469	3E 0.436	36 0.450	EC 0.433	F8 0.456	28 0.458	4D 0.446	37 0.437	E0 0.442	C7 0.438	68 0.453

FIGURE 10 – Résultat de l'attaque par corrélation

Le résultat montre que la valeur de corrélation pour la première clé potentielle est bien supérieure à celle de la deuxième clé potentielle, ce qui démontre que l'attaque a bien fonctionné. Notre première partie de clé recouvrée est `[0xEA, 0x79, 0x79, 0x20, 0xC8, 0x71, 0x44, 0x7D, 0x46, 0x62, 0x5F, 0x51, 0x85,0xC1, 0x3B, 0xCB]`

Information

Cela nous donne deux informations

— Le bootloader déchiffre correctement les données

— L'algorithme de chiffrement utilisé est bien **AES**

Maintenant que nous avons récupéré la clé de **Ronde 14**, il faut être en capacité de retrouver la seconde moitié de la clé. Afin d'y arriver, nous allons devoir comprendre le fonctionnement d'**AES-256** comparé à **AES-128** :

— **AES-256** utilise une clé deux fois plus grande

— **AES-256** se compose de 14 rondes au lieu de 10 pour **AES-128**

Vu que l'on a fait une première attaque **CPA** pour récupérer la clé à la 14ème ronde (K_{14}), nous allons l'utiliser comme étape intermédiaire pour récupérer la clé à la ronde 13 (K_{13}).

Nos ordres d'opérations ressemblent à ça :

— `AddRoundKey` (avec K_{14})
— `InvShiftRows`
— `InvSubBytes`
— `AddRoundKey` (avec K_{13})
— `InvMixColumns`
— ...

Seulement, l'opération **InvMixColumns** complique l'attaque car elle combine 4 octets ensemble. Une attaque directe nécessiterait de tester 2^{32} possibilités *(4 milliards)*. Une solution est de se reposer sur la propriété de **linéarité** de l'opération **MixColumns** :

$$\texttt{MixColumns}(A + B) = \texttt{MixColumns}(A) + \texttt{MixColumns}(B) \tag{1}$$

Ce qui nous permet d'au lieu de faire ceci :

$$X_{13} = \texttt{InvSubBytes}(\texttt{InvMixColumns}(\texttt{InvShiftRows}(K_{14} \oplus K_{13}))) \tag{2}$$

Nous permet de décomposer l'opération de la manière suivante :

$$X_{13} = \texttt{InvSubBytes}(\texttt{InvMixColumns}(\texttt{InvShiftRows}(K_{14})) \oplus \texttt{InvMixColumns}(\texttt{InvShiftRows}(K_{13}))) \tag{3}$$

On introduit une clé intermédiaire $K_{13'} = \texttt{InvMixColumns}(\texttt{InvShiftRows}(K_{13}))$ Puis on récupère $K_{13} = \texttt{InvSubBytes}(\texttt{InvMixColumns}(\texttt{InvShiftRows}(K_{14})) \oplus K_{13'})$

Cette astuce nous permet de faire passer la complexité de `InvMixColumns` de 2^{32} à 2^8.

Maintenant que cette étape n'est plus un problème, nous pouvons récupérer la deuxième partie de la clé !

On est donc maintenant certain que le bootloader utilise **AES-256** pour chiffrer son firmware. L'objectif maintenant est de détecter quel type de bloc est utilisé, pour ce faire, nous allons essayer de déchiffrer le contenu du firmware avec la clé récupérée :

```
from Crypto.Cipher import AES
cipher = AES.new(bytes(btldr_key), AES.MODE_ECB)
print(bytearray(cipher.decrypt(encrypted_firmware[:16])))
# Sortie: bytearray(b'lY\x98\x96Zr\xf4\xd7\x9dI\x97\x9e\xe7l?\x92')
print(bytearray(cipher.decrypt(encrypted_firmware[16:32])))
# Sortie: bytearray(b"\xb7\xa4\x10M\x91\xea\xf2\xf8'\xab\xc7y\xa4\xd5p\x8d")
```

Comme on peut l'observer, nos données n'ont pas de sens particulier. On peut donc éliminer l'hypothèse que le bootloader utilise un chaînage basique **ECB**, car la clé recupéré ne nous permet pas dirèctement de déchiffrer le firmware. Afin de trouver le bon mode, regardons à quoi ressemble la fin de l'opération de chiffrement, sur notre trace de consommation électrique :

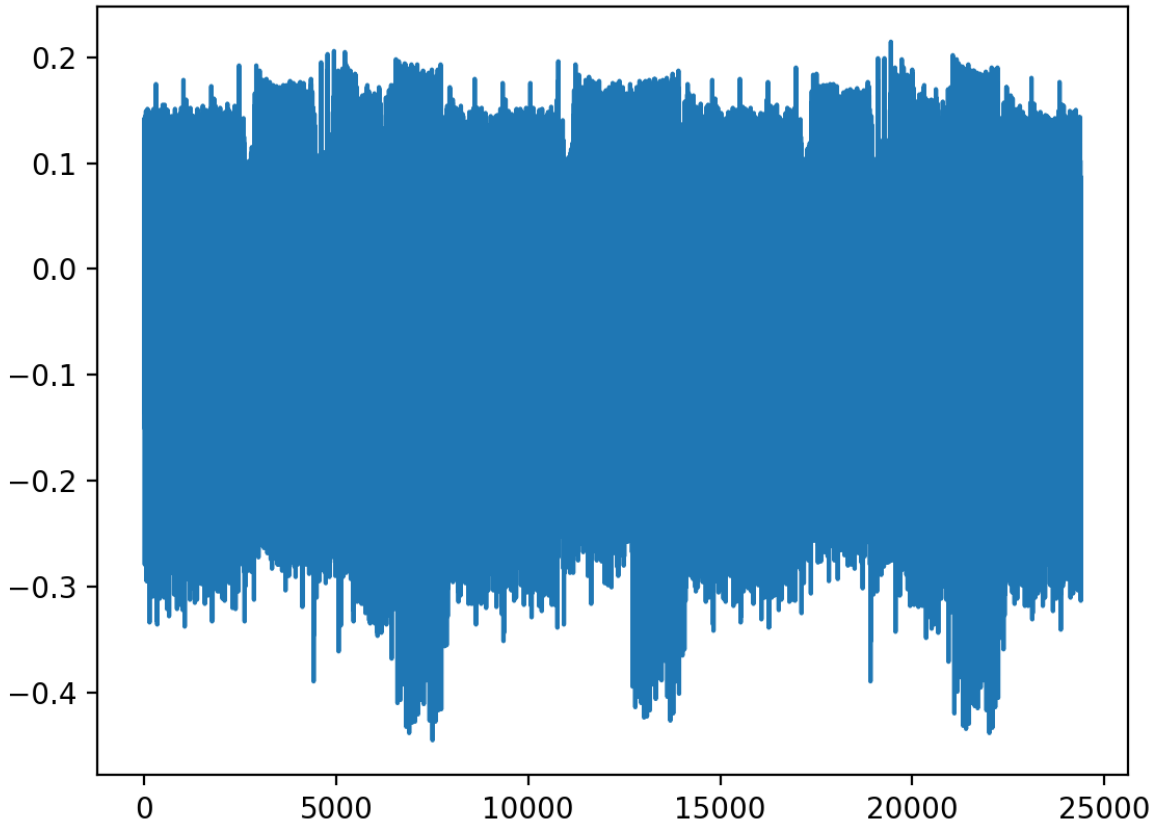


FIGURE 11 – Trace de consommation électrique à la fin de l'opération

Les chercheurs à l'origine de ce lab nous annoncent la chose suivante :

> *It might be hard to pick out, but you should be able to find* 16 `XOR` *short operations just after the end of the last round of AES. It might be using CBC mode, which means it'll be using the ciphertext as part of the encryption and decryption of subsequent blocks.*

Effectivement les 16 opérations XOR sont caractéristiques du mode de chaînage CBC :

$$\texttt{Texteclair} = \text{AES}^{-1}(\texttt{BlocActuel}) \oplus \texttt{BlocPrécédent} \tag{4}$$

On peut donc partir sur l'hypothèse que le bootloader utilise **AES-256** en mode **CBC**.

4.5 Attaque sur le vecteur d'initialisation

Le **bootloader** utiliserait donc le vecteur IV de la manière suivante :

$$PT = DR \oplus IV \tag{5}$$

Où $DR = \text{AES}^{-1}(CT)$.

Pour pouvoir retrouver le texte clair, il nous faut retrouver ce vecteur, pour ce faire, nous allons procéder à une attaque **DPA**.

4.5.1 Principe de l'attaque

L'attaque consiste à récupérer l'IV bit par bit en analysant les différences dans les traces de consommation électrique. Voici comment on va procéder :

Exemple sur le premier bit :

- **Séparation des traces en deux groupes** :
  - Groupe 0 : Traces où `(DR[0] & 0x01) = 0` (premier bit de DR est 0)
  - Groupe 1 : Traces où `(DR[0] & 0x01) = 1` (premier bit de DR est 1)
- **Calcul des moyennes** :
  - Calculer la trace moyenne pour le Groupe 0
  - Calculer la trace moyenne pour le Groupe 1
- **Analyse des différences** :
  - Calculer la différence entre les deux moyennes (`Moyenne_Groupe1 - Moyenne_Groupe0`)
  - Cette différence fera apparaître un pic à l'endroit précis où l'opération $\oplus$ est effectuée
- **Détermination de la valeur du bit d'**IV :
  - Si le pic est positif : le bit d'IV est probablement 0
    - Car si $IV[bit] = 0$, alors $PT[bit] = DR[bit]$, donc la consommation suit DR directement
  - Si le pic est négatif : le bit d'IV est probablement 1
    - Car si $IV[bit] = 1$, alors $PT[bit] = \neg DR[bit]$, donc la consommation est inversée par rapport à DR

Une fois avoir réalisé l'attaque pour un seul bit, on peut l'automatiser pour les 16 octets du vecteur :

```
Pour chaque octet de 0 à 15:
    Pour chaque bit de 0 à 7:
        Séparer les traces selon (DR[octet] & (1 << bit))
        Calculer les moyennes des deux groupes
        Analyser la différence pour déterminer la valeur du bit dans l'IV
```

4.6 Déchiffrement du firmware

Nous possédons maintenant tous les secrets nécessaires afin de déchiffrer le firmware, il ne reste plus qu'à tester :

```python
# Pour le premier bloc uniquement
cipher = AES.new(bytes(btldr_key), AES.MODE_ECB)  # On utilise ECB car on implemente CBC manuellement
first_pt = cipher.decrypt(encrypted_firmware[:16])
first_pt = [first_pt[i] ^ btldr_IV[i] for i in range(16)]
```

```python
def decrypt_firmware(encrypted_data, aes_key, iv):
    cipher = AES.new(bytes(aes_key), AES.MODE_ECB)
    plaintext = []
```

```
    # Premier bloc avec IV
    block = cipher.decrypt(encrypted_data[:16])
    for i in range(16):
        plaintext.append(block[i] ^ iv[i])

    # Blocs suivants
    for block_index in range(1, len(encrypted_data) // 16):
        start = block_index * 16
        end = start + 16
        previous_block = encrypted_data[start-16:start]
        current_block = encrypted_data[start:end]

        decrypted = cipher.decrypt(current_block)
        for i in range(16):
            plaintext.append(decrypted[i] ^ previous_block[i])

    return plaintext
```

Pour rappel voici le mode de fonctionnement de **AES** en mode **CBC**

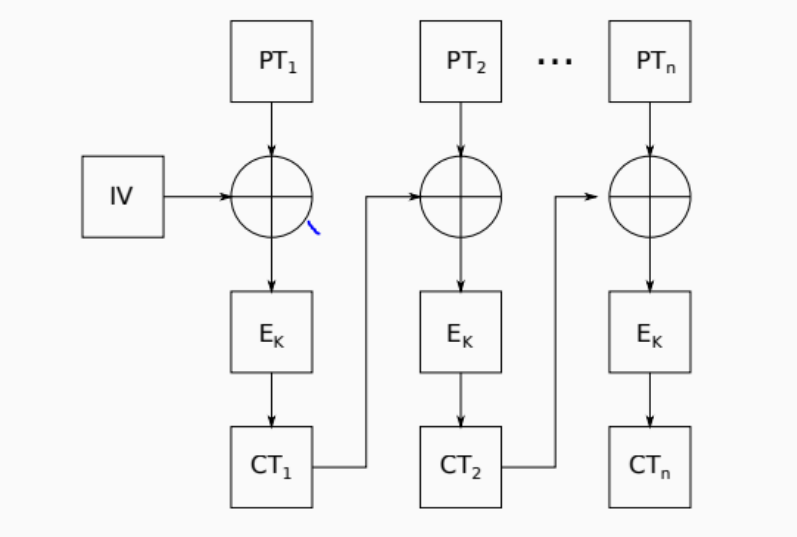


FIGURE 12 – Mode de fonctionnement AES en mode CBC

En déchiffrant le bootloader, on peut y trouver une en-tête contenant la chaîne de caractères suivantes : `deadbeefaabbccddeeff0011`. Elle représente probablement une **signature** utilisée pour vérifier l'**authenticité** du firmware.

4.7 Conclusion

En conclusion, nous avons pu appliquer les techniques d'attaques par canaux auxiliaires dans un environnement assez réaliste : Un bootloader qui utilise du chiffrement pour protéger la mise à jour de firmware est courant, comme le fait d'avoir certaines contraintes comme le jitter qui rendent l'attaque plus complexe. Enfin, nous avons pu voir comment les attaques **CPA** et **DPA** ont pu être utilisées pour récupérer des secrets d'un système qui pouvait paraître 'sécurisé'.

5 Deep Learning-based Side-channel Analysis

L'Intelligence Artificielle (IA) est un nouvel aspect dans les attaques par canaux auxiliaires avec profilage, en particulier pour étudier la consommation électromagnétique et électrique. L'utilisation du Machine Learning et du Deep Learning a permis la mise en place d'attaques plus performantes, pouvant être utilisées sur des mesures à très haute dimension et qui sont résistantes à la déformation du signal comme la gigue *(pour les CNN)* [2].

Pendant ce semestre, nous avons décidé de nous concentrer sur les attaques de canaux auxiliaires utilisant le DL. Le papier "Study of Deep Learning Techniques for Side-Channel Analysis and Introduction to ASCAD Database" [2] a permis un rapide développement des méthodes de DL, grâce à la mise à disposition d'une base de données utilisée pour réaliser des benchmarks, afin de comparer l'ensemble des modèles proposés dans les différents papiers. De plus, quelques modèles de base sont fournis, afin de tester la base de données et de démontrer l'efficacité du DL pour les attaques par canaux auxiliaires avec profilage.

Ce semestre, nous nous sommes concentrés sur la compréhension et l'utilisation de la base de données ASCAD, car elle est un élément central dans le développement des attaques par canaux auxiliaires utilisant du DL. Nous ne parlerons pas des différents datasets proposés lors des conférences Conference on Cryptographic Hardware and Embedded Systems (CHES) depuis 2015 qui ont également joué un rôle important pour le développement de nouveaux modèles et de nos connaissances dans cette branche des attaques par canaux auxiliaires, car nous avions également envie de consacrer du temps ce semestre pour découvrir d'autres aspects des attaques par canaux auxiliaires.

5.1 Travail réalisé

L'objectif de cette section est à partir du papier "A Practical Tutorial on Deep Learning-based Side-channel Analysis" [3] de réaliser des labs au format cours, mélangeant théorique et pratique, afin d'avoir une bonne compréhension du sujet.

Pour faire cela, nous avons créé trois Jupyter lab :

1. **Introduction_Setup :** Ce lab est divisé en trois grandes parties :
   — Introduction aux attaques par canaux auxiliaires pour AES : Présentation des attaques par canaux auxiliaires et d'AES ;
   — Introduction au Deep Learning : Présentation des CNN et MLP, avec la librairie Tensorflow ;
   — DLSCA : Définition du DLSCA et explication de comment intéragir avec la base de données ASCAD.
2. **Exploration_Donnees_Et_Preprocessing :** Dans ce lab, nous allons montrer le format des données utilisées et comment réaliser un prétraitement sur celles-ci. Nous aborderons également comment faire de l'augmentation des données et de la sélection et du ré-échantillonnage des caractéristiques.
3. **Attaques_DLSCA_Et_Ameliorations :** Enfin, pour le dernier lab, une fois nos données pré-traité, nous pouvons passer à la phase d'attaque. Celle-ci se divise en deux parties :

- Des attaques avec deux modèles MLP et un modèle CNN avec des paramètres données ;
- Explication sur la manière d'améliorer ses paramètres et modèles :
  - *Explicabilité :* Présentation de techniques de visualisation basique, permettant de savoir quels sont les points d'intérêt dans la trace qui contribuent à une bonne prédiction dans le réseau de neurones.
  - *Recherche des hyperparamètres*
  - *Méthodes d'Ensemble :* Présentation de l'utilisation de cette technique combinant plusieurs algorithmes d'apprentissage pour obtenir de meilleurs résultats prédictifs.

5.2 Rappel des termes techniques pour le DL

Dans cette partie, nous allons donner la définition de certains mots-clés utilisés dans l'apprentissage profond, afin d'avoir des bases communes pour la suite de cette section.

Definition 5.1 (Epoch) *Dans le contexte de l'entraînement d'un modèle, l'epoch est un terme utilisé pour référer à une itération où le modèle voit tout le training set pour mettre à jour ses coefficients [4].*

Definition 5.2 (Lot *(Batch)*) *Ensemble d'exemples utilisés dans une itération d'entraînement. La taille du lot (batch) détermine le nombre d'exemples dans un lot [5].*

Definition 5.3 (Réseaux de neurones convolutifs) *Réseaux de neurones utilisant des données tridimensionnelles pour les tâches de classification d'images et de reconnaissance d'objets. Ils se distinguent des autres réseaux neuronaux par leurs performances supérieures avec des entrées de signaux d'image, de parole ou audio [6].*

Definition 5.4 (Perceptron multicouche) *Un perceptron multicouche possède au moins trois couches : une couche d'entrée, au moins une couche cachée, et une couche de sortie. Chaque couche est constituée d'un nombre (potentiellement différent) de neurones. L'information circule de la couche d'entrée vers la couche de sortie uniquement : il s'agit donc d'un réseau à propagation directe (feedforward). Les neurones de la dernière couche sont les sorties du système global [7].*

5.3 ASCAD

Comme évoqué précédemment, la base de données ASCAD a joué un rôle important dans le développement des attaques par canaux auxiliaires utilisant l'apprentissage profond *(Deep Learning-based Side-channel Analysis (DLSCA))*, en fournissant un dataset public, pouvant être utilisé comme référence à travers les différents chercheurs, afin de développer des attaques de plus en plus performantes.

Depuis 2018, ASCAD est toujours très utilisée pour comparer les modèles proposés par la communauté scientifique et en 2025 un papier intitulé : "A Practical Tutorial on Deep Learning-based Side-channel Analysis" [3] a été publié, dans le but de guider les lecteurs novices et confirmés sur la réalisation de DLSCA. Afin d'accompagner ce papier, un dépôt GitHub proposant un Jupyter Notebook a été publié le 11 mars 2025, nous guidant dans les différentes étapes pour réaliser des attaques DLSCA.

Nous nous sommes basés sur ce papier pour réaliser notre travail sur les DLSCA.

5.4 Caractéristiques de la base de données

La base de données ASCAD est composée de deux datasets :
— **ASCADv1:** AES masque d'ordre 1 implémenté sur un ATMEGA8515 ;
— **ASCADv2:** AES masque affine implémenté sur un STM32.

Il y a également trois scripts proposés, afin de tester la base de données et d'avoir des exemples de perceptron multicouche *(MLP)* et réseaux de neurones convolutifs *(CNN)* appliqués aux attaques par canaux auxiliaires :
— ASCADv1 : Clé fixe ;
— ASCADv1 : Clé variable ;
— ASCADv2 : Clé variable.

Enfin, trois scripts Python sont fournis, afin de faciliter l'utilisation de la base de données :
— `python ASCAD_generate.py path_to_parameters_file` : Script utilisé pour générer la base de données ASCAD, à partir de toutes les traces de consommations disponible.
— `python ASCAD_train_models.py path_to_parameters_file` : Script utilisé pour entraîner un modèle.
— `python ASCAD_test_models.py path_to_parameters_file` : Script utilisé pour tester les modèles entraîné.

5.4.1 ASCADv1

Dans la première version de ASCAD, utilisant un ATMEGA8515, nous avons deux datasets proposés :

Caractéristiques	Taille des données	Nombre de traces pour l'entraînement	Nombre de traces pour tester	Nombre total de traces
Les traces sont **synchronisées** et la **même clé** est utilisée pour toutes les traces	7.3 GB	$50\ 000$	$10\ 000$	$60\ 000$
Les traces ne sont **pas synchronisées** et la **clé varie** en fonction des traces [1]	71 GB	$200\ 000$	$100\ 000$	$300\ 000$

TABLE 1 – Base de données ASCADv1

5.4.2 ASCADv2

Dans la deuxième version de ASCAD, utilisant un SMT23, nous avons uniquement un dataset proposé :

1. $\frac{1}{3}$ des traces utilisent une clé fixe, et $\frac{2}{3}$ utilisent une clé aléatoire.

Caractéristiques	Taille des données	Nombre total de traces	Nombre de points par trace
Les clés et textes clairs utilisé pour créer les différentes traces sont aléatoires	807 GB	800 000	1 000 000

TABLE 2 – Base de données ASCADv2

5.5 Problèmes rencontrés

Nous avons fait face à un problème principal, qui nous a fait perdre beaucoup de temps.

Lorsque nous voulions charger un modèle fourni par ASCAD, nous avions un problème pour charger le fichier `h5`. Suite à plusieurs recherches, le problème vient de la version de `tensorflow` et `keras` utilisée lors de la création des fichiers `h5`. La version de `keras` utilisée pour générer ces fichiers est la `2.1.1`, nous sommes actuellement en `2.18.0`. Et pour `tensorflow`, la version utilisée est la `2.3.0`, or, nous sommes actuellement en `2.18.0`.

Pour installer `tensorflow 2.3.0`, nous devons utiliser une version de Python comprise entre `3.5` et `3.8` [8]. L'objectif était d'extraire le modèle h5 au format JSON [9] qui ne comporte pas de problème de compatibilité, peu importe la version de `tensorflow` ou `keras` utilisée.

Nous avons donc essayé de créer un `Docker`, afin d'avoir un conteneur avec la bonne version de Python et les librairies nécessaires à l'extraction des paramètres des modèles proposés par ASCAD. Cependant, malgré de nombreux essais, nous faisions face au même souci, lorsque nous essayions de lancer le conteneur : `ERROR: Could not find a version that satisfies the requirement tensorflow==2.3.0 (from versions: none) ERROR: No matching distribution found for tensorflow==2.3.0 .`

```
FROM python:3.8

WORKDIR /usr/src/app

RUN apt-get update && apt-get install -y \
    git \
    wget \
    libpq-dev build-essential

RUN pip3 install --upgrade pip

COPY requirements.txt ./
RUN pip3 install --upgrade --no-cache-dir
    -r requirements.txt

COPY convert_model.py ./

CMD [ "python", "./convert_model.py" ]
```

Listing 1 – Dockerfile

```
Cython
numpy
h5py
keras==2.4.3
tensorflow==2.3.0
```

Listing 2 – requirements.txt

```
import keras
```

```python
from keras.models import load_model
import tensorflow as tf
import json
import os
import numpy as np

print("Keras version:", keras.__version__)
print("TensorFlow version:", tf.__version__)

model_path = '/data/
    mlp_best_ascad_desync0_node200_layernb6_epochs200_classes256_batchsize100.h5'

try:
    model = load_model(model_path)

    model.summary()

    model_json = model.to_json()
    with open('/data/model_architecture.json', 'w') as f:
        f.write(model_json)
    print("Model architecture saved as JSON")

    model.save_weights('/data/model_weights.h5')
    print("Model weights saved separately")

    tf.keras.models.save_model(model, '/data/saved_model_tf')
    print("Model saved")

    print("Conversion completed successfully!")

except Exception as e:
    print(f"Error during model conversion: {str(e)}")
```

Listing 3 – convert_model.py

Suite à une discussion avec monsieur Aragon, nous avons décidé d'utiliser les environnements virtuels de Python, cependant, cette fois aussi, nous avons fait face à la même erreur.

Après avoir passé de nombreuses heures pour essayer d'extraire les données de ses modèles, nous avons décidé d'abandonner cette idée, et de se concentrer sur l'utilisation de la base de données ASCAD pour l'entraînement de modèles de deep learning.

5.6 Modèles proposés et cas de figure étudié

Les modèles les plus utilisés pour attaquer la base de données ASCAD sont les Convolutional Neural Network (CNN) et Multilayer Perceptron (MLP).

Dans la suite de cette section, nous allons les étudier et en présenter trois appliquées à la base de données ASCAD.

Ces modèles vont être entraînés et testés avec le **scénario Optimized Points of Interest (OPOI)**, car il a été utilisé très régulièrement dans les papiers de recherche traitant de DLSCA.

Definition 5.5 (Optimized Points of Interest (OPOI)) *Avec le scénario OPOI, nous n'utilisons qu'un petit sous-ensemble d'échantillons provenant des traces brutes. Nous allons nous concentrer sur le sous-ensemble correspondant au moment où ont lieu les opérations réalisant la fuite des données. Nous pouvons faire cela, car nous avons un contrôle total sur l'appareil de profilage.*

Les principaux avantages sont que la charge de calcul et le nombre de points n'apportant pas d'informations utiles sont plus faciles à gérer [3].

Il y a quatre modèles de fuite principaux :

- Le Poids de Hamming (HW) : Nombre de bits qui sont à 1 [10] ;
  - Il y a 9 classes.
- La Distance de Hamming (HD) : Compte le nombre de transitions de $0 \rightarrow 1$ et $1 \rightarrow 0$ qui ont lieu dans un circuit numérique pendant un certain intervalle de temps [10] ;
  - Il y a 9 classes.
- L'identité (ID) : Utilisant directement l'IV [3] ;
  - Il y a 256 classes.
- Un modèle de fuite par bit où nous sélectionnons un bit spécifique de l'IV [3].

Pour la suite de cette section, nous allons nous concentrer sur le modèle de fuite : **identité *(ID)***.

5.6.1 Perceptron Multicouche (MLP)

MLP est un algorithme d'apprentissage supervisé, qui apprend une fonction $f : R^m \rightarrow R^o$ en s'entraînant sur un dataset, où m est le nombre de dimensions des données en entrée et o le nombre de dimensions pour les sorties *(e.g. nombre de classes)*.

Soit $X = x_1, x_2, \ldots, x_m$ les caractéristiques *(features)* et y l'objectif *(target)*.

Une fois X et y données, le modèle peut apprendre à réaliser une **classification** ou une **régression** grâce à une **fonction non-linéaire**.

Entre l'entrée et la sortie, il peut y avoir **une ou plusieurs couches non-linéaires** *(hidden layers)*.

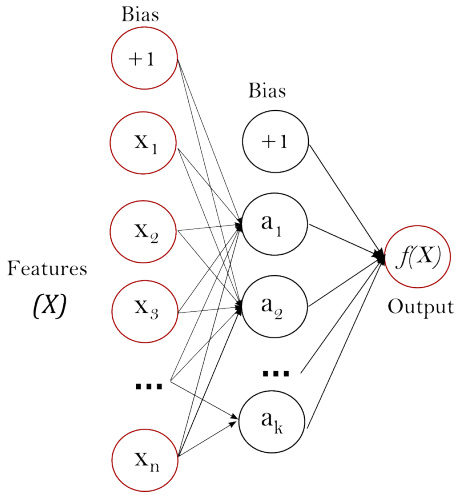


FIGURE 13 – Une couche cachée MLP [11]

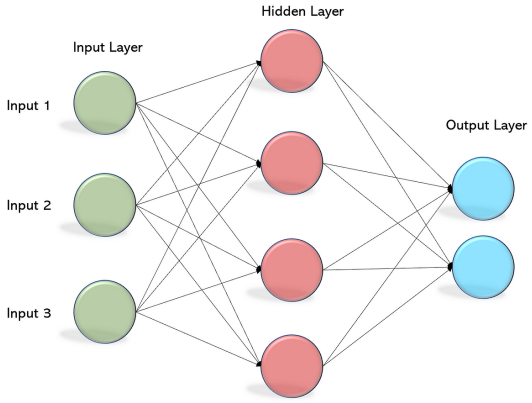


FIGURE 14 – MLP [12]

Dans l'image de gauche, la couche d'entrée (couche gauche), est un ensemble de neurones $\{x_i \mid x_1, x_2, \ldots, x_m\}$ représentant les classes en entrée. Chaque neurone dans la couche cachée *(hidden layer)* transforme la valeur de la couche précédente grâce à une **somme linéaire pondérée** $w_1 x_1 + w_2 x_2 + \cdots + w_m x_m$ et une fonction d'activation non-linéaire $g(.) : R \to R$ comme la fonction hyperbolic tan. La couche de sortie reçoit les valeurs de la dernière couche cachée et les transforme en valeur de sortie *(prédiction)*.

```python
# Model taked from Perin, Guilherme, Lichao Wu, and Stjepan Picek.
# "Exploring feature selection scenarios for deep learning-based side-channel analysis."
# IACR Transactions on Cryptographic Hardware and Embedded Systems 2022.4 (2022): 828-861.
# https://github.com/AISyLab/feature_selection_dlsca/blob/master/experiments/ASCADr/OPOI/best_models.py
def best_mlp_id_opoi_1400_ascadr(classes, number_of_samples):
    # Best multilayer perceptron for ASCAD variable key dataset
    # Number of points-of-interest: 1400
    # Leakage model: ID
    # POI interval: [80945, 82345]
    # Number of parameters: 34236

    batch_size = 100
    tf.random.set_seed(83545)
    model = Sequential(name='best_mlp_id_opoi_ascadv_1400')
    model.add(Dense(20, activation='selu', kernel_initializer='random_uniform', input_shape=(number_of_samples,)))
    model.add(Dense(20, activation='selu', kernel_initializer='random_uniform'))
    model.add(Dense(20, activation='selu', kernel_initializer='random_uniform'))
    model.add(Dense(classes, activation='softmax'))
    optimizer = Adam(learning_rate=0.0005)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model, batch_size
```

Listing 4 – MLP: Optimized Points of Interest - 2022

Meilleur modèle MLP proposé dans le papier d'introduction à ASCAD

Ce modèle est proposé dans le papier présentant la base de donnée ASCAD en 2020, comme étant le meilleur modèle MLP qu'ils ont trouvé dans cette recherche :

> *"Benchmarks reported in this section confirms that the architecture* `MLP(6, 200, ReLU)` *leads to good compromise efficiency versus computational time when trained with the procedure* `Training(200, 100, RMSProp,` 10^{-5}`)." [2]`

Cependant, nous pouvons modifier le nombre d'epoch de 200 à 800, ce qui entraîne un ralentissement du temps nécessaire pour entraîner notre modèle, mais une meilleure performance par la suite :

> *"We insist on the fact that MLPbest has a decent SCA-efficiency with* 200 *epochs but the latter efficiency continues to improve when the number of epochs increases until* 800 *epochs (in our experiments we did not notice any significant improvement after* 800 *epochs). Hence, depending on the amount of time allocated to the training of* MLP_{best}*, it may be interesting to increase the number of epochs in the range* $[200..800]$*." [2]*

Le papier définit les MLP de la manière suivante : `MLP(nLayer, nUnits, activation_function)`, avec l'entraînement comme ceci : `Training(nepochs, batch_size, optimizer, learning_rate)`. Selon la documentation de tensorflow, les couches entièrement connectées sont représentées par la classe `Dense` [13] :

```
tf.keras.layers.Dense(
    units,
    activation=None,
    use_bias=True,
    kernel_initializer='glorot_uniform',
    bias_initializer='zeros',
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    lora_rank=None,
    **kwargs
)
```

Listing 5 – Définition de la classe `Dense`

Nous allons donc définir nos couches de la manière suivante : `Dense(200, activation='relu')`. Il faut légèrement modifier notre code pour la couche d'entrée et de sortie, afin de définir le nombre d'éléments en entrée et de classes en sortie.

```
# Model taken from Benadjila, Ryad, et al.
# "Deep learning for side-channel analysis and introduction to ASCAD database."
# Journal of Cryptographic Engineering 10.2 (2020): 163-188.
def mlp_best(classes, number_of_samples):
  batch_size = 300
  node=200
  layer_nb=6
  model = Sequential()
  model.add(Dense(node, input_dim=number_of_samples, activation='relu'))
  for i in range(layer_nb-2):
    model.add(Dense(node, activation='relu'))
  model.add(Dense(classes, activation='softmax'))
  optimizer = Adam(learning_rate=0.0001)
  model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy
    '])
  return model, batch_size
```

Listing 6 – MLP_best: ASCAD - 2020

5.6.2 CNN

```python
# Model taked from Perin, Guilherme, Lichao Wu, and Stjepan Picek.
# "Exploring feature selection scenarios for deep learning-based side-channel analysis."
# IACR Transactions on Cryptographic Hardware and Embedded Systems 2022.4 (2022): 828-861.
# https://github.com/AISyLab/feature_selection_dlsca/blob/master/experiments/ASCADr/OPOI/best_models.py
def best_cnn_id_opoi_1400_ascadr(classes, number_of_samples):
    # Best Convolutional Neural Network for ASCAD variable key dataset
    # Number of points-of-interest: 1400
    # Leakage model: ID
    # POI interval: [80945, 82345]
    # Number of parameters: 87632

    batch_size = 300
    tf.random.set_seed(511628)
    model = Sequential(name='best_cnn_id_opoi_ascadv_1400')
    model.add(Conv1D(kernel_size=46, strides=23, filters=8, activation='selu', input_shape=(number_of_samples, 1), padding='same'))
    model.add(MaxPool1D(pool_size=2, strides=2, padding='same'))
    model.add(BatchNormalization())
    model.add(Conv1D(kernel_size=50, strides=25, filters=16, activation='selu', padding='same'))
    model.add(MaxPool1D(pool_size=6, strides=6, padding='same'))
    model.add(BatchNormalization())
    model.add(Conv1D(kernel_size=44, strides=22, filters=32, activation='selu', padding='same'))
    model.add(MaxPool1D(pool_size=2, strides=2, padding='same'))
    model.add(BatchNormalization())
    model.add(Flatten())
    model.add(Dense(200, activation='selu', kernel_initializer='he_uniform'))
    model.add(Dense(classes, activation='softmax'))
    optimizer = Adam(learning_rate=0.001)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model, batch_size
```

Listing 7 – CNN: Optimized Points of Interes - 2022

Nous avons pu parler brièvement au premier semestre des Convolutional Neural Network (CNN) comme étant très prometteurs dans dans l'analyse de traces pour les attaques par canaux auxiliaires. Des modèles de plus en plus performants continuent de se développer encore très récemment comme nous avons pu le voir dans cette section. Si l'utilisation des méthodes **SPA, DPA, CPA** sont les piliers des attaques **SCA**, l'apprentissage profond est lui une méthode encore en plein développement.

5.6.3 Comparaison des différents modèles

Definition 5.6 (Supposition de l'entropie(Guessing Entropy (GE))) *Cela correspond au rang moyen des clés.*

Le calcul de la moyenne est effectué pour améliorer la qualité statistique (c'est-à-dire pour réduire l'effet des traces spécifiques utilisées) de l'attaque [3].

L'objectif d'une attaque par canaux auxiliaires est de minimiser le rang de la clé et les suppositions de l'entropie et de maximiser le taux de réussite et l'information perçue.

Nous voulons savoir le nombre de traces nécessaires pour extraire la bonne clé. Lorsque la supposition de l'entropie est égale à 0 cela indique que le bon candidat à la clé est toujours classé en premier.

Modèle	Epoch	Taille Batch	GE Moyenne	GE Median	Trace à partir de quand la GE est égal à 0
`best_mlp_id_opoi_1400_ascadr`	100	100	0	0	Jamais, est à 0.02 pour la $879^{ème}$ trace
`mlp_best`	200	100	11.01	6	Jamais, est à 15.2 pour la $941^{ème}$ trace
`best_cnn_id_opoi_1400_ascadr`	100	300	0	0	$360^{ème}$ trace

TABLE 3 – Comparaison des différents modèles

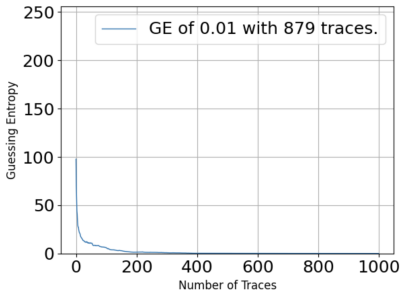


FIGURE 15 – ... `best_mlp_id_opoi_1400_ascadr`

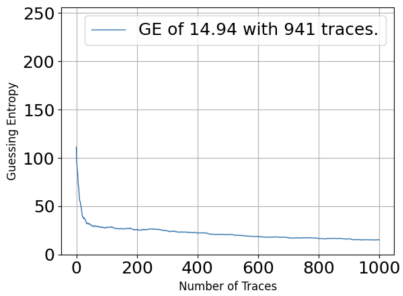


FIGURE 16 – ... `mlp_best`

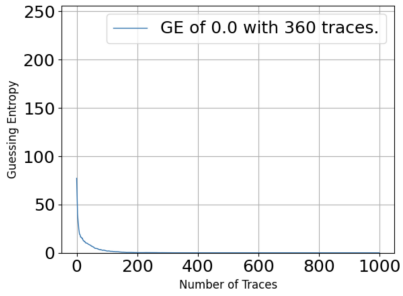


FIGURE 17 – ... `best_cnn_id_opoi_1400_ascadr`

Nous pouvons en conclure, que le meilleur modèle performant le mieux, est `best_cnn_id_opoi_1400_ascadr`, avec seulement 360 traces nécessaires pour avoir un taux de succès de 100%.

6 Conclusion

Au cours de ce semestre, nous avons pu explorer en profondeur le domaine des attaques par canaux auxiliaires SCA appliquées aux chiffrements AES et RSA. Cette étude a été structurée en deux grands axes complémentaires : d'une part, la mise en œuvre concrète d'attaques sur des dispositifs embarqués comme le **ChipWhisperer**, et d'autre part, l'exploration des attaques beaucoup plus récentes et en plein essor fondées sur l'apprentissage profond.

Dans un premier temps, nous avons mené des attaques classiques de type SPA, DPA et CPA sur des implémentations logicielles et matérielles d'AES et de RSA. Ces attaques nous ont permis de comprendre les fuites physiques que peuvent générer les opérations cryptographiques et comment exploiter ces fuites de manière efficace, même avec un nombre limité de traces. L'étude du **bootloader AES-256** a particulièrement mis en lumière l'intérêt stratégique de ces attaques dans des contextes réels où les protections mises en place sont supposées robustes.

Nous avons ensuite abordé les problématiques liées aux conditions de capture de traces non idéales telles que le jitter, qui réduisent significativement l'efficacité des attaques. Pour y remédier, nous avons étudié plusieurs méthodes de resynchronisation, en particulier la **SAD (Sum of Absolute Differences)** et le **Dynamic Time Warping (DTW)**, qui se sont révélées indispensables pour réaligner les traces avant toute analyse statistique fiable.

Enfin, dans la seconde partie du semestre, nous avons approfondi les approches récentes basées sur le **deep learning**. À travers la base de données ASCAD et les outils proposés dans la littérature récente, nous avons expérimenté plusieurs modèles de réseaux de neurones (MLP et CNN) pour des attaques par profilage. Ces approches se sont montrées particulièrement robustes face au bruit et aux désynchronisations, prouvant leur efficacité dans des scénarios de plus en plus réalistes. Le scénario Optimized Points of Interest (OPOI) a notamment permis de concentrer les efforts d'apprentissage sur les portions pertinentes des traces.

Ce travail nous a permis de consolider nos compétences en cryptanalyse expérimentale, tout en intégrant des outils modernes issus du machine learning. Il constitue une base solide pour approfondir, à l'avenir, des aspects encore plus avancés de la sécurité lié aux systèmes embarqués.

En résumé, ce semestre a permis d'être une extension plus concrète, appuyé sur la base de ce qui a déjà été fait au premier semestre, en explorant plein d'axes intéressants et en se confrontant aux problématiques réelles liées à ce domaine.

Acronyms

AES Advanced Encryption Standard. 2, 30

CHES Conference on Cryptographic Hardware and Embedded Systems. 21

CNN Convolutional Neural Network. 23, 25, 29, 31

CPA Correlation power analysis. 2, 30

CRC cyclic redundancy check. 13

DL Deep Learning. 1, 21, 22

DLSCA Deep Learning-based Side-channel Analysis. 22

DPA Differential power analysis. 2, 30

IA Intelligence Artificielle. 21

ML Machine Learning. 21

MLP Multilayer Perceptron. 23, 25, 31

OPOI Optimized Points of Interest. 25, 26, 31

RSA Rivest–Shamir–Adleman. 30

SCA Side-Channel Attacks. 30

SPA Simple power analysis. 2, 30

Glossaire

Attaques par analyse de consommation électrique Les attaques par analyse de consommation électrique exploitent le fait que la consommation électrique instantanée d'un dispositif cryptographique dépend des données qu'il traite et des opérations qu'il effectue [10]. 2

Attaques par canaux auxiliaires Attaques passives et non-invasives où l'attaquant va observer et mesurer les caractéristiques analogiques de l'implémentation logicielle ou matérielle d'un algorithme de chiffrement, afin d'extraire la clé de chiffrement. Les principales caractéristiques analogiques utilisées en SCA sont le **temps d'exécution**, la **consommation électrique** et les **émissions électromagnétiques**. [14] [10]. 21

Attaques par canaux auxiliaires avec profilage Une attaque par profilage consiste en deux étapes. Premièrement, l'adversaire se procure une copie du matériel cible et définit les fuites physiques. Deuxièmement, il réalise l'attaque sur la cible, afin de retrouver la clé [2] . 21

Batch Ensemble d'exemples utilisés dans une itération d'entraînement. La taille de lot (batch) détermine le nombre d'exemples dans un lot[5] . 22

Distance de Hamming Compte le nombre de transitions de $0 \rightarrow 1$ et $1 \rightarrow 0$ qui ont lieu dans un circuit numérique pendant un certain intervalle de temps [10]. 26

Epoch Dans le contexte de l'entraînement d'un modèle, l'epoch est un terme utilisé pour référer à une itération où le modèle voit tout le training set pour mettre à jour ses coefficients [4]. 22

Gigue Mesure la variation de la latence au cours du temps. $Gigue = latency(n) - latency(n-1)$, où la mesure $n-1$ est prise à un temps t et n à un temps $t + \delta t$ [15] . 21

Jitter Le jitter est un effet qui apparait lorsqu'une capture de trace de courant est effectué, elle viens des perturbations électrique d'un système. Elle peuvent rendre l'analyse de trace plus complexe et nécéssitent d'être resynchronisé . 14, 16

Perceptron Multicouche (Multilayer Perceptron [MLP]) Un perceptron multicouche possède au moins trois couches : une couche d'entrée, au moins une couche cachée, et une couche de sortie. Chaque couche est constituée d'un nombre (potentiellement différent) de neurones. L'information circule de la couche d'entrée vers la couche de sortie uniquement : il s'agit donc d'un réseau à propagation directe (feedforward). Les neurones de la dernière couche sont les sorties du système global [7]. 22, 23

Poids de Hamming Nombre de bits qui sont à 1 [10]. 26

Réseaux de Neurones Convolutifs Ils utilisent des données tridimensionnelles pour les tâches de classification d'images et de reconnaissance d'objets. Ils se distinguent des autres réseaux neuronaux par leurs performances supérieures avec des entrées de signaux d'image, de parole ou audio [6]. 22, 23

Références

[1] C. O-Flynn, “Chipwhisperer.” Available at https://www.newae.com/chipwhisperer.

[2] R. Benadjila, E. Prouff, R. Strullu, E. Cagli, and C. Dumas, “Deep learning for side-channel analysis and introduction to ascad database,” *Journal of Cryptographic Engineering*, vol. 10, no. 2, pp. 163–188, 2020.

[3] S. Karayalcin, M. Krcek, and S. Picek, “A practical tutorial on deep learning-based side-channel analysis,” *Cryptology ePrint Archive*, 2025.

[4] A. A. et Shervine Amidi, “Pense-bête de petites astuces d’apprentissage profond.” Available at https://stanford.edu/~shervine/l/fr/teaching/cs-230/pense-bete-petites-astuces-apprentissage-profond.

[5] Google, “Glossaire du machine learning.” Available at https://developers.google.com/machine-learning/glossary?hl=fr.

[6] IBM, “Qu’est-ce qu’un convolutional neural networks (cnn) ?.” Available at https://www.ibm.com/fr-fr/topics/convolutional-neural-networks.

[7] Wikipedia, “Perceptron multicouche.” Available at https://fr.wikipedia.org/wiki/Perceptron_multicouche.

[8] Tensorflow, “Tensorflow version et version python correspondante.” Available at https://www.tensorflow.org/install/source?hl=fr#cpu.

[9] F. C. Neel Kovelamudi, “Save, serialize, and export models.” Available at https://keras.io/guides/serialization_and_saving/.

[10] T. P. Stefan Mangard, Elisabeth Oswald, *Power Analysis Attacks : Revealing the Secrets of Smart Cards*. Springer, 1 ed., 2007.

[11] Scikit-Learn, “1.17. neural network models (supervised).” Available at https://scikit-learn.org/stable/modules/neural_networks_supervised.html.

[12] A. Mohanty, “Multi layer perceptron (mlp) models on real world banking data.” Available at https://becominghuman.ai/multi-layer-perceptron-mlp-models-on-real-world-banking-data-f6dd3d7e998f.

[13] Tensorflow, “tf.keras.layers.dense.” Available at https://www.tensorflow.org/api_docs/python/tf/keras/layers/Dense.

[14] J. H. J.H. Silverman, Jill Pipher, *An Introduction to Mathematical Cryptography*. Springer, 2 ed., 2014.

[15] P.-F. Bonnefoi, “Cours - protocoles et programmation réseau.” Available at https://p-fb.net/master1/proto_prog/cours/Cours_PPRes.pdf.