Stage été 2020 - Récapitulatif du travail effectué

Jean Destribois

Février 2021

Table des matières

1 Introduction

Pendant l'été 2020 j'ai eu l'occasion de faire un stage dans le laboratoire LMV de l'UFR des Sciences de Versailles ayant comme référent le Professeur Louis Goubin. L'objectif du stage étant de travailler sur des implémentations de l'AES en boîte blanche et de réussir à trouver le moyen d'en casser certaines.
Le travail se constituait essentiellement de recherche et documentation (lecture de papiers sur les implémentations en boîte blanche et sur les attaques), l'exploitation et la compréhension de certains outils (notamment ceux fournis par Side-Channel Marvels décris plus tard) et des tentatives d'attaques sur diverses implémentations.
Dans ce document, je décrit les résultats qui m'ont semblés intéressants obtenus pendant ce stage. Le système d'exploitation que j'utilisais était un linux ubuntu.

2 Rappel sur la cryptographie en boîte blanche

2.1 Cryptographie classique

En cryptographie classique, lorsqu'on se met dans la position de l'attaquant, et qu'on souhaite attaquer l'algorithme, on considère potentiellement ces 3 choses :

— On a accès au message clair qu'on envoie à l'algorithme.
— On a accès au chiffré que l'algorithme renvoie.
— On connait le fonctionnement de l'algorithme.

Sachant ça, on peut essayer diverses entrées de l'algorithme mais surtout analyser son fonctionnement et en faire des déductions qui pourrait porter préjudice à la sécurité de ce dernier.

2.2 Cryptographie en boîte grise

Cette fois ci, on considère que l'algorithme est implémenté physiquement sur un appareil (par exemple dans une carte à puce) et on se permet plus de choses : l'exploitation des canaux auxiliaires. Les canaux auxiliaires (ou side channel) correspondent à tout ce qu'on peut faire physiquement sur l'appareil pour obtenir des informations. Par exemple analyser la consommation d'énergie d'une carte à puce lors d'une l'exécution de l'algorithme (SPA et DPA/CPA) ou bien analyser le temps d'exécution pour différents messages donnés (Timing attack) ou même injecter une faute pendant l'exécution et analyser les chiffrés fautés avec le chiffré juste (DFA).
Pour contrer ces attaques on ne modifie pas le fonctionnement de l'algorithme mais plutôt la manière de l'implémenter.

2.3 Cryptographie en boîte blanche

On considère maintenant que l'algorithme n'est plus implémenté dans un circuit mais est implémenté de manière logiciel (dans un ordinateur ou un smart-phone par exemple). On considère également que l'attaquant a tous les droits sur la machine.
De la même manière qu'en cryptographie en boîte grise, l'attaquant va utiliser des sortes de canaux auxiliaires pendant l'exécution du programme et les exploiter. Par exemple, au lieu de mesurer la consommation d'énergie, l'attaquant pourrait enregistrer les traces mémoires de l'exécution et les analyser (DCA). Ou au lieu d'injecter une faute dans une carte à puce à l'aide d'un laser, l'attaquant pourrait injecter une faute directement dans l'exécutable. Tout se fait de manière logiciel et plus physique.

3 Les outils de Side-Channel Marvels

Dans cette section je présenterai les différents outils de Side-Channel Marvels qui m'ont étés utiles pour la réalisation de certaines attaques. Tous ces outils sont accessibles à cette adresse :

https://github.com/SideChannelMarvels.

3.1 Le dépôt Tracer

Le dépôt Tracer est composé de deux plugins : TracerPin pour Intel Pin et TracerGrind pour Valgrind. L'un comme l'autre permettent d'enregistrer la trace mémoire (c'est à dire tout ce qui se passe en mémoire) d'une exécution. L'installation est détaillée dans les fichiers *README.md* présents dans le dépôt cependant je souhaite quand même préciser un problème que j'ai eu : il est nécessaire de compiler les sources de valgrind avec celles de tracergrind et donc de télécharger une archive de valgrind, incorporer le code source de tracergrind et modifier quelques fichiers de configuration avant la compilation et l'installation (tout cela est détaillé dans le dépôt). Dans leur guide, ils proposent d'utiliser l'archive **valgrind-3.12.0.tar.bz2** mais lors de la compilation des sources, j'ai eu une erreur que je n'ai pas réussi à résoudre. Pour remédier à ce problème, j'ai choisi d'utiliser plutôt la version de valgrind la plus récente (pour moi c'était **valgrind-3.16.1.tar.bz2**). Il est nécessaire ensuite de modifier 3 fichiers dans les fichiers sources de valgrind. Cette opération est normalement faites avec un **patch** des fichiers **.diff** mais j'ai du faire ces modifications manuellement car le dépôt Tracer de fournissait pas de fichier **.diff** pour cette version de valgrind. Vous pouvez le faire manuellement ou bien utiliser la commande **patch** avec le fichier **valgrind-3.16.1.diff** présent dans mon dépôt si vous avez la même version de valgrind.
Voici ce qu'il y a à modifier :

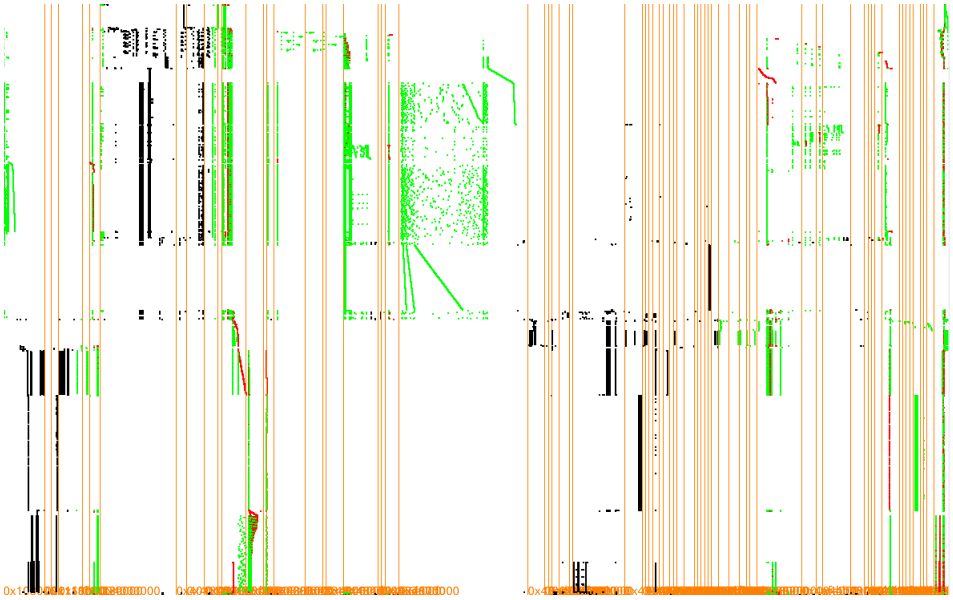
- Dans le fichier valgrind-3.16.1/Makefile.am, aux environs de la ligne 10, rajouter **tracergrind** \ en dessous de **helgrind** \.
- Dans le fichier valgrind-3.16.1/auxprogs/gen-mdg, aux environs de la ligne 58, rajouter **"tracergrind, "** après **"memcheck, "** dans la liste.
- Dans le fichier valgrind-3.16.1/configure.ac, aux environs de la ligne 4861, rajouter à la liste **tracegrind/Makefile** et **tracergrind/tests/Makefile** en dessous de **solaris/Makefile**.

Après cela il suffit de continuer à appliquer les instructions dans le *README.md* du dépôt sauf pour le **patch**.

Je ne parlerai que de TracerGrind dans ce récapitulatif car l'utilisation de ce dernier et celle de TracerPin est assez similaire. Voici un exemple basique de l'utilisation TracerGrind en ligne de commande :

```
valgrind --tool=tracergrind --filter=0x108000-0x4000000 --output=trace.grind
    ./a.out
```

Dans l'ordre des arguments : on précise qu'on veut utiliser l'outil tracergrind, on précise quel espace d'adresse mémoire on souhaite enregistrer (c'est optionnel, on peut également ne rien filtrer et tout prendre), on donne le nom du fichier dans lequel sera stocker la trace mémoire et pour finir on donne l'exécutable à enregistrer.
La trace est stocké directement sous forme binaire. Pour la lire, la stocker sous forme de base de données ou pour la visualiser, TracerGrind propose plusieurs programmes pour ça. Ils sont détaillés dans le dépôt et faciles à utiliser.
Voici par exemple, à quoi ressemble la trace mémoire d'une exécution d'un **ls** en ligne de commande sans filtres :



L'abscisse représente les adresses mémoires et l'ordonnée le temps (de haut en bas). Un point vert correspond à une lecture en mémoire, un point rouge à une écriture en mémoire et un point noir à l'exécution d'une instruction.

3.2 Le dépôt Daredevil

Le dépôt Daredevil contient le programme source permettant d'effectuer la CPA (Correlation Power Analysis) sur des traces stockées sous une forme particulière. Ce programme ne fait pas la différence entre des traces de la consommation d'énergie d'un matériel physique et des traces mémoire d'une exécution logiciel c'est pourquoi le format des données doit être particulier.
Le programme prend en entrée un fichier de configuration dont la syntaxe est décrite dans le dépôt. Je souhaite quand même préciser certaines choses :

3.2.1 Les samples

On considère qu'une trace est composée de "samples", ces samples représentent une donnée dans la trace. Par exemple pour une trace mémoire, un sample peut être une lecture, une écriture ou bien une exécution d'une instruction. La variable *index* dans le fichier de configuration va permettre de dire à Daredevil à partir de quel sample des traces il doit commencer à analyser les données. La variable *nsamples* permet de dire à Daredevil le nombre de sample qu'il doit analyser à partir d'*index*. L'intérêt que ces variables ont à être précisé est qu'on restreint la quantité de travail pour avoir une meilleure précision (par exemple ne cibler que le premier tour de l'AES). On aura donc une CPA plus rapide.

3.2.2 La position

Pour effectuer la CPA, Daredevil a besoin des messages d'entrées qu'on a utilisés pour générer les traces. À partir de ces derniers, Daredevil va construire les "guesses" : Daredevil va appliquer un ou exclusif, octets par octets, entre ces guesses et toutes les clés possibles puis appliquer une transformation grâce aux Lookup Table (LUT). Ainsi pour l'AES il aura simulé la première partie du premier tour pour tous les octets de clé possibles. Il calculera ensuite la corrélation entre la moyenne de ces résultats et la moyenne des données contenues dans chaque traces grâce au coefficient de Pearson. Un coefficient de corrélation sera affecté pour chaque octets de clé et celui ayant un coefficient plus élevé que les autres sera susceptible d'être un octet correct de la clé.
La position dans le fichier de configuration permet de préciser la position dans l'AES qu'on veut

attaquer. Les deux positions qui m'ont étés les plus utiles sont *AES_AFTER_MULTINV* et *AES_AFTER_SBOX*.

3.2.3 L'ordre et window

Daredevil offre la possibilité de faire une attaque d'ordre 1 et d'ordre 2. L'attaque d'ordre 1 est destinée aux implémentations n'utilisant pas de masquage.
Rappel du masquage : Le masquage permet de décomposer les variables sensible de l'algorithme (ici la clé) en plusieurs variables qu'on appelle masques. On va dérouler l'algorithme en utilisant ces masques. Certaines opérations de l'algorithme doivent être donc adaptées pour que le chiffré final reste le même. On dira qu'un masquage d'ordre 1 est la décomposition de la clé en 2 variables, un masquage d'ordre 2 est la décomposition de la clé en 3 variables, etc... L'intérêt du masquage est de modifier les valeurs intermédiaires et ainsi rendre la DPA et la CPA d'ordre 1 inefficace.
La CPA d'ordre 2 permet de casser les implémentations utilisant un masquage d'ordre 1. Daredevil va combiner chaque paires de samples (leur valeurs ayant été soustraite par la moyenne des traces) compris dans un certain intervalle (en terme de sample) lors du calcul de la corrélation. Cet intervalle est appelé "window" et est à préciser dans le fichier de configuration pour une CPA d'ordre supérieur à 1.
Exemple : Si on a au total dans une trace 20 samples, qu'on souhaite effectuer l'attaque seulement sur 10 samples à partir du 5ème, qu'on souhaite faire une CPA d'ordre 2 et que l'intervalle dans lequel on veut qu'il teste toutes les paires possible soit d'une taille de 3, on aura :

- index=4
- nsamples=10
- order=2
- window=3

3.3 Le dépôt Deadpool

Le dépôt Deadpool de Side-Channel Marvels propose un script Python *deadpool_dca.py* très utile permettant d'automatiser l'enregistrement de plusieurs traces et de convertir ces enregistrement en fichier exploitable par Daredevil. Il contient également un certain nombre d'attaques DCA complètes et détaillées (qui ont étés utiles pour moi pour m'entrainer).
Le fichier *deadpool_dca.py* détaille déjà bien comment l'utiliser cependant je souhaite quand même présenter son utilisation. Voici le script typique me permettant de générer des traces :

```
#!/usr/bin/python2

from deadpool_dca import *


def processinput(iblock, blocksize):
    return (None, ['%0*x' % (2*blocksize, iblock)])

def processoutput(output, blocksize):
    return int(output, 16)

t = TracerGrind('./mon_main.out', processinput=processinput,
    processoutput=processoutput, arch=ARCH.amd64, blocksize=16,
    addr_range='0x108000-0x4000000', stack_range='0x1ffefff000-0x1fffffffff')

t.run(1000)

bin2daredevil(configs={'attack_after_sbox':   {'algorithm':'AES',
    'position':'LUT/AES_AFTER_SBOX'},
    'attack_after_multinv':{'algorithm':'AES',
    'position':'LUT/AES_AFTER_MULTINV'},})
```

Ici on appelle le constructeur de la classe *TracerGrind* qui comprend plusieurs paramètres (certains sont optionnels) :

— Le nom de l'exécutable sur lequel on va enregistrer les traces
— Une fonction permettant de savoir la forme de ce que l'exécutable attend en entrée
— Une fonction permettent de savoir la forme de ce que l'exécutable donne en sortie
— L'architecture de la machine
— La taille d'un bloc (ici 16, c'est à dire 16 blocs de 8 bits = 128 correspondant à l'AES-128 bits)
— L'intervalle en terme d'adresse qu'on veut enregistrer
— L'intervalle en terme d'adresse de la pile

Suite à cela, on appelle la méthode *run* en précisant le nombre de traces qu'on veut enregistrer (ici 1000). Et pour finir, on appelle la fonction *bin*2*daredevil* qui transforme les traces enregistrées en fichiers que le programme daredevil accepte pour faire une CPA. Ici on précise qu'on veut que cette fonction produise un fichier de configuration faisant l'attaque à la position après la S-Box et l'autre à la position après la multiplication inverse.

Ce script produit alors, si tout s'est bien passé :

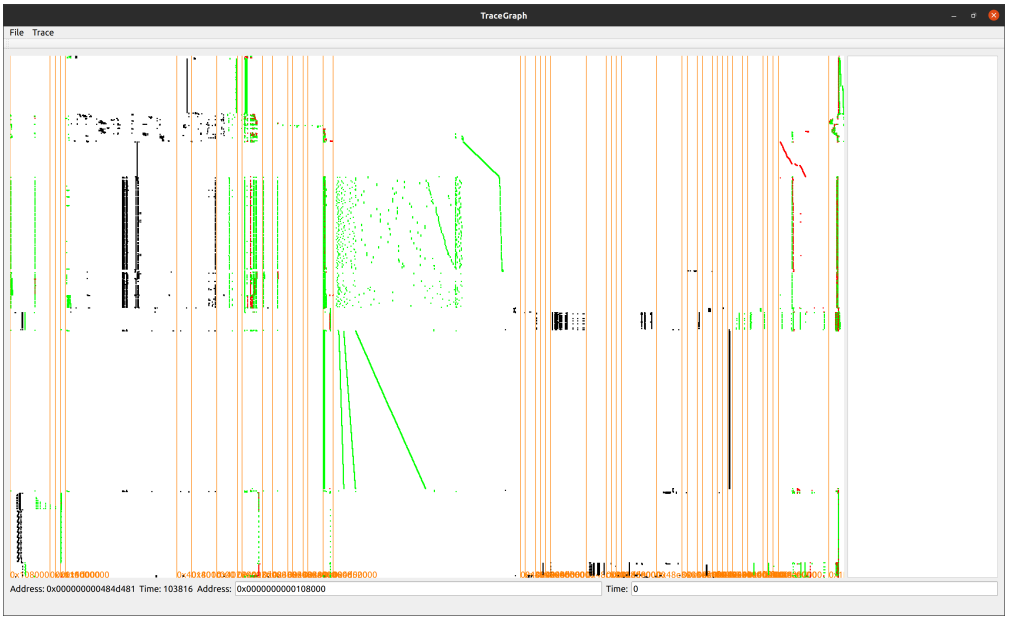
— Des fichiers *.input* et *.output* contenants respectivement les entrées données à l'exécutable et les sorties renvoyées par l'exécutable dans l'ordre. Plusieurs fichiers de la sorte sont générés, seul le nom diffère (utile pour Daredevil).
— Des fichiers *.trace* contenants les données des traces. Par défaut, on aura les fichiers commençants par *mem_data* permettant de faire l'attaque sur les données contenues dans les différentes adresses, les fichiers commençants par mem_addr1 permettant de faire l'attaque sur la valeur des adresses, et les fichiers commençants par *stack* permettant de faire l'attaque sur les données contenues dans la pile.
— Des fichiers $.config$ contenants toutes les informations nécessaires pour lancer une CPA avec Daredevil. Ces fichiers sont à donner en entrée au programme daredevil. Deadpool ne permet pas de générer des fichiers de configuration pour une attaque d'ordre 2, il est donc nécessaire de modifier/créer soit même un fichier de configuration pour faire ce type d'attaque.

4 Mes différentes attaques

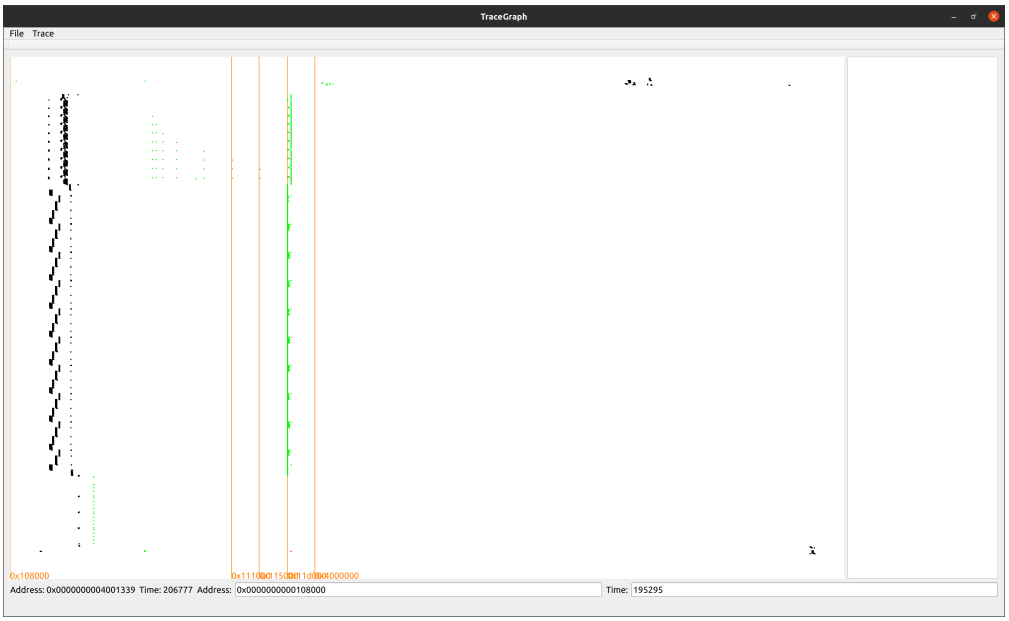
4.1 Implémentation basique de l'AES-128

J'ai pu récupérer une implémentation basique de l'AES-128 (sans protection contre les attaques en boîte blanche) sur ce lien : https://github.com/dhuertas/AES.

Tout d'abord on lance l'enregistrement complet d'une trace mémoire de l'exécution du programme. Voici une capture d'écran de ce qu'on obtient :



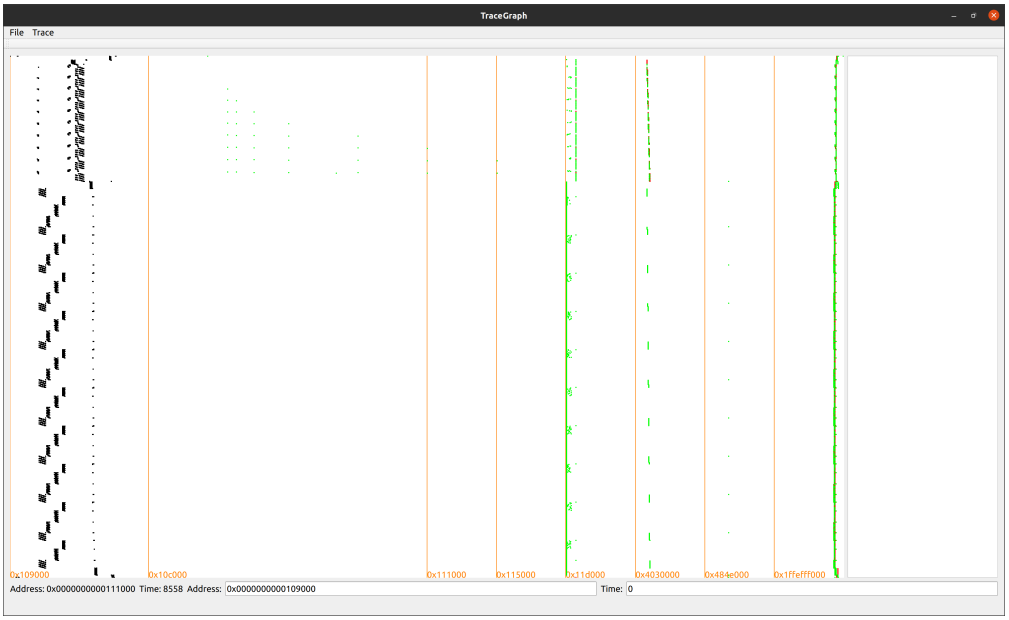
À priori il est assez difficile de distinguer un motif qui pourrait correspondre à l'AES. Cependant, lorsqu'on zoom et qu'on parcours un peu cette représentation graphique, on peut remarquer ceci :



Ici on voit clairement la répétition de 10 même motifs puis la répétition de 9 autres motifs + 1 plus petit à la fin. Cela pourrait vraiment correspondre à la génération des sous-clés et aux 10 tours de l'AES.
On relance alors un enregistrement avec cette fois ci un filtre : on ne garde que les intervalles sur lesquels on observe des lectures/écritures/instructions sur l'espace de temps des motifs qu'on vient de remarquer. Ici le filtre qu'on pose est :

```
"0x108000-4000000, 0x402d000-0x4830000, 0x484d000-0x4852000"
```

Voici ce que nous obtenons :



Cela correspond à ce que nous cherchons. On observe également que la trace comprend bien la pile (tout à droite).
On lance maintenant l'enregistrement de 30 traces avec deadpool pour commencer puis on effectue une CPA avec daredevil avec le fichier de configuration utilisant les données et ayant pour position $AES\_AFTER\_MULTINV$. Voici ce qu'on obtient dans le terminal pour le premier octet :

```
Best 10 candidates for key byte #0 according to sum(abs(bit_correlations)):
 1: 0x14  sum: 5.74918
 2: 0xb2  sum: 5.65233
 3: 0xc5  sum: 5.57721
 4: 0xe5  sum: 5.5731
 5: 0x47  sum: 5.56268
 6: 0x87  sum: 5.54151
 7: 0x6d  sum: 5.52431
 8: 0x42  sum: 5.52314
 9: 0x6e  sum: 5.51673
10: 0x4d  sum: 5.50261

Best 10 candidates for key byte #0 according to highest abs(bit_correlations):
 1: 0x05  peak: 0.874475
 2: 0x14  peak: 0.872872
 3: 0x08  peak: 0.872872
 4: 0xf4  peak: 0.872872
 5: 0xc5  peak: 0.861111
 6: 0xa1  peak: 0.856459
 7: 0x53  peak: 0.817996
 8: 0x64  peak: 0.817996
 9: 0x1b  peak: 0.816497
10: 0xa9  peak: 0.816497
```

Ici on n'observe pas de coefficient de corrélation se démarquant particulièrement des autres (l'écart entre le premier plus élevé et le deuxième plus élevé est assez faible). On obtient un résultat similaire pour tous les autres octets et donc on ne peut pas en déduire la clé.
On essaye alors la même chose mais en utilisant le fichier de configuration ayant pour position AES_AFTER_SBOX et voici ce qu'on obtient pour le premier octet :

```
Best 10 candidates for key byte #0 according to sum(abs(bit_correlations)):
 1: 0x00  sum: 8
 2: 0x82  sum: 5.70328
 3: 0x04  sum: 5.67891
 4: 0x09  sum: 5.66799
 5: 0x7e  sum: 5.64147
 6: 0xa8  sum: 5.54617
 7: 0x74  sum: 5.53941
 8: 0x3c  sum: 5.53057
 9: 0x2e  sum: 5.52619
10: 0x31  sum: 5.52272

Best 10 candidates for key byte #0 according to highest abs(bit_correlations):
 1: 0x00  peak: 1
 2: 0x71  peak: 0.874475
 3: 0x09  peak: 0.870106
 4: 0x21  peak: 0.866667
 5: 0x39  peak: 0.817996
 6: 0x96  peak: 0.817996
 7: 0x87  peak: 0.817996
 8: 0x82  peak: 0.817996
 9: 0xfc  peak: 0.816497
10: 0x3a  peak: 0.816497
```

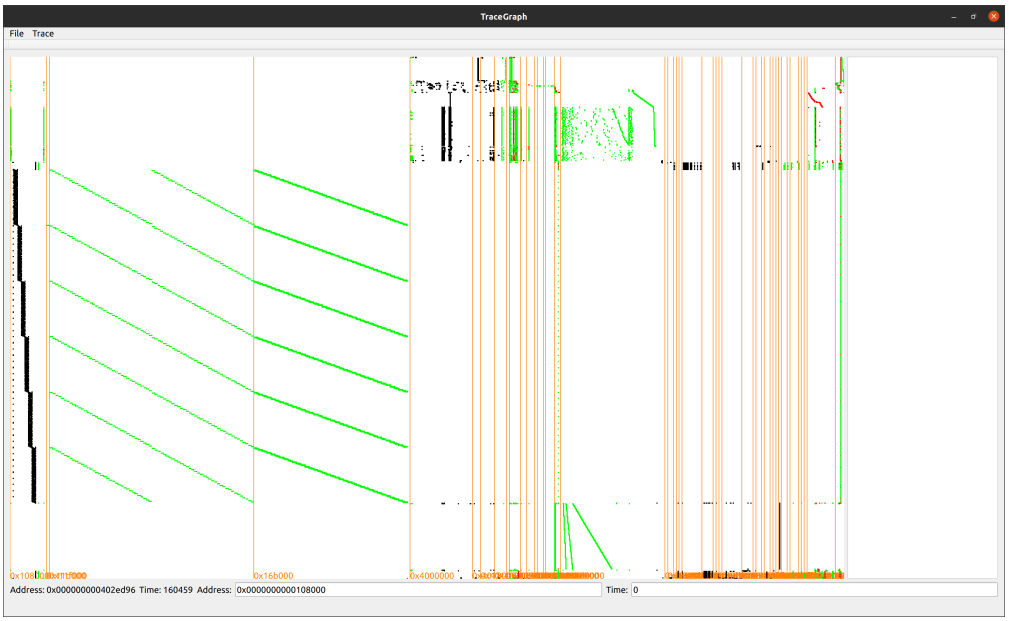
Ici on observe une réelle différence entre le coefficient le plus élevé et le deuxième plus élevé. On en déduit donc que cet octet pourrait vraiment correspondre au vrai premier octet de la clé. On obtient cela pour tous les octets et on peut donc en déduire la clé. Ici la clé obtenue était $000102030405060708090A0B0C0D0E0F$ (choisi par moi-même).

4.2 WhibOx Contest

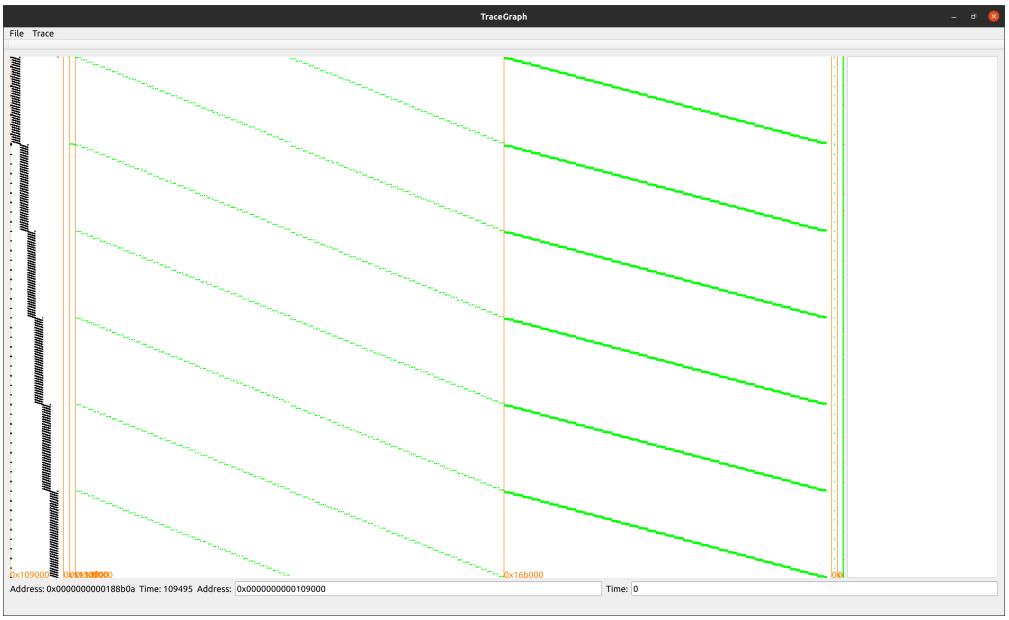
Il est possible de trouver plusieurs implémentations en boîte blanche de l'AES-128 sur le site https://whibox-contest.github.io. J'y suis donc allé pour essayer de casser des implémentations plus protégées qu'une implémentation basique.

4.2.1 WhibOx Contest 2019 numéro 26

Je me suis intéressé à l'implémentation 26 *distracted_leavitt* du WhibOx Contest de 2019. Comme pour l'implémentation basique, on commence par générer une trace complète de l'exécution. Voici ce qu'on obtient :



On remarque ici clairement un motif particulier entre l'adresse $0x108000$ et $0x4000000$. En noir (les instructions) on voit la répétition de 10 motifs qui pourrait correspondre à l'AES-128. On remarque cependant qu'en vert, la quantité de lecture est très importante mais ne ressemble pas à l'exécution d'un AES-128. Ceci à dû à la manière d'implémenter l'AES. En effet l'implémentation comporte la définition de grandes tables dans lesquelles l'algorithme va lire des données pour dérouler l'AES. Ces tables contiennent notamment la clé de manière dissimulé.
Voici maintenant ce qu'on obtient en appliquant le filtre adapté :



On génère pour commencer 30 traces avec ce filtre et on lance daredevil sur les adresses mémoire en ciblant $AES\_AFTER\_MULTINV$. On obtient 4 résultats différents pour les différents octets :

— Cas 1 :

```
Best 10 candidates for key byte #0 according to sum(abs(bit_correlations)):
 1: 0xfc  sum: 6.54827
 2: 0x77  sum: 6.52164
 3: 0x67  sum: 6.51673
 4: 0x0b  sum: 6.50973
 5: 0x07  sum: 6.46086
 6: 0xaa  sum: 6.45003
 7: 0x9d  sum: 6.43024
 8: 0x48  sum: 6.42129
 9: 0x1a  sum: 6.41757
10: 0x79  sum: 6.4108

Best 10 candidates for key byte #0 according to highest abs(bit_correlations):
 1: 0x9d  peak: 1
 2: 0x2e  peak: 0.935414
 3: 0x18  peak: 0.935414
 4: 0x45  peak: 0.934853
 5: 0x32  peak: 0.9337
 6: 0xfc  peak: 0.9337
 7: 0x24  peak: 0.9337
 8: 0x77  peak: 0.931891
 9: 0x1a  peak: 0.875
10: 0xe9  peak: 0.875
```

Quand on regarde la somme des valeurs absolues de la corrélation on ne distingue pas une valeur se distinguant des autres alors que lorsqu'on regarde la valeur absolue de la corrélation, on observe qu'un octet pourrait bien correspondre.

— Cas 2 :

```
Best 10 candidates for key byte #5 according to sum(abs(bit_correlations)):
 1: 0xcf  sum: 6.76961
 2: 0x78  sum: 6.6159
 3: 0x85  sum: 6.54919
 4: 0xd4  sum: 6.53145
 5: 0xec  sum: 6.5218
 6: 0x79  sum: 6.49447
 7: 0x18  sum: 6.46306
 8: 0x69  sum: 6.46128
 9: 0x77  sum: 6.45876
10: 0xd7  sum: 6.45231

Best 10 candidates for key byte #5 according to highest abs(bit_correlations):
 1: 0xcf  peak: 1
 2: 0x03  peak: 0.935414
 3: 0x9d  peak: 0.935414
 4: 0x30  peak: 0.934853
 5: 0x88  peak: 0.934853
 6: 0x4c  peak: 0.9337
 7: 0x64  peak: 0.9337
 8: 0x6a  peak: 0.9337
 9: 0x05  peak: 0.931891
10: 0xe7  peak: 0.875
```

On remarque la même chose que pour le cas précédent sauf que le "meilleur" octet indiqué par la somme des valeurs absolues est le même que celui indiqué par la valeur absolue.

— Cas 3 :

```
Best 10 candidates for key byte #6 according to sum(abs(bit_correlations)):
 1: 0x85  sum: 6.75081
 2: 0xd1  sum: 6.5477
 3: 0x9f  sum: 6.53556
 4: 0x9b  sum: 6.50882
 5: 0x61  sum: 6.49853
 6: 0x64  sum: 6.48858
 7: 0x6c  sum: 6.47032
 8: 0x5f  sum: 6.46802
 9: 0x5d  sum: 6.46542
10: 0x43  sum: 6.45192

Best 10 candidates for key byte #6 according to highest abs(bit_correlations):
 1: 0x85  peak: 1
 2: 0xc6  peak: 1
 3: 0xdf  peak: 0.935414
 4: 0x0e  peak: 0.935414
 5: 0x64  peak: 0.935414
 6: 0xea  peak: 0.934853
 7: 0x88  peak: 0.934853
 8: 0x60  peak: 0.934853
 9: 0x30  peak: 0.9337
10: 0xd1  peak: 0.9337
```

On remarque la même chose que pour le cas précédent sauf que les deux meilleurs octets pour la valeur absolue, ont la même valeur.

— Cas 4 :

```
Best 10 candidates for key byte #9 according to sum(abs(bit_correlations)):
 1: 0x45  sum: 6.71451
 2: 0xa5  sum: 6.49431
 3: 0x2c  sum: 6.48056
 4: 0x5d  sum: 6.47316
 5: 0x73  sum: 6.46743
 6: 0x77  sum: 6.44824
 7: 0x62  sum: 6.44772
 8: 0x43  sum: 6.42933
 9: 0x42  sum: 6.42744
10: 0xe5  sum: 6.4199

Best 10 candidates for key byte #9 according to highest abs(bit_correlations):
 1: 0x77  peak: 0.935414
 2: 0x45  peak: 0.935414
 3: 0x4f  peak: 0.9337
 4: 0x8a  peak: 0.9337
 5: 0x5d  peak: 0.931891
 6: 0x3c  peak: 0.92932
 7: 0xa5  peak: 0.92932
 8: 0xad  peak: 0.92582
 9: 0x02  peak: 0.92582
10: 0x42  peak: 0.875
```

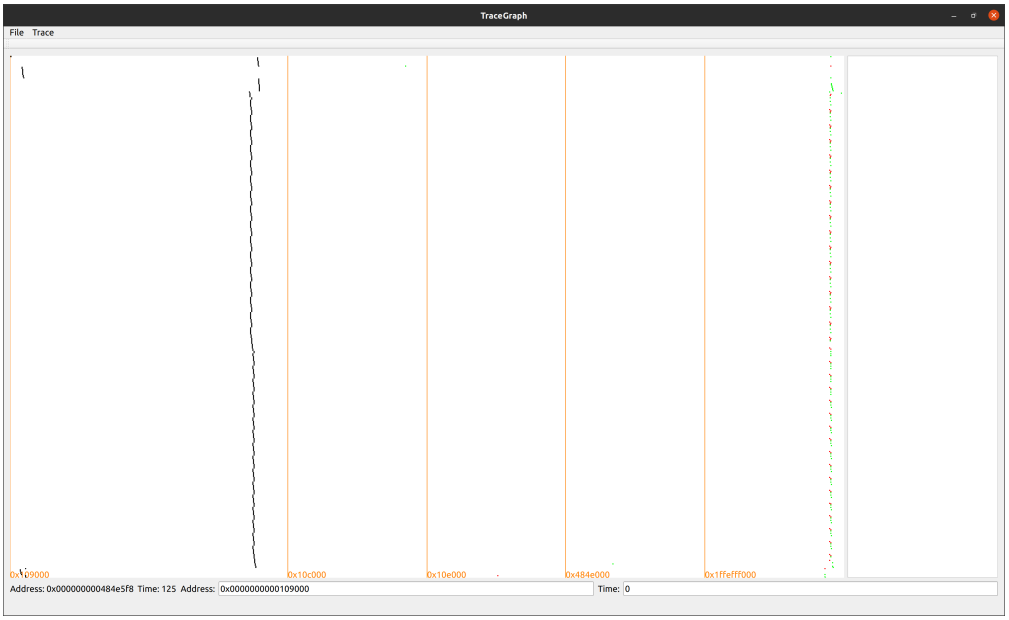
Dans ce cas la valeur $0x45$ pourrait bien correspondre mais il est difficile d'en être sûr car l'écart entre les valeurs n'est pas très marqué.

On peut donc noter la clé potentielle que nous avons obtenu et marquer les octets dont nous ne sommes pas certain. On relance alors plusieurs fois daredevil en changeant la position et le type de donnée que nous attaquons (adresses mémoire, données contenues aux adresses et données de la pile). On croise les différents résultats obtenus et on obtient la clé : $9D797E44B9CF850B21??8406??E3??4E$. Les ?? correspondent aux octets que nous n'avons pas déterminés.
On essaye alors de refaire la même chose mais en augmentant le nombre de traces. Je n'ai pas réussi à obtenir plus de résultat en augmentant le nombre de traces (je suis allé jusqu'à 200 traces). Comme il ne manque que 3 octets à déterminer, j'ai écrit un programme me permettant de faire une recherche exhaustive sur certains octets. Ainsi j'ai pu retrouver la clé complète étant : $9D797E44B9CF850B21DD8406FCE3AC4E$.

4.2.2 WhibOx Contest 2017 numéro 748

L'implémentation 748 "hungry_bardeen" du WhibOx Contest de 2017 a été intéressante car elle a soulevé un problème que je n'ai malheureusement pas réussi à résoudre.
Tout d'abord j'ai suivi la même première étape que pour les implémentations précédentes : générer une trace complète et l'afficher pour déterminer dans quel espace se déroule l'AES. Voici ce qu'on obtient :

On ne remarque à priori rien qui pourrait ressembler à l'AES.
Remarque : Certains motifs sont toujours présents pour tous les enregistrements de traces. Il s'agit, pour la plupart, des chargements des différentes librairies système nécessaire à l'exécution. J'ai alors filtré l'adresse sur l'espace $0x108000 - 0x4000000$ correspondant généralement à l'espace dans lequel se trouve le déroulement de l'AES (au vu de certaines autres implémentations) :

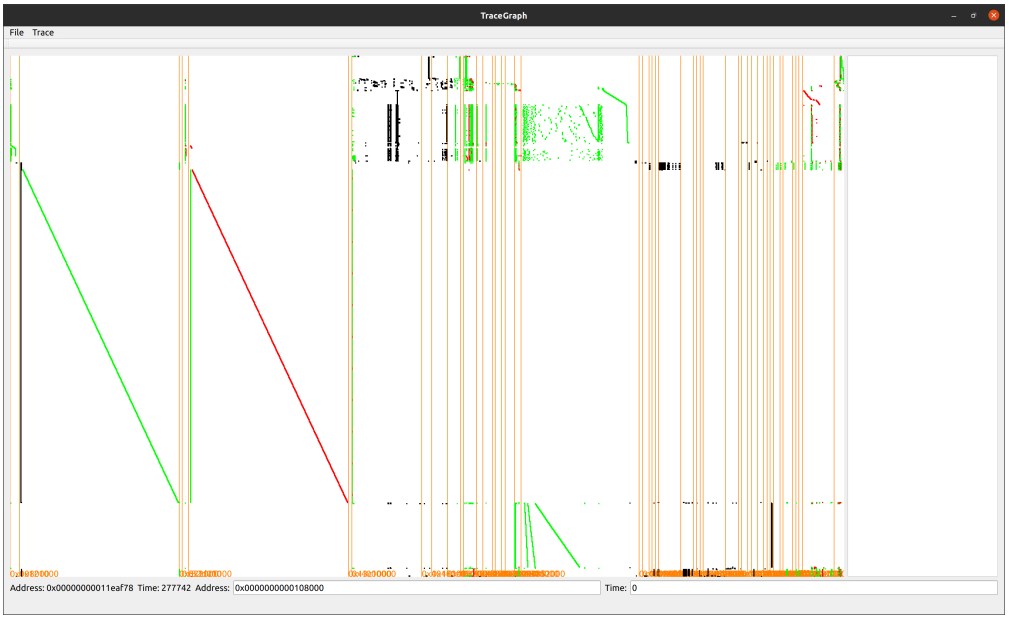


On observe que c'est très pauvre en terme de quantité de lecture et d'écriture comparé aux autres implémentations. Je lance quand même daredevil avec 200 enregistrements différents (filtrés avec ce filtre) mais je n'obtient aucun résultat intéressant. Suite à ça je me suis penché plus sur la trace complète et j'ai essayé de comprendre mieux comment l'implémentation fonctionnait en regardant le code source mais je rien réussi à en déduire.
Je décide de lancer l'enregistrement de 200 traces mémoire mais cette fois ci sans filtres ce qui rend l'attaque moins précise et moins efficace mais au moins me permet de tout englober. En lançant daredevil sur les données contenues dans les différentes adresses, on n'obtient, pour quasiment

tous les octets, aucun résultat se démarquant réellement des autres. 1 ou 2 octets cependant ont l'air d'avoir un score un peu meilleur. Je répète alors la même chose mais cette fois ci avec 1000 enregistrements. On obtient le même genre de résultat que précédemment. Et pareillement avec 2000.
J'ai rencontré ce problème sur plusieurs implémentations.

4.2.3 WhibOx Contest 2017 numéro 777

L'implémentation 777 "adoring_poitras" du WhibOx Contest de 2017 m'a elle aussi posé un problème que j'ai rencontré sur plusieurs implémentations : lorsque je lance l'enregistrement d'une trace mémoire, l'enregistrement ne s'arrête jamais. J'ai longtemps essayé de résoudre ce problème par différents moyens notamment en essayant de paramétrer Valgrind d'une autre manière, en utilisant TracePin et en rajoutant de la puissance à ma machine virtuel mais cela n'a donné aucun résultat.
J'ai essayé plus tard de forcer l'exécution de l'AES à se stopper en ajoutant un processus dans ma fonction *main*() et en ajoutant un timer : le processus père attend un certain nombre de secondes et tue le processus fils exécutant l'AES. La trace mémoire n'est donc pas complète, cependant comme daredevil n'attaque que sur le premier tour de l'AES, on peut espérer que le premier tour s'est au moins déroulé. Le chiffré en sortie devient également erroné mais ce n'est pas un problème car daredevil n'as pas besoin d'utiliser les messages de sorti de l'AES pour effectuer la CPA.
Voici ce qu'on obtient :



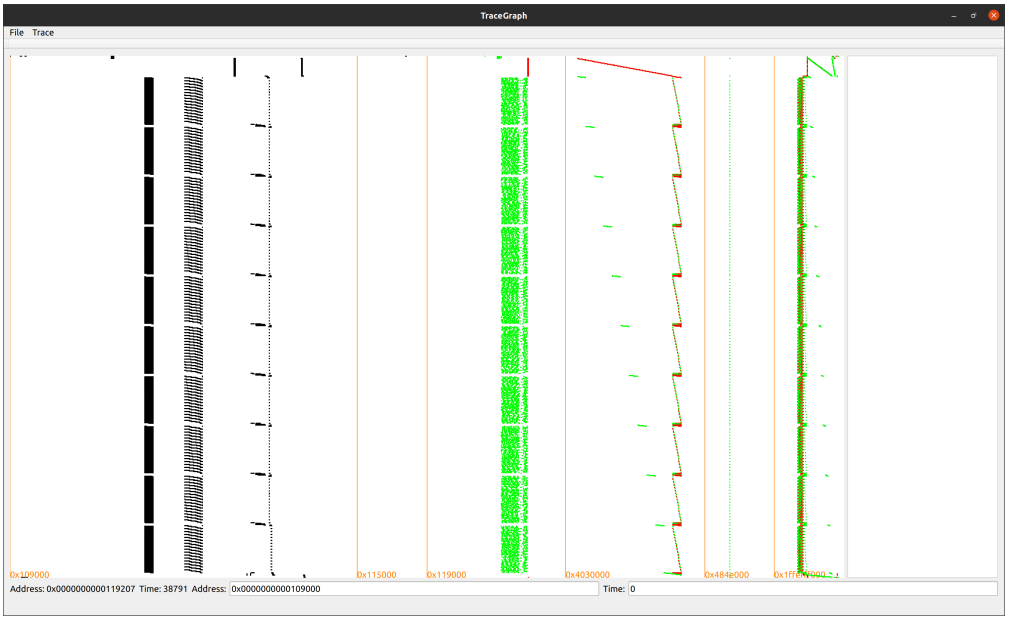
On remarque clairement un motif inhabituel prenant la plus grande place dans cette trace mémoire.
On lance l'enregistrement de 100 traces pour commencer et on exécute daredevil avec les différentes configurations possibles. On obtient le même genre de résultat que pour l'implémentation précédente : aucune déduction possible sur les différents octets de clé sauf quelques suspicions pour certains.
On essaye cette fois ci de lancer une attaque d'ordre 2 (dans le cas où il y aurait un masquage d'ordre 2) avec différents paramètres mais on n'obtient rien non plus.
On décide d'augmenter le nombre de traces à 200 et cette fois ci. Aucun résultat ne se distingue et rien ne va dans le sens des suspicions faites pour 100 traces. Pareillement pour 800 traces.
J'ai essayé longtemps plusieurs configurations possible mais je ne suis pas arrivé à trouver la clé pour cette implémentation.
La solution étant peut être d'augmenter encore le nombre de traces, cependant l'enregistrement de 800 traces et l'exécution de daredevil pour cette implémentation était déjà très long.

J'ai retravaillé sur cette implémentation un peu plus tard et ait constaté quelque chose dont je ne m'étais pas aperçu initialement : le formatage automatique des traces avec deadpool ne se faisait pas bien. Deadpool considérait que la taille des données à analyser était de 4 octets alors qu'en réalité elle était de 8 octets. Ceci était dû au fait que j'utilisais les filtres par défaut proposés par deadpool pour le formatage. J'ai donc relancé l'acquisition de traces en spécifiant les filtres à utiliser pour le formatage (la variable $filters$ en paramètre de la classe $TracerGrind$ et la variable $keywords$ en paramètre de la fonction $bin2daredevil$) toujours en forçant l'arrêt de l'exécution au bout d'une seconde. Cela n'a malheureusement pas donné de résultats exploitables après le lancement de daredevil pour 100 traces. Je n'ai pas essayé avec plus de traces car l'analyse était déjà très longue (environ 18 minutes par octet).

4.3 Implémentation de Rivain-Prouff avec masquage d'ordre 1

Il est possible de trouver sur le dépôt https://github.com/coron/htable plusieurs implémentations du DES et de l'AES utilisant différentes contremesure. Ici je me suis intéressé à l'implémentation utilisant la contremesure de Rivain-Prouff avec un masquage d'ordre 1.
On commence par générer une trace complète :

On reconnait assez bien le motif de l'AES :



On lance alors l'enregistrement de 100 traces avec les filtres mémoire couvrant toute la zone dans laquelle l'AES se déroule. On lance une CPA d'ordre 1 avec daredevil et on obtient aucune information sur les octets de clé. On essaye en utilisant plusieurs positions différentes et plusieurs types de données différents et le résultat reste le même. Ceci est cohérent car l'attaque est d'ordre 1 alors que l'implémentation contient un masquage d'ordre 1.
On essaye maintenant en changeant les fichiers de configuration : On met la variable *order* à 2 et la variable *window* à 10 pour commencer. Voici comment se présente le résultat :

```
[ATTACK] Key byte number 2

Rank   Correlation    Key       Sample(s)
 0.       -0.558277    0x10     61699   61705
 1.        0.548059    0x65    185550  185559
 2.        0.536643    0x97    156881  156890
 3.        0.532755    0x94     16990   16991
 4.       -0.529992    0xde    127768  127776
 5.       -0.525062    0xa2    148942  148947
 6.       -0.524137    0x71    152810  152813
 7.        0.521855    0x80     17342   17345
 8.        0.521531    0xd3    115064  115066
 9.        0.521318    0xda     10461   10469
10.       -0.519089    0x42    169887  169894
11.       -0.518245    0x8     125203  125205
12.        0.517358    0x61    137080  137084
13.        0.517301    0xf1    190793  190796
14.       -0.516811    0x59     10916   10923
15.        0.516135    0x6d     26045   26045
16.       -0.514982    0xca    147713  147718
17.       -0.514931    0x6a     11215   11217
18.       -0.512222    0x13     57882   57883
19.        0.510884    0xec     80969   80971

[INFO] Attack of byte number 2 done in 16.456649 seconds.
```

Aucun résultat ne semble se démarquer. L'idée initiale était que cette "fenêtre" d'analyse (donnée par la variable *window*) n'était pas suffisante. On relance alors daredevil plusieurs fois avec

les même traces en augmentant progressivement la variable *window*. Cela n'amène pas de résultat positif et on remarque que le temps de l'analyse augmente considérablement lorsqu'on incrémente cette variable. Voici la liste des choses faites pour tenter d'obtenir un résultat :

- Augmenter le nombre de trace
- Diminuer le nombre de sample à analyser (essayer de cibler plus précisément le premier tour de l'AES)
- Augmenter la taille de la fenêtre
- Comme on avait accès au code source, restreindre l'exécution de l'AES à l'exécution du premier tour et ainsi obtenir des fichiers de trace moins volumineux

Finalement en "jouant" avec tous ces paramètres tout en gardant une durée de l'attaque faisable, je n'ai eu aucun résultat positif.
Je me suis penché alors plus particulièrement sur la compréhension de ce que faisait daredevil lors d'une CPA du second ordre. J'ai eu, pendant un moment, plusieurs incompréhension concernant la validité du code mais finalement, il me semble bien correct dans ce qu'il fait.
Une autre solution serait peut-être d'augmenter considérable la variable *window* et le nombre de traces par rapport à ce que j'ai fait. Je n'ait donc pas réussi à trouver la clé pour cette implémentation.

5 Liens Utiles

- Utilisation et explication des outils de SideChannelMarvels et attaques par DCA par les créateurs des outils de SideChannelMarvels :
  `https://www.sstic.org/media/SSTIC2016/SSTIC-actes/design_de_cryptographie_white-box_et_a_la_fin_c_es/SSTIC2016-Article-design_de_cryptographie_white-box_et_a_la_fin_c_est_kerckhoffs_qui_gagne-hubain_teuwen_1.pdf`

- Cryptanalyse sur les implémentations en boîte blanche de l'AES :
  `https://link.springer.com/content/pdf/10.1007%2F978-3-540-30564-4_16.pdf`

- Passer outre un encodage affine sur une implémentation en boîte blanche :
  `https://eprint.iacr.org/2019/096.pdf`

- Attaques et contremesures sur les conceptions en boîte blanche (notamment avec masquage) :
  `https://eprint.iacr.org/2018/049.pdf`

- Contre-mesure contre les DCA d'ordre supérieur à 1 :
  `https://eprint.iacr.org/2010/523.pdf`
  `https://link.springer.com/content/pdf/10.1007%2F978-3-642-15031-9_28.pdf`

- Attaques sur des implémentations en boîte blanche :
  `https://eprint.iacr.org/2018/098.pdf`