

Direct Meet-in-the-Middle Attack

Julian Mouthon, Mario Razafinony

26 décembre 2025

Résumé

Ce rapport présente la parallélisation d'un code d'attaque *Meet-in-the-Middle* (MitM) en utilisant MPI et OpenMP.

Table des matières

1	Introduction	1
2	Parallélisation	1
2.1	Distribution des données	2
2.2	Vectorisation	2
3	Phases de l'algorithme	2
3.1	Phase 1 : Construction distribuée du dictionnaire	2
3.1.1	Objectif	2
3.1.2	Étapes détaillées	2
3.2	Phase 2 : Recherche de collisions	2
3.2.1	Objectif	2
3.2.2	Étapes détaillées	3
3.3	Phase 3 : Agrégation des résultats	3
3.3.1	Objectif	3
3.3.2	Étapes détaillées	3
4	Algorithme principal	4
5	Difficultés rencontrées	4
5.1	Approche Boss-Workers	4
5.2	Communications à chaque itération	5

1 Introduction

L'attaque *Meet-in-the-Middle* est une technique qui exploite la possibilité de diviser un chiffrement en deux parties indépendantes. Le but est de chercher des paires de clés (k_1, k_2) telles que :

$$\text{Enc}_{k_1}(\text{Enc}_{k_2}(P_1)) = C_1 \quad \text{et} \quad \text{Enc}_{k_1}(P_0) = \text{Dec}_{k_2}(C_0)$$

où (P_0, C_0) et (P_1, C_1) sont deux paires texte clair-chiffré connues.

2 Parallélisation

Pour maximiser les performances, nous combinons plusieurs niveaux de parallélisme :

- **MPI** : parallélisme multi-processus
- **OpenMP** : parallélisme multi-thread
- **Sharding** : distribution du dictionnaire entre processus
- **Vectorisation** : TODO

2.1 Distribution des données

Le sharding est réalisé par fonction de hachage :

$$\text{shard_id} = h(\text{clé}) \bmod P$$

où P est le nombre de processus MPI. Cela garantit une distribution uniforme des données entre les processus.

2.2 Vectorisation

Nous avons utilisé `#pragma omp parallel for simd` pour l'initialisation des tableaux :

```
#pragma omp parallel for simd
for (u64 i = 0; i < local_dict_size; i++) {
    local_A[i].k = EMPTY;
}
```

Cela permet au compilateur de générer des instructions pour que le processeur puisse traiter plusieurs éléments à la fois.

3 Phases de l'algorithme

3.1 Phase 1 : Construction distribuée du dictionnaire

3.1.1 Objectif

Construire un dictionnaire distribué associant chaque image $f(x) = \text{Enc}_x(P_0)$ à sa préimage x .

3.1.2 Étapes détaillées

1. **Partitionnement de l'espace des clés :**
 - Chaque processus MPI traite un sous-ensemble des clés : $\{x \mid x = \text{rank} \bmod (\text{world_size})\}$
 - Chaque thread OpenMP traite une partie de ce sous-ensemble
2. **Calcul local des paires (image, préimage) :** Pour chaque clé x assignée :

$$z = f(x) = \text{Enc}_x(P_0) \quad \text{où} \quad z \in \{0, 1\}^n$$

- Si z appartient au shard local : insertion directe
- Sinon : on le met dans buffer pour l'envoyer au processus propriétaire plus tard

3. Échange asynchrone des données :

- 1: **Chaque processus :**
- 2: Calcule `send_counts[i]` = nombre d'éléments pour le processus i
- 3: `MPI_Alltoall` : échange des comptes entre tous les processus
- 4: `MPI_Irecv` : poste toutes les réceptions de manière asynchrone
- 5: `MPI_Isend` : envoie toutes les données de manière asynchrone
- 6: `MPI_Wait` : attend que toutes les réceptions soient terminées
- 7: Traite les données reçues (insertion dans le dictionnaire)
- 8: `MPI_Wait` : attend que tous les envois soient terminés

4. Insertion dans les tables de hachage locales

3.2 Phase 2 : Recherche de collisions

3.2.1 Objectif

Trouver toutes les paires (x, z) telles que :

$$f(x) = g(z) \quad \text{et} \quad \text{Enc}_x(\text{Enc}_z(P_1)) = C_1$$

3.2.2 Étapes détaillées

1. **Calcul local des images inverses** : Pour chaque z assigné au processus :

$$y = g(z) = \text{Dec}_z(C_0)$$

- Si y appartient au shard local : recherche locale dans le dictionnaire
- Sinon : mise en buffer pour interrogation à distance

2. **Recherche distribuée des collisions** :

- Interrogation du dictionnaire avec y comme clé
- Pour chaque x trouvé : vérification de la deuxième paire (P_1, C_1)
- Utilisation de la fonction `is_good_pair()` pour validation

3. **Échange asynchrone des requêtes** : Même schéma que la Phase 1 mais avec :

- Envoi des paires (y, z) aux propriétaires de y
- Réception et traitement des résultats
- Tag différent (2) pour distinguer des communications de la Phase 1

4. **Vérification complète des paires** : Pour chaque collision candidate (x, z) :

$$\text{vérifier si } \text{Enc}_x(\text{Enc}_z(P_1)) = C_1$$

- Test complet avec les deux clés
- Filtrage des faux positifs
- Arrêt après avoir trouvé **maxres** solutions

3.3 Phase 3 : Agrégation des résultats

3.3.1 Objectif

Rassembler toutes les solutions trouvées sur le processus racine (rank 0).

3.3.2 Étapes détaillées

1. **Collecte des comptes** :

```
MPI_Gather(&nres_local, 1, MPI_UINT64_T,
          all_nres, 1, MPI_UINT64_T, 0, MPI_COMM_WORLD);
```

Chaque processus envoie son nombre de solutions au processus 0.

2. **Collecte des solutions** :

```
MPI_Gatherv(local_k1, nres_local, MPI_UINT64_T,
            global_k1, counts, displs, MPI_UINT64_T, 0);
MPI_Gatherv(local_k2, nres_local, MPI_UINT64_T,
            global_k2, counts, displs, MPI_UINT64_T, 0);
```

Collecte avec déplacements variables pour gérer des nombres de solutions différents.

3. **Validation finale** (rank 0 uniquement) :

- Vérification que $f(k_1) = g(k_2)$ pour chaque paire
- Vérification complète avec la deuxième paire (P_1, C_1)
- Affichage des solutions valides

4 Algorithme principal

Algorithm 1 Algorithme golden_claw_search

```

1: procedure GOLDEN_CLAW_SEARCH(maxres, k1[], k2[])
2:   Phase 1 : Construction du dictionnaire
3:   for  $x$  tel que  $x \equiv \text{rank} \pmod{\text{world\_size}}$  do
4:      $z \leftarrow f(x)$ 
5:      $\text{shard} \leftarrow z \bmod \text{world\_size}$ 
6:     if  $\text{shard} = \text{rank}$  then
7:       SHARD_DICT_INSERT( $z, x$ )
8:     else
9:       Ajouter ( $z, x$ ) à buffer pour processus  $\text{shard}$ 
10:    end if
11:  end for
12:  Échanger les buffers entre tous les processus
13:  Phase 2 : Recherche de collisions
14:  for  $z$  tel que  $z \equiv \text{rank} \pmod{\text{world\_size}}$  do
15:     $y \leftarrow g(z)$ 
16:     $\text{shard} \leftarrow y \bmod \text{world\_size}$ 
17:    if  $\text{shard} = \text{rank}$  then
18:       $X \leftarrow \text{SHARD\_PROBE\_LOCAL}(y)$ 
19:      for  $x \in X$  do
20:        if IS_GOOD_PAIR( $x, z$ ) then
21:          Ajouter ( $x, z$ ) aux résultats locaux
22:        end if
23:      end for
24:    else
25:      Ajouter ( $y, z$ ) à buffer pour processus  $\text{shard}$ 
26:    end if
27:  end for
28:  Échanger et traiter les buffers
29:  Phase 3 : Agrégation des résultats
30:  Collecter tous les résultats sur le processus 0
31:  return solutions trouvées
32: end procedure

```

5 Difficultés rencontrées

La parallélisation s'est révélée plus complexe que prévu. Avant d'aboutir à la version finale présentée dans ce rapport, plusieurs approches ont été explorées et abandonnées en raison de performances insuffisantes.

5.1 Approche Boss-Workers

Une première implémentation reposait sur un schéma *boss-workers* :

- un processus maître distribuait dynamiquement des blocs de clés aux travailleurs,
- les travailleurs effectuaient les calculs puis renvoyaient leurs résultats au maître.

Cette approche s'est révélée être contre-productive pour plusieurs raisons :

- le processus maître devenait rapidement un **goulot d'étranglement**.
- les communications étaient très fréquentes et de petite taille, ce qui augmentait la latence MPI
- la charge de messages et de synchronisations annulaient complètement les bénéfices du parallélisme.

Cette version parallèle était bien **plus lente que le code séquentiel**.

5.2 Communications à chaque itération

Une deuxième tentative était proche de la version finale, mais avec une stratégie de communication différente :

- chaque paire (z, x) ou (y, z) était envoyée immédiatement à son processus propriétaire à l'aide de `MPI_Send`,
- les communications étaient donc déclenchées à chaque itération de boucle

Cette version a rapidement montré ses limites :

- elle générerait un **nombre extrêmement élevé de messages MPI** et de petites tailles.
- le coût de gestion des communications dominait le temps de calcul
- la saturation du réseau dégradait les performances

Cette version était aussi bien **plus lente que le code séquentiel**.