LIVRABLE : Projet de cryptographie

François Tambidore
60355

2023

TABLE DES MATIÈRES

I. INTRODUCTION

Dans l'univers des cybermenaces, il existe plusieurs types d'attaques notamment les attaques invasives et non invasives. Les attaques invasives tentent de décrypter directement des données dans les composants internes (ex : lorsqu'un individu tente d'intercepter les flux de trafic de données dans le but d'observer et voler les transferts entre 2 réseaux…). Quant aux attaques non invasives, ces dernières n'exploitent que les informations externes peu contrôlables comme une émission quelconque d'une onde (ex : des ondes électromagnétiques, thermiques…) qui permettent de déduire des informations de manière indirecte comme la consommation d'énergie, la durée de fonctionnement etc. L'attaque dont nous allons parler utilise cette méthode.

De même, on parle souvent dans ce domaine très vaste d'attaques actives et passives. Les attaques actives sont généralement des attaques directes qui veulent altérer le fonctionnement d'un système et impacter le service voulu dans des buts lucratifs, politiques ou autres. Ce sont majoritairement les ransomwares, les dénis de service et les chevaux de Troie qui utilisent cette méthode. Les attaques passives en revanche se contentent d'observer le comportement des appareils pendant leur traitement sans les perturber. Un attaquant serait totalement invisible et indétectable par le système attaqué ce qui en fait une manière certes plus lente et moins dangereuse à priori mais elle peut être tout autant efficace et dévastatrice.

Le terme « Side Channel Attack » — que j'appellerai SCA pour simplifier — traduit en français « attaques par canaux auxiliaires » est donc une attaque passive et non-invasive qui exploite les émissions électromagnétiques, thermiques… produites par un ordinateur pour décrypter via des analyses statistiques des mots de passe ou données sensibles. Les SCA sont alors une forme de cryptanalyse, c'est-à-dire une menace que la cryptographie ne peut pas toujours vaincre.

Pour rappel, la cryptographie consiste à masquer un message en le convertissant en un texte crypté et de le diffuser dans des canaux non sécurisés (on l'utilise lors de transactions bancaires, E-mails…). La cryptographie a évolué en de multiples facettes notamment avec les systèmes de clé symétriques (aujourd'hui obsolète utilisant une clé pour envoyer et recevoir un message) et asymétriques (à base de clés publiques et privées). Cette dernière combinée à des techniques de hachage assure l'authenticité, l'intégrité et la confidentialité des données partagées dans le réseau.

La cryptanalyse est le fait de pouvoir restituer un texte en clair à partir de messages chiffrés par des algorithmes et hash, le tout sur un canal non sécurisé. C'est une technique de décryptage avancée qui permet donc de retrouver le message originel avant conversion en texte chiffré. On retrouve dans cette catégorie d'autres attaques relativement connues et généralement vaincues grâce à la cybersécurité notamment Chosen Plaintext Attack, Man In The Middle, Brute Force et attaques par Dictionnaires.

C'est pourquoi les SCA sont généralement considérées comme des méthodes de cryptanalyse latérales très performantes car elles ne cherchent pas à casser les algorithmes de chiffrement en étudiant leurs propriétés mathématiques mais plutôt

analyser les déperditions / fuites d'informations de consommations énergétiques ou temporelles. Dans ce cas-là, un attaquant n'a même pas besoin de connaître ou avoir de quelconques informations sur la clé secrète car les SCA sont capables de contourner les algorithmes de cryptographie et ce, qu'ils soient récents ou non.

Pour représenter visuellement le principe d'une attaque auxiliaire, prenons l'exemple d'un hacker qui voudrait espionner le trajet d'un conducteur. Dans ce cas, l'attaque la plus évidente serait de suivre la voiture ou d'observer le chemin par GPS. Une attaque auxiliaire consisterait ici à faire des mesures de variation de quantité d'essence dans le réservoir, d'étudier le poids de la voiture, son moteur, son couple de rotation… pour déterminer les lieux visités, les distances parcourues, le contenu de la voiture et le nombre de passagers… le tout sans s'en prendre directement à la voiture.

C'est le chercheur Paul Kocher qui a trouvé cette nouvelle méthode d'attaque dans les composants semi-conducteurs CMOS, utilisés en masse dans les circuits des ordinateurs des années 2000. Dans son analyse, il démontrait les failles de sécurité que les SCA relevaient avec des études statistiques…

II. LES DIFFÉRENTS TYPES DE SIDE CHANNEL ATTACK

Aujourd'hui, la majorité des puces et composants des systèmes modernes (téléphones, consoles et PC) ont des modules de protection anti-SCA. De même, les premiers modèles de SCA qui étaient capables de briser les algorithmes d'époque ne sont plus véritablement exploitables. En outre, de la même manière que les systèmes de cryptographie ont évolué pour améliorer la sécurité des données, les systèmes de cryptanalyse ont tout aussi été renforcé au fil du temps pour pouvoir contourner les dernières technologies de chiffrement et rendre difficile la protection des données.

Les SCA sont bien plus courantes aujourd'hui en raison de l'augmentation de la sensibilité des équipements de mesure car les hackers peuvent récolter des données précises provenant d'un système observé dès lors qu'il est en cours d'utilisation. De plus, l'augmentation de la puissance de calcul (comme énoncé par la Loi de Moore) ainsi que le Machine Learning ont drastiquement aidé les hackers à développer leur compréhension et interprétation des données brutes qu'ils extrayaient. Comme je l'énonçais, toute l'évolution technologique a permis aux attaquants de mieux exploiter les changements subtils entre plusieurs systèmes (capacités de composants, puces, processeurs, clés cryptographiques…). Nous allons donc étudier de manière relativement simplifiée les différents types d'attaques par canaux auxiliaires, leurs fonctionnements et se concentrer sur 2 méthodes en particulier.

1. POWER ANALYSIS ATTACK

Découverte en 1996 par des chercheurs israéliens, cette attaque – considérée comme une menace sérieuse par le NIST en 2001 – est probablement la plus populaire parmi les différentes Side Channel Attacks. En effet, le concept est de mesurer la consommation de puissance électrique produite par l'appareil visé pendant qu'il est utilisé. Le principe

est d'observer et analyser ce qu'on appelle des traces de puissances qui résument l'activité de la consommation énergétique d'un ordinateur. En l'occurrence, un hacker tentera d'utiliser cette attaque lorsqu'une opération particulière de chiffrement est effectuée (ex : un transfert de données) et récoltera les traces pour les étudier.

Ainsi, cette catégorie d'attaque est connue pour être très efficace dans l'extraction des clés de chiffrement et ne nécessite pas de matériel couteux pour collecter des traces de puissance. Cependant, cette attaque est assez sensible aux variations de l'environnement et du matériel. C'est-à-dire que les conditions de mesures comme le type d'équipement étudié, son processeur, la température extérieure etc. peuvent influer la consommation et ce pour toute exécution ou processus ; de même, la présence de bruits peut affecter les résultats des mesures. C'est donc une catégorie d'attaque très connue et capable de briser les algorithmes modernes notamment RSA qui est toujours utilisé dans certaines technologies. Les ordinateurs d'aujourd'hui ont des mesures pour brouiller les pistes et éviter qu'un tiers puisse analyser les signatures électriques mais cela ne signifie pas qu'ils sont hors d'état de nuire.

2. ELECTROMAGNETIC ANALYSIS ATTACK

Historiquement, les premières attaques auxiliaires étaient en réalité des attaques électromagnétiques. L'attaque par canal électromagnétique a été découverte par Paul Kocher et son équipe en 1997. Ces derniers ont démontré qu'il était possible d'analyser des signaux électromagnétiques émis par des appareils utilisant des circuits cryptographiques à la même manière que la Power Analysis. En effet, avec un appareil de mesures spécialisé (oscilloscope ou analyseur de spectre), un attaquant peut étudier le comportement du système de chiffrement et déduire les informations cachées.

Ce genre d'attaques est particulièrement efficace car il peut être utilisé sur une kyrielle de circuits cryptographiques notamment les chiffrements symétriques, asymétriques mais aussi les fonctions de hachage ce qui rend très dangereuse cette attaque. Exploiter les ondes électromagnétiques est utile pour mener une attaque sans contact physique et spécifiquement sur les appareils non électroniques comme les cartes à puces. En revanche, la collecte de signaux électromagnétiques est plus complexe car sensible aux interférences et demande un matériel sophistiqué pour se faire.

Pour contrer ce type d'attaque, les constructeurs de composants ont créé des mesures pour rendre les signaux émis moins visibles grâce à des techniques de modulation de signal moins couteuses en énergie, un blindage anti- propagation des ondes.

3. ACOUSTIC ANALYSIS ATTACK

Cette attaque sonore – découverte en 2002 par Serge Vaudenay – consiste à utiliser des techniques d'analyse d'émissions acoustiques pour déduire les données d'entrées dans un circuit. En effet, les sons sont produits par les opérations de chiffrement peuvent être captées par un attaquant grâce à un microphone ou analyseur de spectre sonore. Les variations acoustiques résultent en général du dispositif de chiffrement et donc dans le sens inverse peuvent aider à retrouver les messages originaux. De la même

façon que les attaques électromagnétiques, ce type d'attaque affecte les systèmes de chiffrement symétriques, asymétriques et hachages.

De plus, ce genre d'attaques a tellement évoluée qu'il est possible pour des hackers de reconstruire l'ordre dans lequel un utilisateur a tapé des touches de son clavier. Dès que l'utilisateur se connecte à une plateforme sécurisée, il révèle involontairement son mot de passe car les composants électroniques produisent des sons certes légèrement différents à l'oreille mais tout de même perceptibles avec des enregistrements audios.

C'est donc une attaque très utilisée pour attaquer des systèmes électroniques (claviers, souris etc.) sans aucun contact physique. Néanmoins, enregistrer et analyser des sons est assez complexe car cela demande d'être dans un environnement sans aucun bruit parasite pouvant empêcher la bonne interprétation de touches. Cette attaque dépend énormément des signaux sources et sont donc très sensibles aux variations de bruits et matériels utilisés. Une solution type serait que le système produise lui-même volontairement des bruits parasitant l'écoute de l'attaquant et faussant les résultats.

4. Thermal Analysis Attack

Cette attaque est très similaire à la Power Analysis car – en plus d'avoir été rapportée par les mêmes chercheurs en 1996 – elle peut retrouver des données sensibles en exploitant les déperditions de chaleur associées à la consommation d'Energie de la fonction de cryptographie. Pour se faire, l'attaque analyse la variation des chemins de courant provoqués par les électrons du circuit et en fonction des données de départ. Par conséquent, si mon opération comporte une porte OR, la consommation thermique ne sera jamais égale avec des données en entrée différentes car le courant parcourra des chemins distincts à chaque opération.

Ce type d'attaque est donc très efficace pour extraire des identifiants dans des scénarios particuliers et ne demande pas de matériel très couteux. Cependant, elle dépend du dispositif ciblé car les concepteurs de circuits cryptographiques ont développé des parades pour brouiller les déperditions de chaleur ce qui rend l'attaque difficile à exécuter. Enfin, l'attaque peut être très longue à effectuer car elle a besoin de nombreuses traces thermiques avant de pouvoir déterminer des mots de passe etc.

5. Timing Analysis Attack

Paul Kocher et son équipe ont prouvé en 1996 la possibilité de déduire des clés secrètes en mesurant les temps de fonctionnement d'une opération cryptographiques. Lorsqu'un utilisateur exécute son programme de chiffrement ou que l'attaquant exécute son algorithme d'attaque, on va mesurer le temps de réponse pris pour chaque caractère du mot de passe saisi et cela permet de définir précisément chaque caractère bien plus rapidement qu'avec une attaque brute force.

Cette méthode est donc utile lorsque les autres canaux d'attaque ne sont pas réalisables ou peu efficaces mais peut être affectée par la latence et le bruit natif du système. De même, il peut être complexe de réaliser l'attaque car il faut synchroniser toutes les mesures du temps.

De nombreuses variations de l'attaque ont été conçues à travers le temps ; les constructeurs ont donc inventé des solutions d'uniformisation de temps qui permettent de donner un temps de chiffrement égal quel que soit la valeur en entrée ce qui perturbe très grandement la capacité à analyser les mots de passe.

6. Logical Analysis Attack

Cette attaque a été découverte en 1997 par Paul Kocher et ses collègues et concerne également tous systèmes de chiffrement symétriques, asymétriques et fonctions de hachage. L'attaque exploite le fait que les circuits cryptographiques ont des comportements spécifiques dus à des erreurs dans leur conception électrique. En effet, des imperfections dans la fabrication du produit ou logiciel que l'on ne soupçonnera pas en temps normal peuvent être remarquées par un attaquant et l'aider à comprendre le processus de chiffrement et retrouver les bits d'entrée.

L'efficacité de l'attaque peut dépendre de si les informations logiques sont disponibles et exploitables. Cela s'explique du fait que l'attaque logique agit comme une attaque physique car elle doit accéder physiquement au circuit cryptographique pour collecter les données qu'elle souhaite exploiter. Cela signifie qu'elle ne peut être évitée avec des mesures logicielles. C'est pour cela que des chercheurs ont montré qu'avec une conception plus robuste, une meilleure fabrication des circuits et des contrôles plus sûr, on pouvait renforcer la protection contre ce type de SCA.

7. Fault induction attack

Découverte en 2006 par des chercheurs de Cambridge, cette attaque serait très efficace pour retrouver une clé non annoncée. En effet, l'attaque utilise une méthode de cryptanalyse puissante qui consiste à trafiquer un gadget afin de lui faire effectuer des opérations erronées. Ces dernières peuvent ainsi faire fuir des données sur les paramètres secrets.

On dit que cette catégorie d'attaque se décompose en 2 modèles d'attaque par faute, « permanente » et « transitoire ». Une erreur dite permanente va endommager le crypto système (physiquement par exemple en coupant des câbles, des jonctions…) de manière non réversible, il faut donc soit réparer le système soit le changer car il risque de reproduire les mêmes erreurs plus tard.

On parle aussi d'erreur transitoire où le module de chiffrement sera perturbé uniquement pendant le traitement en cours. Autrement dit, les perturbations (fréquence ou tension anormale, ondes radioactives…) influencent directement les opérations et causeront une erreur de calcul à un moment précis ce qui permettra de récupérer des informations.

En contrepartie, même si cette attaque peut affecter temporairement ou non un système, la difficulté est d'abord d'accéder au circuit physique pour pouvoir saboter quoi que ce soit. De plus, cela demande de bonnes connaissances en circuits électriques pour réaliser l'attaque et causer des erreurs (en affectant la mémoire par exemple).

8. AUTRES TYPES D'ATTAQUES MOINS POPULAIRES

Les attaques mentionnées au-dessus sont considérées comme les plus populaires et les dangereuses ; néanmoins, il existe d'autres attaques par canaux auxiliaires comme :

Optique : A l'aide d'indices visuels, on peut rassembler des informations sensibles. C'est en quelque sorte une catégorie qui concerne le shoulder surfing, espionnage, social engineering…

Mémoire Cache : En abusant de la mise en cache de la mémoire (processus permettant d'optimiser les performances), il est possible d'acquérir un accès et de voir des informations « invisibles » ou bloquées. De nombreuses vulnérabilités Spectre et Meltdown auraient utilisé ce canal pour contaminer les processeurs Intel.

Faiblesse du Hardware : En ayant une connaissance parfaite du modèle ciblé, ses caractéristiques, composants etc., un hacker peut provoquer des comportements ou erreurs à l'insu de l'utilisateur et du système. De cette manière, on peut forcer un changement dans la mémoire vive et la puce et récupérer les données que l'on veut. C'est le principe qu'a utilisé l'attaque par martelage de mémoire.

La plupart des attaques peu connues sont aujourd'hui irréalisables car des mesures ont été adoptés universellement dans les composants qui a corrigé ces failles.

On peut identifier des paramètres qui peuvent influer sur les résultats quelle que soit l'attaque choisie :

- Algorithmes de chiffrement attaqués (DES, AES, RSA) ;
- Variation de puissance ou temps d'exécution causé par les opérations du crypto système ;
- Corrélation entre les opérations et la clé : une corrélation forte implique que l'on peut supposer plus facilement les bits de la clé ;
- Modèles de puissance ou temps : caractéristique dans les traces de puissance ;
- Nombre de traces collectées : Plus l'on augmente ce nombre, plus l'on a de chances de restituer la clé secrète avec une marge d'erreur faible ;
- La présence de contremesures : par ex la randomisation, l'atténuation de signaux…

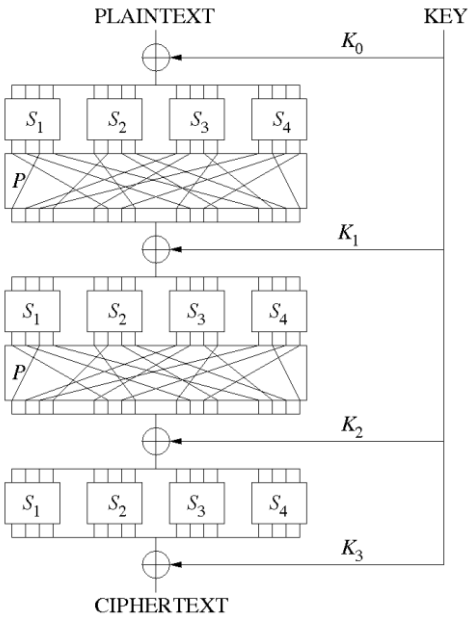
Il existe aussi des paramètres que l'on va d'amblée négliger comme l'environnement, le type et la vitesse du processeur, l'optimisation du compilateur (-O2 par ex).

Pour faire mon étude, j'ai choisi de développer sur les 2 catégories les plus populaires : la Timing et Power Analysis. Au vu de mes compétences et le temps que j'avais à disposition, j'ai préféré me concentrer sur les versions les plus basiques de ces types de SCA.

9. L'ALGORITHME DATA ENCRYPTION STANDARD

Pour me simplifier la compréhension, j'ai voulu essayer de briser l'algorithme DES car il s'agit d'un algorithme de chiffrement symétrique basique mais obsolète (créé en 1977) et remplacé par AES en 2001 puis RSA. En effet, les 64 bits accordés par DES étaient insuffisants face aux puissances de calcul des ordinateurs modernes d'où la venue de AES qui proposait des clés de 256 bits.

L'algorithme DES utilise des pseudo random permutations, ce qui veut dire qu'au lieu de transformer les valeurs de bit, on va les mélanger entre elles de manière aléatoire. Plus précisément, on exploite les « Substitutions Permutation Networks » qui divisent les 128 bits de la clé en Substitution boxes. Voici un exemple de SPN :



Ici, on altère le texte en clair pour le diviser en 4 S-boxes contenant chacun une partie de la clé. On applique ensuite une permutation pour tout mélanger et réordonner les bits de la clé, ce qui va nous donner des résultats uniques à chaque fois. On appelle rounds, le nombre de fois où l'on répète ces étapes.

D'autre part, on considère que le nombre de SPN correspond au nombre de substitutions qui ont eu lieu dans une altération. Dans la figure 1, on remarque l'opération SPN ne se déroule qu'une seule fois par round, c'est-à-dire que pour chaque bit de la clé K1,K2,Ki, on altère une seule fois avec la permutation toutes les S-boxes.

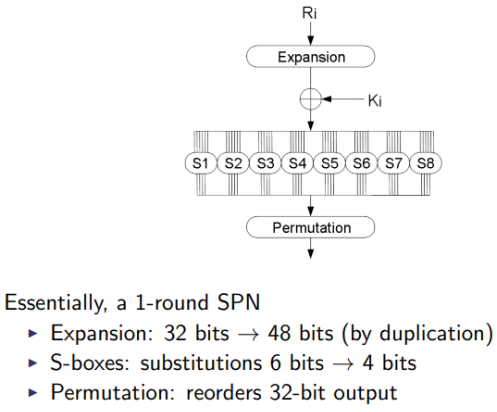


Figure 1 Un round de SPN dans un DES classique

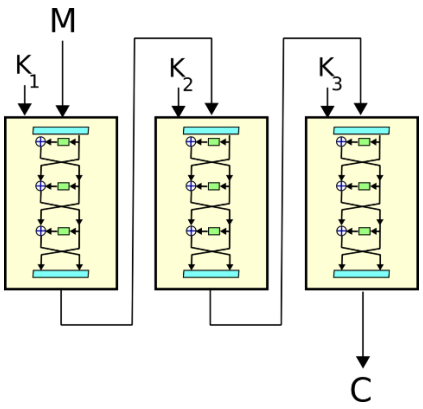


Figure 2 Des rounds dans un triple DES

L'algorithme DES utilisait seulement 16 rounds et seulement une opération de type SPN. En conséquence, avec l'ascension de la puissance de calculs des ordinateurs étant de plus en plus puissant, les hackers pouvaient briser cet algorithme. C'est pour cela que la version Triple DES (composée de 3 opérations de SPN par round) a été créé en 1998 mais n'a pas fait long feu face à l'algorithme AES considéré comme plus rapide.

Autrement dit, ce système est bijectif ; chaque antécédent en entrée a une unique image en sortie. Ensuite, il faut savoir que l'on utilise la méthode Cipher Block Chaining pour chiffrer et combiner les blocs dans des chaines. En outre, avant de chiffrer le bloc, on doit faire un XOR du bloc avec le bloc chiffré précédent. De cette manière, le résultat du chiffrement de ces blocs dépend par transitivité de tous les blocs précédents.

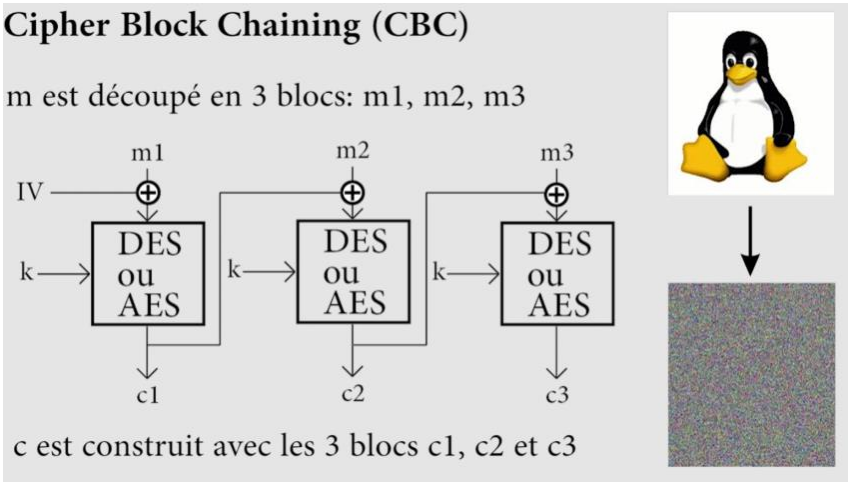


Figure 3 Schématisation du principe de CBC

Dans cette image, on remarque que le 1er bloc n'a pas de bloc précédent, c'est pour cela que l'on introduit un vecteur d'initialisation IV que l'on combine avec le bloc 1 de la clé pour former le bloc chiffré c1 et le processus se répète mais avec les nouveaux blocs et parties de la clé k.

Maintenant que l'on a compris plus ou moins le fonctionnement de DES, nous pouvons tenter de le programmer.

III. TIMING ATTACK

1. DESCRIPTIF

L'attaque la plus facile à réaliser est la Simple Time Analysis (STA). Comme je l'ai expliqué ce genre d'attaques affecte les smart card, les machines de votes électroniques etc. et exploite le temps de réponses du système de chiffrement ce qui peut dévoiler involontairement le secret comme le montre le schéma ci-dessous.

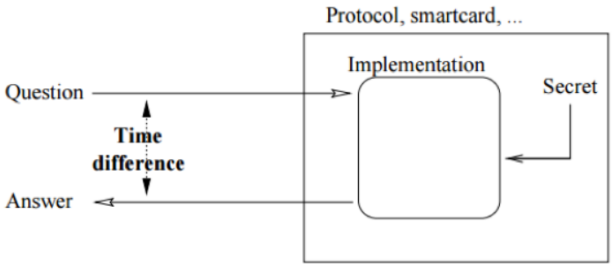


Figure 4 Schématisation de fuite de temps

Pour bien comprendre le but d'une STA et ce qui la différencie avec du brute force :

1er scenario : Si on considère qu'un mot de passe comporte 16 caractères et que des majuscules anglaises, cela donne 26 valeurs possibles pour chacun des 16 caractères du mot de passe. Autrement dit, l'attaque brute force demanderait 26^16 tentatives différentes ce qui serait très long à calculer. En général, pour un mot de passe de longueur n et une plage de caractères de taille k, cracker le mot de passe aura une complexité de O(kn) tentatives.

2ème scenario : En utilisant une Timing Attack, le hacker profitera du fait que lorsque le programme vérifie l'égalité des chaînes de caractères, la comparaison se termine dès qu'il trouve un caractère qui ne correspond pas.

Par exemple : un attaquant fait 3 tentatives et mesure le temps de réponse. L'attaquant essaie la chaine "AAAA" et mesure 0,2 ms, puis essaie "BAAA" et mesure 0,5 ms, enfin essaie "CAAA" et mesure à nouveau 0,2 ms. L'attaquant peut être certain que le premier caractère est "B" car l'algorithme a pris plus de temps, cela veut dire que B correspondait mais que le A suivant ne correspondait pas d'où la différence de temps.

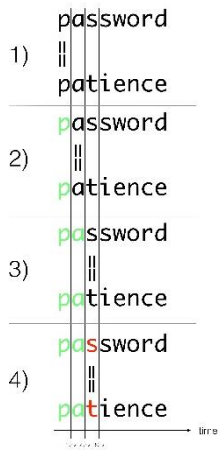


Figure 5 Démonstration de l'attaque STA

Selon des études, la vérification du mot de passe complet ne nécessite pas l'itération de toutes les combinaisons possibles de chaînes de taille n. Il suffit d'itérer sur toutes les valeurs possibles de chaque caractère (k), et de répéter cette opération n fois ce qui donnerait une complexité de linéaire k + k + k + … = n*k = O(n).

En conclusion, la complexité de temps diminue énormément car au lieu de comparer toute la chaine, on compare chaque caractère individuellement.

2. IMPLÉMENTATION

a. Idée de départ

Pour essayer de me faire apprendre ces attaques de manière intuitive, j'ai eu l'idée de créer une sorte de mini-jeu en python où un joueur proposerai un mot de passe (comprenant les lettres et chiffres seulement) qui serait stocké dans une variable. Puis, l'ordinateur essayerait de cracker mon mot de passe en utilisant l'attaque auxiliaire de temps et pourrait déduire de la fragilité de mon mot de passe selon le temps pris par la console.

Je trouvais l'idée intéressante car même si elle n'est pas réaliste et va forcément négliger la plupart des facteurs de risques que j'ai introduits plus haut, c'est une méthode que ludique et qui me montre les raisonnements basiques d'un hacker débutant. Pour m'aider, j'ai implémenté en premier lieu une version sans DES.

b. Sans chiffrement DES

```
def staAttack(mdp): #fonction de STA
    charset = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789'
    # on initialise des variables et listes vides qu'on remplira au fur et à mesure de l'attaque pour déduire des mesures de temps etc.
    mdp_length = len(mdp)
    mdpCracked = ''
    total_time_of_attack = 0
    charactersList = []
    timeList = []

    for i in range(mdp_length):
        start_time = time.time() #Pour chaque caractère, on va mesurer le temps pris pour exécuter l'attaque
        for char in charset:
            time.sleep(0.1) #Je fais exprès de freiner l'attaque de temps pour avoir des temps visibles
            if char == mdp[i]: #Si correspondance, on mesure
                mdpCracked += char
                end_time = time.time() #on mesure les temps pris avant de passer au prochain caractère
                time_taken_for_particular_character = end_time - start_time # On mesure la différence de temps et on l'ajoute dans la liste et temps total
                total_time_of_attack += time_taken_for_particular_character
                charactersList.append(char)
                timeList.append(f"{time_taken_for_particular_character:.6f}") #on ajoute aux listes le caractère en question et son temps associé
                print(mdpCracked)
                break

    attack_table = PrettyTable() #on créé un tableau qui résumera les temps pris STA pour trouver chaque caractère un par un
    attack_table.field_names = ["Caractères", "Temps associés"]
    for char, time_taken_for_particular_character in zip(charactersList, timeList):
        attack_table.add_row([char, time_taken_for_particular_character]) #on associe dans des lignes le caractère trouvé et son temps respectif de recherche
    print("\nTemps pris pour les caractères:")
    attack_table.align = "r"
    print(attack_table) #On affiche le tableau
```

Au lancement du programme, je peux saisir un mot de passe de taille lambda et sans symboles ou caractères spéciaux. On initialise des listes et variables vides qui vont stocker progressivement les temps pris, les caractères retenus par le programme… A partir d'un temps précis au lancement, la fonction parcourt chaque caractère du mot de passe cible. Elle applique la méthode expliquée plus haut qui au lieu de faire une brute force fait une comparaison de caractères selon le temps. Le programme teste séquentiellement chaque caractère du champ de caractères autorisés que j'ai défini préalablement.

Chaque supposition de caractère est suivi d'un délai simulé, réalisé à l'aide de la fonction *time.sleep (0.1)* – importé du module *time* – de freinage. Ce délai de 0.1 seconde permet de simuler le temps pris pour chaque essai, indépendamment du caractère testé. C'est aussi une manière de réguler la vitesse de calcul du programme car plus l'on réduit le délai, plus il exécutera vite les instructions du code. Comme je l'expliquais plus haut, si le premier caractère testé est correct, l'algorithme va prendre plus de temps car il passe au suivant et il va aussi enregistrer la valeur devinée pour restituer progressivement le mot de passe. Comme prévu, si le caractère testé est incorrect, le programme réitère la boucle jusqu'à ce qu'une correspondance soit trouvée.

J'ai voulu afficher en temps réel le mot de passe deviné caractère par caractère ce qui me permet de suivre le processus de recherche de correspondance. Le programme va donc pour chaque caractère identifié enregistrer le temps pris dans un tableau de variables *« attack_table »*. Ce tableau (créé avec la bibliothèque *prettytable* et les exemples sur internet) sera affiché en fin de session pour montrer le temps parcouru pour chaque itération correcte de la boucle et résumer les temps pris par l'algorithme.

Enfin, je voulais aussi déterminer le caractère qui a pris le plus de temps à trouver avec la fonction max qui retrouve le temps le plus élevé dans le tableau.

```
Entrez un mot de passe à attaquer: f9X
Lançement de l'attaque STA
f
f9
f9X

Temps pris pour les caractères:
+------------+----------------+
| Caractères | Temps associés |
+------------+----------------+
|          f |       0.602746 |
|          9 |       6.221169 |
|          X |       5.022543 |
+------------+----------------+
```

```
Le mot de passe a été trouvé:  f9X
Temps pris pour le déterminer:  11.846458 seconds
Le caractère '9' a été le plus long à trouver: 6.221169 seconds
```

c. Avec chiffrement DES

Après avoir établi une version basique sans procédé de chiffrement, j'ai tenté d'implémenter des fonctions de chiffrement DES et déchiffrement à l'aide de modules dédiés notamment « *py.cryptodome* ». En premier lieu, j'ai dû faire une fonction de remplissage (*padding*) pour faire en sorte que la taille du mot de passe saisit soit toujours adaptée en des blocs de 8 octets – avant d'entrer dans la fonction de chiffrement – comme on énonçait le principe des SPN et S-boxes. Grâce à la documentation en ligne de cette librairie, j'ai pu créer des fonctions de padding, de chiffrement et déchiffrement :

```python
def pad(data):  #fonction qui remplit automatiquement le mdp pour ensuite appliquer le chiffrement DES
    block_size = DES.block_size
    padding_size = block_size - len(data) % block_size #ca calcule le remplissage nécessaire en obtenant le reste de la divsion euclidienne
    padding = bytes([padding_size]) * padding_size #on applique le remplissage avec la taille nécessaire
    return data + padding

def encryptDES(data, key): #fonction de chiffrement DES, elle applique la méthode classique des 16 rounds
    data = pad(data)
    iv = get_random_bytes(DES.block_size)  #initialization vector : il est totalement arbitraire et se multiple avec un XOR aux parties du mot en clair
    cipher = DES.new(key, DES.MODE_CBC, iv) #On génère une encryption avec le mode CBC et en prenant les IV
    encrypted = iv + cipher.encrypt(data)
    return encrypted

def decryptDES(encrypted_data, key):  #fonction inverse au chiffrement DES, elle utilise la même clé d'où le fait que le déchiffrement est instantané
    iv = encrypted_data[:DES.block_size]
    cipher = DES.new(key, DES.MODE_CBC, iv) #on amorce le déchiffrement
    decrypted = cipher.decrypt(encrypted_data[DES.block_size:]) #le résultat redonne le mot de passe en clair
    return decrypted
```

L'encryption se fait avec la méthode CBC que j'ai abordé plus haut. On combine avec un vecteur d'initialisation aléatoire le bloc 1 de la clé. Le déchiffrement procède de la même manière mais dans l'ordre inverse ; c'est-à-dire que l'on utilise le vecteur d'initialisation qui est inclus dans les données chiffées pour retrouver les S-boxes et ainsi obtenir le mot de passe de départ.

Puis j'ai repris la fonction de « *staAttack* » précédemment faite pour mener une attaque mais avec cette fois-ci des versions cryptées et décryptées. Le programme va donc mesurer une nouvelle fois le temps pris pour chiffrer tous les caractères du mot de passe que j'ai saisis au lancement. Ensuite, pour donner une représentation visuelle, j'ai créé un tableau associant pour chaque caractère son temps de chiffrement respectif et sa transition en bloc chiffré avant de lancer l'attaque.

```
Entrer le mot de passe à attaquer : f9X
Clé de chiffrement: b'<\xbf\xf9\x80\x11x\xfe\xbd'
Chiffrement DES en cours...

Table de temps et caractères lors du chiffrement :
+-----------+----------------------+----------------------------------+
| Caractère | Temps de chiffrement |        Caractère chiffré en bloc |
+-----------+----------------------+----------------------------------+
|         f |             0.002427 | 8b5a12eb6356bd99b57365836cfdbea5 |
|         9 |             0.000062 | 6098014886df86bce8fabbc912d367bf |
|         X |             0.000052 | 5c7ba3729ef7c2bfcc200155e378080a |
+-----------+----------------------+----------------------------------+
```

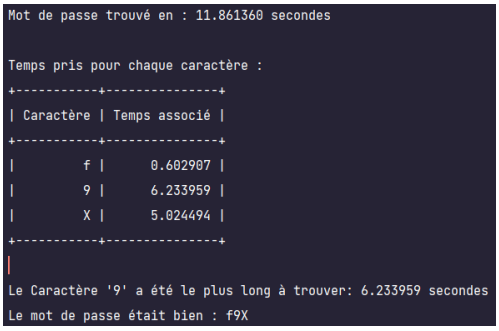
Enfin, la fonction main appelle les fonctions que j'ai créé à tour de rôle (*encryptDES, decryptDES, et staAttack*), les convertit au format adéquat et arrange sous forme de tableau pour avoir des indices visuels.

```python
if __name__ == "__main__":
    mdp_to_attack = input("Entrer le mot de passe à attaquer : ") #On récupère la saisie utilisateur
    encryption_key_generation = get_random_bytes(DES.block_size) #on génère la clé
    encryption_time_list = []  # On crée une liste qui regroupe les temps de chiffrement avec DES

    print(f"Clé de chiffrement: {encryption_key_generation}")
    print("Chiffrement DES en cours...")
    encrypted_chars_list = []  #liste qui rassemble tous les caractères lorsqu'ils sont chiffrés avec la clé
    for char in mdp_to_attack:
        encrypted_time = timeit.timeit(lambda: encryptDES(char.encode(), encryption_key_generation), number=1) #on utilise timeit car la librairie basique n'est pas précise
        encryption_time_list.append(f"{encrypted_time:.6f}") #on ajoute à la liste le temps de chaque caractère
        encrypted_data = encryptDES(char.encode(), encryption_key_generation) #on applique la fonction de chiffrement à la clé + caractère en question
        encrypted_chars_list.append(encrypted_data.hex())  # On convertit en hexadécimal les valeurs de chiffrement

    print("\nTable de temps et caractères lors du chiffrement :")
    encryption_table = PrettyTable() #on créé un tableau visuel qui rassemblera les listes liées au chiffrement avant l'attaque
    encryption_table.field_names = ["Caractère", "Temps de chiffrement", "Caractère chiffré en bloc"]
    for char, enc_time, enc_char in zip(mdp_to_attack, encryption_time_list, encrypted_chars_list):
        encryption_table.add_row([char, enc_time, enc_char])  #on ajoute au tableau des lignes avec les paramètres (quel caractère, temps, versions chiffrée)

    encryption_table.align = "r"   #on aligne à la droite les résultats du tableau
    print(encryption_table) #On affiche le 1er tableau

    print("\nLançement de l'attaque STA...")
    startAttack = staAttack(mdp_to_attack, encryption_key_generation, encryption_time_list)  #On appelle la fonction d'attaque STA sur la variable de mdp

    print(f"Le mot de passe était bien : {startAttack}")
```

Dans la théorie, c'est comme ça que je voulais reproduire le processus ; mais en pratique, je n'ai pas vraiment réussi à faire deviner le mot de passe chiffré ni la clé avec une attaque STA. En effet, j'ai beau avoir compris le principe général de la méthode d'attaque, je ne comprenais pas comment parvenir à faire de comparaisons sur la clé ou le mot de passe chiffré alors avec mes moyens.

Mon code n'est pas réaliste car il ne parvient pas à effectuer d'attaque sur la clé de chiffrement. Lors de mes tentatives, le code tournait en boucle et ne parvenait pas à trouver de valeurs correspondantes ou finissait par s'arrêter sans avoir déterminé un paramètre. J'ai aussi eu droit à de nombreuses erreurs que je n'ai pas réussi à résoudre n'ayant pas les compétences et connaissances nécessaires.

Dans d'autres essais, lorsque je lançais l'algorithme de STA, il trouvait des caractères sans lien avec mon mot de passe saisi. En outre, le programme n'est pas opérationnel avec du chiffrement car il ne cherche pas à retrouver ou déchiffrer la véritable clé mais directement le mot de passe comme si le chiffrement n'avait pas eu lieu. C'est pour cette raison que les résultats ci-dessous sont à priori quasi-similaires au code d'avant.



Au final même si mon attaque avec chiffrement n'est en aucun cas exhaustive ni représentative de la réalité, j'ai tout de même pu m'essayer à manipuler des clés et avoir un aperçu de à quoi ressemblerait une véritable attaque. Je suis conscient que mon attaque n'exploite pas la clé comme demanderait une vraie STA et que le code « triche » en faisant un déchiffrement instantané. Cependant, ces problèmes et échecs m'ont tout de même fait apprendre le principe de Timing Analysis. Je ne comprenais pas comment corriger le code et les erreurs qui allaient suivre. Par manque de temps, j'ai préféré passer à la 2ème catégorie d'attaque.

3. CONTREMESURES CONTRE LES TIMING ATTACK

Durant mes recherches, j'ai relevé dans des articles scientifiques des anti-mesures pertinentes notamment utiliser des algorithmes invulnérables aux analyses de temps comme (SHA – 3, ChaCha20, Salsa20 contrairement à DES, AES et MD5) ou encore utiliser des fonctions de Hash.

Selon des experts, les circuits qui cherchent à optimiser le temps de réponse et la productivité par l'utilisation de tables de consultation précalculées… risquent de faire fuiter eux-mêmes des données temporelles. Diversifier le nombre de plateformes supportées par son système n'est pas non plus recommandé car des fuites concernent l'aspect physique du système ; les plateformes ne peuvent pas empêcher des problèmes spécifiques au crypto système.

D'autres chercheurs ont aussi pensé à des idées de « masquage », où l'on modifie les valeurs intermédiaires lors du chiffrement de sorte que même si ces dernières sont identifiées, elles ne permettront pas de pouvoir déduire sur les données principales. Mais leur coût est très élevé ce qui rend cette idée peu pratique pour certains systèmes.

Mettre des mécanismes de comparaison de strings à temps constants. C'est un concept simple que Paul Kocher a noté mais qui n'est pas facile à réaliser à cause des optimisations de la mémoire vive, cache… qui ne sont pas vraiment réglables chez les constructeurs. Paul Kocher a donc pensé à intégrer dans les circuits cryptographiques des modules de retard qui peuvent créer du délai volontairement entre chaque entrée de l'algorithme. Cependant, l'idée n'est pas non plus idéale car l'ajout de bruit artificiel ne peut contrer des résultats d'analyse venants de multiples traces.

IV. POWER ANALYSIS

La deuxième catégorie de SCA que je souhaitais découvrir était celle liée aux analyses de puissance. Comme je l'expliquais plus haut, l'attaquant va chercher à analyser les variations de consommation de puissance provenant du système de cryptographie.

1. DESCRIPTIF

En faisant une corrélation entre ces variations de puissance et les informations sensibles notamment la clé, un attaquant peut être capable de récupérer la clé secrète. Par exemple, si le circuit cryptographique effectue une opération XOR (Ou exclusif, utilisé en opération binaire) sur 2 nombres, la consommation énergétique utilisée pour exécuter l'opération sera différente et laissera une « trace de consommation » respective pour chaque nombre. Autrement dit, chaque opération ne demandera pas exactement la même consommation électrique. Paul Kocher a donc démontré que l'on pouvait déduire les données en entrée et la clé secrète utilisée grâce à ces empreintes. On distingue 2 d'attaques, les Simple Power Analysis et les Differential Power Analysis.

a. Simple Power Analysis

Cette version de l'attaque est la plus basique car elle ne fait qu'observer des traces de puissance et identifie visuellement les variations associées aux opérations du crypto système. La figure ci-dessous tirée d'un article scientifique montre la trace de consommation d'énergie d'un appareil qui utilise le chiffrement AES. On peut voir un schéma qui se répète 10 fois ce qui correspond littéralement aux 10 rounds de l'algorithme AES (soit 6 rounds de moins que DES).

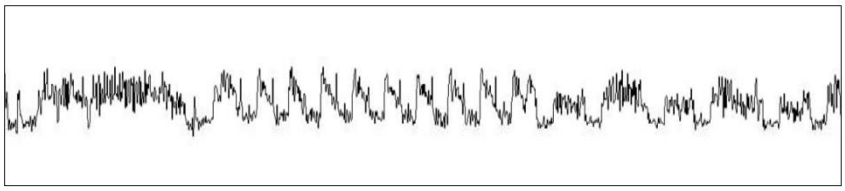


Figure 6 Attaque SPA sur un chiffrement AES sur une smart card

À partir de ce graphique, un attaquant est censé pouvoir faire son analyse statistique ; c'est cette étape que je vais tenter de reproduire dans l'implémentation. D'autre part, il faut savoir que le flux d'instructions dépend des données.

Par Exemple : l'opération d'exponentiation modulaire peut être faite avec un algorithme à bases de puissances au carré ou de multiplication. Si l'opération de carré est implémentée différemment de l'opération de multiplication, l'algorithme de chiffrement sera optimisé pour ne faire que cette opération. En conséquence, cela va drastiquement accélérer le code et l'on pourra notifier la différence avec des modèles de consommations différentes. Alors la trace de puissance d'une exponentiation donne directement la valeur (secrète) de l'exposant.

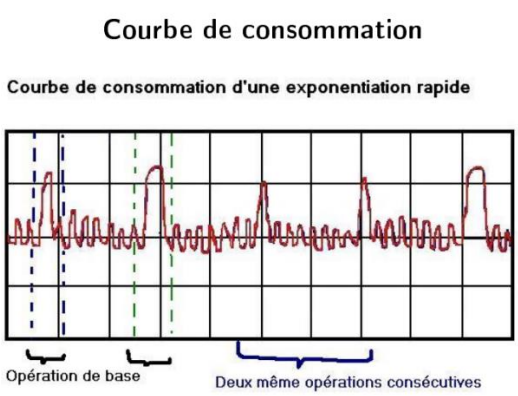


Figure 7 Courbe 2 de SPA

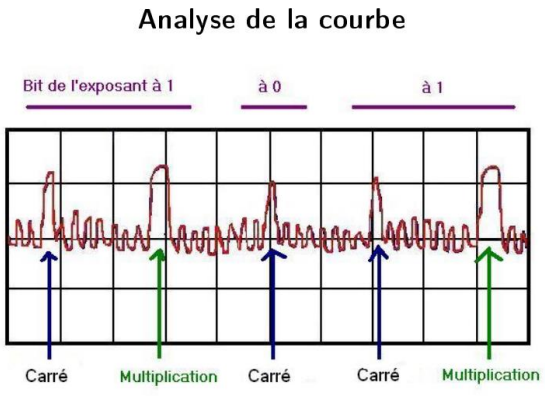


Figure 8 Courbe 1 de SPA

Les scientifiques en ont déduit que la plupart des opérations conditionnelles dépendant des paramètres secrets sont à risque. Cependant, ce modèle d'attaque n'est plus utilisé aujourd'hui car les systèmes modernes ont des mesures classiques « anti-fuite de consommation ». De même, j'avais parlé d'uniformisation de puissance comme contremesure ce qui a causé l'obsolescence des SPA car elles ne sont plus capables de restituer des secrets alors que les variations de puissance sont peu visibles.

b. Differential Power Analysis

Les DPA sont des modèles bien plus répandus et dangereux que les SPA. Contextuellement, lorsque les constructeurs ont établi des mesures anti-SPA, les attaques SPA – étant bien plus efficaces et craintes par les systèmes de cryptographie contemporains notamment RSA – faisaient déjà leur apparition et menaçaient la sécurité des identifiants.

Le principe de cette attaque est de faire une conjecture sur la dépendance des données avec les schémas de consommation d'énergie. On parle donc de dépendance par rapport au secret utilisé et on cherche à vérifier la légitimité de l'hypothèse. D'après des études, la conjecture concerne la valeur d'un ou plusieurs bits du secret.

En outre, on exploite ces dépendances de données avec des statistiques puissantes ; cela a conduit à une classe d'attaques appelées « différentielles ». Dans ce genre d'attaques, l'attaquant veut simuler plusieurs fois l'exécution d'un algorithme et constater les changements d'état des portes logiques OR, AND, XOR… pour déduire les variations.

Certains documents que j'ai consultés parlaient de point critique, c'est-à-dire « l'instant où l'opération cible entre la clé secrète ou un morceau de la clé secrète est utilisé en interaction avec le message connu » ; ce point critique dépendrait donc d'une petite partie de la clé secrète ce qui permettrait à des hackers de restituer la clé.

D'autres articles parlaient de poids de Hamming (le nombre de bits différents de 0) et mesures particulières mais en dépit de mes connaissances très faibles en électronique et en traitement de signal, je n'ai pas vraiment compris tous ces concepts.

De manière très simplifiée et selon d'au cours officiels (pour la compréhension du lecteur et la mienne), le principe du DPA est de rassembler des traces de consommation en visant un bit de la clé (on choisit donc entre les états de transition 0 ou 1). On va donc faire une hypothèse sur la clé et séparer en 2 groupes les résultats de consommation selon le bit choisi.

Selon les messages et le chiffrement de l'algorithme, les 2 groupes pourront être distincts selon la présence ou non des bits. Plus une clé contient de bits 0, plus ce sera visible dans des traces de consommation. Si un attaquant arrive à deviner le bon bit, on pourra voir une certaine égalité des moyennes en certains points. En outre, peut confirmer la conjecture émise sur le bit si les résultats des moyennes de trace sont distinguables et font une moyenne égale. Voici un exemple théorique provenant de l'étude de Paul Kocher de ce à quoi on pourrait ressembler les résultats des hypothèses avec l'algorithme DES.

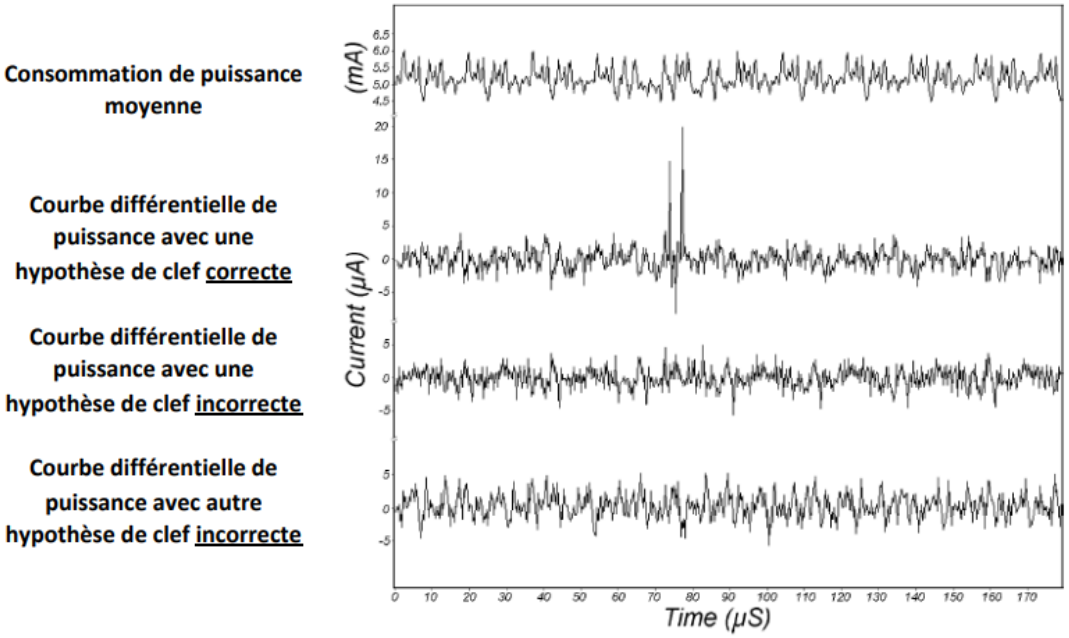


Figure 9 Exemple de DPA sur l'algorithme DES

Dans la Figure, on peut voir des traces créées à partir de textes en clair connus entrant une fonction de chiffrement DES le tout sur une carte à puce. Les 3 dernières traces montrent la vérification de l'hypothèse émise au départ. La 2e courbe montre un pic particulier qui prouve que la supposition de clé était correcte tandis que les 2 dernières ont des suppositions différentes et incorrectes. Cela prouve que le DPA attack est capable de déterminer la trace où l'on peut voir le signal lié à la clé même avec un peu de bruit à condition que l'on ait suffisamment d'échantillons et traces (ici, les graphiques ont été fait avec 1000 échantillons).

D'après les recherches, il est possible d'extraire par des méthodes statistiques la consommation liée à ce bit de la consommation globale et de tirer des informations sur la clé. D'un autre côté, il existe une technique plus avancée pouvant mesurer la corrélation (CPA) mais je n'ai pas réussi à cerner toutes les métriques ni comment les

avoir. Constatant le temps passé à essayer de comprendre ces articles, j'ai décidé de me contenter de cette compréhension simpliste du DPA et se limiter à l'essentiel.

2. TENTATIVE D'IMPLÉMENTATION

Pour la partie implémentation, au vu de mes difficultés de compréhension des DPA, j'ai préféré étudier uniquement les SPA pour m'aider à comprendre le fonctionnement primaire de cette catégorie d'attaque. J'ai donc mené des recherches dans des articles scientifiques mais aussi des forums pour m'aider à partir sur de bonnes bases mais je n'ai rien trouvé de vraiment exploitable ni compréhensible pour un débutant. Mon objectif était de faire ma propre interprétation des SPA et de chercher à avoir des graphiques similaires à ceux que nous avons vu dans la partie précédente sans aborder la partie liée aux analyses statistiques. L'objectif était donc de reproduire une trace de consommation lorsqu'un message passe par un appareil de cryptographie (avec soit le courant soit le voltage) et de l'afficher selon le temps comme avec les exemples au-dessus.

Pour ce programme, j'ai repris à nouveau le module « *py.cryptodome* » (ainsi que des librairies de temps et d'aléatoire et la fameuse librairie de graphiques « *matplotlib.pyplot* ». Le code commence donc par la même fonction de chiffrement DES, pad et de génération de clés DES en 8 bits. Puis repris une fonction sur internet permettant de pour récupérer les expressions régulières des utilisateurs de sorte à autoriser des messages types sans caractères spéciaux. En effet, puisque je parlais de chiffrement de message cette fois, je me disais que cela pouvait être une bonne idée de cette fois prendre une librairie dédiée à des messages types. En utilisant la libraire *« regular expression »* et la formulation adéquate, je pouvais vérifier via des conditions si le message écrit était cohérent, auquel cas on pouvait passer à l'étape suivante.

```
def getUserInput(): #fonction permettant juste de vérifier qu'un msg a bien été saisi ou respecte les le charset définit
    msgListe = [] #Liste qui peut garder plusieurs msgliste différents écrits à la suite
    while True:
        msg = input("Entrez le msg ou appuyez sur Entrée pour terminer : ") #on attend la saisie utilisateur
        if not msg:
            break
        if re.match("^[A-Za-z0-9]+$", msg): #si le mdp respecte les conditions,
            msgListe.append(msg.encode())          #on peut alors convertir en bytes le msg et passer à la suite
        else:
            print("caractères autorisés : majuscules, minuscules ou chiffres")
    return msgListe
```

Puis j'ai fait une fonction SPA à partir des études et docs en ligne.

```
def spaAttack(data): #fonction de l'attaque Simple power Analysis
    # Initialisation de listes vides de traces et mesures de courant qui seront remplies au fur et à mesure
    traces = []
    mesures_courant = []

    start_time = time.time()

    for round_num in range(16): #Il aurait fallu que la boucle mesure les 16 rounds en théorie lors du chiffrement DES pour mesurer les temps etc.
        current_time = time.time()
        time_diff = (current_time - start_time) * 1e-3

        for block_num in range(8): #de même, cette boucle aurait du observer les 8 blocs de 64 bits chiffrés dans chaque round pendant le chiffrement
            power_consumption = random.uniform(0, 5)
            traces.append(power_consumption)
            #En dépit de valeurs exhaustives, les valeurs de puisssances et consommations sont ici totalement aléatoires
            current_measurement = random.uniform(-0.5, 2.5)
            mesures_courant.append(current_measurement)

    end_time = time.time()
    time_associated_power = [(t - start_time) * 1e-3 for t in range(len(traces))]  #on génère le nombre traces nécessaires pour associer les temps
    min_time = min(time_associated_power) #on cherche le temps exact où la 1ère mesure a été enregistrée
    time_associated_power = [t - min_time for t in time_associated_power]  #cette différence sert uniquement à afficher des temps "cohérents" après 0

    # l'attaque est censée effectuer ces observations pendant le chiffrement, ici nous manquons de réalisme
    # le but était de m'aider à visualiser l'attaque même si ce n'est en aucun cas réaliste

    return time_associated_power, traces, mesures_courant
```

Ici, on commence par générer des tableaux vides de traces, mesures de courant et on mesure le temps où la collecte commence. Ces tableaux seront remplis au fur et à mesure que l'on fait chaque étape du chiffrement DES et ce pour chaque byte pendant les 16 rounds DES. Ça va donc mesurer le temps relatif au début du lancement pour avoir des mesures de temps précises des opérations, ce qui donne des séries de temps pour chaque calcul de l'algorithme. Après cela, on génère donc une clé aléatoire et récupérons la saisie de l'utilisateur. Vous pouvez remarquer que pour rendre « réaliste » l'analyse, j'ai fait exprès de simuler aléatoirement la puissance de consommation de chaque byte et de la comprendre entre -0,5 et 2,5 µAmpères.

```
key = get_random_bytes(DES.block_size)  #on appelle la fonction de la librairie pycryptodome
msgListe = getUserInput()  #on appelle la fonction de vérification de msgListe bien saisis
keyHexa = key.hex()
#print("Clé DES générée aléatoirement:", key)
print("Clé DES générée aléatoirement:", keyHexa)
time.sleep(1)

if not msgListe:
    print("Pas de message entré")
else:
    timeList = []  #On va créér des listes de traces, temps et mesures de courant qui seront utilisées dans un graphique de puissance
    tracesList = []
    mesures_de_courantList = []

    for msg in msgListe:
        encrypted_msg = encryptDES(msg, key)  #on chiffre avec la fonction dédiée
        time_associated_power, traces, mesures_courant = spaAttack(encrypted_msg)  #on appelle la fonction SPA pour avoir les valeurs retournés
        timeList.append(time_associated_power)
        tracesList.append(traces) #Pas utilisée
        mesures_de_courantList.append(mesures_courant) #on ajoute des valeurs déduites de l'attaques (même si les valeurs restent aléatoires)
```

On appelle la fonction de la libraire « pycryptodome » pour générer aléatoirement une clé DES qui sera convertie en hexa pour la voir concrètement. Puis, j'ai créé des tableaux rassemblant tous les temps et traces relatives au message chiffré qui pourraient servir à l'affichage des données dans le graphique.

```
for i, (time_associated_power, traces, mesures_courant) in enumerate(
        zip(timeList, tracesList, mesures_de_courantList)):  #on reprend les données des listes pour générer un graphique de consommation
    plt.figure(figsize=(12, 6))
    # notez que les traces ne sont pas exploités dans le graphique, il aurait été utile d'en avoir plusieurs différentes pour avoir plus de précision

    plt.plot(time_associated_power, mesures_courant, label=f'msg {i + 1}', color='blue')
    plt.xlabel("Temps (µs)")
    plt.ylabel("Courant (µA)")
    plt.legend()
```

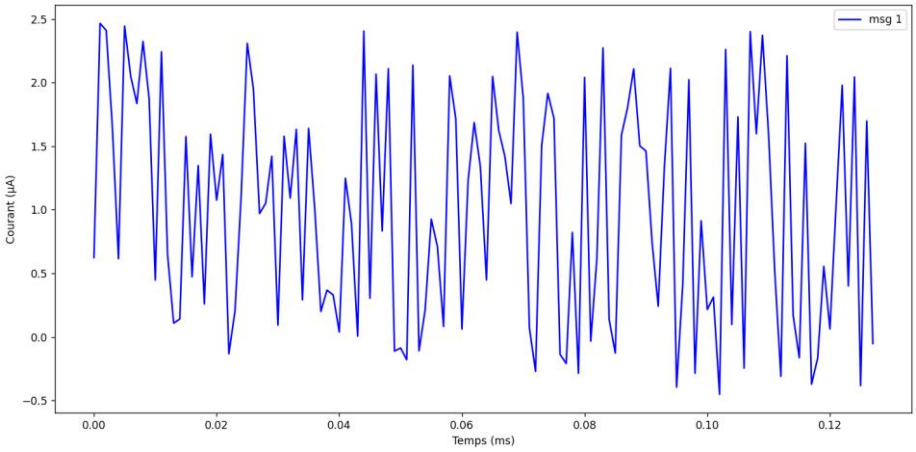
J'ai eu 2 idées de graphique, montrer la consommation de puissance selon des échantillons représentant une mesure de consommation de puissance à un moment donné pendant le chiffrement DES. Sur le papier, je pensais que c'était une bonne idée mais vu que je n'avais pas de modèles précis, j'ai fini par l'abandonner et le supprimer, c'est pour cette raison que l'on peut voir que des *« tracesList »* qui ne sont au final pas utilisées dans les graphiques…

Le 2ème graphique reprend justement les graphiques des articles scientifiques ; il affiche le courant électrique en µA ou mV en fonction du temps en (µ ou mili) secondes. J'ai donc cherché à reproduire les graphiques obtenus par les scientifiques qui fournissent des informations sur les opérations consommatrices de puissance, ce qui peut aider à identifier la clé secrète utilisée pour le chiffrement (dans un cas réel où à l'inverse de mon programme, les mesures ne sont pas faites aléatoirement).

Voilà un exemple de comment je voulais concevoir mon code :

```
Entrez le msg ou appuyez sur Entrée pour terminer : $Test
caractères autorisés : majuscules, minuscules ou chiffres
Entrez le msg ou appuyez sur Entrée pour terminer : Alice va chez Bob dans 5 min
Entrez le msg ou appuyez sur Entrée pour terminer :
Clé DES générée aléatoirement: 8e9ddf7e1fd672a8
```

Quand on écrit et envoie le message, le programme SPA se lance et tente de mesurer la consommation prise par le chiffrement de tout le message. Voici donc à quoi ressemblerait le graphique généré via le chiffrement de ce message.



Bien entendu, étant donné que mes mesures de courant sont générées aléatoirement, on n'a pas vraiment de lien ou de pic caractéristique d'une opération de chiffrement. Dans les cas réels, les chercheurs et hackers pourraient déterminer des changements d'état interne (passage de bit 0 à 1), des opérations répétées, des perturbations pendant le chiffrement, des déphasages etc.

Au moment où j'ai décidé de passer à ce code ci, j'étais conscient que je n'aurais pas assez de compréhension ou connaissances dans ce sujet là pour faire une implémentation parfaite et faire d'analyse statistique. Par conséquent, à l'instar du code de l'attaque Timing Analysis, le code que j'ai construit n'est ni réaliste ni véritablement fonctionnel car il est simulé aléatoirement. Néanmoins, j'ai pu entrevoir brièvement les raisonnements et les questions à se poser dans l'implémentation de ce genre d'attaques.

3. CONTREMESURES

Dans les articles scientifiques et leçons que j'ai consultés, j'ai relevé quelques contremesures pouvant freiner ce type de SCA. L'objectif principal de ces contremesures est de faire disparaitre une quelconque relation entre les consommations d'énergie et les informations secrètes. On note que ce n'est pas non plus nécessaire de rendre toutes les données traitées par le circuit de chiffrement indépendantes des analyses de puissance. En effet, certaines valeurs ne permettront pas de pouvoir vérifier ses suppositions d'état ; c'est le cas des entrées ou sorties des S-boxes dans l'algorithme AES qui sont inexploitables par un attaquant.

Dans le cas des SPA, une première méthode serait de protéger les valeurs directement liées à la clé qui affectent l'exécution du programme et la puissance. Par exemple, en empêchant les branchements conditionnels [1], on serait en mesure de réduire les chances de déduire des données secrètes.

Une autre approche concernant le DPA serait de protéger le dispositif de chiffrement via des signaux parasites ; ces bruits pourraient dissimuler ainsi les signaux primaires. Cependant, ce n'est pas une méthode très utile car le DPA exploite des analyses statistiques avancées pour extraire des données des traces. On peut alors distinguer 2 catégories de mesures, celles qui cherchent à masquer les données (masking) et celles qui veulent les cacher dans d'autres segments (hiding). Tandis que le masking est une méthode permettant de limiter au mieux les fuites et leurs impacts, le hiding est une technique de stéganographie qui permet de cacher dans d'autres pièces ou composants (exemple : images, sons) les données. Cela dit, ces catégories restent pour autant complémentaires et peuvent dépendre du matériel, logiciel ou méthodologie utilisée.

Comme 3ème approche consisterait à concevoir des appareils de cryptographie à partir d'hypothèses réalistes sur le matériel sous-jacent. En effet, les procédures non linéaires de mises à jour des clés peuvent être employées pour empêcher une éventuelle corrélation entre les transactions et les traces de consommation. Paul Kocher pensait donc que le hachage d'une clé de 160 bits avec SHA devrait nuire aux informations que l'attaquant a rassemblé. De même, dans le cas des systèmes à clé publiques, une utilisation très poussée des processus d'exponentiation modulaire pourrait palier une accumulation de données avec les opérations effectuées. Des compteurs d'utilisation

[1] Branchements conditionnels : « structures de contrôle » ou plus simplement l'exécution d'une boucle qui auraient pour but de rompre une séquence d'instructions dans un programme. D'après les articles, ces branchements seraient dépendant de données secrètes dans l'implémentation du chiffrement.

des clés pourraient potentiellement empêcher la collecte de plusieurs échantillons de traces etc.

Enfin, une dernière idée serait de concevoir un crypto système tolérant face aux fuites de données. C'est-à-dire que le constructeur doit définir les taux de fuite et les fonctions pouvant être révélées sans nuire à la sécurité du système. Cette approche est assez intrigante car les fonctions de fuite peuvent être analysées et exploitées à l'instar d'un oracle [2] et fournir des informations sur les données utilisées, la limite de données leakable acceptable.

Notez qu'il existe d'autres moyens de sécuriser des algorithmes soi-même contre les SCA proposées dans des articles très spécifiques et assez complexes (qui en un mot rendent soit aléatoire des paramètres de l'algorithme soit invisibles les données publiques) pour que je puisse en parler. J'ai donc retenu les mesures généralisées.

V. CONCLUSION

1. RÉCAPITULATIF DES ATTAQUES

Nous avons vu de manière globale les attaques par canaux auxiliaires. Ce genre d'attaques exploitent les failles d'un algorithme de chiffrement comme DES, AES, RSA… pour pouvoir récupérer des informations secrètes involontairement leakés par le système. Nous avons vu plusieurs catégories comme les attaques sonores, électromagnétiques, thermiques… Les catégories que nous avons étudiées en particulier sont les attaques temporelles et d'analyses de consommation. Ces attaques sont les plus populaires et les plus étudiées par les scientifiques ce qui m'a incité à les étudier. Tandis que l'attaque de temps exploite les temps de réponse dans les processus des algorithmes, l'attaque de puissance mesure la consommation électrique utilisée par le crypto système pour effectuer ses opérations de chiffrement.

2. DIFFICULTÉS RENCONTRÉES

J'ai eu de nombreuses difficultés lors de mes recherches et la composition du code et rapport. En effet, j'avais énormément de peine à comprendre les articles scientifiques que j'ai consultés, à concevoir un code cohérent car je ne savais pas vraiment ce que je pouvais essayer de faire compte tenu du temps qu'il me restait et mes connaissances limitées dans le domaine. De même, je cherchais continuellement à avoir une vision simplifiée pour m'aider à comprendre globalement les caractéristiques des attaques sans rentrer dans tous les détails car j'avais conscience que je risquais de m'y perdre.

[2] Oracle : Algorithme de chiffrement automatique et à la demande. C'est une boite noire qui utilise exactement le même procédé de chiffrement que le crypto-système sans que l'attaquant le sache. A l'aide du message calculé en sortie de l'oracle, l'attaquant peut tenter de deviner le résultat du véritable algorithme sans avoir la moindre information sur l'algorithme.

C'est pour cette raison que pour mes codes, j'ai voulu faire quelque chose de certes peu concret ou réaliste mais qui reste dans mes moyens.

Dans le cas des STA, il aurait fallu que j'arrive à faire en sorte que ce soit le chiffrement DES qui subisse l'attaque de temps. Je ne comprenais pas comment faire cette opération et comment améliorer le code.

Du côté des SPA, les résultats que j'ai obtenus n'ont pas vraiment de lien réaliste entre les mesures et le temps pris. Il aurait fallu que je puisse mesurer les consommations directement dans la fonction de chiffrement. Cela m'aurait contraint à la faire moi-même.

En outre, pour améliorer mes programmes et les rendre réaliste, il aurait fallu que je le relie correctement les opérations de chiffrement sans mettre d'aléatoire forcé, que j'ai plus de compétences en codes pour résoudre les multiples erreurs que j'ai eu et que je n'ai pas pu résoudre et aussi plus d'expériences et connaissances dans ces types d'attaques particulièrement difficiles à cerner pour un débutant.

J'ai aussi eu une réflexion sur la bibliothèque *pycryptodome* ; vu son nombre d'utilisateurs, elle a très certainement été optimisée et préparée pour des réelles utilisations et est donc censée être insensible aux attaques auxiliaires de manière générale.

Même si ces codes ne sont pas représentatifs de la réalité ni très pertinents, vous pouvez les retrouver ici : https://github.com/NeospinX/Projet-Side-Channel-Attacks

3. CE QUE J'AI NÉANMOINS APPRIS

Malgré les échecs que j'ai reconnus, j'ai relevé plusieurs points intéressants quant à mon apprentissage. Premièrement, j'ai pu découvrir le concept de Side Channel Attacks, dont je ne soupçonnais pas l'existence jusqu'à maintenant ce qui en fait un plus par rapport à mes connaissances générales liées aux cyber attaques.

J'ai pu faire une sorte de comparatif des différentes attaques, avoir une idée de leur fonctionnement, de leurs méthodes d'attaques et comment les constructeurs ont établi des systèmes incluant des mesures anti-SCA petit à petit.

Concernant les attaques Power Analysis et Timing, j'ai pu en apprendre plus sur ces catégories différentes, complexes mais intrigantes. Je n'avais jamais idée que la puissance perdue causée par les composants et l'algorithme de chiffrement pouvait révéler secrètement des données à notre insu. Il en va de même pour les attaques de temps que je considérais comme « une attaque brute force avec des indices de temps ».

Que le projet soit réussi ou non n'avait donc pas grande importance ; travailler sur ce projet en simultané avec mon stage en entreprise fut très difficile pour moi en raison de mes travaux et autres livrables scolaires. J'ai donc souvent été pressé dans la composition de ce rapport et du projet. Néanmoins, ce que j'ai découvert dans ce projet par rapport aux attaques auxiliaires pourra potentiellement me servir à l'avenir pour ma culture générale en tant que consultant en cabinet de conseil.

VI. BIBLIOGRAPHIE

Side-channel Attacks:

Introduction to Side-Channel Attacks | François-Xavier Standaert
https://www.allaboutcircuits.com/technical-articles/understanding-side-channel-attack-basics/
https://www.synetis.com/attaques_auxiliaires/
https://www.techtarget.com/searchsecurity/definition/side-channel-attack
https://web.maths.unsw.edu.au/~lafaye/CCM/crypto/des.htm
https://www.di.ens.fr/~mrossi/docs/slides-stage.pdf
Side Channel Attacks | Khaled Alashik and Amet Efe
Side Channel Attacks | Labsticc.univ-ubs.fr
https://www.di.ens.fr/~mrossi/docs/rapport-stage.pdf

Timing Attacks:

https://medium.com/spidernitt/introduction-to-timing-attacks-4e1e8c84b32b
https://venafi.com/blog/what-are-timing-attacks-and-how-do-they-threaten-encryption/
https://sqreen.github.io/DevelopersSecurityBestPractices/timing-attack/python#:~:text=Timing%20attacks%20can%20occur%20when,to%20find%20the%20next%20characters
https://blog.sqreen.com/developer-security-best-practices-protecting-against-timing-attacks/

DPA Attack:

Differential Power Analysis | Paul Kocher
https://www.emse.fr/~nadia.el-mrabet/Presentation/Cours5_SCA.pdf
http://www.goubin.fr/papers/dpafinal.pdf
https://pdfs.semanticscholar.org/e92d/50e613fc38849cf39940eaeea914869c6c51.pdf
https://www.tandfonline.com/doi/full/10.1080/23742917.2016.1231523

Autres:

https://pycryptodome.readthedocs.io/en/latest/
https://github.com/topics/side-channel-attacks?l=python
https://scalib.readthedocs.io/en/stable/
https://zetcode.com/python/prettytable/
A Simple and Differential Power Analysis Attack Resistance Circuit for Smart Card Reader Using an Integrated Power Spike Vanisher
https://he-arc.github.io/livre-python/pycrypto/index.html
https://www.dlitz.net/software/pycrypto/api/2.6/Crypto.Cipher.DES-module.html
A Practical Implementation of the Timing Attack