SFA - Laboratoire 2

Fault attacks

Maxime Chantemargue, Ruben Pereira Lopes, Charles Matrand

22 mai 2024

Table des matières

Auteurs : Charles Matrand, Maxime Chantemargue, Ruben Pereira Lopes

Lab setup

Nous utilisions la carte ChipWisperer, même carte qu'on utilise depuis déjà quelques laboratoires. ChipWisperer est une carte open-source qui permet de faciliter l'étude des side channel et fault attacks. Nous allons étudier pour ce laboratoire une fonction qui chiffre en utilisant AES128. Le but est d'obtenir la clé de chiffrement utilisée.

Pour la recherche des paramètres optimaux, nous nous sommes basés sur le notebook fourni. La " Characterization loop" pour déterminer les paramètres et une loop qui fourni les ciphertexts fautés qui seront ensuite utilisés pour la recherche de bytes.

Pour la recherche des bytes de clés, nous allons utiliser l'outil "PhoenixAES" : https://github.com/SideChannelMarvels/JeanGrey/tree/master/phoenixAES. Pour l'installation :

```
import phoenixAES
```

Et le code utilisé (celui du repository, avant analyse) :

```
import phoenixAES

with open('tracefile', 'wb') as t:
    t.write("""
74657374746573747465737474657374 BF9B06F11DF478145B8300FE440B0D06
74657374746573747465737474657374 BF9BDDF11D527814568300FE440B0DFA
74657374746573747465737474657374 BF9BF9F11DAC78145F8300FE440B0D67
74657374746573747465737474657374 BF9BF0F11DBB78140C8300FE440B0DEE
74657374746573747465737474657374 BF9BF9F11DAC78145F8300FE440B0D67
74657374746573747465737474657374 BF9B69F11DBD7814E68300FE440B0DAE
74657374746573747465737474657374 BF9BF0F11DBB78140C8300FE440B0DEE
74657374746573747465737474657374 BF9B90F11D4178149D8300FE440B0DE2
74657374746573747465737474657374 BF9BCFF11D0478140E8300FE440B0D28
74657374746573747465737474657374 FD9B06F11DF478E15B831AFE44C40D06
74657374746573747465737474657374 BA9B06F11DF4787B5B83E8FE44020D06
74657374746573747465737474657374 579B06F11DF478565B8364FE446F0D06
74657374746573747465737474657374 579B06F11DF478565B8364FE446F0D06
74657374746573747465737474657374 BF9B065C1DF4B6145B1800FE9E0B0D06
74657374746573747465737474657374 BF9B065C1DF4B6145B1800FE9E0B0D06
74657374746573747465737474657374 BF9B06251DF454145BC200FE060B0D06
74657374746573747465737474657374 BF9B06941DF4C3145BFB00FED20B0D06
74657374746573747465737474657374 BF9B12F11D977814DD8300FE440B0D21
74657374746573747465737474657374 BF9B90F11D4178149D8300FE440B0DE2
74657374746573747465737474657374 BF9BCFF11D0478140E8300FE440B0D28
74657374746573747465737474657374 BF9BDDF11D527814568300FE440B0DFA
74657374746573747465737474657374 BFFB06F1E2F478145B8300AB440B7906
74657374746573747465737474657374 BF5D06F142F478145B830049440B7306
""".encode('utf8'))

phoenixAES.crack_file('tracefile')
```

Ce code permettra de nous fournir les bytes de la clé en fonction de ciphertexts fautés spécifiquement ciblés (détaillé dans les prochains points).

Attack methodology

1 - Recherche de paramètres optimaux

Notre premier objectif est de trouver les paramètres de glitch qui permettent de trigger le plus de glitches sans faire crash totalement la chip.

Les paramètres testés sont les suivants :

- `ext_offset` : nombre de cycle d'horloge à attendre avant d'injecter la faute (ici depuis le trigger de début d'AES)
- `offset` : permet de placer le glitch (pourcentage d'une clock)
- `width` : la durée du glitch (pourcentage d'une clock)

Les glitchs utilisent la méthode de glitch de voltage MOSFET déjà implémentée sur la ChipWhisperer.

Notre code est une série de boucles imbriquées. Au sein de ces boucles, nous avons un code qui interagit avec la chip et envoie un array vide à chiffrer. Et nous recevons la réponse de la chip.

Cette réponse est traitée par le code et elle va nous print des indications sur l'efficacité des glitchs, celle-ci est composée des caractères suivants :

- `.` : signifie que la valeur de compteur à la sortie de l'exécution est la même que celle attendue (2500)
- <`chiffre`> : signifie que la valeur de compteur est différente de celle attendue, la valeur du compteur est affichée.
- `*` : signifie que la chip a crashé

Le `count` en dessous de la réponse indique les paramètres utilisés (`ext_offset`,`offset`, `width`) et le nombre de fois ou le programme a glitché pour ces paramètres. Pour rappel, cela indique donc le nombre de fois où il n'y a pas de crash et que la valeur du compteur est différente de 2500. Le maximum de `count` est de 5 car pour chaque combinaison de paramètres, on itère 5 fois dessus.

```
1  ..............................X
      ....................................*00002499*...
2  Count 01    ext_offset 01   Width 35.90 Offset -27.60
3  ....................................*00002499*.*00002499**00002499*

4  Count 03    ext_offset 01   Width 35.80 Offset -27.50
5  ..........................................*00002499*.*00002499*.
6  Count 02    ext_offset 01   Width 35.80 Offset -27.40
7  ..............................................*00002499*.*00002499**00002499*

8  Count 03    ext_offset 01   Width 35.90 Offset -27.30
9  ....................................*00002499**00002499*
10 Count 02    ext_offset 01   Width 35.80 Offset -27.20
```

2 - Recherche du round AES optimal

Une fois qu'on a trouvé les paramètres d'`offset` de `width` en regardant pour quelles valeurs, nous obtenions des counts élevés. C'est-à-dire un nombre de glitchs élevé. Nous avons cherché le `ext_offset`, c'est-à-dire le bon décalage depuis le trigger du début d'AES qui nous donne les ciphertexts fautés. On expliquera lequel on a trouvé plus bas dans la partie **Glitch parameters**.

3 - Recherche de ciphertexts fautés

Nous voulons attaquer le round 9 d'AES car grâce aux propriétés d'AES, une quelconque modification d'un byte avant le MixColumn du round 9 permet l'obtention d'un pattern "diagonal" dans le ciphertext final (en rouge dans le schéma ci-dessous). Si la faute est avant le MixColumn du round 8, tout le ciphertext sera modifié, si la faute est trop tard, il n'y aura qu'un seul byte qui sera modifié.

L'altération d'un byte ici sera faite grâce à un glitch.

Il faut par contre récupérer des ciphertexts avec des diagonales différents :

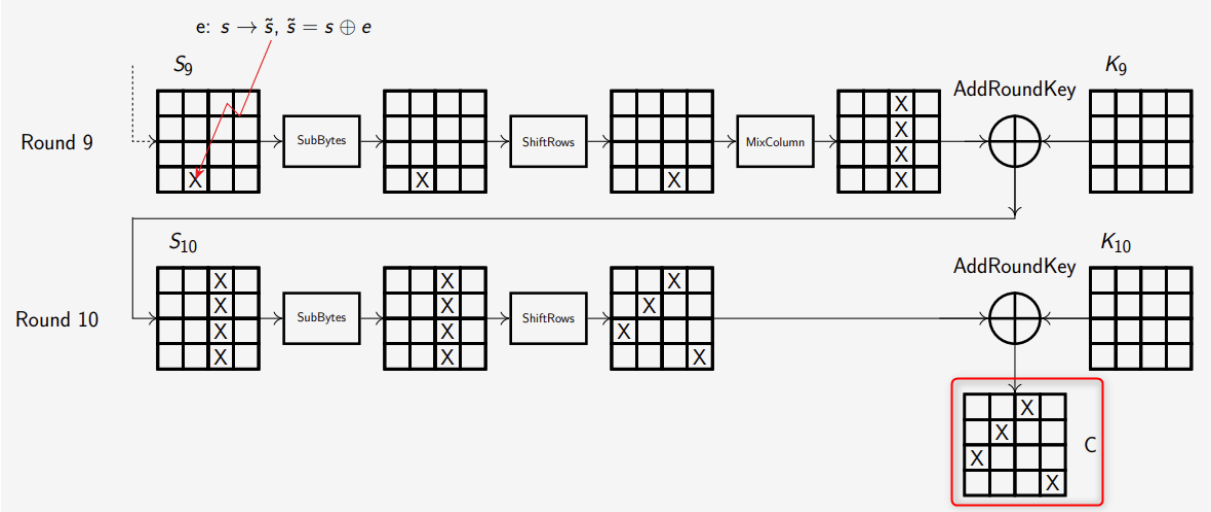

Figure 1: Error propagation in AES128

Error propagation in AES128

Voici les diagonales qui nous intéressent et qui couvriront tous les 16 bytes :

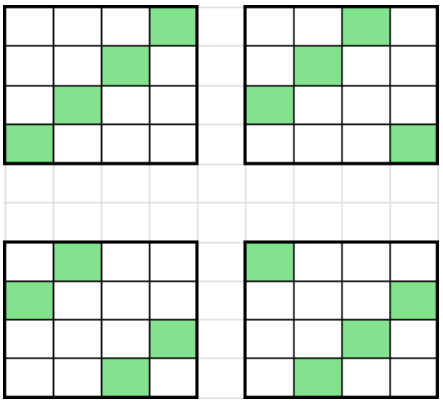

Figure 2: Diagonales ciblées

Diagonales ciblées

4 - Recherche de bytes de clés avec PhoenixAES

Après avoir trouvé nos ciphertexts fautés avec les diagonales ci-dessus (en utilisant un XOR avec le ciphertext original), nous utilisons PhoenixAES.

Cette librairie permet de trouver (une fois ce pattern identifié sur au moins 2 ciphertexts) la k10 AES qui a été utilisée. Cela est effectué en utilisant les propriétés d'AES, on envoie à PhoenixAES le ciphertext original, les ciperthexts fautés (avec diagonales) et leurs plaintexts correspondants.

5 - Master key

La dernière étape, récupérer simplement le flag à partir de la K10 ressortie par PhoenixAES en utilisant les fonctions données par la classe `sca_training`.

Glitch parameters

En faisant du fine tuning, nous avons d'abord trouvé des bons paramètres de `Width` et d'`Offset` en suivant la méthodologie expliquée plus tôt dans le rapport.

En "plotant" les résultats sur un simple graphe :

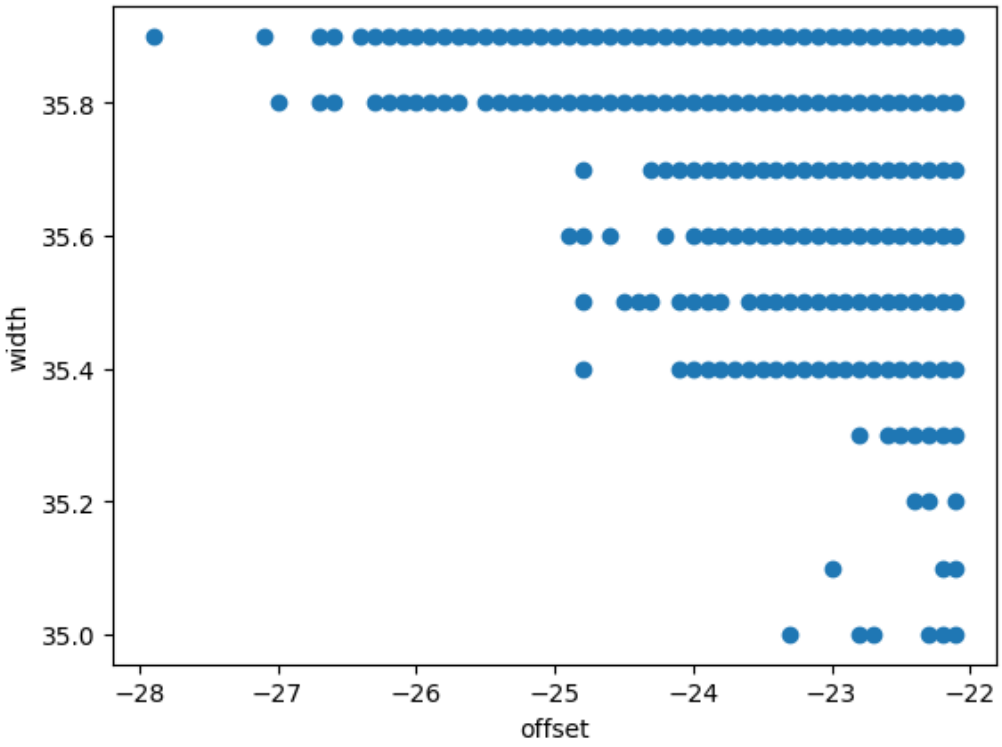

Figure 3: Untitled

Et sur une heatmap :

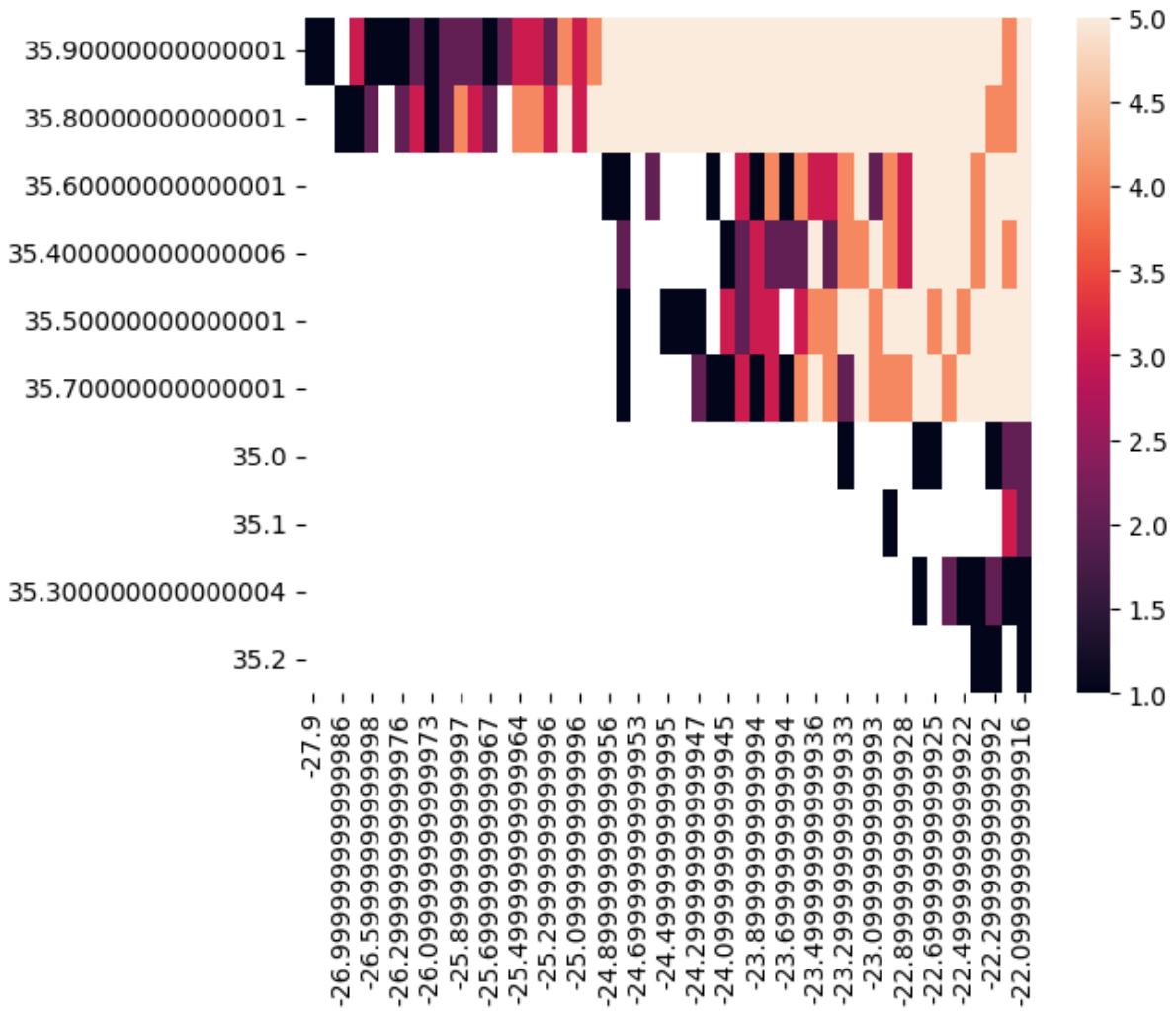

Figure 4: Untitled

Les paramètres trouvés sont donc :

- `Width` : (35, 36)
- `Offset` : (-23.80, -22.0)

Après avoir trouvé les bons ranges de `Width` et d'`Offset`, nous avons cherché le bon `ext_offset` .

En regardant le graphe de la consommation de courant donné dans les slides du laboratoire depuis le trigger. Nous avons identifié que le round 9 se trouvait dans le range 5600 à 6010 (en vert ci-dessous).

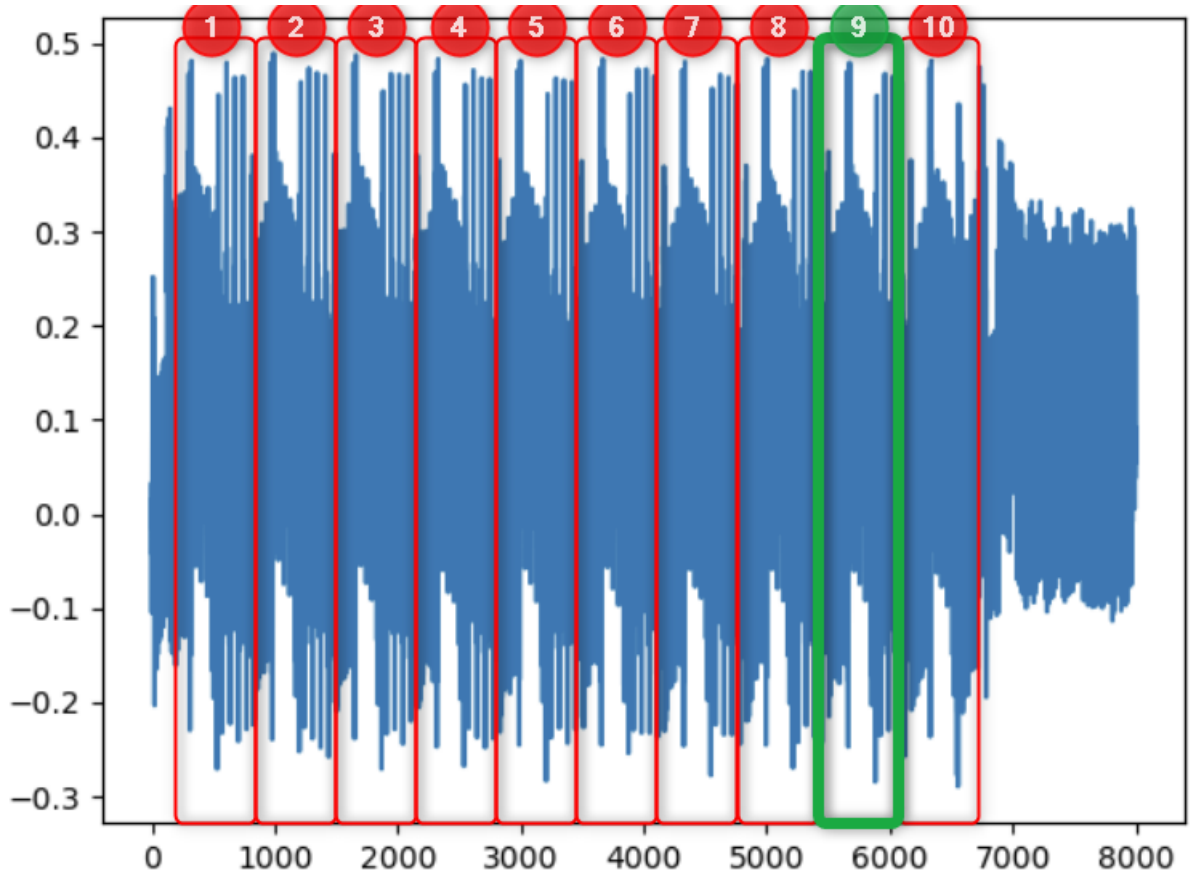

Figure 5: Graphe de consommation avec séparation des round AES

Graphe de consommation avec séparation des round AES

Relevant faulted ciphertexts used

Voici la liste des ciphertexts (plaintext = `0123456789abcdef`) utilisés avec PhoenixAES (le premier étant l'original et le reste les ciphertexts fautés) :

```
# Cipher original
b6a0d09b603785de90f30b516b341e54

# Diagonal 1
b6a01d9b608885de2cf30b516b341eb0 (xor : 0000
    cd0000bf0000bc000000000000e4)
b6a0819b605e85de8ef30b516b341e8f (xor : 00005100006900001
    e000000000000db)

# Diagonal 2
b6dad09bfc3785de90f30be06b341754 (xor : 007
    a00009c000000000000b100000900)
b623d09b373785de90f30b416b342854 (xor :
    00830000570000000000001000003600)

# Diagonal 3
```

```
a9a0d09b603785b590f32e516b3f1e54 (xor : 1
    f0000000000006b00002500000b0000)
90a0d09b6037854690f34f516b611e54 (xor :
    26000000000000980000440000550000)
70a0d09b6037854390f3d2516bf31e54 (xor :
    c60000000000009d0000d90000c70000)

# Diagonal 4
b6a0d0e8603761de900d0b51d8341e54 (xor : 000000730000
    e40000fe0000b3000000)
b6a0d003603743de90c90b51e8341e54 (xor : 000000980000
    c600003a000083000000)
```

Nous avons du rajouter un ciphertext fauté en plus (3, au lieu 2) pour une diagonale, car le résultat était incomplet sur l'outil.

Sortie de l'outil :

```
Last round key #N found:
F9169C42CB5A6DA13033AAE4866C114E
```

AES key

La clé AES obtenue avec les fonctions de `sca_training` est donc `SCA{Gl1tch__AES}`.