Chaînes arc-en-ciel
Attaque par compromis temps-mémoire sur AES-128

Gabriel Blanchot Pierre Fromonot
Master 2 CRYPTIS – Université de Limoges

Année universitaire 2025–2026

Table des matières

1 Introduction

On considère un AES-128 dont la clé secrète est restreinte à un espace réduit de n bits (avec $n \ll 128$). Concrètement, seuls les n premiers bits de la clé sont significatifs, les $128 - n$ bits restants étant fixés à zéro. L'espace de clés effectif est donc de taille $N = 2^n$.

L'objectif est de retrouver la clé secrète k^* à partir d'un chiffré $y = f(k^*)$, où f désigne le chiffrement AES-128 avec un clair fixé et connu. Pour cela, on implémente une attaque par *chaînes arc-en-ciel* (rainbow tables), un compromis temps-mémoire introduit par Oechslin [2] en amélioration des tables de Hellman [1].

Notre implémentation en C utilise les tailles de clés suivantes :

— $n = 8$ bits ($N = 256$) et $n = 16$ bits ($N = 65\,536$) pour les tests,
— $n = 24$ bits ($N \approx 16{,}7 \times 10^6$) pour une attaque jouable,
— $n = 40$ bits ($N \approx 1{,}1 \times 10^{12}$) comme cible finale.

2 La fonction à sens unique f

2.1 Principe

On fixe un bloc clair P de 128 bits (connu de l'attaquant). La fonction f est définie par :

$$f : \mathcal{K} \to \mathcal{C}, \qquad k \mapsto \text{AES-128}_k(P)$$

Autrement dit, $f(k)$ chiffre le clair fixé P en utilisant la clé k. Cette fonction joue le rôle de la fonction à sens unique : étant donné $y = f(k^*)$, retrouver k^* revient à inverser AES.

2.2 Implémentation

L'AES-128 est implémenté suivant le standard NIST (FIPS 197) avec 10 tours. La fonction `f` prend une clé `ki` de 16 octets, effectue l'expansion de clé (`KeyExpansion`) puis chiffre le clair fixé :

Listing 1 – Fonction f – chiffrement AES avec clé réduite

```
void f(key ki, const key round_keys[], int rounds){
    State s;
    bytes_to_state(ki, s);
    AES_Encrypt(s, round_keys, rounds);
    state_to_bytes(s, ki);
}
```

Remarque : En pratique, les `round_keys` sont pré-calculées une seule fois à partir de la clé maître. Ici, le clair fixé est directement la clé `ki` elle-même (la clé est à la fois utilisée comme entrée du chiffrement et comme clé de chiffrement via les round keys). Cela simplifie l'implémentation sans changer le principe de l'attaque.

3 Les fonctions de réduction R_j

3.1 Rôle de la réduction

Après chaque application de f, on obtient un chiffré de 128 bits. Pour construire une chaîne, il faut ramener ce chiffré dans l'espace des clés de n bits. C'est le rôle de la *fonction de réduction* R_j.

Dans une table arc-en-ciel, chaque colonne j utilise une fonction de réduction **différente** R_j. C'est la distinction fondamentale avec les tables de Hellman classiques, où une seule fonction R est utilisée par table.

3.2 Construction de R_j

Chaque R_j est composée de deux étapes :

1. **Troncature + tweak** (`reduction`) : on extrait les n bits de poids fort du chiffré, puis on applique un XOR avec un *tweak* dépendant de j :

$$\texttt{reduced}_j = \texttt{trunc}_n(c) \oplus \texttt{tweak}(j)$$

2. **Expansion** (`expand`) : on reconstruit une clé de 128 bits en plaçant les n bits réduits dans les premiers octets et en complétant par des zéros.

Le tweak est dérivé de l'indice de colonne j par un générateur pseudo-aléatoire de type xorshift :

Listing 2 – Dérivation du tweak pour $n = 16$ bits

```c
static uint32_t derive_tweak16(unsigned long int i){
    uint32_t s = (uint32_t)i + 0x9e3779b9U;  /* constante dor e */
    s ^= s << 13;
    s ^= s >> 7;
    s ^= s << 17;
    uint32_t t = s ^ (s << 8) ^ (s >> 16);
    return t & MASK16;
}
```

Point crucial : la fonction `expand` ne doit **pas** appliquer le tweak inverse. Si elle le faisait, le tweak s'annulerait et toutes les R_j seraient identiques, ce qui réduirait la table arc-en-ciel à une table de Hellman et provoquerait des fusions massives de chaînes.

Listing 3 – Réduction et expansion pour $n = 16$ bits

```c
uint16_t reduction16(const key k, unsigned long int i){
    uint32_t r = ((uint32_t)k[0] << 8) | ((uint32_t)k[1]);
    uint32_t tweak = derive_tweak16(i);
    return (uint16_t)((r ^ tweak) & MASK16);
}

void expand16(uint16_t reduced, key out, unsigned long int i){
    (void)i;  /* PAS de tweak inverse */
    out[0] = (reduced >> 8) & 0xFF;
    out[1] = reduced & 0xFF;
    memset(out+2, 0, 14);
```

```
}

void R16(key k, unsigned long int i){
    uint32_t reduced = reduction16(k, i);
    expand16(reduced, k, i);
}
```

La même structure est déclinée pour $n = 8, 24$ et 40 bits.

4 Phase de pré-calcul

4.1 Construction de la table

On choisit m clés de départ $k_0^{(1)}, \ldots, k_0^{(m)}$ aléatoirement dans l'espace réduit $\mathcal{K}$ et on construit m chaînes de longueur t :

$$k_0 \xrightarrow{f} c_0 \xrightarrow{R_0} k_1 \xrightarrow{f} c_1 \xrightarrow{R_1} k_2 \longrightarrow \cdots \xrightarrow{f} c_{t-1} \xrightarrow{R_{t-1}} k_t$$

On ne stocke que les extrémités (k_0, k_t) pour chaque chaîne.

Listing 4 – Génération de la table arc-en-ciel

```
#define T 500    /* longueur des chaines */
#define M 500    /* nombre de chaines */

typedef struct {
    key start;   /* k_0 */
    key end;     /* k_T */
} Chain;

void get_table(Chain *table, const key round_keys[]){
    for(int i = 0; i < M; i++){
        key k;
        get_random_key(k, 0);        /* k_0 aleatoire dans l'espace
           reduit */
        memcpy(table[i].start, k, 16);

        for(int t = 0; t < T; t++){
            f(k, round_keys, 10);    /* c_t = f(k_t) */
            R16(k, t);               /* k_{t+1} = R_t(c_t) */
        }

        memcpy(table[i].end, k, 16);
    }
}
```

4.2 Paramètres choisis

Pour les tests avec $n = 16$ bits ($N = 65\,536$), on prend $m = t = 500$. Le facteur de couverture est $c = mt/N = 500 \times 500/65\,536 \approx 3{,}8$, ce qui donne une probabilité de succès théorique d'environ 88% (voir section 8).

— **Mémoire** : $m = 500$ couples (k_0, k_t).
— **Coût du pré-calcul** : $m \times t = 250\,000$ évaluations de f.
— **Temps d'attaque** : au plus $t(t+1)/2 = 125\,250$ évaluations de f.

5 Phase d'attaque

5.1 Algorithme

Étant donné le chiffré cible $y = f(k^*)$, on cherche la clé k^*.

L'algorithme parcourt chaque position hypothétique j (de $t-1$ à 0) où k^* pourrait se trouver dans une chaîne :

1. **Calcul du candidat de fin :** on applique R_j au chiffré y, puis alternativement f et $R_{j+1}, \ldots, R_{t-1}$ pour obtenir un endpoint candidat.
2. **Recherche dans la table :** on compare cet endpoint avec les k_t stockés.
3. **Reconstruction :** en cas de correspondance, on reconstruit la chaîne depuis k_0 en appliquant alternativement f et R_j. On compare après chaque f (et avant R_j) avec le chiffré cible y. Si $f(\text{prev}) = y$, alors `prev` est la clé cherchée.

Points importants pour l'implémentation :
— Dans la phase de recherche, on applique d'abord R_j (pour passer du chiffré à une clé), puis $f + R_{j+1}$, etc.
— Dans la reconstruction, la comparaison se fait **après** f mais **avant** R_j : en effet, y est un chiffré (sortie de f), il faut donc comparer avec une sortie de f, pas avec une clé réduite.

Listing 5 – Fonction d'attaque

```
int attack(const key target, Chain *table,
           const key round_keys[]) {
    uint8_t candidate[16], work[16], prev[16];

    for (int pos = T - 1; pos >= 0; --pos) {
        /* Phase 1 : calcul de l'endpoint candidat */
        memcpy(candidate, target, 16);
        R16(candidate, pos);            /* R_pos en premier */
        for (int t = pos + 1; t < T; ++t) {
            f(candidate, round_keys, 10);
            R16(candidate, t);
        }

        /* Phase 2 : recherche dans la table */
        for (int i = 0; i < M; ++i) {
            if (memcmp(candidate, table[i].end, 16) != 0)
                continue;

            /* Phase 3 : reconstruction */
            memcpy(work, table[i].start, 16);
            for (int j = 0; j < T; ++j) {
                memcpy(prev, work, 16);
                f(work, round_keys, 10);
```

```
24
25                  if (memcmp(work, target, 16) == 0) {
26                      /* prev est la cle cherchee */
27                      return 1;
28                  }
29                  R16(work, j);
30              }
31              /* sinon : faux positif */
32          }
33      }
34      return 0;   /* echec */
35  }
```

5.2 Faux positifs

Un *faux positif* survient lorsqu'un endpoint candidat correspond à un endpoint stocké dans la table, mais la clé cible k^* ne se trouve pas dans cette chaîne. Cela arrive quand deux chaînes différentes convergent vers le même endpoint sans passer par les mêmes clés intermédiaires (collision de fin de chaîne).

Les faux positifs augmentent le temps de l'attaque mais ne produisent pas de résultat incorrect : la vérification par reconstruction les élimine.

6 Tests et résultats expérimentaux

6.1 Vérification de l'AES

L'implémentation AES est validée avec le vecteur de test officiel du NIST (FIPS 197, Appendix A.1) :

- Clé : `2b7e1516 28aed2a6 abf71588 09cf4f3c`
- Clair : `3243f6a8 885a308d 313198a2 e0370734`
- Chiffré attendu : `3925841d 02dc09fb dc118597 196a0b32`

Le chiffrement et le déchiffrement produisent les résultats attendus.

6.2 Vérification de la réduction

On vérifie que pour toute clé k et tout indice j :

$$\texttt{reduction16}(\texttt{expand16}(\texttt{reduction16}(k, j), j), j) = \texttt{reduction16}(k, j)$$

c'est-à-dire que la composition réduction $\circ$ expansion $\circ$ réduction est idempotente.

On vérifie également que $R_i \neq R_j$ pour $i \neq j$ en s'assurant que les tweaks sont bien distincts.

6.3 Résultats de l'attaque ($n = 16$ bits)

Avec les paramètres $m = 500$, $t = 500$, $N = 2^{16} = 65\,536$:

- Facteur de couverture : $c = mt/N \approx 3{,}8$.
- Probabilité théorique (avec collisions) : $\approx 88\,\%$.
- **Résultat expérimental : 9 succès sur 10 exécutions** (90%).

— Nombre moyen de faux positifs : ≈ 1 par attaque.
— Temps de pré-calcul : < 1 seconde.
— Temps d'attaque : < 1 seconde.

Ces résultats sont cohérents avec la théorie développée dans la section suivante.

7 Étude du compromis temps-mémoire

7.1 Couverture de l'espace de clés

Chaque chaîne de longueur t traverse t clés distinctes (en l'absence de collisions). Avec m chaînes, le nombre maximal de clés couvertes est $m \times t$.

Pour couvrir l'intégralité de l'espace $\mathcal{K}$, on veut donc :

$$m \times t \;\geq\; N \tag{1}$$

En pratique, on prend $m \times t$ légèrement supérieur à N pour compenser les collisions inévitables entre chaînes.

7.2 Courbe de compromis

Notons $M = m$ la mémoire (nombre d'entrées stockées) et $T_{\text{att}} = t^2/2$ le temps d'attaque en ligne. Avec la contrainte de couverture $m \times t = N$, on a $t = N/m$, d'où :

$$\boxed{T_{\text{att}} \;=\; \frac{t^2}{2} \;=\; \frac{N^2}{2\,M^2}} \qquad \text{soit} \qquad T_{\text{att}} \cdot M^2 \;=\; \frac{N^2}{2} \tag{2}$$

C'est la **courbe de compromis temps-mémoire**. On peut librement régler le curseur entre les deux extrêmes :

— *Plus de mémoire, moins de temps* : augmenter M (plus de chaînes, plus courtes).
— *Moins de mémoire, plus de temps* : diminuer M (moins de chaînes, plus longues).

7.3 Paramètres optimaux

Proposition 1. *Au point d'équilibre $M \approx T_{att}$, les paramètres optimaux satisfont :*

$$\boxed{M \;\approx\; T_{att} \;\approx\; N^{2/3}}$$

Démonstration. En imposant $M = T_{\text{att}}$ dans la relation (2) :

$$M \cdot M^2 = \frac{N^2}{2} \quad\Longrightarrow\quad M^3 = \frac{N^2}{2} \quad\Longrightarrow\quad M = \left(\frac{N^2}{2}\right)^{1/3} = \frac{N^{2/3}}{2^{1/3}} \;\approx\; 0{,}79 \cdot N^{2/3}$$

La longueur de chaîne correspondante est :

$$t = \frac{N}{M} \approx \frac{N}{N^{2/3}} = N^{1/3}$$

Le coût du pré-calcul reste $m \times t = N$ (comparable à une recherche exhaustive), mais il n'est effectué **qu'une seule fois** et peut être réutilisé pour attaquer autant de chiffrés que l'on souhaite. □

Remarque 1. *Le gain par rapport à la force brute est considérable. Par exemple, pour* $N = 2^{40}$ *:*

Méthode	***Mémoire***	***Temps en ligne***
Force brute	$\mathcal{O}(1)$	$2^{40} \approx 10^{12}$
Table complète	2^{40}	$\mathcal{O}(1)$
Rainbow table	$\mathbf{2^{27} \approx 10^8}$	$\mathbf{2^{27} \approx 10^8}$

On passe d'un coût de 10^{12} *à environ* 10^8 *en temps et en mémoire : un gain d'un facteur* $\approx 10\,000$*.*

Attention : les paramètres optimaux $m = t = N^{1/3}$ ne correspondent qu'à une *seule* table arc-en-ciel. Avec ces paramètres, le facteur de couverture est $mt/N = N^{-1/3} \ll 1$, ce qui donne une probabilité de succès très faible pour une seule table. Hellman propose d'utiliser t tables distinctes (chacune avec sa propre famille de fonctions de réduction) pour atteindre une probabilité globale d'environ 80%. En revanche, pour atteindre 80–90% avec une *seule* table, il faut $mt \approx 3N$ à $4N$.

8 Analyse de la probabilité de succès

8.1 Modèle sans collisions

En l'absence de collisions, chaque chaîne couvre exactement t clés distinctes. La probabilité qu'une clé secrète k^* choisie uniformément *ne soit pas* couverte par la table est :

$$P(\text{échec}) = \left(1 - \frac{1}{N}\right)^{m \times t} \approx e^{-mt/N} \tag{3}$$

Avec $m \times t = N$ (couverture exacte) : $P(\text{succès}) \approx 1 - e^{-1} \approx 63\,\%$.

Pour atteindre une probabilité plus élevée, on prend $m \times t = c \cdot N$ avec $c > 1$:

$$P(\text{succès}) \approx 1 - e^{-c}$$

8.2 Modèle avec collisions

En pratique, des collisions réduisent la couverture. Lorsque deux chaînes produisent la même clé à la même colonne j, elles *fusionnent* et couvrent les mêmes clés pour le reste de la chaîne.

Soit m_j le nombre de clés **distinctes** à la colonne j. L'évolution est donnée par la récurrence (analogue au paradoxe des anniversaires) :

$$\boxed{m_0 = m, \qquad m_{j+1} = N\left(1 - e^{-m_j/N}\right)} \tag{4}$$

À chaque étape, les m_j entrées produisent $m_{j+1} < m_j$ sorties distinctes. La probabilité de succès tenant compte des collisions est alors :

$$P(\text{succès}) = 1 - \prod_{j=0}^{t-1}\left(1 - \frac{m_j}{N}\right) \approx 1 - \exp\left(-\frac{1}{N}\sum_{j=0}^{t-1} m_j\right) \tag{5}$$

8.3 Probabilité avec les paramètres optimaux

Proposition 2. *Avec un facteur de couverture $c = mt/N \approx 3$ à 4 et des fonctions de réduction bien choisies, la probabilité de succès atteint environ* **86 %** *à* **90 %**.

Justification. En résolvant la récurrence (4) par l'approximation continue $\frac{dm}{dj} \approx -\frac{m^2}{2N}$, on obtient :

$$m(j) = \frac{2N \cdot m_0}{m_0 \cdot j + 2N}$$

La couverture totale est alors :

$$\Sigma = \sum_{j=0}^{t-1} m_j \;\approx\; \int_0^t m(j)\,dj \;=\; 2N \ln\!\left(\frac{m_0 \cdot t}{2N} + 1\right) \;=\; 2N \ln\!\left(\frac{c}{2} + 1\right)$$

La probabilité de succès devient :

$$P(\text{succès}) \approx 1 - e^{-\Sigma/N} = 1 - e^{-2\ln(c/2+1)} = 1 - \frac{1}{(c/2+1)^2}$$

Application numérique pour différentes valeurs de $c = mt/N$:

$c = mt/N$	Σ/N (couverture effective)	P(succès)
1	$\approx 0{,}81$	$\approx 56\,\%$
2	$\approx 1{,}39$	$\approx 75\,\%$
3	$\approx 1{,}83$	$\approx 84\,\%$
4	$\approx 2{,}20$	$\approx 89\,\%$
5	$\approx 2{,}50$	$\approx 92\,\%$

Ainsi, avec $c \approx 3{,}8$ (nos paramètres : $m = 500$, $t = 500$, $N = 65\,536$), on atteint une probabilité de succès d'environ **88 %**, ce qui est cohérent avec le résultat expérimental de 9/10 succès. □

Remarque 2. *Pour augmenter la probabilité au-delà de 90 % sans augmenter excessivement m ou t, on peut utiliser plusieurs tables arc-en-ciel indépendantes (avec des familles de réduction différentes). Avec ℓ tables, chacune de probabilité p :*

$$P_\ell(\textit{succès}) = 1 - (1-p)^\ell$$

Par exemple, $\ell = 2$ tables avec $p = 75\,\%$ donnent $P_2 \approx 93{,}7\,\%$.

9 Importance des fonctions de réduction distinctes

L'utilisation de fonctions de réduction *différentes* à chaque colonne est essentielle. Si toutes les R_j sont identiques (table de Hellman classique), deux chaînes qui collisionnent à n'importe quelle position fusionnent définitivement, ce qui augmente considérablement le taux de collisions et réduit drastiquement la couverture effective.

Dans une table arc-en-ciel, deux chaînes ne peuvent fusionner que si elles collisionnent à la **même colonne** j (car R_j est la même pour les deux chaînes à cette colonne). Si elles se croisent à des colonnes différentes, elles divergent aussitôt car $R_i \neq R_j$.

Vérification expérimentale : avec des R_j identiques (bug dans `expand` qui annulait le tweak), nous observions un taux de succès de 2/10 avec des milliers de faux positifs. Après correction (tweak préservé, toutes les R_j distinctes), le taux passe à 9/10 avec en moyenne 1 seul faux positif, confirmant l'importance cruciale de cette propriété.

10 Résumé

Paramètre	**Notation**	**Valeur optimale**
Espace de clés	N	—
Nombre de chaînes (mémoire)	m	$\sim N^{2/3}$
Longueur des chaînes	t	$\sim N^{1/3}$
Temps d'attaque en ligne	$T_{\text{att}} \approx t^2/2$	$\sim N^{2/3}$
Pré-calcul (une seule fois)	$m \times t$	$\sim N$
Courbe de compromis	$T_{\text{att}} \cdot M^2$	$= N^2/2$
Probabilité de succès (1 table)	P	≈ 86–$90\,\%$ (avec $c \approx 4$)

Les chaînes arc-en-ciel réalisent un compromis temps-mémoire efficace : au lieu de payer N en temps (force brute) ou N en mémoire (table complète), on ne paie que $N^{2/3}$ pour les deux, au prix d'un pré-calcul de coût N effectué une seule fois.

Références

[1] M. E. Hellman, *A Cryptanalytic Time-Memory Trade-Off*, IEEE Transactions on Information Theory, vol. 26, no. 4, pp. 401–406, 1980.

[2] P. Oechslin, *Making a Faster Cryptanalytic Time-Memory Trade-Off*, Advances in Cryptology – CRYPTO 2003, LNCS vol. 2729, pp. 617–630, Springer, 2003.