Rapport de laboratoire #2

CRY - Chiffrement symétrique

Alexis Martins

1er mai 2023

Table des matières

1 CBC

1.1 Description de l'algorithme

AES-256 avec le mode opératoire CBC :

- **Taille d'un bloc :** 128 bits (16 octets)
- **Taille de la clé :** 256 bits (32 octets)
- **Taille de l'IV :** 128 bits (16 octets)

1.2 Description de l'attaque

Premièrement, j'ai commencé par analyser l'algorithme qui nous était proposé. J'ai remarqué que le problème ne pouvait pas venir de l'exécution d'AES directement, car c'était fait via une librairie sûre (pycryptodome). Le problème ne venant pas du chiffrement en tant que tel, je me suis attardé sur les paramètres qui lui étaient passés.

J'ai remarqué que l'IV n'était pas totalement aléatoire, car l'algorithme avait simplement un compteur qu'il incrémentait de 1 à chaque fois. Or, si on se réfère au cours l'IV ne doit pas pouvoir être déterminé à l'avance. Cela causerait une vulnérabilité et c'est ce qui s'est passé ici.

Ce que l'on veut atteindre, c'est d'avoir un message clair qui une fois chiffré soit identique au message chiffré de référence (voir schéma plus bas). Pour rappel, CBC XOR le premier bloc avec un IV et ensuite fait le chainage classique avec le cipher précédent et le bloc clair actuel avant d'y appliquer AES. Donc ce que l'on veut surtout, c'est que l'entrée du premier bloc AES soit identique à l'entrée du premier bloc du message chiffré de référence. Après, il suffit que la suite du message soit identique au message clair qui a été utilisé pour créer le chiffré de référence.

On a en notre possession les éléments suivants :

- Un message **M** composé de blocs $M_1, M_2, M_3, M_4, ..., M_n$ (On part du principe que c'est exactement le message utilisé pour créer le chiffré de référence)
- Un message chiffré de référence **C** composé des blocs $C_1, C_2, C_3, C_4, ..., C_n$
- Un IV de référence $IV_{\text{réf}}$
- L'IV utilisé pour chiffrer le message suivant **IV+1**

Le message chiffré de référence a été créer de la manière suivante :

$$C_1 = AES(IV_{réf} \oplus M_1)$$

$$C_2 = AES(C_1 \oplus M_2)$$

$$\dots$$

$$C_n = AES(C_{n-1} \oplus M_n)$$

$$Résultat : C = C_1||C_2||...||C_n$$

Comme dit, on veut avoir le même premier bloc en entrée d'AES et ensuite il faut simplement la même suite de texte clair, ainsi on retrouvera le même chiffré. Il faut que malgré l'application de l'IV par l'algorithme,

on arrive à retrouver $IV_{réf} \oplus M_1$ en entrée du premier bloc AES. Pour réussir à retrouver cette valeur avec l'application de l'IV incrémenté que l'on connait, on doit créer le message suivant :

$$(M_1 \oplus (IV + 1) \oplus IV)||M_2||M_3||M_4||...||M_n$$

De cette manière lorsque l'algorithme va se faire, on va réussir à annuler l'IV incrémenté. Car si on XOR une valeur avec elle-même, on obtient 0.

$$C_1 = AES((IV + 1) \oplus (M_1 \oplus (IV + 1) \oplus IV_{réf})) = AES(IV_{réf} \oplus M_1)$$

$$C_2 = AES(C_1 \oplus M_2)$$

$$\dots$$

$$C_n = AES(C_{n-1} \oplus M_n)$$

$$Résultat : C = C_1||C_2||...||C_n$$

On peut voir ci-dessous un schéma de l'explication précédente. On retrouve à gauche le chiffrement du message de référence et à droite le chiffrement du message bruteforcé. On y voit les deux $IV + 1$ qui s'annulent et qui permettent d'avoir le même premier bloc en entrée d'AES que le message de référence.

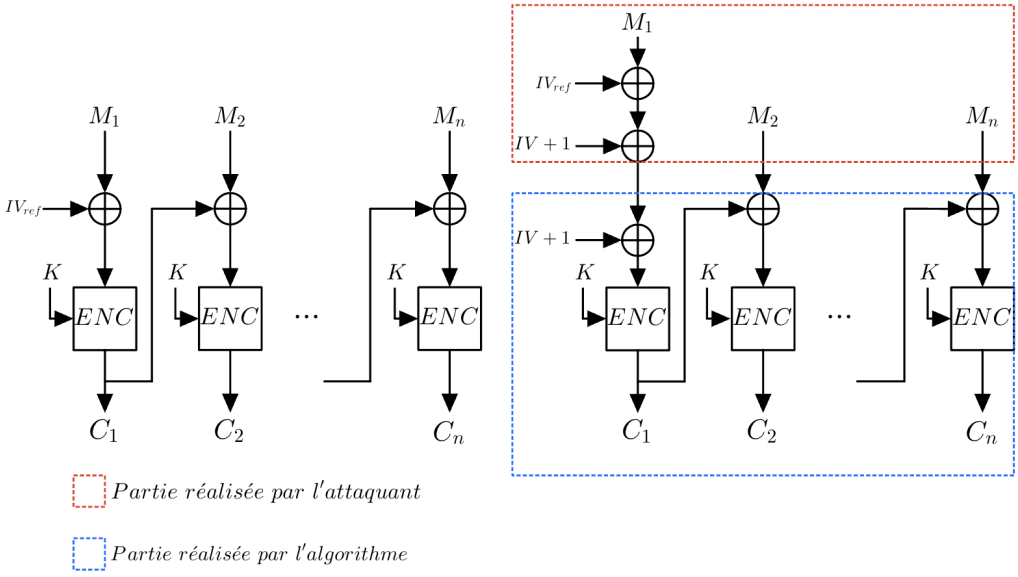

Figure 1: Comparaison du chiffrement du message de référence et du message bruteforcé.

Voici le code Python qui a été utilisé pour réaliser l'attaque :

```python
def breaker(MY_KEY_ID, ct, iv):

    ct = b64decode(ct)
    IV = b64decode(iv)

    # Récupère la prochaine valeur de l'IV utilisée par l'oracle
    next_IV = getNextIV(MY_KEY_ID)

    # Bruteforce toutes les valeurs possibles du salaire.
    for i in range(0, 3000):

        m = b"Le salaire journalier du dirigeant USB est de " + str(i).
            encode() + b" CHF"

        """
        Le XOR avec l'IV de référence permet d'avoir la même valeur que
            celle utilisée par l'oracle pour chiffrer le message de référ
            ence.
        Le XOR avec l'IV suivant permet d'annuler le XOR qui va être effectu
            é par l'oracle.
        """
        m = strxor(strxor(next_IV, IV), m[0:16]) + m[16:]

        (_, cipher) = real_oracle(MY_KEY_ID, m)
        next_IV = increaseIV(next_IV)
        if cipher == ct:
            print(i)
            break
```

Tout cela m'a permis de trouver le salaire du dirigeant USB qui est de 94 CHF.

1.3 Correction de la vulnérabilité

Pour corriger la vulnérabilité, il faut que l'IV soit aléatoire et qu'il ne soit pas possible de le prédire. Je me suis donc permis de simplement utiliser la fonction qui était proposée par la librairie qui utilise déjà un IV aléatoire.

```python
def oracle_fix(key, plaintext):
    cipher = AES.new(key, mode = AES.MODE_CBC)
    return (cipher.iv, cipher.encrypt(pad(plaintext, AES.block_size)))
```

1.4 AES-CTR

Non, on ne peut pas réaliser la même attaque sur le mode opératoire CTR. Car c'est l'algorithme qui gère l'entrée d'AES avec le nonce et le compteur. On ne peut donc pas prédire le chiffré qui va sortir de la boite et donc on ne peut pas annuler la valeur avec un XOR.

2 CCM

2.1 Différences avec CCM

L'algorithme qui nous est donné possède les différences suivantes :

- L'entrée de CBC-MAC ne possède pas de nonce, la longueur du message et les données authentifiées. Il n'y a pas directement de padding fait par l'algorithme, mais il n'accepte que des messages de bonne longueur.
- La valeur du compteur utilisé est remise à zéro au lieu d'utiliser 0 pour CBC-MAC et la suite des compteurs pour le chiffrement.

2.2 Déchiffrement de CCM

J'ai simplement inversé les opérations qui étaient faites dans le chiffrement. Déchiffrer au lieu de chiffrer que ça soit pour le tag ou le message (en réinitialisant le compteur à 0). Finalement, j'ai vérifié que le tag fourni correspondait bien au tag calculé avec le message déchiffré.

```python
def decrypt_ccm(ciphertext: bytes, key: bytes, tag: bytes, nonce: bytes) -> bytes:

    if len(key) != 16:
        raise Exception("Only AES-128 is supported")

    plain = AES.new(key, mode = AES.MODE_CTR, nonce = nonce)
    plaintext = plain.decrypt(ciphertext)

    plain = AES.new(key, mode=AES.MODE_CTR, nonce=nonce)
    if tag != plain.encrypt(cbcmac(plaintext, key)):
        raise Exception("Invalid tag")

    return plaintext
```

2.3 Attaque sur CCM

Ci-dessous un petit schéma pour aider à comprendre l'attaque expliquée.

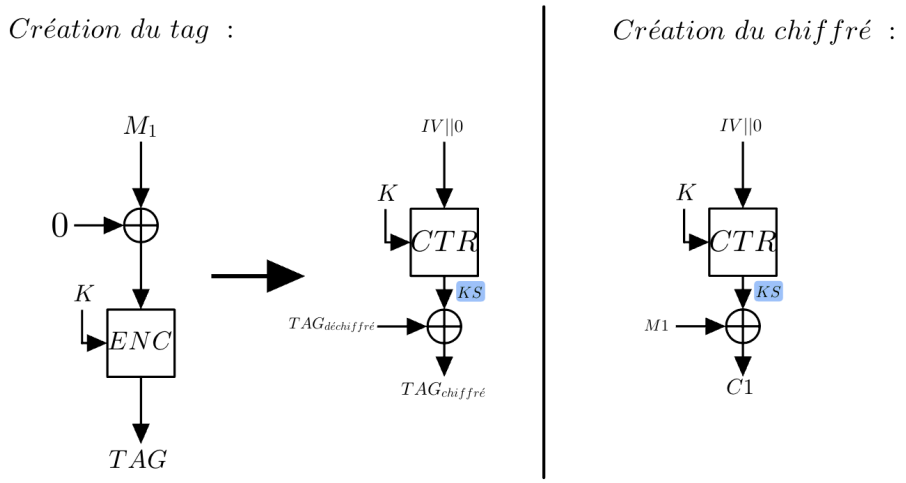

Figure 2: Schéma de CCM sur un bloc.

Pour cette partie, j'ai eu besoin des informations suivantes :

- Un message **M1** composé d'un bloc
- Un message **M2** composé de deux blocs $M2_1, M2_2$
- Un chiffré **C1** composé d'un bloc
- Un chiffré **C2** composé de deux blocs $C2_1, C2_2$
- Un tag clair $TAG1_{Déchiffré}$ qui est le tag généré par CBC-MAC pour le message M1
- Un tag chiffré $TAG1_{Chiffré}$ qui est le tag chiffré
- Le keystream $KS1$ qui est la sortie du bloc AES-CTR qui chiffre M1 et $TAG1_{Chiffré}$
- Le keystream $KS2$ qui est la sortie du bloc AES-CTR qui chiffre M2 et $TAG2_{Chiffré}$. Il a une longueur de 2 blocs $KS2_1, KS2_2$

La première partie consistait à remarquer que le compteur qui était ré-initialisé permettait de déchiffrer le tag chiffré. Pour cela, il fallait remarquer que la sortie (keystream) du bloc AES qui chiffre le tag était similaire à celle du bloc AES qui chiffre le premier bloc du message. Cela venant de la ré-initialisation du compteur.

$$M1 \oplus C1 = KS1$$

$$KS1 \oplus TAG1_{Déchiffré} = TAG1_{Chiffré} \Leftrightarrow TAG1_{Déchiffré} = KS1 \oplus TAG1_{Chiffré}$$

$$TAG1_{Déchiffré} = M1 \oplus C1 \oplus TAG1_{Chiffré}$$

Maintenant, on peut procéder à une attaque similaire à celle réalisée en classe sur CBC-MAC. On va créer un nouveau message clair dont le tag correspond à celui du tag déchiffré que l'on possède. On passe donc du premier message qui fait un bloc à un message qui fait deux blocs.

$$M_{forgé} = M1||(M1 \oplus TAG_{Déchiffré})$$

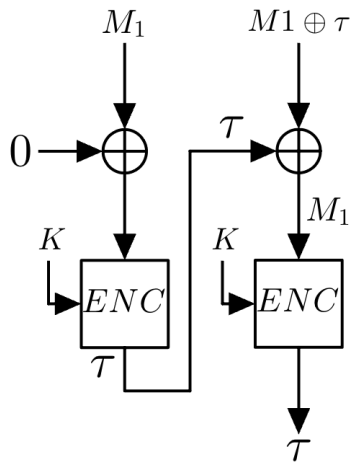

Figure 3: Exemple d'attaque sur CBC-MAC.

Le problème à ce stade, c'est qu'on possède un message et un tag qui ne sont pas chiffrés. Donc si lors du déchiffrement, on fait une vérification, on est sûr que ça ne passera pas. Le moyen que j'ai trouvé pour chiffrer ce nouveau message de deux blocs, c'est de récupérer le keystream du second message chiffré qu'on avait à notre disposition et qui faisait 2 blocs. Ainsi, je peux me permettre de chiffrer le nouveau message de deux blocs et pour le tag, je ne prendrai que le premier bloc du keystream.

$$KS2 = M2 \oplus C2$$

$$C_{forgé} = M_{forgé} \oplus KS2$$

$$TAG_{forgé} = TAG1_{Déchiffré} \oplus KS2_1$$

Ainsi, on récupère le nouveau tag chiffré et le nouveau message chiffré. Il ne faut pas oublier que l'IV correspondant au keystream utilisé, c'est maintenant l'IV du second message, car c'est avec celui-ci que le second keystream que l'on a récupéré a été créé.
Les résultats de l'attaque sont les suivants :

```
Tag : b'SBg9+0/Y1AFUOMrrNm1yvg=='
Chiffré : b'qnaNVBlYqPI3/NtZGaQUTfIp9ht1GKJcl4tGR9F7AOQ='
IV : b'ETRuSjIOL2g='
```

2.4 Correction de l'implémentation de CCM

Durant le cours, on nous a pas mal expliqué qu'il fallait vraiment faire attention lors de l'implémentation d'algorithmes. En effet, il est très facile de faire des erreurs et de se retrouver avec un algorithme qui n'est plus sûr. Il est beaucoup mieux d'utiliser une implémentation déjà existante et qui a été testée.

```
def correct_ccm(message: bytes, key: bytes) -> tuple:

    cipher = AES.new(key, mode = AES.MODE_CCM)
    ciphertext, tag = cipher.encrypt_and_digest(message)
    return (cipher.nonce, ciphertext, tag)

def decrypt_correct_ccm(ciphertext: bytes, key: bytes, nonce: bytes, tag:
    bytes) -> bytes:

    plain = AES.new(key, mode = AES.MODE_CCM, nonce = nonce)
    plaintext = plain.decrypt_and_verify(ciphertext, tag)
    return plaintext
```

3 Bruteforce intelligent

3.1 Attaque Meet-in-the-middle

L'attaque réalisée s'appelle Meet-in-the-middle. Le principe est assez simple au lieu de chercher à déchiffrer deux fois le texte chiffré, on déchiffre une fois le chiffré et on chiffre une fois le clair. Cela a comme impact de réduire radicalement la complexité de l'attaque. On ne réalise plus que 2^{17} opérations au lieu de 2^{32}.
Pour faire cette attaque, on chiffre alors le texte clair avec toutes les clés possibles et on déchiffre le texte chiffré avec toutes les clés possibles jusqu'à retrouver une valeur correspondante. On met la valeur intermédiaire du chiffrement en tant que clé du dictionnaire et la valeur associée sera la clé utilisée. Lors du déchiffrement, si on retrouve une valeur qui correspond à une clé du dictionnaire (même valeur intermédiaire), alors on a trouvé les deux clés.

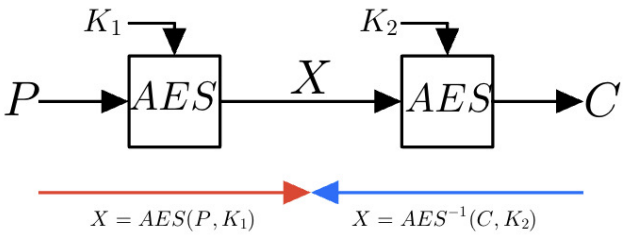

Figure 4: Schéma d'une attaque Meet-in-the-middle.

```
K1 = b'D9EAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA='
K2 = b'IjkAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA='
```

3.2 Complexité algorithmique de l'attaque

Pour l'attaque, on doit premièrement calculer toutes les clés possibles avec leur résultat de chiffrement. Cela prend approximativement 2^{16} opérations. Ensuite, il faut vérifier en déchiffrant le texte chiffré avec toutes les clés possibles jusqu'à retrouver une correspondance avec une des valeurs du dictionnaire. Cela prend dans le pire des cas 2^{16} opérations. On retrouve donc une complexité finale de $O(2 \cdot 2^{16}) = O(2^{17})$. La complexité mémoire est quant à elle de $O(2^{16})$.
Si on compare à une attaque de bruteforce plus classique, on aurait une complexité de $O(2^{32}) = O(2^{16} \cdot 2^{16})$, mais une complexité mémoire de $O(1)$. Cela venant du fait que l'on calcule directement toutes les possibilités de l'enchaînement des deux clés.

3.3 Complexité d'un MITM avec 3-key 3DES

Un Meet-in-the-middle sur un algorithme avec 3 clés implique de réaliser 2^{112} opérations au lieu de 2^{168}.
On l'explique en faisant le même raisonnement que précédemment. On calcule toutes les possibilités de chiffrement de la première clé, suivi du déchiffrement avec la seconde clé et on stocke le résultat dans un dictionnaire.
Ensuite, on déchiffre le chiffré avec la clé restante et on s'arrête dès qu'on trouve une correspondance avec une des valeurs du dictionnaire. Le double chiffrement/déchiffrement implique d'avoir 2^{112} opérations et le déchiffrement implique d'avoir 2^{56} opérations. On a finalement une complexité de $O(2^{112} + 2^{56})$ que l'on peut simplifier en $O(2^{112})$.