Chiffrement homomorphique appliqué au Machine Learning

Protection de l'information

Justin BOSSARD Tom MAFILLE

Le chiffrement homomorphique

Définition générale

Le chiffrement homomorphique est une forme de cryptographie qui permet d'effectuer des opérations sur des données chiffrées, sans jamais avoir besoin de les déchiffrer.

L'avantage est qu'un serveur (ou un tiers) peut manipuler les données sans jamais voir leur contenu, ce qui est crucial pour la confidentialité. Il y a de nombreuses applications, notamment dans le cloud, le médical, la finance ou l'intelligence artificielle, et dans notre cas la reconnaissance d'image. Concrètement, le résultat d'une opération entre deux membres cryptés doivent donner un résultat qui, une fois décrypté, donne le résultat qu'aurait eu l'opération sur les deux membres avant l'opération de cryptage.

Définition formelle

D'une manière plus formelle, considérons deux messages clairs m_1 et m_2 et $\star$ une opération simple telle que l'addition ou la multiplication. Un schéma de chiffrement E est dit homomorphe si, pour ces deux messages m_1 et m_2, et l'opération $\star$, on a :

$$E(m_1) \star E(m_2) = E(m_1 \circ m_2)$$

Soit l'opération entre le crypté de m_1 et le crypté de m_2 donne un résultat qui correspond au crypté d'une opération entre m_1 et m_2. Pour un tiers qui effectue le calcul, aucune information n'a fuitée : néanmoins, des opérations ont été réalisées sur les nombres. On peut donc déléguer le calcul sans crainte de fuites de données.

Maintenant, nous allons nous pencher plus précisément sur le fonctionnement du chiffrement homomorphe et les étapes clés qu'il implique d'un point de vue général. En effet, pour que le chiffrement homomorphe fonctionne, plusieurs fonctions clés doivent être utilisées. Ces fonctions permettent respectivement de générer des clés, de chiffrer des données, de réaliser des calculs sur ces données, et de les déchiffrer une fois les opérations terminées. Tout type de chiffrement homomorphe implique ces 4 étapes clés. Il est tout de même bon de noter que c'est seulement la base indéfectible de ce qui constitue le chiffrement homomorphe et que d'autres étapes peuvent être engagées suivant les différents types de chiffrement homomorphes que nous aborderons par la suite.

Voici ces étapes :

1. En premier lieu, la fonction de génération des clés

C'est la première étape d'un système de chiffrement homomorphe. Elle génère deux types de clés : - Clé publique : utilisée pour chiffrer les données. - Clé privée : utilisée pour déchiffrer les données.

Les clés sont générées à partir de paramètres cryptographiques tels que des grands nombres premiers, et elles permettent de garantir que seules les personnes possédant la clé privée peuvent déchiffrer les données. Nous ne

nous attarderons pas sur cela étant donné que nous avons étudié ça en cours. On peut tout de même noter que les clés publiques et privées doivent être générées avec une structure mathématique qui permettra d'effectuer certaines opérations sur les données chiffrées, on ne peut pas les choisir au hasard. On ne rentrera pas dans ces détails dans cet exposé, mais par exemple, si on veut autoriser des additions dans le monde chiffré, comme dans le chiffrement de Paillier que nous verrons tout à l'heure, on choisit une structure qui rend cela possible — ici, un groupe de Z mod n^2. Pour CKKS que nous verrons également, c'est encore plus complexe, car on doit pouvoir faire des multiplications, des divisions approximées, et gérer la précision. Les clés sont donc construites autour de polynômes, avec des paramètres très spécifiques (représentation des vecteurs de nombres réels ou complexes comme des polynômes dans un anneau modulo un cyclotomique pour les plus curieux et matheux).

2. Ensuite, la fonction de chiffrement

La fonction de chiffrement prend un message en clair et le transforme en un message chiffré à l'aide de la clé publique. Le chiffrement doit être conçu de manière à préserver la sécurité du message, tout en permettant l'application d'opérations sur le message chiffré. Les messages sont généralement représentés sous forme d'entiers, mais elles peuvent également être sous forme de vecteurs ou de flottants suivant la complexité de la fonction homomorphe mise en jeu. Le chiffrement transforme ces données en un format difficilement lisible sans la clé privée.

Le chiffrement homomorphe encode les données de manière à ce que certaines opérations effectuées sur les chiffrés aient un équivalent direct sur les données en clair. C'est la structure même du chiffrement (et non juste la clé publique) qui rend cela possible.

3. Puis la fonction d'évaluation (opérations sur les données chiffrées)

La fonction d'évaluation permet de réaliser des calculs ou des opérations sur les données chiffrées. Ces opérations peuvent être de différents types, selon le type de chiffrement homomorphe : - Addition homomorphe : Ajouter deux valeurs chiffrées, ce qui donnera une nouvelle valeur chiffrée représentant la somme des messages en clair.

`- Multiplication homomorphe : Multiplier deux valeurs chiffrées pour obtenir un résultat chiffré correspon`

Ces opérations sont réalisées sur les données chiffrées, et l'idée est de préserver la sécurité des données tout en effectuant les calculs nécessaires. Il existe différents niveaux de fonctionnalité selon les schémas (chiffrement partiellement homomorphe, totalement homomorphe, etc.) que nous allons étuier dans la prochaine slide.

C'est ici que l'homomorphie est véritablement exploitée. Les schémas homomorphes sont construits de manière à permettre des opérations dans l'espace chiffré qui ont un sens dans l'espace en clair. C'est cette étape qui distingue un chiffrement classique d'un chiffrement homomorphe.

4. Et enfin la fonction de déchiffrement

La fonction de déchiffrement permet de récupérer les messages en clair à partir des données chiffrées après qu'une opération a été effectuée. Cette fonction utilise la clé privée pour déchiffrer le résultat, et elle doit être conçue de manière à garantir que le déchiffrement donne le bon résultat des calculs réalisés sur les données chiffrées.

Par exemple, après avoir ajouté deux messages chiffrés, la fonction de déchiffrement permet de récupérer le résultat de cette addition sur les messages en clair.

Avant de passer à la suite, nous allons ouvrir une parenthèse : c'est important de noter un certain nombre de points concernant le chiffrement homomorphe. Il est nécessaire de savoir qu'il existe plusieurs principes à respecter pour que le chiffrement homomorphe soit correct. Il ne s'agit pas uniquement de trouver ces fonctions qui correspondent et le tour est joué. En plus de respecter les conditions cryptographiques basiques (chiffrement sûr ou presque-sûr, besoins des systèmes d'information (concernant la confidentialité, l'authentification, l'intégrité, la non-répudiation et la disponibilité) et respectant le principe de Kerchkhoff), le chiffrement doit remplir une condition supplémentaire, qui est la suivante.

1. Correction (Correctness)

Le principe fondamental du cryptage homomorphique est la correction. Cela signifie que les opérations réalisées sur des données chiffrées doivent produire des résultats corrects lorsqu'elles sont décryptées. En d'autres termes, si on applique une opération sur des données chiffrées, le déchiffrement du résultat doit correspondre à l'opération effectuée sur les données originales. Cela garantit que le chiffrement homomorphe fonctionne de manière fiable et qu'il n'introduit aucune erreur durant les calculs.

Maintenant, on peut se dire que cela est naturel lorsqu'il y a homomorphie. Alors pourquoi la correction n'est pas garantie par simple homomorphie ?

Une fonction de chiffrement peut être homomorphe sur une certaine opération, mais ne pas garantir la correction dans toutes les conditions, notamment à cause de deux facteurs principaux : - Le bruit introduit dans le chiffrement Dans la plupart des schémas de chiffrement homomorphes modernes (comme BGV ou CKKS que nous allons voir...), chaque opération (addition ou multiplication) augmente le bruit dans le chiffré. Ce bruit est d'erreur introduite pour assurer la sécurité du chiffrement: tant que le bruit reste en dessous d'un certain seuil, le déchiffrement est correct ; mais si le bruit devient trop grand (après de nombreuses opérations), le déchiffrement peut échouer, même si la fonction reste homomorphe au sens formel. - La précision et l'encodage (cas du chiffrement de nombres réels)

```
Dans des schémas comme CKKS, qui permettent de traiter des nombres flottants (approximatifs), la correctio
Le résultat déchiffré est proche (mais pas toujours exactement égal) à celui qu'on aurait obtenu en clair.
Une fonction peut être homomorphe sans être "correcte" dans tous les cas.
```

Pour assurer la correction, on peut utiliser le bootstrapping. Qu'est-ce que c'est le bootstrapping ? Le bootstrapping est une technique clé pour rendre le chiffrement homomorphe praticable sur des données de taille plus importante ou pendant plusieurs étapes de calcul. Comme on vient de l'aborder, les schémas de chiffrement homomorphes souffrent souvent d'un croissance exponentielle du bruit au fur et à mesure que des opérations sont effectuées. Ce bruit peut rendre les données inutilisables pour des opérations supplémentaires.

Le bootstrapping consiste à réinitialiser le bruit à un niveau acceptable, de sorte que l'on puisse continuer à effectuer des calculs sur les données chiffrées sans que le bruit ne compromette le résultat final. C'est un processus coûteux en ressources, mais il est nécessaire pour rendre les calculs sur des données chiffrées pratiquement réalisables à grande échelle.

Pour résumer, il faut donc ajouter une condition explicite de correction, qui garantit que le déchiffrement d'un calcul homomorphe donne bien le bon résultat (ou un résultat approché admissible), tant que certaines conditions sont respectées. C'est pour cela que dans les définitions formelles, on distingue : * Homomorphisme : structurelle (préserve les opérations) * Correction : fonctionnelle (le résultat est bon) * Sécurité : l'attaquant n'apprend rien

Nous pouvons maintenant fermer cette parenthèse et nous attaquer à la suite concernant les chiffrements homomorphes partiel et complet.

Comme nous l'avons rapidement abordé précédemment, il existe plusieurs classes parmi les fonctions d'évaluation du chiffrement homomorphe. Cette fonction doit être soigneusement choisie pour permettre des opérations sur les données chiffrées tout en assurant que le résultat soit correctement déchiffrable.
On peut ainsi distinguer deux types de chiffrements homomorphes selon les calculs pouvant être effectués : - Les chiffrement homomorphes partiels, qui désignent l'ensemble des chiffrements homomorphes valides pour une seule opération (addition ou multiplication) - Les chiffrements homomorphes complets ou chiffrements entièrement homomorphes (FHE), qui désignent l'ensemble des chiffrements homomorphes valides pour l'addition *et* la multiplication d'entiers

Un chiffrement homomorphe complet est donc plus fort qu'un chiffrement homomorphe partiel, car la complétude d'une fonction d'évaluation implique ainsi sa partialité. Bien qu'on puisse penser qu'une multiplication d'entiers est une simple addition successive, effectuer cette méthode n'est pas viable lorsqu'il s'agit de grands nombres. En pratique ce sont donc les fonctions d'évaluation complètes qui sont utilisées dans la plupart des cas.

/! Il est bon de noter que nous ne désignons que des entiers dans le chiffrement homomorphe partiel. Lorsque des fonctions d'évaluation sont utilisées pour du machine learning, il est impératif que celles-ci soient complètes dans un premier temps, et capables d'opérations sur des flottants dans un second temps. Nous parlerons plus tard de ce cas de figure d'opérations sur des flottants et considérerons des entiers pour la suite.

Pour illustrer le propos sur ces deux catégories de chiffrements homomorphes, nous allons prendre un exemple de chiffrement partiel et de chiffrement complet et montrer leur fonctionnement, en cryptant ensemble deux messages et en effectuant des opérations sur eux.

Le chiffrement partiel que nous allons étudier en exemple est le chiffrement Pailler. Le chiffrement complet sera le chiffrement BGV (Brakerski-Gentry-Vaikuntanathan).

Considérons tout d'abord la fonction homomorphe partielle. La chiffrement que nous avons choisi ici s'appelle le chiffrement Paillier.

Le chiffrement Paillier

Le chiffrement Paillier est un exemple de schéma de chiffrement homomorphe partiel qui permet uniquement d'effectuer des additions sur des nombres entiers. C'est l'un des systèmes de chiffrement homomorphe les plus populaires ; c'est un chiffrement à clé publique et basé sur la difficulté du problème du logarithme discret dans les groupes multiplicatifs.

Voici son fonctionnement :

Comme les autres chiffrement homomorphes, Pailler repose sur un chiffrement à clé publique/clé privée. La clé publique permet de chiffrer les données, et la clé privée est utilisée pour les déchiffrer. Nous allons aborder étapt par étape de façon détaillée les méthodes sur lesquelles reposent ce chiffrement.

1. Génération des clés

Pour la génération de clé, on peut choisir deux grands nombres premiers distincts p et q. On calcule n=p×q (le module, qui est utilisé dans le processus de chiffrement). On calcule λ = PPCM(p-1,q-1), où PPCM est le plus petit multiple commun. On choisit un entier dans Znˆ2, tel que g ait un ordre dans le groupe Zn *(Cela signifie qu'il doit être un générateur du groupe multiplicatif Zn*, ou avoir des propriétés similaires). En d'autres termes, on doit avoir g premier avec n^2 et PPCM(L(gˆλ mod(nˆ2)),n)=1. Choisir g=n+1 est généralement privilégié. La clé publique est constituée des valeurs (n,g), et la clé privée est λ, qui est utilisée pour le déchiffrement.

2. Chiffrement des messages

Supposons que nous voulons chiffrer un message m (un entier compris entre 0 et n). On choisit un nombre aléatoire r tel que r∈Zn*, c'est-à-dire r doit être un entier entre 1 et n-1, et r doit être premier avec n. On calcule ensuite le chiffrement de m avec la formule suivante : $E(m)=g^{m\times r}n \bmod(n\hat{}2)$

Le message m est donc chiffré en produisant deux composantes : gˆm et rˆn. La valeur chiffrée E(m) est un entier qui représente le message de manière secrète. Cette valeur peut être envoyée à un serveur sans que celui-ci ne puisse connaître le message réel m.

3. Opérations sur les messages chiffrés (Addition)

L'une des caractéristiques principales de Paillier est son additionnalité homomorphe comme nous l'avons vu. Si nous avons deux messages chiffrés, E(m1) et E(m2), nous pouvons effectuer une opération d'addition sur ces messages chiffrés sans jamais les déchiffrer. Supposons que nous ayons deux messages m1 et m2, avec leurs versions chiffrées respectives E(m1) et E(m2). La propriété homomorphe additive du chiffrement Paillier nous permet de ajouter les messages chiffrés : E(m1)*E(m2)=E(m1+m2) mod(nˆ2) L'addition des deux valeurs chiffrées donne une nouvelle valeur chiffrée qui représente la somme des deux messages en clair. Ce résultat peut ensuite être déchiffré pour obtenir la somme réelle m1+m2.

4. Déchiffrement du message

Pour déchiffrer un message c chiffré, on utilise la clé privée. L'étape de déchiffrement fonctionne de la manière suivante :

- On calcule L(cˆ λ mod(nˆ2), où L(x) est la fonction définie par : L(x)=(x-1)/n Soit : L(cˆλ mod(nˆ2)) = (cˆλ mod(nˆ2)-1)/n
- On effectue un calcul similaire avec le générateur g, connu publiquement : L(gˆλ mod($n^{2)}$=(($^{g}\lambda$ mod(nˆ2)) -1)/n

Puis on calcule son inverse modulo n : μ=1/(L(gˆλ mod(nˆ2))) mod(n)

Le message clair m est obtenu en multipliant les deux résultats précédents modulo n : m= (L(cˆλ mod(nˆ2))*μ) mod(n)

Soit la formule finale générale : m = ((cˆλ mod(nˆ2)-1)/n * 1/((gˆλ mod(nˆ2) -1) /n)) mod(n)

Cela permet d'extraire le message m = m1 +m2 en clair à partir de la version chiffrée c.

Exemple concret avec Paillier

Prenons maintenant un exemple simple pour illustrer le processus de chiffrement et d'addition avec Paillier.

1. Génération des clés On choisit p=7 et q=11 (exemple simple). On calcule n=p×q=7×11=77. On calcule λ=PPCM(7-1,11-1)=PPCM(6,10)=30. On choisit g=78 = n+1 comme générateur de façon arbitraire (il correspond car = premier avec $n^2 = 77^2$ et PPCM(L(gˆλ mod(nˆ2)),n)=1).

2. Chiffrement du message

- On chiffre m1=3 avec un r=5 (choisi aléatoirement) : E(3) = $78^{3\times5}77$ mod(77ˆ2) = 2390
- On chiffre m2=5 avec un r'=8 : E(5)=$78^{5\times8}77$ mod(77ˆ2) = 1366

3. Addition des messages chiffrés

On additionne les deux valeurs chiffrées : E(3)*E(5) = E(3+5) = E(8) Or E(3)*E(5) = E(3) * E(5) mod(n^2) (C'est notre opération *) = 2390*1366 mod(77^2) = 3790 = c Donc E(8) = c = 3790

4. Déchiffrement du message:

Calculs intermédiaires : - cˆλ mod(nˆ2) = 3790ˆ30 mod(77^2) = 694 - L(cˆλ) = 77694-1 = 9 - gˆλ mod(nˆ2) = 2311 - L(gˆλ) = 772311-1 = 30 - Inverse modulaire de 30 mod(77) = 18

D'où : -> m=L(cˆλ) * 1/L(gˆλ) mod(n)=9*18 mod(77) = 8 = 5+3.

Le chiffrement BGV

Passons maintenant à un exemple de fonction homomorphe complète : le chiffrement BGV.

Etant un chiffrement complet, BGV permet non seulement des additions, mais aussi des multiplications sur les données chiffrées, comme nous l'avons vu tout à l'heure. Il repose sur la difficulté d'un problème mathématique moderne appelé Learning With Errors (LWE) ou sa version à structure algébrique Ring-LWE. Sans rentrer davantage dans les détails, il fonctionne sur des polynômes dans des anneaux, pour les plus intéressés.

Voici les grandes étapes de son fonctionnement :

1. **Génération des clés** On fixe des paramètres : n : degré du polynôme (doit être une puissance de 2 pour l'efficacité). q : un grand entier premier (modulo). Il faut qu'il soit grand, sinon on a des débordements et le résultat final est biaisé. (Grand = Pour une sécurité classique de 128 bits, les implémentations modernes recommandent pour BGV q≈ 2ˆ50 à 2ˆ200, selon la profondeur des circuits). f(x) : un polynôme cyclotomique, généralement de la forme f:x->xˆn + 1.

On choisit :

- Une clé secrète s(x)∈ R_q, petit polynôme aléatoire (petit = de degré <= n et des coeffs petits, généralement dans {-1;0;1}. On veut en effet garder le produit s*r peu bruyant).
- Une clé publique pk=(a(x),b(x)) constituée de a(x)∈ R_q (aléatoire) et : b(x) = -a(x)*s(x) + e(x) mod(q)

où e(x) est un petit bruit (petit <=> |e(x)*r(x)|<=δ).

La clé publique est donc : pk=(a(x),b(x)), et la clé secrète est s(x).

2. **Encodage et chiffrement**

Pour chiffrer un message m(x)∈ R_q (avec petits coefficients) :

- On commence par encoder m avec un facteur d'échelle δ={pipe_bas}q/t {pipe_bas}tel qu'on ai m'(x)=m(x)δ. *L'objectif est que m'»e*r, pour assurer que le bruit ne perturbe pas le message lors du déchiffrement.
- On prend un petit vecteur aléatoire r(x)∈ R_q
- On calcule : c_0(x)=b(x)*r(x)+m'(x) mod(q) c_1(x)=a(x)*r(x) mod(q)

Le chiffré est donc : c(x)=(c_0(x),c_1(x))

3. **Calculs sur les messages chiffrés**

Additions : - (c_0,c_1) + (c_0',c_1') = (c_0 + c_0', c_1 + c_1')

Multiplications : - Le produit nécessite une étape supplémentaire (appelée relinearization) car le degré du chiffré augmente. La relinearisation utilise des clés de rélinearisation (générées à partir de la clé secrète) pour ramener le chiffré à deux composantes tout en préservant l'homomorphisme. Le schéma BGV applique un

traitement pour revenir à une forme standard à 2 composantes. Concrètement : Quand on multiplie deux chiffrés : $c^{(1)=(c\_0}(1)$,c_$1^{(1)),c}(2)=(c_0^{(2),c_1}(2))$

On obtient : $c^{(1)c^{(2)=(c_0}(1)}$c_0(2), c_$0^{(1)c_1^{(2)+c_1}(1)}$c_0(2), c_1ˆ(1)*c_1ˆ(2))

→ C'est un triplet, donc on sort du format à 2 composantes. → Il faut "relineariser" pour revenir à un format à deux composantes.

4. **Déchiffrement**

- Le détenteur de la clé privée peut supprimer le bruit et extraire le message clair à partir du polynôme :

Si c(x)=(c0,c1), alors le message clair est obtenu par : m'(x)=c_0(x)+c_1(x)*s(x) mod(q) m(x) ={pipe_bas}m'(x)/δ [pipe]

Sous réserve que le bruit ne soit pas trop grand, ce message est exact. (Remarque : le bruit augmente à chaque opération homomorphe, surtout les multiplications. Le schéma BGV inclut donc des techniques comme le modulus switching pour maintenir le bruit dans des bornes correctes tout au long du calcul. Nous n'aborderons pas cela ici.)

Application du chiffrement BGV

Paramètre	Valeur	Justification
t	64	Message clair : on veut 4, 5, 9, 20 $\in$ [0, 63]
q	65537	Grand modulo premier, $q > \Delta^2$
Δ	{pipe_bas}q / t{pipe_bas} = 1024	Facteur d'échelle, assure que $m' \gg bruit$
s	1	Clé secrète simplifiée
a	1234	Échantillon aléatoire de $\mathbb{Z}_q$
e	1	Bruit minimal pour garantir exactitude
r	1	Aléa petit et constant pour simplicité

1. Génération de clés On calcule : b=-a*s+e=-1234*1+1=-1233 mod(65537)=64304 -> Clé publique : pk=(a(x)=12345,b(x)=20424) -> Clé secrète : s(x)=1
2. Chiffrement (on veut faire 4+5) Encodage avec δ=1024 : m_1=4*Rightarrow*m_1'=4*1024 = 4096 m_2=5Rightarrowm_2'=5*1024 = 5120

Chiffrement Rappel : c_0=b*r+m',c_1=a*r

Pour m_1 : c_0ˆ(1) = 64304+4096 = 68400 mod(65537) = 2863 c_1ˆ(1) = 1234

Pour m_2 : c_0ˆ(2) = 64304+5120 = 69424 mod(65537) = 3887 c_1ˆ(2)=1234

3. Addition : (c_$0^{(1)+c_0}$(2), c_$1^{(1)+c_1}$(2))=(2863+3887, 1234+1234)=(6750, 2468)

Déchiffrement : m'ˆ+=c_0+c_1*s = 6750+2468 = 9218 mod(65537) m={pipe_bas}9218/1024[pipe]={pipe_bas}9.002[pipe]=9

4. Multiplication

Produit brut (avant relinearisation) : Formule du produit: c_0ˆ× = c_0ˆ(1)*c_0ˆ(2) = 2863*3887 = 11128481 mod(65537) = 52728 c1ˆ× = c_$0^{(1)c_1^{(2)+c_1}(1)}$c_0(2) = 2863*1234 + 1234*3887 = 3532942 + 4796558 = 8329500 mod(65537) = 6301 c2ˆ ×= c_1ˆ(1)*c_1ˆ(2) = 1234*1234 = 1522756 mod(65537) = 15405

Triplet : (c_0,c_1,c_2)=(52728, 6301, 15405)

Relinearisation (simplifiée ici) : On simule que c_2*s est injecté dans c_0 c_0'=c_0+c_2*s = 52728+15405 = 68133 mod(65537) = 2596 c_1'= c_1 = 6301

5. Déchiffrement final

m'ˆ× = c_0'+c_1'*s = 2596+6301=8897 mod(65537) = 8897

On utilise : $m = pipe_b asm'Ö\delta^2[pipe] = pipe_b as8897/1024^2[pipe] = 0$

:(ça marche pas.... Pourquoi ?

Toute l'info a été perdue dans la réduction modulo q car le produit chiffré a dépassé q. Il faut m_1$m_2\ \delta$ ^2 < < q.

Trouver un q minimal, c'est le noise budget (ici on n'aborde pas ça).

m_1'm_2'=$4096$5120=20971520 plus grand que 65537

En refaisant avec q = 2^36 = 68719476736 :

b = -as+e = -$1234$1+1 = -1233 mod(68719476736) = 68719475503 -> Clé publique : pk=(a(x)=12345,b(x)=68719475503) -> Clé secrète : s(x)=1

Rappel : c_0=br+m',c_1=ar

Pour m_1 : c_0^(1) = 68719475503+4096 = 68719479599 mod(68719476736) = 2863 c_1^(1) = 1234

Pour m_2 : c_0^(2) = 68719475503+5120 = 68719480623 mod(68719476736) = 3887 c_1^(2)=1234

Et la multiplication

Produit brut (avant relinearisation) : Formule du produit: c_0^× = c_0^(1)c_0^(2) = $2863$3887 = 11128481 mod(68719476736) = 11128481 c_1^× = c_0$^{(1)c_1^{(2)}+c_1^{(1)}}$c_0(2) = 2863$1234$ + $1234$3887 = 3532942 + 4796558 = 8329500 mod(68719476736) = 8329500 c_2^×= c_1^(1)c_1^(2) = $1234$1234 = 1522756 mod(68719476736) = 1522756

Triplet : (c_0,c_1,c_2)=(11128481, 8329500, 1522756)

Relinearisation (simplifiée ici) : On simule que c_2s *est injecté dans* c_0 c_0'=c_0+c_2s = 11128481+1522756 = 12651237 mod(68719476736) = 12651237 c_1'= c_1 = 8329500

Déchiffrement final :

m'^× = c_0'+c_1'*s = 12651237+8329500 = 20980737 mod(68719476736) = 20980737

On utilise : $m = pipe_bas m'Ö\delta^2[pipe] = pipe_bas20980737/1024^2[pipe] = 20$

Ça fonctionne !