application de MPC pour AES

mettre AES dans un contexte de boîte blanche grâce à MPC

Caisson Mathis

Capsule Team
INRIA - IRISA - University of Rennes 1

Jun. 21, 2022

Summary

Introduction au MPC

qu'est ce que veut dire MPC

Définition : $\boldsymbol{M}$ulty $\boldsymbol{P}$arty $\boldsymbol{C}$omputation

MPC ou multy party compuation est une méthode de calcul sécurisé dans un réseau de machine permettant de calculer le résultat d'une fonction sur des données distribuées et privés d'un partie à l'autre, le tout sans jamais révéler ces données aux autres partis

Typage des variables MPC

- Variable secrète : $\{x\}$ est une variable connue uniquement du parti qui a fournis la variable x
- Variable publique : $[x]$ est une variable connue de tout les partis et visible

Objectif d'un calcul multipartite

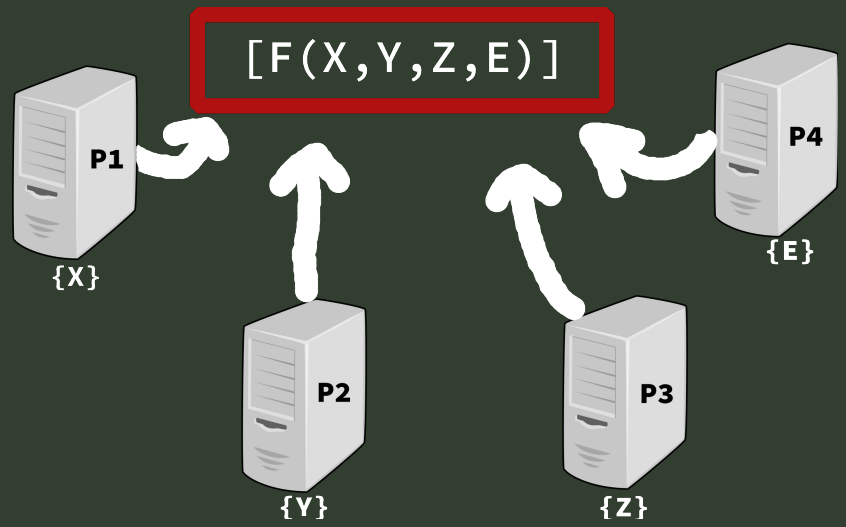

Figure : Objectif à atteindre dans le cadre d'un calcul multipartite

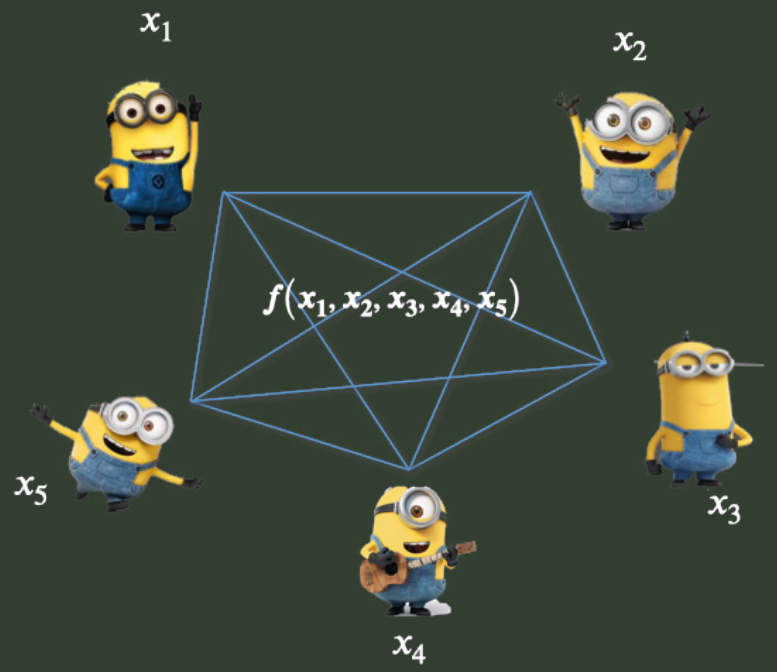

Figure : schéma plus réaliste des communications effectuées.

Qu'est ce que le chiffrement homomorphe

Définition : chiffrement $\boldsymbol{homomorphe}$

soit F une fonction quelconque évaluable d'arrité $\boldsymbol{n}$ avec $n \in \mathbb{Z}^+$,
$\delta \; et \; \gamma$ respectivement les fonctions de déchiffrement et de chiffrement,
M désigne un message clair et C un message chiffré
Alors une fonction de chiffrement homomorphe doit respecter les propriétés suivantes :

- $correction : \forall i \in \mathbb{Z}^+ : \delta(\gamma(M_i) \; = \; M_i$
- $homorphisme : \gamma(F(M_1, ..., M_n)) \; = \; F(\gamma(M_1), ..., \gamma(M_n)) = F(C_1, ..., C_n)$

Calcul d'une fonction en MPC

protocole d'application d'une fonction avec MPC

Pour pouvoir appliquer une fonction sur N variables secrètes distribuées entre N partis

- Séparer localement et aléatoirement la valeur en N fragments pour chaque parti possèdant une donnée
- Distribuer à chaque partis (y compris soit même) un des fragments obtenus
- Appliquer localement la fonction avec pour paramètre les fragments obtenus des autres partis
- Reconstruire le résultat en communiquant entre tout les partis les résultats de la fonction pour chaque parti attention on doit avoriune fonction permettant de respécter les propriété du chiffrement homomorphe

Schéma de calcul en MPC

Example de calcul d'une fonction sur 2 valeurs secrètes parmis 3 partis

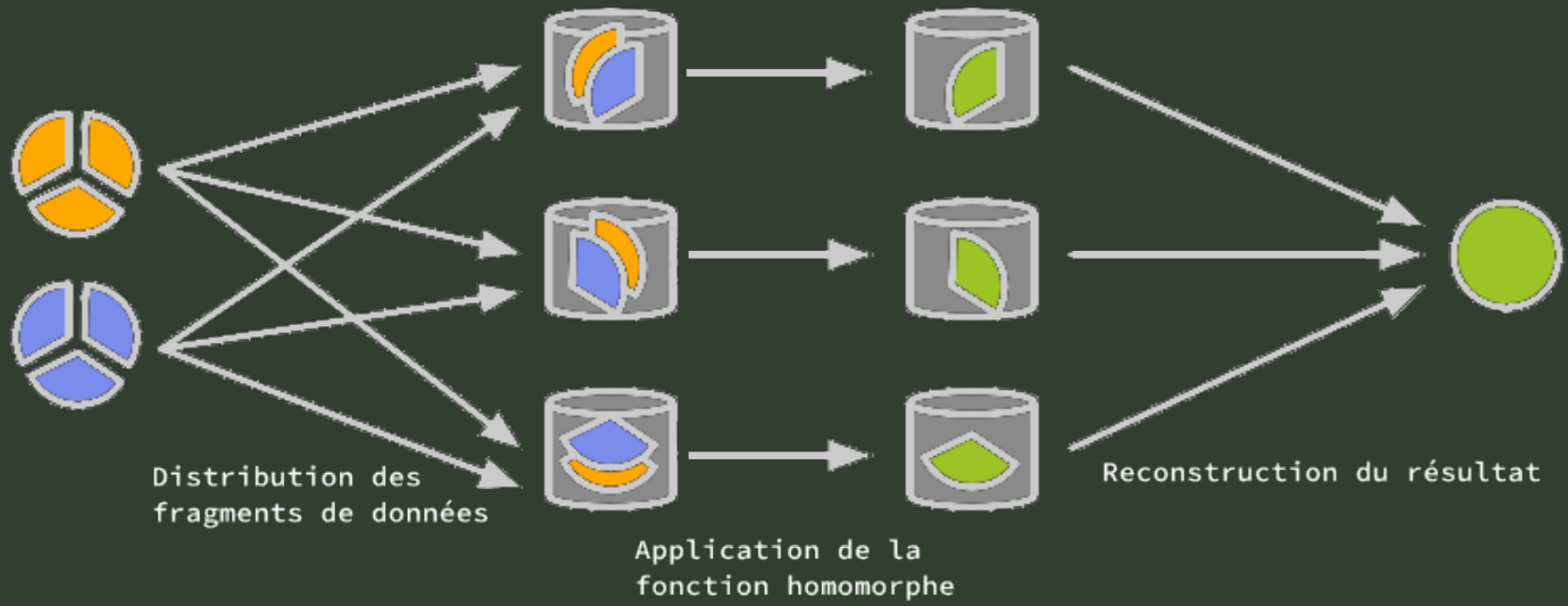

Figure : Application sécrurisée d'une fonction homomorphe sur des valeurs secrètes.

utilisation d'un framework MPC

SPDZ un protocol basé pour MPC

Définition : SPDZ

SPDZ est un framework permettant d'effectuer du Multy Party COmputation en rajoutant en plus de cela des protocoles pour chaque opérateur et un test de constistance des données avec la présence de vérification du système par MAC.
SPDZ distingue deux phases :

- la phase offline : sert avant tout à préparer les calculs sur des valeurs secrètes en générant de l'aléatoire que ce soit pour les MAC mais aussi pour la séparation des valeurs secrétes et fragments.
- la phase online : sert à récolter les différents fragments des valeurs secrètes entre partis et à y appliquer les diverses fonctionnalités requises dans le programme. C'est une partie du protocole qui doit être optimisée et efficace.

SPDZ, vérification par MAC

Vérification de MAC avec 3 partis

soit f une fonctionnalité de la cryptographie homomorphe et p un entier premier aléaoire publique.

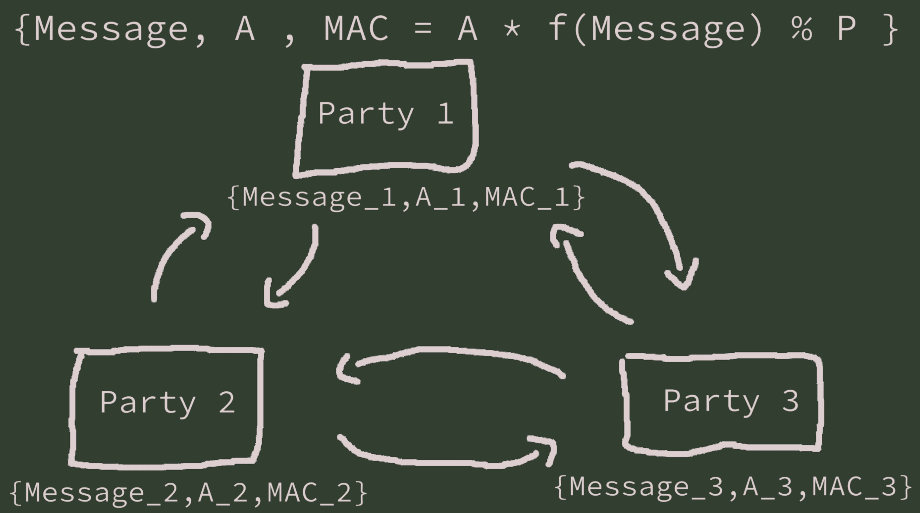

Figure : Echange de fragments de message,MAC et clé entre 3 partis

pour la vérification de mac on a le protocole suivant :

- $\forall i, P_i \; compute \; f(Message_i)$
- $f(Message)' = \sum_i f(Message_i)$
- $\forall i, P_i \; compute \; d_i = ((A_i * f(Message)')\%p) - MAC_i$

si jamais on a $(\sum_i d_i) == 0$ alors le message n'a pas été compromis dans la communication ou alors l'attaquant a reussi casser le MAC

MP-SPDZ : un fork de SPDZ2

Quest ce que MP-SPDZ ?

MP-SPDZ est un framework permettant de programmer facilement des système MPC parmis un panel de protocoles disponibles soit a travers un réseau ou bien des threads.

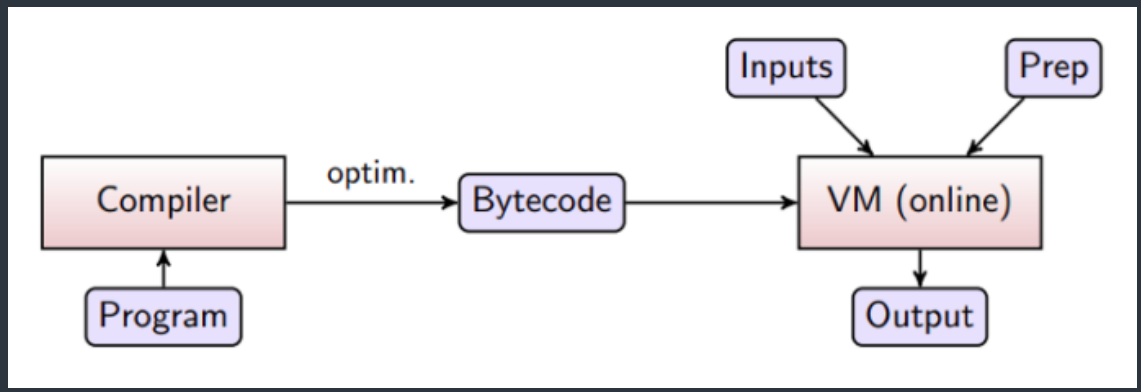

Figure : Architecture compilateur utilisé

Ainsi on peut soit programmerr avec une interface haut niveau en python ou bas niveau en C++ pour convertir notre code directement en bytecode avant d'étre exécuter en phase online dans des VM assurant un isolement entre les partis et empechant la fuite d'information.

Comment programmer en MP-SPDZ ?

opérateurs classiques

```
1  a = sint(1)
2  b = sint(2)
3  def test(actual, expected):
4      actual = actual.reveal()
5      print_ln('expected %s, got %s', expected, actual)
6
7  test(a + b, 3)
8  test(a * b, 2)
9  test(a - b, -1)
```

Default bit length : 64
Default security parameter : 40
Program requires at most : 1 integer triples & 3 virtual machine rounds

Comment programmer en MP-SPDZ ?

opérateurs classiques

Running /home/red/mp-spdz-0.3.2/Scripts/../spdz2k-party.x 0 tutorial -b 1 -pn 14682 -h localhost -N 2

Running /home/red/mp-spdz-0.3.2/Scripts/../spdz2k-party.x 1 tutorial -b 1 -pn 14682 -h localhost -N 2

Using security parameter 40
expected 3, got 3
expected 2, got 2
expected -1, got -1
Time = 0.00341053 seconds
Data sent = 0.175036 MB in 79 rounds (party 0)
Global data sent = 0.350072 MB (all parties)

Comment programmer en MP-SPDZ ?

Espace fini, tableau et entrées

```
1  secret_array = Array(16,sgf2n)
2  @for_range_opt(16)
3  def _whatever(i):
4      secret_array[i] = (sgf2n.get_input_from(0))+(sgf2n.get_input_from(1))
5
6  result = sum(secret_array)
7  print_ln("my galois sum is %s : ",result.reveal())
8  print_ln("my galois multiplication of 5*5 %s :",(sgf2n(5)*cgf2n(5)).reveal())
```

Default bit length : 64
Default security parameter : 40
Program requires at most : 16 gf2n inputs from player 0 & 2 virtual machine rounds

Comment programmer en MP-SPDZ ?

Espace fini, tableau et entrées

Running /home/red/mp-spdz-0.3.2/Scripts/../mascot-party.x 0 tutorial -b 64 -pn 15768 -h localhost -N 2

Running /home/red/mp-spdz-0.3.2/Scripts/../mascot-party.x 1 tutorial -b 64 -pn 15768 -h localhost -N 2

Using security parameter 40
my galois sum is 0x49 :
my galois multiplication of 5*5 0x11 :
The following timing is inclusive preprocessing. Time = 0.00899926 seconds
Data sent = 1.33827 MB in 31 rounds (party 0)
Global data sent = 1.36351 MB (all parties)

Comment executer notre programme MP-SPDZ ?

Choix du protocole à utiliser :

Nous voulons travailler sur le système de sécurité le plus contraignant ici une majorité non honnête et malicieuse. Par conséquent il faut choisir des portocol dit fait pour être dans un environnement de "Malicious, dishonest majority"

- MASCOT : c'est un protocole qui se base sur des transferts inconscients le tout avec un espace limité par un modulo d'un nombre entier premier généré aléatoirement
- SPDZ2K : c'est un protocole qui a pour but d'améliorer MASCOT toujours avec des transferts inconscients mais c'ette fois ci un espace limité par un module de 2^k avec souvent k $= 64$ mais dépend avant tout de l'architecture

Comment executer notre programme MP-SPDZ ?

Choix des paramètres de compilation :

Pour pouvoir optimiser au mieux notre programme il nous faut choisir minutieusement nos paramètres que ce' soit pour la sécurité ou encore pour l'optimisation du temps d'execution :

- -M / –preserve-mem-order paramètre qui vise à préserver l'ordre des instructions dans la mémoire. Attention provoque une petite perte de performance mais fiabilise l'excecution du programme
- -D / –dead-code-elimination paramètre qui va venir supprimer tout les bout de programmes inutilisés au lancement.
- -I / –flow-optimization paramètre qui va automatiquement transformer toutes les boucles pythons en boucles optimisées MP-SPDZ, fait gagner beaucoup de temps d'execution mais transforme tout les indices entier en regint.

AES implementation with MPC

Comment fonctionne AES ?

Traiter les entrées pour AES au départ

pour simplifier les choses nous allons tenter de reproduire avec MP-SPDZ la version dite ECB d'AES avec une clé de 256 bits :

- tout les partis donnent en entrée du MPC 32 mots de 8 bits stockés chacun dans des variables de type SGF2N
- on applique ensuite une opération de réduction avec pour opérateur le xor réunissant toute les clés individuelles en une seule clé globale secrète définie dans un $GF2^8$ stocké dans un Array
- Définir un parti qui va fournir le nombre de blocs fournis puis le message à chiffrer pour ensuite découper ce message en blocs de 16 mots de 8 bits réprésentés par des Array de SGF2N
- Enfin à partir des blocs de messages obtenus et de la clé secrète, appliquer les différentes étapes du chiffrement AES avec les opérateurs de SGF2N

Comment fonctionne AES ?

Traiter les entrées pour AES au départ

```
secret_key = Array(32,sgf2n)
secret_key.assign_all(sgf2n(0))
for i in range(32):
    secret_key[i] ^= (sgf2n.get_input_from(0))
    secret_key[i] ^= (sgf2n.get_input_from(1))

#ici on fixe la taille de notre message initial à un seul bloc mais quand on
#ne voudra plus hardcoder on devra utiliser des socket clients
blocs  = []
number_of_blocs = 1
@for_range_opt(number_of_blocs)
def _ok(i):
    secret_message = Array.create_from(sgf2n.get_input_from(0,size=16))
    blocs.append(secret_message)

for bloc in blocs:
    ...
```

Comment fonctionne AES ?

les différentes étapes d'AES en MPC

Si l'on suit les specifications données par le nist il nous faudra pour réaliser notre algorithme en MPC réaliser 5 fonctions qui vont influencer directement l'état de notre bloc ou de notre clé. ces 5 fonctions sont :

- l'expansion de la clé : le but est de passer d'une clé de 256 bits à une clé de 240 mots de 8 bits.
- l'évaluation de la sbox qui consiste à effectuer une substitution non linéaire de nos 16 mot de notre état.
- le décalage des lignes consistant à faire une rotation des lignes de notre état en fonction de l'indicie de la ligne.
- l'ajout de la clé du round consistant en un xor d'une partie de la clé expansée avec notre état
- le mélange des colonnes qui consiste à réaliser une multiplication de galois entre les colonnes de notre état et une matrice de constante

Implémentation de AES avec MP-SPDZ

la fonction ShiftRows

dans cette fonction on ne fait que simplement bouger les valeurs de palce dans notre état, on intervient pas sur les valeurs mais uniquement sur leur place donc aucun soucis avec des nombres secrets SGF2N :

```
1  def rotate(word, n):
2      #positive to rotate to left and negative to rotate right
3      ...
4  def shiftRows(state):
5      @for_range_opt(4)
6      def _someother(i):
7          t = Array(4,sgf2n)
8          for j in range(4):
9              t[j] = state[i+4*j]
10         ns = rotate(t,i)
11         for j in range(4):
12             state[i+4*j] = ns[j]
```

Implémentation de AES avec MP-SPDZ

la fonction AddRoundKey

pour la fonction addround key on donne en paramètre un état et une clé de round qui sont des Array de 16 mots de type SGF2N. Par conséquent on a grâce à MP-SPDZ l'opérateur xor déjà fourni pour faire un xor entre 2 valeurs secrètes dans un champs de galois tout en gardant le secret entre les valeurs. la fonction est donc assez triviale et très peu couteuse :

```
1 def addRoundKey(state,round_key):
2     @for_range_opt(len(state))
3     def _(i):
4         state[i] = state[i] ^ = round_key[i]
```

Implémentation de AES avec MP-SPDZ

la fonction MixColumns

Pour mixcolumns il nous faut effectuer une multiplication de galois entre une colonne de notre état et une matrice constante :

$$\begin{bmatrix} b_{i+1}^0 \\ b_{i+1}^1 \\ b_{i+1}^2 \\ b_{i+1}^3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \times \begin{bmatrix} b_i^0 \\ b_i^1 \\ b_i^2 \\ b_i^3 \end{bmatrix} \tag{1}$$

Projection dans le bon espace

Néanmoins pour réaliser notre multiplication de galois il faut se trouver dans $\mathbb{F}_{2^8}$ et nous nous trouvons de par SPDZ et son paramètre de sécurité dans $\mathbb{F}_{2^{40}}$. Il va alors nous falloir réaliser une fonction de projection $\mathbb{F}_{2^{40}} \hookrightarrow \mathbb{F}_{2^8}$ effectuer notre multplication de galois puis revenir dans l'espace initial avec la fonction inverse $\mathbb{F}_{2^8} \hookrightarrow \mathbb{F}_{2^{40}}$

Implémentation de AES avec MP-SPDZ

Projection $\mathbb{F}_{2^{40}} \hookrightarrow \mathbb{F}_{2^8}$

```
1  @vectorize
2  def ApplySecretEmbedding(x):
3      in_bytes = x.bit_decompose(8)
4
5      out_bytes = [sgf2n(0) for _ in range(8)]
6
7      out_bytes[0] = sum(in_bytes[0:8])
8      out_bytes[1] = sum(in_bytes[idx] for idx in range(1, 8, 2))
9      out_bytes[2] = in_bytes[2] + in_bytes[3] + in_bytes[6] + in_bytes[7]
10     out_bytes[3] = in_bytes[3] + in_bytes[7]
11     out_bytes[4] = in_bytes[4] + in_bytes[5] + in_bytes[6] + in_bytes[7]
12     out_bytes[5] = in_bytes[5] + in_bytes[7]
13     out_bytes[6] = in_bytes[6] + in_bytes[7]
14     out_bytes[7] = in_bytes[7]
15
16     return sgf2n.bit_compose(out_bytes)
```

Implémentation de AES avec MP-SPDZ

Projection $\mathbb{F}_{2^8} \hookrightarrow \mathbb{F}_{2^{40}}$

```
1  @vectorize
2  def ApplySecretInverseEmbedding(x):
3      in_bytes = x.bit_decompose(40)
4
5      out_bytes = [sgf2n(0) for _ in range(8)]
6
7      out_bytes[7] = in_bytes[7]
8      out_bytes[6] = in_bytes[6] + out_bytes[7]
9      out_bytes[5] = in_bytes[5] + out_bytes[7]
10     out_bytes[4] = in_bytes[4] + out_bytes[5] + out_bytes[6] + out_bytes[7]
11     out_bytes[3] = in_bytes[3] + out_bytes[7]
12     out_bytes[2] = in_bytes[2] + out_bytes[3] + out_bytes[6] + out_bytes[7]
13     out_bytes[1] = in_bytes[1] +  out_bytes[3] + out_bytes[5] + out_bytes[7]
14     out_bytes[0] = in_bytes[0] + sum(out_bytes[1:8])
15
16     return sgf2n.bit_compose(out_bytes)
```

Implémentation de AES avec MP-SPDZ

Fonction mixColumn pour une colonne quelconque de l'état

```
1  def mixColumn(column):
2      temp = Array(4,sgf2n)
3      kv = Array(2,cgf2n)
4      kv[0] = ApplyClearEmbedding(cgf2n(0x2))
5      kv[1] = ApplyClearEmbedding(cgf2n(0x3))
6      for i in range(4):
7          temp[i] = ApplySecretEmbedding(column[i])
8      column[0] = ApplySecretInverseEmbedding(temp[0]*kv[0] + temp[1]*kv[1]
9      + temp[2] + temp[3])
10     column[1] = ApplySecretInverseEmbedding(temp[1]*kv[0] + temp[2]*kv[1]
11     + temp[3] + temp[0])
12     column[2] = ApplySecretInverseEmbedding(temp[2]*kv[0] + temp[3]*kv[1]
13     + temp[0] + temp[1])
14     column[3] = ApplySecretInverseEmbedding(temp[3]*kv[0] + temp[0]*kv[1]
15     + temp[1] + temp[2])
```

Implémentation de AES avec MP-SPDZ

l'évaluation d'une table T publique par indice secret

On cherche à accéder à une table T de taille ll avec un indice x secret, dans notre cas de type SGF2N. Pour cela on va tenter de calculer une seconde Table à partir d'un masque aléatoire secret s de sorte que :

$$[Table(s)] = \left([T[s]], [T[s \oplus 1]], [T[s \oplus 2]], \dots , \left[T\left[s \oplus \left(2^l - 1\right)\right]\right]\right)$$

Ainsi lorsque l'on voudra accéder à un élément avec un indice secret, on génére notre masque aléatoire secret s et grâce à lui et T le tableau auquel on veut accéder, on peut générer une Table masquée. Il ne nous reste alors plus qu'à calculer $h = (x \oplus s)$. Enfin on accéde à notre Table secrète générée avec l'indice h qu'on a rendu publique de sorte que $[Table(S)][h] = T[h \oplus s] = T[x \oplus s \oplus s] = T[x]$

Implémentation de AES avec MP-SPDZ

Comment générer $[Table(s)]$?

Tout d'abord on va venir générer un masque aléatoire s sur 8 bits, cette valeur va alors passer dans un demultiplexeur pour obtenir s' avec les bits de s' à 0 sauf pour le bit s'_s qui est à 1. Enfin on vient calculer la Table de la manière suivante :

$$\forall i \ \in 0..\left(2^l - 1\right), \ [Table(s)][i] \ = \ [T[i \oplus s]]$$

$$= \ [T[i]] \times [s'_0] \ + [T[i \oplus 1]] \times [s'_1] \ + \ \ldots + \left[T\left[i \oplus \left(2^l - 1\right)\right]\right] \times [s'_{2^l-1}]$$

Implémentation de AES avec MP-SPDZ

Fonction démultiplexeur

```
1  def demux(s):
2      l = len(s)
3      p = [1-s[0],s[0]]
4      for j in range(1,l):
5          t=[]
6          for bit in p:
7              t.append(bit*s[j])
8          addLeft = [*([sgf2n(0,size=nparallel)]*(2**j)),*t]
9          addRight = []
10         for (bp,bt) in zip(p, t):
11             addRight.append(bp-bt)
12         addRight = [*addRight,*([sgf2n(0,size=nparallel)]*(2**j))]
13         tempP = []
14         for (bl,br) in zip(addLeft, addRight):
15             tempP.append((bl+br))
16         p = tempP
17     return p
```

Implémentation de AES avec MP-SPDZ

Fonction génération table secrète avec masque aléatoire

```
1  def generateSSTable(T):
2      s =sgf2n.get_random_input_mask_for(0)[0]
3      s = s.bit_decompose()[0:int(math.log2(len(T)))]
4      s= Array.create_from(s)
5      sp=Array.create_from(demux(s))
6      Table = Array(2**(len(s)),sgf2n)
7      Table.assign_all(sgf2n(0))
8      @for_range_opt([2**(len(s)),2**(len(s))])
9      def _(i,j):
0          Table[i] += T[i^j]*sp[j]
1      return (sgf2n.bit_compose(s),Table)
```

Implémentation de AES avec MP-SPDZ

évaluation d'une table avec indice secret

```
1  class globalVars():
2      generated,s = MemValue(0), MemValue(sgf2n(0))
3      Table,prevT = Array(256,sgf2n), MemValue(0)
4  def boxEvaluation(T,x):
5      #partie faite en sorte pour
6      if (globalVars.generated.read()==0):
7          (st,Tablet) = generateSSTable(T)
8          globalVars.s.write(st)
9          globalVars.Table.assign(Tablet)
10         globalVars.generated.write(1)
11         globalVars.prevT.write(T.address)
12     else
13         if (globalVars.prevT.read()!=T.address):
14             (st,Tablet) = generateSSTable(T)
15             globalVars.s.write(st)
16             globalVars.Table.assign(Tablet)
17             globalVars.prevT.write(T.address)
18     return globalVars.Table[h].reveal()
```

Implémentation de AES avec MP-SPDZ

Fonction subBytes

Maintenant que la fonction d'évaluation d'une table publique avec indice secret est réalisée, il nous est assez facile d'effectuer la fonction de substitution non linéaire. En effet il nous suffit d'évaluer avec pour indice secret la valeur de notre état, notre sbox étant stockée dans la classe box.

Fonction subBytes

```
1 def subBytes(state):
2     @for_range_opt(len(state))
3     def _(i):
4         state[i] = boxEvaluation(box.sbox,state[i])
```

Implémentation de AES avec MP-SPDZ

Fonction expandAESKey

Nous nous situons dans le cadre d'un schéma de clé de 256 bits par conséquent la fonction d'expansion de clé va nous générer une clé de 240 mots de 8 bits qui va ensuite pouvoir être découper en 15 morceaux de 16 mots de 8 bits chacun grâce à la fonction create round key qui va nous séléctionner la clé d'un round en fopnction de son indice de round.

Fonction createRoundKey

```
1  def createRoundKey(expandedKey, n):
2      na = Array(16,sgf2n)
3      @for_range_opt(16)
4      def _ohoh(i):
5          na[i] = expandedKey[n*16+i]
6      return na
```

Fonction expandAESKey

Fonction expandAESKey

```
1  def expandAESKey(cipherKey, Nr = 14, Nb = 4, Nk = 8):
2      round_key,temp = Array(240,sgf2n),Array(4,sgf2n)
3      @for_range_opt([Nk,4])
4      def _whatever(i,j):
5          round_key[(4*i)+j] = cipherKey[(4*i)+j]
6      @for_range_opt(Nb * (Nr + 1)-Nk)
7      def _perhaps(i):
8          @for_range_opt(4)
9          def _ohno(j):
0              temp[j] = round_key[(i-1+Nk) * 4 + j]
1          if i % Nk == 0:
2              #rotate the 4 bytes word to the left
3              temp.assign(rotate(temp,1))
4              #now substitute word
5              temp[0] = boxEvaluation(box.sbox,temp[0])
6              temp[1] = boxEvaluation(box.sbox,temp[1])
7              temp[2] = boxEvaluation(box.sbox,temp[2])
8              temp[3] = boxEvaluation(box.sbox,temp[3])
```

Fonction expandAESKey

Fonction expandAESKey

```
        temp[0] = temp[0] ^ box.rcon[((i+Nk)//Nk)]

        # since we're using a 256-bit key -> add an extra sbox transform
        if i % Nk == 4:
            temp[0] = boxEvaluation(box.sbox,temp[0])
            temp[1] = boxEvaluation(box.sbox,temp[1])
            temp[2] = boxEvaluation(box.sbox,temp[2])
            temp[3] = boxEvaluation(box.sbox,temp[3])

        @for_range_opt(4)
        def _idk(j):
            round_key[4 * (i+Nk) + j] = round_key[4 * i + j] ^ temp[j]
    return round_key
```

Fonction aesRound

effectuer un round d'AES

Maintenant que nos fonctions utiles pour réaliser un round d'AES son terminées il ne nous reste plus qu'à les appeler dans le bon ordre pour pouvoir effectuer un round d'AES. Enfin il nous faudra effectuer 14 rounds pour pouvoir compléter notre schéma de chiffremnet AES complet

Fonction expandAESKey

```
1 def aesRound(state, roundKey):
2     subBytes(state)
3     shiftRows(state)
4     mixColumns(state)
5     addRoundKey(state, roundKey)
```

Fonction principale de chiffrement AES ECB 256

Fonction expandAESKey

```
1  aesMain(expandedKey,state, numRounds=14):
2  roundKey = createRoundKey(expandedKey, 0)
3  addRoundKey(state,roundKey)
4
5  @for_range_opt(numRounds-1)
6  def _ok(i):
7      createRoundKey(expandedKey,i+1)
8      aesRound(state,roundKey)
9      globalVars.generated.write(0)
10 #last round
11 roundKey = createRoundKey(expandedKey,numRounds)
12 subBytes(state)
13 shiftRows(state)
14 addRoundKey(state,roundKey)
```

Performances de l'algorithme

Comparaison MASCOT vs SPDZ2K

nous allons tenter de faire un benchmark de performance que ce soit au niveau du chiffrement ou au niveau dechiffrement.

- Compilation : 2 minutes 22 secondes pour 2700000 lignes
- nombre de bits et triples GF2N : 54528 bits et 219456 triples

MASCOT

- Chiffrement : 1.64797 secondes pour 4368 rounds avec 166.469 MB échangés
- Déchiffrement : 1.86601 secondes pour 4455 rounds avec 178.071 MB échangés

SPDZ2K

- Chiffrement : 1.156 secondes pour 4372 rounds avec 166,482 MB échangés
- Déchiffrement : 1.31247 secondes pour 4459 rounds avec 178.084 MB échangés

Conclusion

Comment Améliorer l'implémentation ?

Des opérations couteuses en temps et en mémoire

- La fonction d'évaluation de table par indice secret

Cette fonction est une fonction qui prend beaucoup de temps, on est obligé de généré une grande quantité de masques aléatoire ainsi que la création d'une table entière ce qui provoque un temps de calcul accru et c'est la principal source de perte de temps dans le programme.

- la fonction de mélange de colonne et les multiplications de galois

la fonction de mixcolumns est la deuxième fonction qui prend le plus de temps car on est obligé à des fins de sécurité pour la vérification de la MAC de projetter notre espace fini constament. Il faudrait alors optimiser la fonction de projection ou bien réfléchir à travailler constamment dans $\mathbb{F}^{2^{40}}$ et revoir toutes les fonctions d'AES pour travailler dans cet espace.

Comment continuer l'intégration

Comment faire avancer le projet ?

Actuellement, tout les tests ont été fait avec uniquement 2 partis calculatoires en local sur un ordinateur et c'était le parti 0 qui fournissait le message à chiffrer ou déchiffrer de manière fixe dans le code. Les prochaines étapes seraient donc :

- généraliser l'infrastructure sur un réseau physique avec plusieurs machines
- améliorer encore la sécurité en proposant de l'AES CBC et plusieurs taille de clés possibles
- Mise en place de dockers et de réseaux de docker pour le calcul avec la mise en place d'une couche SSL
- Mise en place d'une application cliente externe aux machines calculatoires

References I

1. Alliance, M. *MPC Alliance wiki.* https://wiki.mpcalliance.org.
2. *Chiffrement homomorphe.* https://fr.wikipedia.org/wiki/Chiffrement_homomorphe.
3. Dahl, M. *The SPDZ Protocol, Part 1 Secure Computation using Precomputed Triples.* https://mortendahl.github.io/2017/09/03/the-spdz-protocol-part1/.
4. *github MP-SPDZ.* https://github.com/data61/MP-SPDZ.
5. inpher. *What is Secure Multiparty Computation ?.* https://inpher.io/technology/what-is-secure-multiparty-computation/.

References II

6. Keller, M. *MP-SPDZ's documentation.* https://mp-spdz.readthedocs.io/en/latest/Compiler.html.
7. Marcel Keller Valerio Pastro, D. R. *Overdrive : Making SPDZ Great Again.* https://eprint.iacr.org/2017/1230.pdf.
8. Pullonen, P. *SPDZ : Multiparty Computation from Somewhat Homomorphic Encryption..* https://courses.cs.ut.ee/MTAT.07.014/2019_fall/uploads/Main/spdz.pdf.
9. *Rijndael MixColumns.* https://en.wikipedia.org/wiki/Rijndael_MixColumns.

The End