université de BORDEAUX

RAPPORT DE TER

Protocole d'interrogation anonyme de base de données

CHARBONNIER ELOUAN, MONDIN ESTELLE,
RAPHEL ELSA

Master CSI 2023

Enseignant : CASTAGNOS Guilhem

Établissement : UNIVERSITÉ DE BORDEAUX

I Introduction

Au cours de ce projet, nous souhaitons sécuriser les échanges entre une base de données et un client par le biais d'un protocole cryptographique particulier.

Une base de données est un conteneur informatique pouvant stocker un grand nombre d'informations, facile d'accès, à gérer et mettre à jour par un serveur. Le serveur est un dispositif souvent informatique en capacité de manipuler la base de donnée.

Outils très intéressants, elles sont notamment utilisées dans de nombreux secteurs comme la finance, l'administration publique ou le secteur médical. Manipulant des données sensibles, il est nécessaire de protéger l'accès des utilisateurs afin de contrôler les actions et échanges d'informations.

Lors d'un échange entre le serveur et un client, celui-ci souhaite récupérer un champ de la base de données qui l'intéresse, c'est-à-dire une information à une position particulière. On peut le voir comme une position dans un tableau. On suppose que le client connaît la position de l'information qui l'intéresse dans la base de données.

Le client, après identification de l'élément intéressant, envoie une requête au serveur pour le récupérer. Celui-ci renvoie en retour ce qu'on lui a demandé.

Dans certains cas, par souci de sécurité, le client souhaite ne pas divulguer la position ni l'élément consulté, y compris au serveur. En effet, les bases de données sont sujettes à de nombreuses menaces de sécurité comme l'élévation de privilège, injection SQL, déni de service, etc [1]. Il est donc nécessaire de chiffrer l'information que le client demande pour ne pas être reconnaissable par le serveur.

Ainsi, si le serveur ignore ce qu'il transmet, il sera difficile à un attaquant de retrouver l'information demandée par notre client, même s'il a accès à la base de données.

Afin de respecter cet anonymat de requête, le protocole cryptologique PIR (Private Information Retrieval) fût mis en place. Nous implémenterons ce protocole et l'exécuterons sur une base de données.

La base de données utilisée sera un tableau à 1 dimension constitué de 64 éléments de 1024 bits. Chacun de ces éléments sont générés aléatoirement. Nous implémenterons ce programme en langage C.

Afin de manipuler des entiers de grande taille, nous utiliserons la librairie gratuite GMP [2]. Les principales applications cibles de GMP sont la recherche en cryptographie, les applications de sécurité Internet ou encore la recherche en algèbre computationnelle.

GMP est conçu pour être aussi rapide que possible malgré la manipulation de grands nombres ou la complexité des calculs demandés (exponentiation rapide). On obtient cette vitesse grâce à des algorithmes rapides et un code assembleur hautement optimisé, utilisé pour les boucles internes les plus courantes de nombreux processeurs.

Table des matières

II Cryptosystème de Paillier

Pour la bonne mise en œuvre du protocole, nous devons garantir la sécurité des éléments chiffrés. Pour cela, nous utilisons un problème mathématique reconnu difficile, la factorisation des grands nombres premiers [3], jusqu'à présent impossible en temps raisonnable pour de grand nombre arbitraires de 1024 bits et plus. De nombreux protocoles de chiffrement se basent sur ce problème comme le chiffrement RSA largement utilisé de nos jours pour sécuriser les transactions bancaires. En effet, il est facile de multiplier des nombres, cependant il est plus difficile de faire l'inverse, c'est-à-dire partir d'un grand nombre et de retrouver ses diviseurs premiers. Le cryptosystème de Paillier utilise ce problème afin de garantir la sécurité des informations chiffrées.

Le cryptosystème de Paillier se base sur un algorithme asymétrique amélioré, c'est-à-dire un chiffrement utilisant une clé publique et un déchiffrement utilisant une clé privée.

Le chiffrement asymétrique, nous assure la confidentialité du message, seulement le destinataire possède la clé privée pour lire le message clair. La clé privée n'est pas partagée entre les destinataires. Pour ce qui est de la rapidité, nous trouverons un moyen de l'optimiser dans notre implémentation du protocole par l'utilisation de plusieurs dimensions pour la base de données.

Le cryptosystème se fait en 3 étapes :

— la génération des clés

— le chiffrement

— le déchiffrement

II.1 Génération des clés

Nous utiliserons trois clés. Deux clés publiques N, g et une clé privée λ. Comme expliqué dans l'introduction de cette partie, l'utilisation du problème de la factorisation de grands nombres sera utilisé pour la génération des clés. Nous sélectionnons ainsi deux grands nombres premiers, indépendants et aléatoires p et q. Lorsqu'on parle de grands nombres entiers, cela signifie au-dessus de 250 chiffres, sinon le système ne serait plus sécurisé, les ordinateurs modernes étant capables de les retrouver en quelques minutes.[3] Afin de manipuler de grand nombre de bits, nous utiliserons la librairie GMP et générerons des clés de 1024 bits.

La clé publique, visible par tous, sera $N = p * q$ et la clé privée sera le produit de $p - 1$ et $q - 1$, on nommera cette clé privée λ :

$$\lambda = (p - 1) \cdot (q - 1). \tag{1}$$

La fonction de génération des clés, que nous avons appelé `key_gen()`, contient trois parties, la première consiste à initialiser la graine aléatoire et les variables qui vont être utilisées, c'est-à-dire p, q, N, g et λ. On initialise aussi N^2 afin de ne pas avoir à l'initialiser dans toutes les fonctions.

```
void key_gen(mpz_t public_key_N, mpz_t public_key_g, mpz_t private_key,
             mpz_t public_key_N2) {

  do_seed();
  mpz_t p, q, lamb, N, N2, g, tmp, p2, q2;
  mpz_inits(p, q, lamb, N, N2, g, tmp, p2, q2, NULL);
```

Dans la seconde partie, nous calculons tous les paramètres dont nous avons besoin et on place leurs valeurs dans les arguments de la fonction.

```
  // Generation of p and q, great prime numbers with SIZE bits
  mpz_urandomb(p, state, SIZE - 1);
  mpz_urandomb(q, state, SIZE - 1);
  mpz_nextprime(p, p);
  mpz_nextprime(q, q);

  // Generation of the public key N
  mpz_mul(N, p, q);

  //  Generation of the private key lambda
  mpz_sub_ui(p2, p, 1);
  mpz_sub_ui(q2, q, 1);
  mpz_mul(lamb, p2, q2);

  // Compute of N^2
  mpz_pow_ui(N2, N, 2);

  mpz_add_ui(g, N, 1);

  mpz_set(public_key_g, g);
  mpz_set(public_key_N, N);
  mpz_set(private_key, lamb);
  mpz_set(public_key_N2, N2);
```

Enfin, dans la troisième partie, nous libérons la mémoire des variables temporaires que nous avons créés afin de simplifier le code.

```
  mpz_clears(p, q, lamb, N, N2, g, tmp, p2, q2, NULL);
}
```

II.2 Chiffrement

On pose m le message à chiffrer et r un entier aléatoire compris strictement entre 0 et N, qu'on appellera l'aléa, on suit la procédure du chiffrement de Paillier qui nous donne le chiffré c, on note ϵ cette fonction de chiffrement :

$$\begin{array}{rcccc} \epsilon & : & (\mathbb{Z}\backslash N\mathbb{Z}, \times) \times (\mathbb{Z}\backslash N\mathbb{Z}^*, \times) & \mapsto & (\mathbb{Z}\backslash N^2\mathbb{Z}^*, \times) \\ & & (m, r) & \mapsto & (1+N)^m \cdot r^N \end{array} \tag{2}$$

Dans notre cas, nous allons noter $g = N + 1$.

Cependant, nous choisissons plutôt d'utiliser cette fonction de chiffrement qui va grandement simplifier nos calculs :

$$\begin{array}{rccc} \epsilon & : & (\mathbb{Z}\backslash N\mathbb{Z}, \times) \times (\mathbb{Z}\backslash N\mathbb{Z}^*, \times) & \mapsto & (\mathbb{Z}\backslash N^2\mathbb{Z}^*, \times) \\ & & (m, r) & \mapsto & (1 + N \cdot m) \cdot r^N \end{array} \tag{3}$$

Montrons pourquoi nous pouvons affirmer que $(1+N)^m \equiv 1 + N \cdot m \mod N^2$:

Preuve : *Tout d'abord, on rappelle la Formule du Binôme de Newton :* $(a+b)^n = \sum_{k=0}^{n} \binom{n}{k} a^{n-k} b^k$. *On applique cette formule à notre cas et on a :*

$$\begin{aligned} (1+N)^m &= \sum_{k=0}^{m} \binom{m}{k} 1^{m-k} N^k \\ &= \sum_{k=0}^{m} \binom{m}{k} N^k \\ &= \binom{m}{0} \cdot 1 + \binom{m}{1} N + \binom{m}{k} N^2 + \cdots + \binom{m}{m} N^m \\ &= 1 + m \cdot N + (\frac{m(m-1)}{2}) N^2 + \cdots + N^m \end{aligned}$$

En faisant la congruence modulo N^2*, on a,*

$$= 1 + m \cdot N \mod N^2$$

Donc, on a bien $(1+N)^m \equiv 1 + N \cdot m \mod N^2$.

La fonction de chiffrement de Paillier que nous avons implémenté, nommée `encrypt()`, est la suivante :

```
1  void encrypt(mpz_t m, mpz_t public_key_N, mpz_t c, mpz_t public_key_N2) {
2
3  // Initialization of variables
4    mpz_t s, r;
5    mpz_inits(s, r, NULL);
6
7  // Generation of the random with at most SIZE bits
8    mpz_rrandomb(r, state, 2 * SIZE - 1);
9
10 // Encrypting m by following the algorithm
11   mpz_mul(c, m, public_key_N);
12   mpz_add_ui(c, c, 1);
13   mpz_powm(r, r, public_key_N, public_key_N2);
14   mpz_mul(c, c, r);
15   mpz_mod(c, c, public_key_N2);
16
17 // Clearing all temporary variables
18   mpz_clears(r, s, NULL);
19 }
```

II.3 Déchiffrement

Pour déchiffrer un message chiffré c reçu, il faut tout d'abord définir la fonction L :

$$L \ : \ x \ \mapsto \ \tfrac{x-1}{N}$$

Par la suite, on doit appliquer la fonction de déchiffrement suivante :

$$\begin{array}{rccc} D & : & (\mathbb{Z}\backslash N^2\mathbb{Z}^*, \times) & \mapsto & (\mathbb{Z}\backslash N\mathbb{Z}, +) \\ & & c & \mapsto & L(c^\lambda \mod N^2) \cdot \mu \end{array} \tag{4}$$

Où $\mu = (L(g^\lambda \mod N^2))^{-1} \mod N$ et on note $m = D(c)$, on rappelle que $g = N + 1$.

Nous avons légèrement changé la procédure, en effet, on s'est rendu compte que $\mu = (L(g^\lambda \mod N^2))^{-1} \mod N$ est équivalent à prendre $\mu = \lambda^{-1} \mod N$, comme démontré dans la preuve suivante[4].

Nous allons maintenant prouver que notre fonction de déchiffrement nous permet bien de retrouver le message en clair :

Preuve : *Soit c un chiffré de m avec l'aléa r, on a :*

$$c^\lambda = c^{\phi(N)}$$

Où $\phi(N) = (p-1) \cdot (q-1)$ *est l'indicatrice d'Euler de N.*

$$= g^{m \cdot \phi(N)} \cdot r^{N \cdot \phi(N)}$$

Or $N \cdot \phi(N) = \phi(N^2)$

$$= g^{m \cdot \phi(N)} \cdot r^{\phi(N^2)}$$

Or Deplus, $r^{\phi(N^2)} \equiv 1 \mod N^2$ *d'après le théorème d'Euler*

$$\begin{aligned} &\equiv g^{m \cdot \phi(N)} \mod N^2 \\ &\equiv 1 + (m \cdot \phi(N)) \cdot N \mod N^2 \end{aligned}$$

On représente $c^\lambda \mod N^2$ *par un entier x.*
On a $x = a + bN$ *la décomposition en base N de x. Dans notre cas,* $a = 1$ *et* $b \equiv m \cdot \phi(N)$*. Ainsi, on a :*

$$\begin{aligned} L(g^{m \cdot \phi(N)}) &= \frac{g^{m \cdot \phi(N)} - 1}{N} \\ &= m \cdot \phi(N) \mod N \end{aligned}$$

Ici, on peut voir qu'il suffit d'utiliser l'inverse de la clé privée $\lambda = \phi(N)$ pour retrouver le message. Il suffit de calculer $b \cdot \phi(N)^{-1}$ pour retrouver m.

La fonction `decrypt()` correspond à la procédure de déchiffrement décrite ci-dessus :

```
1  void decrypt(mpz_t c, mpz_t private_key, mpz_t m2, mpz_t public_key_N,
2               mpz_t public_key_N2) {
3
4    // Initialization of variables
5    mpz_t mu;
6    mpz_init(mu);
7
8    // L(c^lambda mod N^2)
9    mpz_powm(c, c, private_key, public_key_N2);
10   mpz_sub_ui(c, c, 1);
11   mpz_div(c, c, public_key_N);
12
13   // Generation of mu
14   mpz_invert(mu, private_key, public_key_N);
15
16   // Multiplication of c by mu, modulo N
17   mpz_mul(c, c, mu);
18   mpz_mod(m2, c, public_key_N);
19
20 // Clearing all temporary variables
21   mpz_clear(mu);
22 }
```

II.4 Homomorphisme

Le cryptosystème de Paillier comporte certaines propriétés qui vont particulièrement nous intéresser dans la suite. En effet, le cryptosystème de Pailler est ce qu'on appelle un homomorphisme additif, cela correspond, en partie, à la propriété ci-dessous qui va nous permettre de déchiffrer le produit de deux messages chiffrés :

Soient c_1 et c_2 les deux chiffrés, respectivement, de m_1 et m_2, on a :

$$\begin{aligned} c_1 \cdot c_2 &= (1+N)^{m_1} {r_1}^N \cdot (1+N)^{m_2} {r_2}^N \mod N^2 \\ &= (1+N)^{m_1+m_2} (r_1 \cdot r_2)^N \mod N^2 \end{aligned}$$

Ainsi $D(c_1 \cdot c_2) = m_1 + m_2$.
Où N correspond à la clé publique, r_1, r_2 à deux aléas utilisés dans le chiffrement et D la fonction de déchiffrement.

Donc $c_1 \cdot c_2$ est le chiffré du message $m_1 + m_2$ avec l'aléa $r_1 + r_2$. Comme on le verra plus tard, lorsque nous allons chiffrer des bases de données entières, cette propriété va nous aider à rendre anonyme la requête du client, mais aussi à masquer la réponse du serveur.

La propriété suivante va nous permettre d'implémenter un chiffrement qui respecte l'anonymat du serveur ainsi que du client :

Soient c un chiffré de m, $k \in \mathbb{Z}/N\mathbb{Z}$ et D la fonction de déchiffrement, on a :

$$D(c^k \mod N^2) = k \cdot m \mod N$$

En déchiffrant c^k, nous allons obtenir le message en clair $k \cdot m$. Cette propriété va être très importante lorsqu'on implémentera PIR.

II.5 Lien avec PIR

Le protocole PIR, afin d'anonymiser les échanges des éléments de la base entre le serveur et le client, utilise, met en place un système de chiffrement homomorphe additif de Paillier.

Car ainsi, les chiffrés sont indistinguables, c'est-à-dire qu'il n'est pas possible de distinguer les chiffrés des messages clairs eux-mêmes distincts.

Il est également possible de remélanger un chiffré en rajoutant un chiffrement de zéro au chiffré existant ($2^{ème}$ propriété de l'homomorphisme). Cette propriété de re-randomisation est importante dans de nombreuses constructions visant à préserver la vie privée, étant donné qu'elle rend intraçable un message ainsi remélangé.

III Protocole PIR

La sécurité du cryptosystème de Paillier démontré, nous allons donc l'utiliser dans le protocole PIR (Private Information Retrieval)[5].

Le protocole s'exécute en 3 étapes :

— le chiffrement des indices de la base de données par le client afin d'indiquer anonymement son choix d'élément. (*request_client()*)

— le chiffrement du message par la base de données avec les indices chiffrés et son renvoi au client (*server_return()*)

— le déchiffrement du message par le client (*decrypt()*)

Le protocole d'échange de requête s'effectue sans que la base de données ne sache la demande explicite du client. Dans l'introduction, nous initialisons la base de données par un tableau de 64 éléments, nous ferons varier sa taille lors de nos expériences. La structure de la base de données sera toujours la même, un tableau d'1 dimension.

Cependant, le client peut choisir de quelle manière il souhaite voir la base de données et ces éléments. Cela influencera sa manière de retourner le tableau d'indice chiffré. Comme un seul tableau de la taille de la base de donnée, à l'image de la structure de la base de données ou en plusieurs tableaux représentant des coordonnées.

Pour l'instant, le client considère la base de données en 1 dimension.

III.1 Requête du client

Le client demande un certain indice de la base de donnée au serveur sans le révéler de façon claire. La première partie du protocole est de chiffrer l'indice de l'élément qui nous intéresse. Pour indiquer cet élément au serveur, nous lui envoyons un tableau de la taille de sa base de donnée en 1D.

À l'intérieur de ce tableau, des zéros et un 1 chiffrés, la position du 1 dans le tableau correspond à la position du message voulu dans la base de donnée.

d_0	d_1	d_2	.....	d_k	d_l	d_q	d_p

(a) Détermination de l'indice

0	0	0	.....	0	0	1	0

(b) Marquage de l'indice voulut

216..54	156..15	321..51	.....	754..35	024..51	147..37	452..73

(c) Chiffrement

FIGURE 1 – Chiffrement du tableau d'indice

Afin de ne pas connaître l'indice en question, ces 0 et 1 sont chiffrés grâce à la fonction de chiffrement de Paillier. Notamment, chaque élément du tableau envoyé est chiffré par un aléa différent, donc tous les 0 sont chiffrés différemment. Pour chaque indice, on appelle le chiffrement de Paillier, ceci afin de ne pas trouver de corrélation entre les indices chiffrés, notamment les zéros et ainsi retrouver l'indice 1. Pour ce faire, nous initialisons une nouvelle seed par l'appel à la fonction `do_seed()`.

```
1  void prng_init(unsigned int seed) { srand(seed); }
2
3  void do_seed() {
4    unsigned int seed;
5    gmp_randinit_mt(state);
6
7    prng_init(time(NULL));
8    gmp_randseed_ui(state, seed);
9  }
10
```

Après chiffrement, les éléments ne ressemblent plus à des 0 ou 1, mais sont des entiers aléatoires de 1024 bits impossibles à déchiffrer sans la clé secrète du client qui est `public_key_N` et `public_key_N2`.

Suite aux descriptions ci-dessus, notre implémentation de la première partie du protocole PIR est :

```
1  mpz_t *request_client(int index, mpz_t public_key_N, mpz_t m, mpz_t c,
2                        mpz_t public_key_N2, int size) {
3    clean_seed(state); // cleen the previous seed in the main
4    do_seed();
5    mpz_t *request = calloc(size, sizeof(mpz_t));
6
7    for (int i = 0; i < size; i++) {
8      mpz_set_ui(m, (i == index) ? 1 : 0);
9      encrypt(m, public_key_N, c, public_key_N2);
10     mpz_set(request[i], c);
11   }
12
13   return request;
14 }
```

Le tableau renvoyé, ces indices chiffrés permettent d'anonymiser la requête. Le serveur ne sait pas à quel indice se trouve le message souhaité par le client. Comme exprimé auparavant, la clé privée doit être assez grande pour garantir la sécurité du chiffrement de Paillier, ici, c'est notre cas avec une taille de 1024 bits.

III.2 Retour du serveur

Le serveur reçoit le tableau d'indices chiffrés, désormais, il doit renvoyer l'élément demandé. Le serveur a connaissance des éléments clairs dans sa base de données, mais ne connaît pas l'indice clairement souhaité par le client, car il est chiffré. Il va utiliser une

propriété du cryptosystème de Pailler pour partager le message clair sans connaître lequel, grâce à l'homomorphisme additif ($2^{ème}$ propriété de l'homomorphisme).

Le serveur a pour chacun de ses messages clairs m dans sa base, un indice chiffré lui venant du client, il lui renvoie un nombre β de taille modulo N^2, tel que :

$$\beta = \prod_{i=0}^{taille\_bdd-1} c_i^{m_i} \tag{5}$$

où c_i correspond aux indices chiffrés et m_i les messages clairs de la base de données.

Nous traduisons cette formule dans notre implémentation par :

```
1   void server_return(mpz_t *request, mpz_t response, mpz_t *c,
2                      mpz_t public_key_N2, int size) {
3
4   mpz_t tmp;
5   mpz_inits(tmp, NULL);
6   mpz_set_ui(response, 1);
7
8   for (int i = 0; i < size; i++) {
9     mpz_powm(tmp, request[i], c[i], public_key_N2);
10    mpz_mul(response, tmp, response);
11    mpz_mod(response, response, public_key_N2);
12  }
13
14  mpz_clear(tmp);
15  return;
16 }
```

Par l'homomorphisme additif, le client retrouvera facilement le m clair qu'il souhaite. L'information est bien conservée par le produit grâce à l'homomorphisme additif. Soit D la fonction de déchiffrement de Paillier, on a :

$$D(c_1^{m_1} \cdot c_2^{m_2}) = k_1 \cdot m_1 + k_2 \cdot m_2 \tag{6}$$

III.3 Déchiffrement

Le client reçoit β de taille modulo N^2, pour obtenir le message clair qu'on lui a envoyé, il déchiffre β grâce à la fonction de déchiffrement de Paillier.

```
1       decrypt(response, private_key, m2, public_key_N, public_key_N2);
```

Possédant les clés privées du chiffrement, le client va décoder β :

$$D(\beta) = \sum_{i=0}^{taille\_bdd} D(c_i) \cdot m_i \tag{7}$$

par propriété d'homomorphisme additif.

Seul le message m_{index} avec D(c_i)= 1 sera rendu. Les messages m_i du serveur ayant été envoyés en clairs, le client recevra ce qu'il attendait :

$$m_i \cdot 0 + m_i \cdot 0 + ...(m_{index}) \cdot 1 + m_i \cdot 0 + ... + m_i \cdot 0 = m_{index} \tag{8}$$

Nous retrouvons en clair l'élément que nous désirions.

III.4 Expériences

Afin de tester notre protocole, nous commençons nos expériences avec une base de taille 64 avec des entiers de 1024 bits. Le client demande à obtenir la dernière case de la base de données.

Dans la réalité des bases de données, les informations qu'elle contient, peuvent et sont souvent très lourdes comme des chaînes de caractères. Dans notre cas, nous nous contenterons d'entiers de grande taille.

Pour challenger notre programme, nous augmentons la taille de la base de données et des clés. Nous obtenons les temps suivants :

Taille de BDD \ Taille p , q (bits)	**512**	**1024**	**2048**	**4096**
64	0.273	1.849	12.103	72.718
128	0.516	3.624	23.256	136.197
256	1.024	7.318	45.948	264.570
512	1.966	14.499	90.721	518.306
729	2.298	8.996	128.936	733.333
4096	15.576	114.743	720.847	4082.819

FIGURE 2 – Temps d'exécution du protocole PIR 1D (en s)

Nous remarquons que plus les clés et la taille de la base de données sont grandes, plus le temps augmente.

IV Optimisation de PIR

IV.1 2D

Nous avons introduit PIR en 1 dimension. Cependant, lors de sa requête, le client renvoie un tableau de chiffré, de la même taille que celle de la base de donnée. En retour, le serveur retourne un entier modulo N^2, β. Nos expériences nous ont montrées que lorsque nous augmentions les tailles, cela prenait beaucoup de temps.

Nous cherchons désormais à optimiser cette requête pour rendre l'échange plus rapide. Précédemment, le client considérait la base de donnée comme un tableau d'une seule ligne contenant toutes les données. Dorénavant, nous la considérerons comme un tableau à 2 dimensions comme ceci :

1	2	3
4	5	6
7	8	9

FIGURE 3 – Représentation du serveur de taille 9 en 2D

IV.1.1 Requête du client

De cette manière, le client demandera au serveur le contenu d'une case aux indices i et j. Nous choisissons en exemple la case d'indice 6 de coordonnées (2,3).Pour cela, le client va envoyer 2 tableaux avec les indices chiffrés par Paillier, un pour l'indice i, l'autre pour j. Ces tableaux de taille la racine carrée du nombre de cases (ici 3), sont composés d'éléments chiffrés de taille modulo N^2.

lignes	colonnes
1	1
2	2
3	3

lignes	colonnes
0	0
1	0
0	1

lignes	colonnes
152..25	134..45
165..84	563..53
498..65	987..45

TABLE 1 – Illustration des étapes de chiffrement des tableaux d'indices en 2D

Le client communique ces deux tableaux chiffrés au serveur. Cela se traduit par l'implémentation suivante :

```
1   void request_client_2d(int index_width, int index_length, mpz_t public_key_N,
2                          mpz_t m, mpz_t c, mpz_t *request1, mpz_t *request2,
3                          mpz_t public_key_N2, int size) {
4   clean_seed(state); // cleen the previous seed in the main
5   do_seed();
6
7   // row
8   for (int i = 0; i < size; i++) {
9     mpz_init(request1[i]);
10    mpz_set_ui(m, (i == index_width) ? 1 : 0);
```

```
    encrypt(m, public_key_N, c, public_key_N2);
    mpz_set(request1[i], c);
  }

  // column
  for (int i = 0; i < size; i++) {
    mpz_init(request2[i]);
    mpz_set_ui(m, (i == index_length) ? 1 : 0);
    encrypt(m, public_key_N, c, public_key_N2);
    mpz_set(request2[i], c);
  }
}
```

IV.1.2 Retour du serveur

Le serveur, quant à lui, va utiliser le premier tableau et appliquer PIR en 1 dimension sur chaque colonne de la base. Cela lui donnera donc trois chiffrés (c_1,c_2,c_3) si nous considérons notre exemple. Après l'application de PIR, ces chiffrés seront modulo N^2.

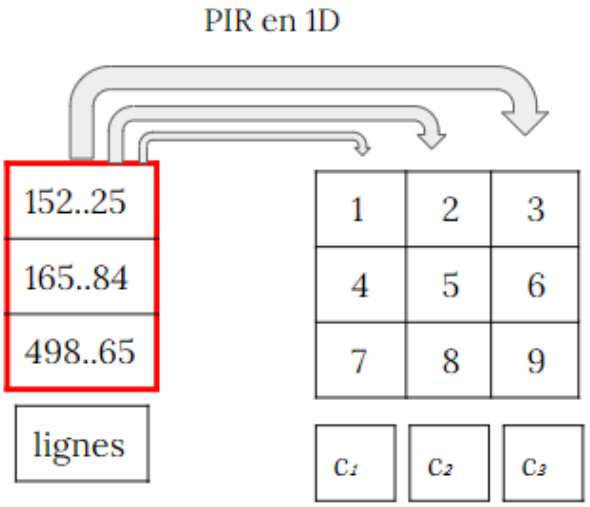

FIGURE 4 – Application de PIR 1D en 2D

Comme suit, notre implémentation :

```
  void server_return_2D(mpz_t *request1, mpz_t *request2, mpz_t response,
                        mpz_t public_key_N, mpz_t a_cipher, mpz_t b_cipher,
                        mpz_t public_key_N2, int size, int slice) {

mpz_t *col = calloc(size, sizeof(mpz_t));
mpz_t *cipher_square = calloc(size, sizeof(mpz_t));

int index2 = 0;
for (int j = slice; j < slice + (int)pow(size, 2); j += size) {
  int var = 0;

  for (int index = j; index < j + size; index++) {

    mpz_set(col[var], server[index]);
    var++;
  }
```

```
17
18      server_return(request1, response, col, public_key_N2, size);
19      mpz_set(cipher_square[index2], response);
20
21      index2++;
22    }
```

Pour considérer le deuxième tableau de coordonnées, ici colonnes, il faut re-appliquer PIR en 1 dimension. Afin de l'appliquer, nous devons transformer (c_1,c_2,c_3) en modulo N . Pour ceci, nous considérons chaque chiffré comme $c_i = a_i + b_i N$, avec a_i et b_i de taille modulo N désormais. Nous construisons un tableau avec les a_i et un avec les b_i pour tous les chiffrés obtenus.

c_1	
a_1	b_1

c_2	
a_2	b_2

c_3	
a_3	b_3

a_1	a_2	a_3

b_1	b_2	b_3

TABLE 2 – Illustration de génération des tableaux a et b

Par l'implémentation :

```
1   //initialization
2   mpz_t *a = calloc(size, sizeof(mpz_t));
3   mpz_t *b = calloc(size, sizeof(mpz_t));
4
5   //  creation of array a and b
6   for (int i = 0; i < size; i++) {
7     mpz_mod(a[i], cipher_square[i], public_key_N);
8     mpz_fdiv_q(b[i], cipher_square[i], public_key_N);
9   }
```

Les indices chiffrés maintenant modulo N, nous pouvons appliquer PIR en 1 dimension entre le deuxième tableau, les colonnes et les tableaux de a_i et de b_i. Ainsi, on obtiendra un $a_{chiffre}$ et un $b_{chiffre}$, de taille modulo N^2, que le serveur renvoie au client.

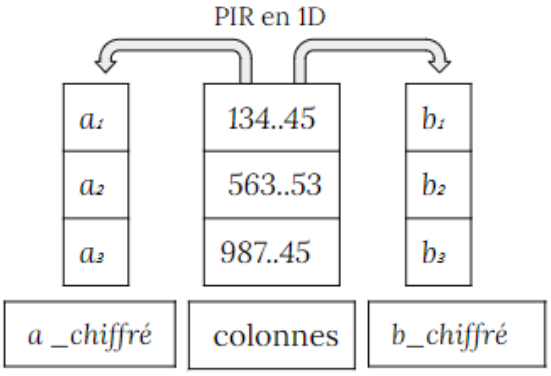

FIGURE 5 – Schéma du chiffrement de $a_{chiffre}$ et $b_{chiffre}$

```
1   server_return(request2, a_cipher, a, public_key_N2, size);
2   server_return(request2, b_cipher, b, public_key_N2, size);
```

IV.1.3 Déchiffrement

Pour obtenir le contenu de la case, notre m clair, il nous suffit de déchiffrer $a_{chiffré}$ et $b_{chiffré}$ avec Paillier pour les ramener à un modulo N et de recomposer $a_{dechiffre} + b_{dechiffre} \cdot N$, pour déchiffrer notre message.

```
void decrypt_2d(mpz_t a_cipher, mpz_t b_cipher, mpz_t a, mpz_t b,
                mpz_t private_key, mpz_t m2, mpz_t public_key_N,
                mpz_t public_key_N2) {
  decrypt(a_cipher, private_key, a, public_key_N, public_key_N2);
  decrypt(b_cipher, private_key, b, public_key_N, public_key_N2);

  mpz_mul(b, b, public_key_N);

  mpz_add(b, a, b);

  decrypt(b, private_key, m2, public_key_N, public_key_N2);
}
```

IV.1.4 Expériences

Testons désormais l'efficacité de la 2D :

Taille de BDD \ Taille p , q (bits)	**512**	**1024**	**2048**	**4096**
64	0.222	1.435	9.470	57.701
256	0.646	4.586	29.107	169.049
729	1.643	11.662	74.369	425.533
4096	8.301	60.186	383.555	2169.540

FIGURE 6 – Temps d'exécution du protocole PIR en 2D (en s)

Nous remarquons que par rapport aux résultats de la 1D, pour une base de données de taille 64, la 2D est un peu plus rapide. Cependant, un appel à la 2D sur des bases de données plus grandes est 2 fois plus rapide. Le protocole PIR en 2D est plus rapide qu'en 1D et cet écart augmente avec la taille.

IV.2 3D

Nous avons mis en place une optimisation de PIR en 2 dimensions. Cependant, par des opérations similaires, on peut considérer la base de donnée comme un tableau en 3, 4, D dimensions. Pour un D de plus en plus grand. Pour nous aider à comprendre, nous avons commencé par implémenter PIR en 3 dimensions, nous prenons pour exemple une base de donnée de taille 64.

IV.2.1 Requête du client

Pour cela, le client doit maintenant envoyer trois tableaux de taille la racine cubique de la taille de la base de donnée, représentant les indices chiffrés par Paillier i, j et k. Les éléments sont de taille modulo N^2. On note l cette racine.

```
    void request_client_3d(int index_width, int index_length, int index_deep,
                        mpz_t public_key_N, mpz_t m, mpz_t c, mpz_t *request1_3d,
                        mpz_t *request2_3d, mpz_t *request3, mpz_t public_key_N2,
                        int size) {

//same code as 2D

  // depth
  for (int i = 0; i < size; i++) {
    mpz_init(request3[i]);
    mpz_set_ui(m, (i == index_deep) ? 1 : 0);
    encrypt(m, public_key_N, c, public_key_N2);
    mpz_set(request3[i], c);
  }
}
```

On envoie donc plus de messages, 3, mais de taille plus petite qu'en 2D pour cette taille de base de donnée. On considère désormais la base de donnée comme un cube et chaque tranche du cube comme un tableau en 2D. Il y a donc l tranches, et nous appliquons PIR en 2 dimensions sur chaque tranche, avec les tableaux d'indices chiffrés i et j.

ligne	colonne	profondeur
21..57	15..68	59..15
32..56	51..65	62..42
16..87	95..41	78..15
15..44	18..66	44..31

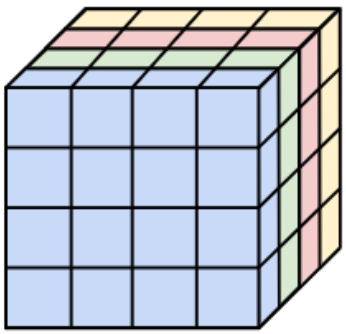

TABLE 3 – Tableaux d'indices chiffrés et Visualisation 3D

IV.2.2 Retour du serveur

Chaque appel à PIR en 2D va nous renvoyer un chiffré a et un chiffré b de taille modulo N^2, que nous stockons dans deux tableaux différents u et v. Ensuite, nous divisons chaque élément de ces tableaux en c = $uu + uv \cdot N$, pour les éléments de u et en c = $vu + vv \cdot N$, pour les éléments de v.

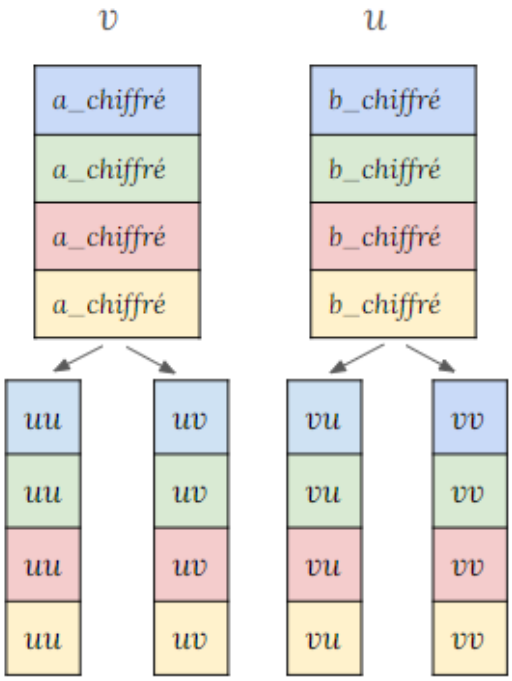

FIGURE 7 – Schéma construction des tableaux uu, uv, vu, vv

```
   void server_return_3D(mpz_t *request1_3d, mpz_t *request2_3d, mpz_t *request3,
                         mpz_t response, mpz_t public_key_N, mpz_t a_cipher,
                         mpz_t b_cipher, mpz_t public_key_N2, mpz_t uu, mpz_t uv,
                         mpz_t vu, mpz_t vv, int size) {

// In u there is all the A of the l dimensions
mpz_t *u = calloc(size, sizeof(mpz_t));

// In v there is all the B of the l dimensions
mpz_t *v = calloc(size, sizeof(mpz_t));

int index = 0;

for (int i = 0; i < SIZE_SERV; i += (int)pow(size, 2)) {
  server_return_2D(request1_3d, request2_3d, response, public_key_N,
                   a_cipher, b_cipher, public_key_N2, size, i);

  mpz_set(u[index], b_cipher);
  mpz_set(v[index], a_cipher);
  index++;
}

mpz_t *uud = calloc(size, sizeof(mpz_t));
mpz_t *uvd = calloc(size, sizeof(mpz_t));
mpz_t *vud = calloc(size, sizeof(mpz_t));
```

```
26  mpz_t *vvd = calloc(size, sizeof(mpz_t));
27
28  for (int i = 0; i < size; i++) {
29    // u(d)
30    mpz_mod(uvd[i], u[i], public_key_N);
31    mpz_fdiv_q(uud[i], u[i], public_key_N);
32
33    // v(d)
34    mpz_mod(vvd[i], v[i], public_key_N);
35    mpz_fdiv_q(vud[i], v[i], public_key_N);
36  }
```

Nous avons quatre tableaux de chiffrés : *uu*, *uv*, *vu* et *vv*. À ce moment-là, le dernier tableau que le client a envoyé, où l'indice *k* est chiffré, entre en jeu. On va utiliser PIR en 1D avec le tableau *k* et tous les tableaux *uu*, *uv*, *vu* et *vv*. Ceci nous renverra donc quatre chiffrés.

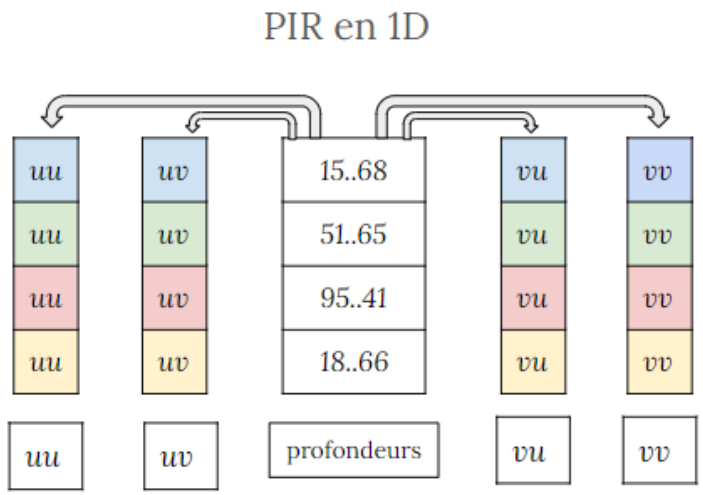

FIGURE 8 – Schéma du chiffrement de *uu*, *uv*, *vu* et *vv* chiffrés

```
1  server_return(request3, uu, uud, public_key_N2, size);
2  server_return(request3, uv, uvd, public_key_N2, size);
3  server_return(request3, vu, vud, public_key_N2, size);
4  server_return(request3, vv, vvd, public_key_N2, size);
```

IV.2.3 Déchiffrement

Pour les déchiffrer, nous procédons de la manière suivante, avec D considérée comme la fonction de déchiffrement de Paillier :

$$x(i,j,k) = D(D(D(uu) * N + D(uv)) * N + D(D(vu) * N + D(vv))) \tag{9}$$

Ce qui se traduit dans notre code par :

```
1  void decrypt_3d(mpz_t uu, mpz_t uv, mpz_t vu, mpz_t vv, mpz_t private_key,
2                  mpz_t m3, mpz_t public_key_N, mpz_t public_key_N2) {
3
```

```
4   mpz_t Duu, Dvv, Duv, Dvu, addvu_vv, adduu_uv, Dvun, Duun, A, An, B, finale;
5   mpz_inits(Duu, Dvv, Duv, Dvu, addvu_vv, adduu_uv, Dvun, Duun, A, An, B,
6             finale, NULL);
7
8   decrypt(vv, private_key, Dvv, public_key_N, public_key_N2);
9   decrypt(uv, private_key, Duv, public_key_N, public_key_N2);
10  decrypt(vu, private_key, Dvu, public_key_N, public_key_N2);
11  decrypt(uu, private_key, Duu, public_key_N, public_key_N2);
12
13  mpz_mul(Dvun, Dvu, public_key_N);
14  mpz_mul(Duun, Duu, public_key_N);
15
16  mpz_add(addvu_vv, Dvun, Dvv);
17  mpz_add(adduu_uv, Duun, Duv);
18
19  decrypt(adduu_uv, private_key, A, public_key_N, public_key_N2);
20  decrypt(addvu_vv, private_key, B, public_key_N, public_key_N2);
21  mpz_mul(An, A, public_key_N);
22
23  mpz_add(finale, An, B);
24  decrypt(finale, private_key, m3, public_key_N, public_key_N2);
25
26  mpz_clears(Duu, Dvv, Duv, Dvu, addvu_vv, adduu_uv, Dvun, Duun, A, An, B,
27             finale, NULL);
28  }
```

IV.2.4 Expériences

Examinons nos nouvelles expériences en 3D :

Taille de BDD \ Taille p , q (bits)	**512**	**1024**	**2048**	**4096**
64	0.282	1.885	12.277	73.602
512	1.362	9.777	62.59	357.881
729	1.862	13.408	85.235	487.077
4096	9.119	66.684	416.458	2356.087

FIGURE 9 – Temps d'exécution du protocole PIR en 3D (en s)

Les résultats de la 2D étant meilleurs qu'en 1D, nous nous attendions à une amélioration de nos temps en 3D par rapport à la 2D, ce n'est pas le cas. Pour les tailles de bases de données que nous pouvons comparer : 64, 729, 4096 ; la 2D est en moyenne 1,215 fois plus rapide que la 3D. Ce n'est pas une différence significative bien qu'elle existe.

De plus pour certains arguments la dimension 1 est meilleure. Par exemple pour une taille de base de données de 64 et de taille 729 avec p et q de taille 1024 bits. La 3D n'est vraiment pas efficace pour des tailles de bases de données trop petites. Les courbes ci-dessous montrent bien ce phénomène :

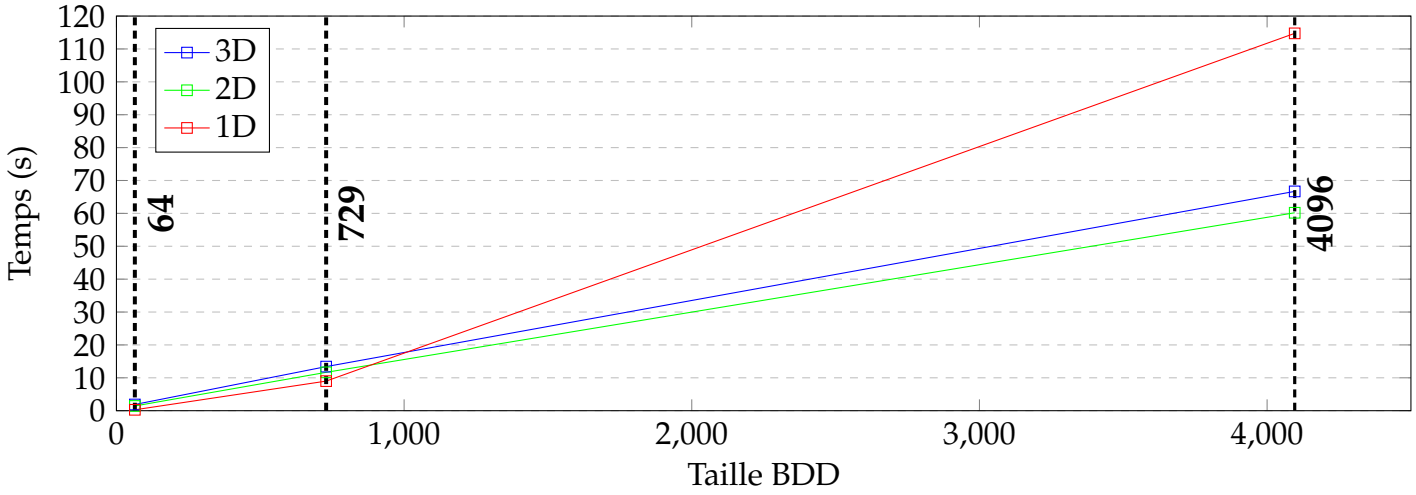

FIGURE 10 – Temps d'exécution du protocole PIR en fonction de la taille de la base de données avec p et q de 1024 bits

Ainsi, nous pouvons expliquer ces résultats décevant par le fait que peut-être les éléments choisis pour effectuer les expériences en 3D ne sont pas optimaux en cette dimension, la taille de la base de données n'est pas assez grande pour visualiser l'optimisation. En effet, le calcul des vecteurs est très chronophage par rapport aux calculs de chiffrement et déchiffrement.

IV.3 Dimension D

Pour faciliter l'implémentation, le client va toujours considérer la base de données comme un tableau à une dimension, et va vouloir accéder à une case x dans ce tableau. Le programme calcule donc les indices de cette case en dimension D afin de créer les tableaux de chiffrés. Pour ceci, on remarque que l'indice de x =

$$x_0 + x_1 \cdot l + x_2 \cdot l^2 + ... + x_d \cdot l^d \tag{10}$$

donc des divisions successives par l nous permettront de trouver les indices puis de les chiffrer.

On déduit de la construction précédente que si on veut appliquer PIR en D dimensions, il faudra utiliser l (la racine d-ième de la taille de la base de données) fois PIR en dimension D-1, et ainsi de suite pour la dimension D-1, jusqu'à arriver à la dimension 3 que nous avons déjà implémentée. Cela se résume donc à des appels récursifs de PIR.

V Conclusion

Au fil de nos expériences à travers la dimension 1, 2 et 3 du protocole PIR, nous remarquons une amélioration de nos résultats malgré la complexification des arguments du système. Cette amélioration n'est pas significative entre la deuxième et troisième dimension. D'après nos expériences, nous supposons que l'amélioration des temps n'est pas linéaire et dépend des arguments du système.

Nous supposons qu'avec des dimensions supérieures, l'optimisation est meilleure pour des tailles de base de donnée supérieures. Hypothèse que nous pourrions expérimenter dans un second temps avec l'implémentation de la dimension D.

Malgré des résultats d'expériences en 3D qui nous étonnent, l'objectif de mettre en place le protocole cryptographique PIR est réussi. Notre client reçoit bien du serveur l'élément qu'il a demandé sans que celui-ci ne l'identifie parmi sa base. De plus, une amélioration est remarquée dans l'exécution du protocole entre la première et la deuxième dimension.

Afin de challenger notre système cryptographique, nous pourrions le confronter à une réelle base de données et continuer de tester son efficacité [6].

VI Bibliographie

Références

[1] Alexandra Henzinger. Simplepir : Simple and fast single-server private information retrieval. https://www.usenix.org/system/files/sec23_slides_henzinger.pdf.

[2] The gnu mp bignum library documentation. https://gmplib.org/gmp-man-6.3.0.pdf.

[3] La factorisation des grands entiers : de fermat au code rsa. https://www.tangente-mag.com/article.php?id=7059#:~:text=Aujourd'hui%2CgrÃćceauxordinateurs,plusgrandentierproduitde.

[4] Guilhem Castagnos. Cours de cryptologie, le chiffrement de paillier (1999). https://www.math.u-bordeaux.fr/~gcastagn/Crypto/CoursCrypto-23-24.pdf. Page 35-37.

[5] Rafail Ostrovsky and William E. Skeith III. A survey of single-database pir : Techniques and applications. https://eprint.iacr.org/2007/059.pdf.

[6] Le top 10 des menaces de sécurité des bases de données. https://www.globalsecuritymag.fr/IMG/pdf/Top_Ten_Database_Threats_IMPERVA.pdf.

[7] Gihyuk Ko. Private information retrieval. https://course.ece.cmu.edu/~ece733/lectures/20-pir.pdf.

[8] Yan-Cheng Chang. Single database private information retrieval with logarithmic communication. https://eprint.iacr.org/2004/036.pdf.

[9] Pascal Paillier. Public-key cryptosystems based on composite degree residuosity classes. https://link.springer.com/content/pdf/10.1007/3-540-48910-X_16.pdf.

[10] Demonstration of the paillier cryptosystem use in a voting application. https://web.archive.org/web/20120218222103/http://security.hsr.ch/msevote/paillier.