Comparaison Homomorphic Encryption

abdel-malik fofana

November 2023

Contents

1 CONCRETE **2**

2 HELIB **2**

3 PALISSADE (OPENFHE) **4**

4 SEAL **5**

5 TFHE **7**

6 Conclusion **7**

1 CONCRETE

Voici la documentation de concrete : https://docs.zama.ai/concrete/getting-started/installing Concrete permet de faire du chiffrement homomorphique , je l'ai utilisé en python Voici le code (fournis dans le fichier également avec les modifications pour calculer le temps d'execution): https://docs.zama.ai/concrete/getting-started/quick_start

On lance le python sur le docker contenant toute les librairies que l'on peut installer depuis la documentation :

```
root@a56a10d9a48c:/# python3 test.py
4+4 =  8
Temps que prend l'addition en python sans concrete :  1.5020370483398438e-05
4*4 =  16
Temps que prend la multiplication en python sans concrete :  2.1457672119140625e-06
4 + 4 = 8 = 8
l'algorithme addition avec Concrete à pris :  0.03432059288024902  sec
4 + 4 = 16 = 16
l'algorithme multiplication avec Concrete à pris :  0.03179430961608887  sec
root@a56a10d9a48c:/# 
```

Voici les resultats apres avoir comparer le temps d'execution de chaque fonction:

Sans concrete :

Addition en python normalement $= 1.5$ x10^5sec

Multiplication en python normalement $= 2.1$ x 10^6sec

Avec concrete :

Addition en python avec concrete $= 0.03432059288$ sec

Multiplication en python avec concrete $=0.0317943096$ sec

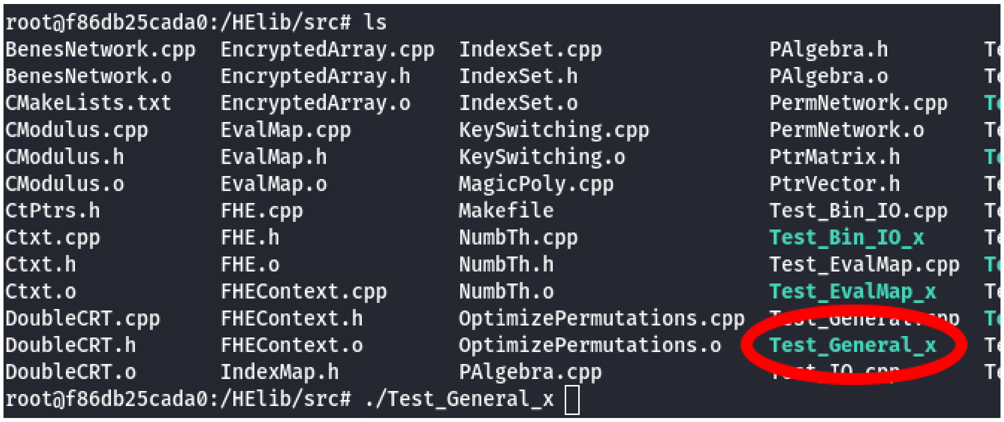
2 HELIB

Voici le lien ou telecharger Helib : https://github.com/homenc/HElib/tree/aes

Voici le lien du docker où j'ai telecharger le helib avec un test de performance déja fait :

https://hub.docker.com/r/kenmaro/helib

Voici le test de performance déja fourni :



il y a plusieurs fonction qui font la multiplication et l'addition voici quelque une

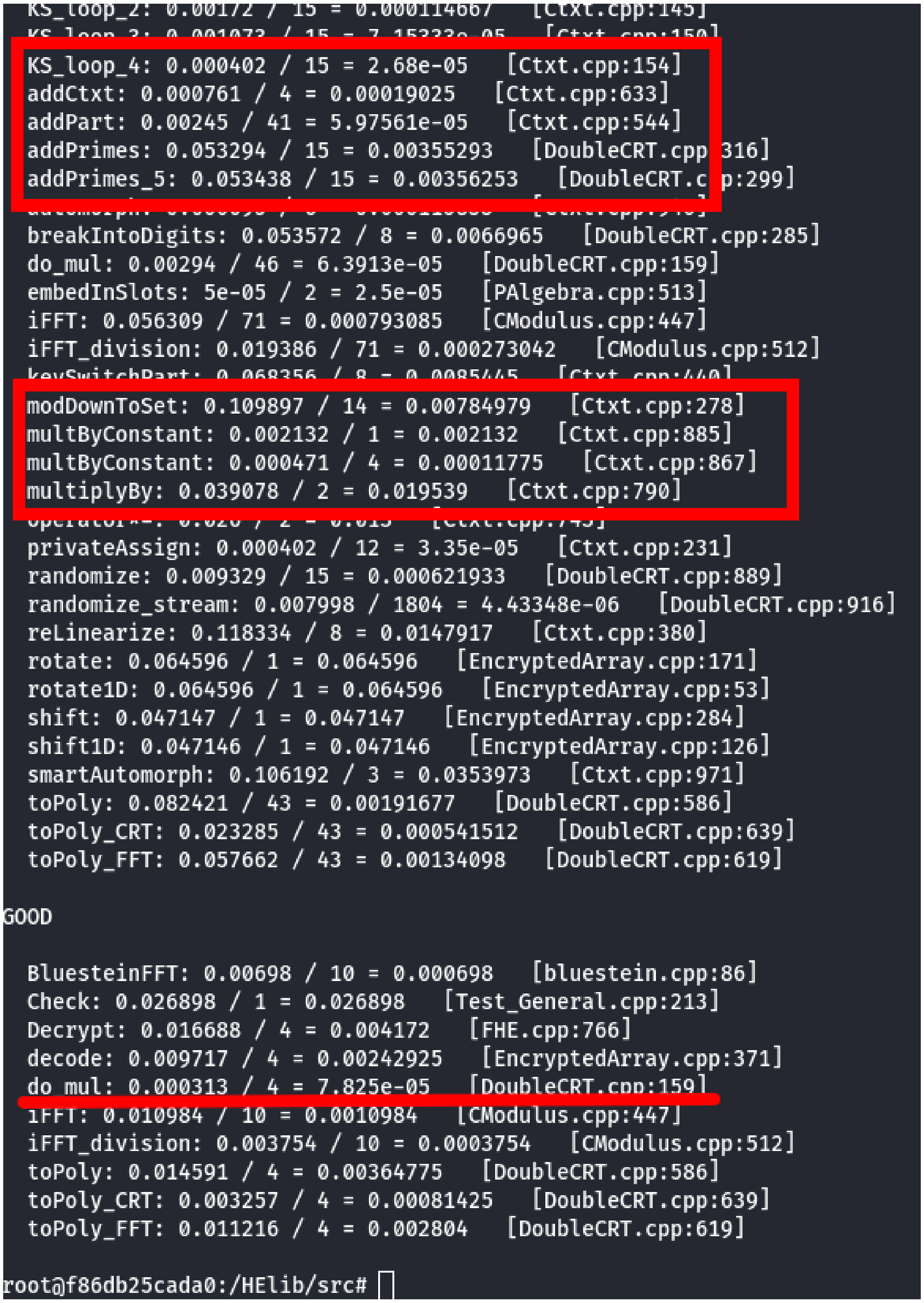
```
Une partie du code :
 mul(ea, p1, p0);     // c1.multiplyBy(c0)
     c1.multiplyBy(c0);
     if (!noPrint) CheckCtxt(c1, "c1*=c0");
     debugCompare(ea,secretKey,p1,c1);

     add(ea, p0, const1); // c0 += random constant
     c0.addConstant(const1_poly);
     if (!noPrint) CheckCtxt(c0, "c0+=k1");
     debugCompare(ea,secretKey,p0,c0);

     mul(ea, p2, const2); // c2 *= random constant
     c2.multByConstant(const2_poly);
     if (!noPrint) CheckCtxt(c2, "c2*=k2");
     debugCompare(ea,secretKey,p2,c2);
```

Je vous est fournis le code dans le dossier pour le voir plus en detail

Voici une partie de l'execution:



```
KS_loop_2: 0.00172 / 15 = 0.000114667   [Ctxt.cpp:145]
KS_loop_4: 0.000402 / 15 = 2.68e-05   [Ctxt.cpp:154]
addCtxt: 0.000761 / 4 = 0.00019025   [Ctxt.cpp:633]
addPart: 0.00245 / 41 = 5.97561e-05   [Ctxt.cpp:544]
addPrimes: 0.053294 / 15 = 0.00355293   [DoubleCRT.cpp:316]
addPrimes_5: 0.053438 / 15 = 0.00356253   [DoubleCRT.cpp:299]
breakIntoDigits: 0.053572 / 8 = 0.0066965   [DoubleCRT.cpp:285]
do_mul: 0.00294 / 46 = 6.3913e-05   [DoubleCRT.cpp:159]
embedInSlots: 5e-05 / 2 = 2.5e-05   [PAlgebra.cpp:513]
iFFT: 0.056309 / 71 = 0.000793085   [CModulus.cpp:447]
iFFT_division: 0.019386 / 71 = 0.000273042   [CModulus.cpp:512]
modDownToSet: 0.109897 / 14 = 0.00784979   [Ctxt.cpp:278]
multByConstant: 0.002132 / 1 = 0.002132   [Ctxt.cpp:885]
multByConstant: 0.000471 / 4 = 0.00011775   [Ctxt.cpp:867]
multiplyBy: 0.039078 / 2 = 0.019539   [Ctxt.cpp:790]
privateAssign: 0.000402 / 12 = 3.35e-05   [Ctxt.cpp:231]
randomize: 0.009329 / 15 = 0.000621933   [DoubleCRT.cpp:889]
randomize_stream: 0.007998 / 1804 = 4.43348e-06   [DoubleCRT.cpp:916]
reLinearize: 0.118334 / 8 = 0.0147917   [Ctxt.cpp:380]
rotate: 0.064596 / 1 = 0.064596   [EncryptedArray.cpp:171]
rotate1D: 0.064596 / 1 = 0.064596   [EncryptedArray.cpp:53]
shift: 0.047147 / 1 = 0.047147   [EncryptedArray.cpp:284]
shift1D: 0.047146 / 1 = 0.047146   [EncryptedArray.cpp:126]
smartAutomorph: 0.106192 / 3 = 0.0353973   [Ctxt.cpp:971]
toPoly: 0.082421 / 43 = 0.00191677   [DoubleCRT.cpp:586]
toPoly_CRT: 0.023285 / 43 = 0.000541512   [DoubleCRT.cpp:639]
toPoly_FFT: 0.057662 / 43 = 0.00134098   [DoubleCRT.cpp:619]

GOOD

BluesteinFFT: 0.00698 / 10 = 0.000698   [bluestein.cpp:86]
Check: 0.026898 / 1 = 0.026898   [Test_General.cpp:213]
Decrypt: 0.016688 / 4 = 0.004172   [FHE.cpp:766]
decode: 0.009717 / 4 = 0.00242925   [EncryptedArray.cpp:371]
do_mul: 0.000313 / 4 = 7.825e-05   [DoubleCRT.cpp:159]
iFFT: 0.010984 / 10 = 0.0010984   [CModulus.cpp:447]
iFFT_division: 0.003754 / 10 = 0.0003754   [CModulus.cpp:512]
toPoly: 0.014591 / 4 = 0.00364775   [DoubleCRT.cpp:586]
toPoly_CRT: 0.003257 / 4 = 0.00081425   [DoubleCRT.cpp:639]
toPoly_FFT: 0.011216 / 4 = 0.002804   [DoubleCRT.cpp:619]

root@f86db25cada0:/HElib/src#
```

On en conclu que selon le type d'addition on à : 0.00019025 sec et 5.97561e-05 sec

Pareil selon le type de multiplication on à : 0.00011775 sec, 0.002132sec et 0.019539 sec

3 PALISSADE (OPENFHE)

Voici le github de openfhe : https://github.com/openfheorg/openfhe-development/tree/main

Lorsque l'on lance le benchmark present ici :

benchmark: https://github.com/openfheorg/openfhe-development/blob/main/docker/benchmark.sh

```
root@68371edf5d1c:/openfhe-development/docker# ./benchmark.sh
./benchmark.sh: line 15: /var/www/html/benchmark.html: No such file or directory
./benchmark.sh: line 16: /var/www/html/benchmark.html: No such file or directory
Running /openfhe-development/build/bin/benchmark/IntegerMath
./benchmark.sh: line 20: /var/www/html/benchmark.html: No such file or directory
tee: /var/www/html/benchmark.html: No such file or directory
tee: /var/www/html/benchmark.html: No such file or directory
2023-12-09T18:42:39+00:00
Running /openfhe-development/build/bin/benchmark/IntegerMath
Run on (12 X 4500 MHz CPU s)
CPU Caches:
  L1 Data 48 KiB (x6)
  L1 Instruction 32 KiB (x6)
  L2 Unified 1280 KiB (x6)
  L3 Unified 12288 KiB (x1)
Load Average: 0.39, 1.20, 0.79
***WARNING*** CPU scaling is enabled, the benchmark real time measurements may be noi
-----------------------------------------------------------------------------------
Benchmark                                          Time             CPU   Iterations
-----------------------------------------------------------------------------------
BM_BigInt_constants<M2Integer>                 0.014 us        0.014 us     47865502
BM_BigInt_constants<M4Integer>                 0.013 us        0.013 us     55762592
BM_BigInt_constants<NativeInteger>             0.000 us        0.000 us   1000000000
BM_BigInt_small_variables<M2Integer>           0.186 us        0.186 us      3755078
BM_BigInt_small_variables<M4Integer>           0.226 us        0.226 us      3084161
BM_BigInt_small_variables<NativeInteger>       0.000 us        0.000 us   1000000000
BM_BigInt_large_variables<M2Integer>            1.22 us         1.22 us       572738
BM_BigInt_large_variables<M4Integer>            1.38 us         1.38 us       507317
BM_BigInt_Add<M2Integer>/Small:0               0.023 us        0.023 us     30057048
BM_BigInt_Add<M2Integer>/Large:1               0.026 us        0.026 us     26948869
BM_BigInt_Add<M4Integer>/Small:0               0.020 us        0.020 us     35227313
BM_BigInt_Add<M4Integer>/Large:1               0.024 us        0.024 us     29769746
BM_BigInt_Add<NativeInteger>/Small:0           0.000 us        0.000 us   1000000000
BM_BigInt_Addeq<M2Integer>/Small:0             0.120 us        0.120 us      5846150
BM_BigInt_Addeq<M2Integer>/Large:1             0.613 us        0.613 us      1130656
BM_BigInt_Addeq<M4Integer>/Small:0             0.143 us        0.143 us      4852673
BM_BigInt_Addeq<M4Integer>/Large:1             0.692 us        0.692 us      1009532
BM_BigInt_Addeq<NativeInteger>/Small:0         0.004 us        0.004 us    194531624
BM_BigInt_Mult<M2Integer>/Small:0              0.042 us        0.042 us     16706535
BM_BigInt_Mult<M2Integer>/Large:1              0.091 us        0.091 us      7788890
BM_BigInt_Mult<M4Integer>/Small:0              0.051 us        0.051 us     13725158
BM_BigInt_Mult<M4Integer>/Large:1              0.095 us        0.095 us      7399080
BM_BigInt_Mult<NativeInteger>/Small:0          0.000 us        0.000 us   1000000000
BM_BigInt_Multeq<M2Integer>/Small:0            0.164 us        0.164 us      4263224
```

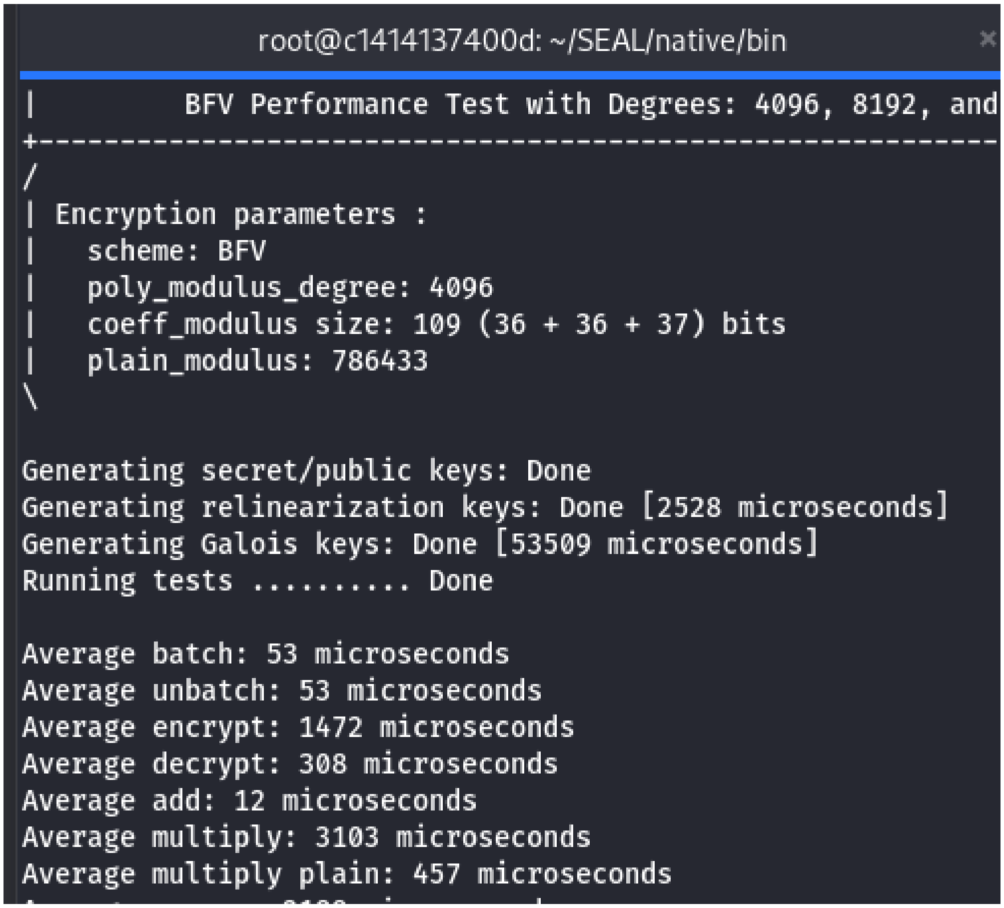
On a beaucoups d'informations , ce qui nous interesse est le bigInt addition et multiplication

Temps addition: 0.020 micro-seconde (2×10-8 secondes) pour BM_BigInt_Add<M4Integer>/Small

Temps multiplication: 0.051 micro-seconde (5.1×10-8 secondes) pour BM_BigInt_Mult<M4Integer>/Small

4 SEAL

Voici une partie du code de SEAL qui teste l'addition et la multiplication et sa performance qui est disponible dans le github:
https://github.com/microsoft/SEAL/blob/3.4.0/native/examples/6_performance.cpp



Voici la partie du code qui nous interesse

```
[Add]
        /*We create two ciphertexts and perform a few additions with them.
        */
        Ciphertext encrypted1(context);
        encryptor.encrypt(encoder.encode(static_cast<uint64_t>(i)), encrypted1);
        Ciphertext encrypted2(context);
        encryptor.encrypt(encoder.encode(static_cast<uint64_t>(i + 1)), encrypted2);
        time_start = chrono::high_resolution_clock::now();
        evaluator.add_inplace(encrypted1, encrypted1);
        evaluator.add_inplace(encrypted2, encrypted2);
        evaluator.add_inplace(encrypted1, encrypted2);
        time_end = chrono::high_resolution_clock::now();
        time_add_sum += chrono::duration_cast<
            chrono::microseconds>(time_end - time_start);

        /*
        [Multiply]
        We multiply two ciphertexts. Since the size of the result will be 3,
        and will overwrite the first argument, we reserve first enough memory
        to avoid reallocating during multiplication.
        */
        encrypted1.reserve(3);
        time_start = chrono::high_resolution_clock::now();
        evaluator.multiply_inplace(encrypted1, encrypted2);
        time_end = chrono::high_resolution_clock::now();
        time_multiply_sum += chrono::duration_cast<
```

```
chrono::microseconds>(time_end - time_start);
```

Apres l'execution du benchmark (qui etait déja present dans le docker) que l'on viens de voir on en conclu que l'addition prend 1,2e-5 secondes et la multiplication prend 0,003103 secondes si la multiplication est simple elle se fait en : 0,000457sec

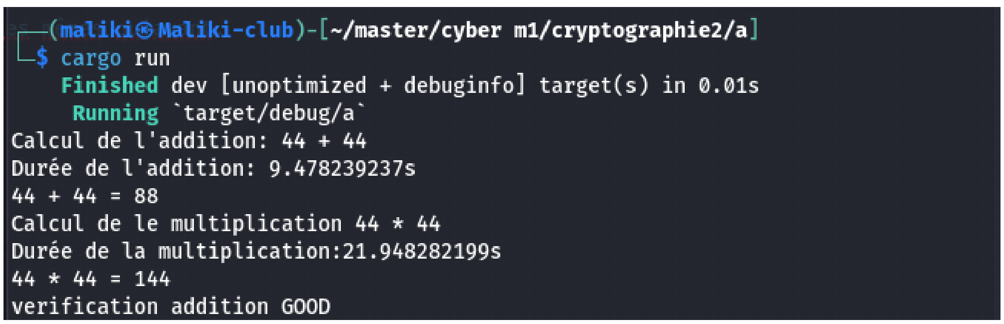
5 TFHE

Voici la documentation de tfhe : https://docs.zama.ai/tfhe-rs/getting-started/installation

Le code pour l'addition et la multiplication est disponible ici : https://docs.zama.ai/tfhe-rs/getting-started/quick_start

J'ai juste ajouté des fonctions qui permettent de calculer le temps d'execution des 2 fonctions.

TFHE est le pire élève comme on peut le voir après le code modifier disponible dans la documentation.



l'addition prend 9.478239237 sec

la multiplication prend 21.948282199 sec

6 Conclusion

Concrete :

- Addition : 0.03432059288 sec

- Multiplication : 0.0317943096 sec

Helib :

- Addition : 5.97561e-05 sec

- Multiplication : 0.00011775 sec

OPENFHE:

- Addition : 2×10-8 sec

- Multiplication : 5.1×10-8 sec

SEAL

- Addition : 1,2e-5 sec

- Multiplication : 0,000457 sec

TFHE :

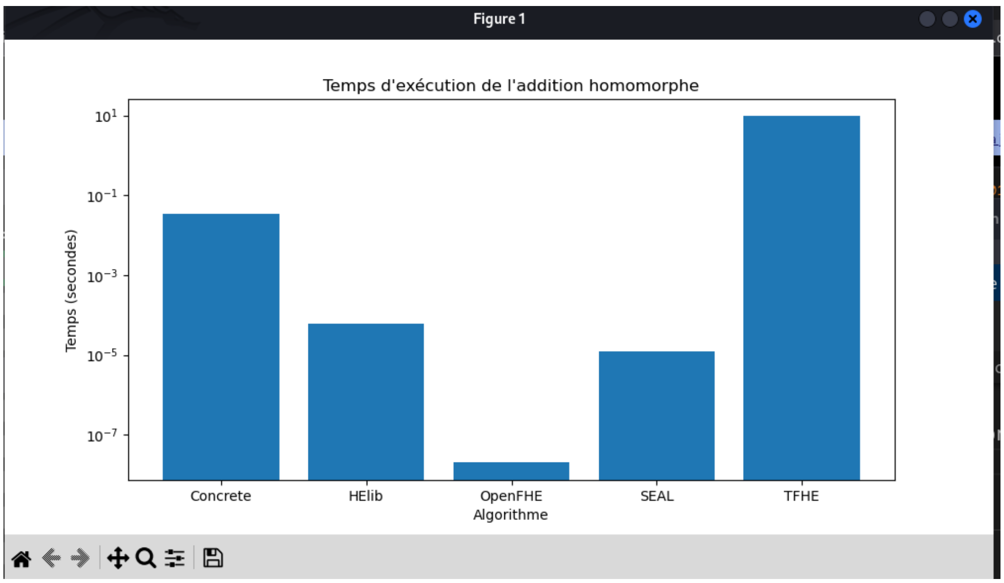
- Addition : 9.478239237 sec

- Multiplication : 21.948282199 sec

nb: Code pour le graphique en python disponible dans le dossier rendu

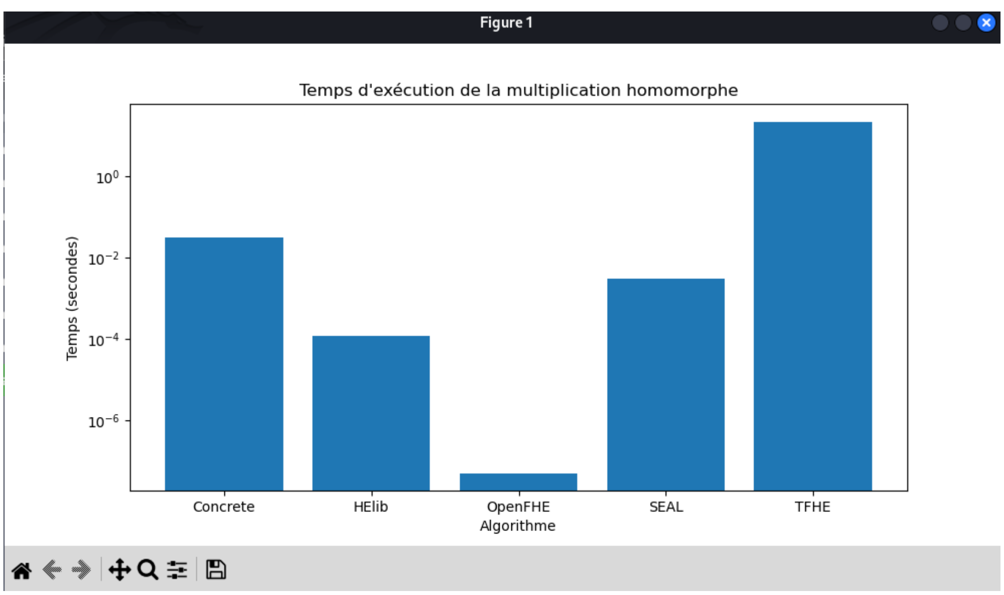
RESULTATS:

Temps execution de la multiplication homomorphe:



Pour l'addition : 1er : openfhe ; 2eme SEAL ; 3 eme HElib; 4eme concrete et 5 eme TFHE (rust)

Temps execution de la multiplication homomorphe:



Pour la multiplication : 1er Openfhe , 2eme HElib ; 3eme SEAL ; 4 eme concrete ; 5eme TFHE (rust)

On en conclu que le meilleur algorithme est OPENFHE c'est le plus rapide, suivit de HELIB ou SEAL , puis parmis les pire on à concrete et surtout TFHE (rust)