Homomorphic Encryption - A Survey and Comparison Of Libraries Using Existing Schemes

Abhijnan Bajpai *Prof. Sivaraman Eswaran* *K Gyanishtaa*
SRN: PES1UG19CS014 SRN: PES1UG19CS175

Computer Science and Engineering, PES University, Bangalore, India

I. Introduction:

Since the advent of computational devices, every bit of data has become immensely crucial for communication. Communication could be between machines, machine to human or between two humans. These interactions must not be intercepted by a third party. Achieving such privacy is not easily possible in human to human interactions. Even in a private space, one can never ensure privacy. In communications that involve computers, "walls have ears" as a phrase doesn't hold out.

Cryptography is the field dedicated to ensuring a secure exchange of information between computers and, as a result, securing the communications between humans too. As communication security over an insecure channel has gotten better, the next step is securing computations delegated to untrusted systems.

The 21st century is called the century of data. Humans are generating information at an unprecedented scale. To ensure faster computations and reduce latencies, using faster external computers for large computations has become a necessary evil. These external computers are often hard to verify and end up being used as a medium to steal huge chunks of data.

Rivest et al. in 1978 proposed to solve this issue through homomorphic encryption [1]. The homomorphic encryption system is a crypto method that allows calculations to be performed on information without decrypting it. The homomorphically coded web search engine, for example, would bring in encrypted search statements and analyze them with the encrypted list of the network. Or the homomorphically coded business information stored in the cloud could permit users to take how much money the employee earned in the second quarter of 2013. But it could receive the encrypted employee name and create the encrypted response, avoiding the privacy issues that normally plague on-line companies that deal with such sensitive information.

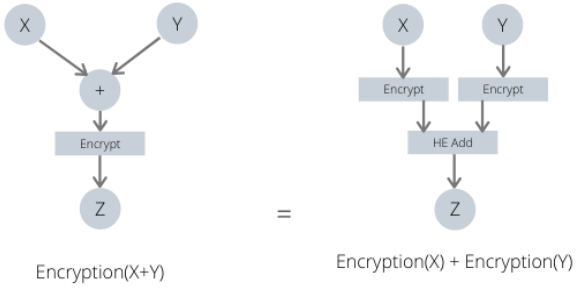


Fig 1. Basic flowchart depicting homomorphic encryption [2]

II. Homomorphic encryption and SEAL

The following is the most common definition. Let M (resp., C) denote the set of the plaintexts (resp., ciphertexts). An encryption scheme is said to be homomorphic if for any given encryption key k the encryption function E satisfies [3]:

$$\forall m_1, m_2 \in \mathcal{M}, \quad E(m_1 \odot_{\mathcal{M}} m_2) \longleftarrow E(m_1) \odot_{\mathcal{C}} E(m_2)$$

for some operators in M and C in C, where ⟵ means "can be directly computed from," that is, without any intermediate decryption.

Plaintext elements in SEAL are polynomials in Rt, just as in textbook-BFV. Ciphertexts in SEAL are tuples of polynomials in Rq of length at least 2. This is a difference to textbook-BFV, where the ciphertexts are always in Rq × Rq.

The goal of relinearization is to decrease the size of the ciphertext back to (at least) 2 after it has been increased by multiplications.In other words, given a size k + 1 ciphertext that can be decrypted, relinearization is supposed to produce a ciphertext of size k, or – when applied repeatedly – of any size at least 2, that can be decrypted using a smaller degree decryption function to yield the same result. This conversion will require a so-called evaluation key (or keys) to be given to the evaluator, as we will explain below [4]:

In BFV, suppose we have a size 3 ciphertext (c_0, c_1, c_2) that we want to convert into a size 2 ciphertext (c_0', c_1') that decrypts to the same result. Suppose we are also given a pair $\texttt{evk} = \left([-(as+e)+s^2]_q, a\right)$, where $a \xleftarrow{\$} R_q$, and $e \leftarrow \chi$. Now set $c_0' = c_0 + \texttt{evk}[0]c_2$, $c_1' = c_1 + \texttt{evk}[1]c_2$, and define the output to be the pair (c_0', c_1'). Interpreting this as a size 2 ciphertext and decrypting it yields

$$c_0' + c_1' s = c_0 + (-(as+e)+s^2)c_2 + c_1 s + a c_2 s = c_0 + c_1 s + c_2 s^2 - e c_2 \,.$$

III. BFV Scheme

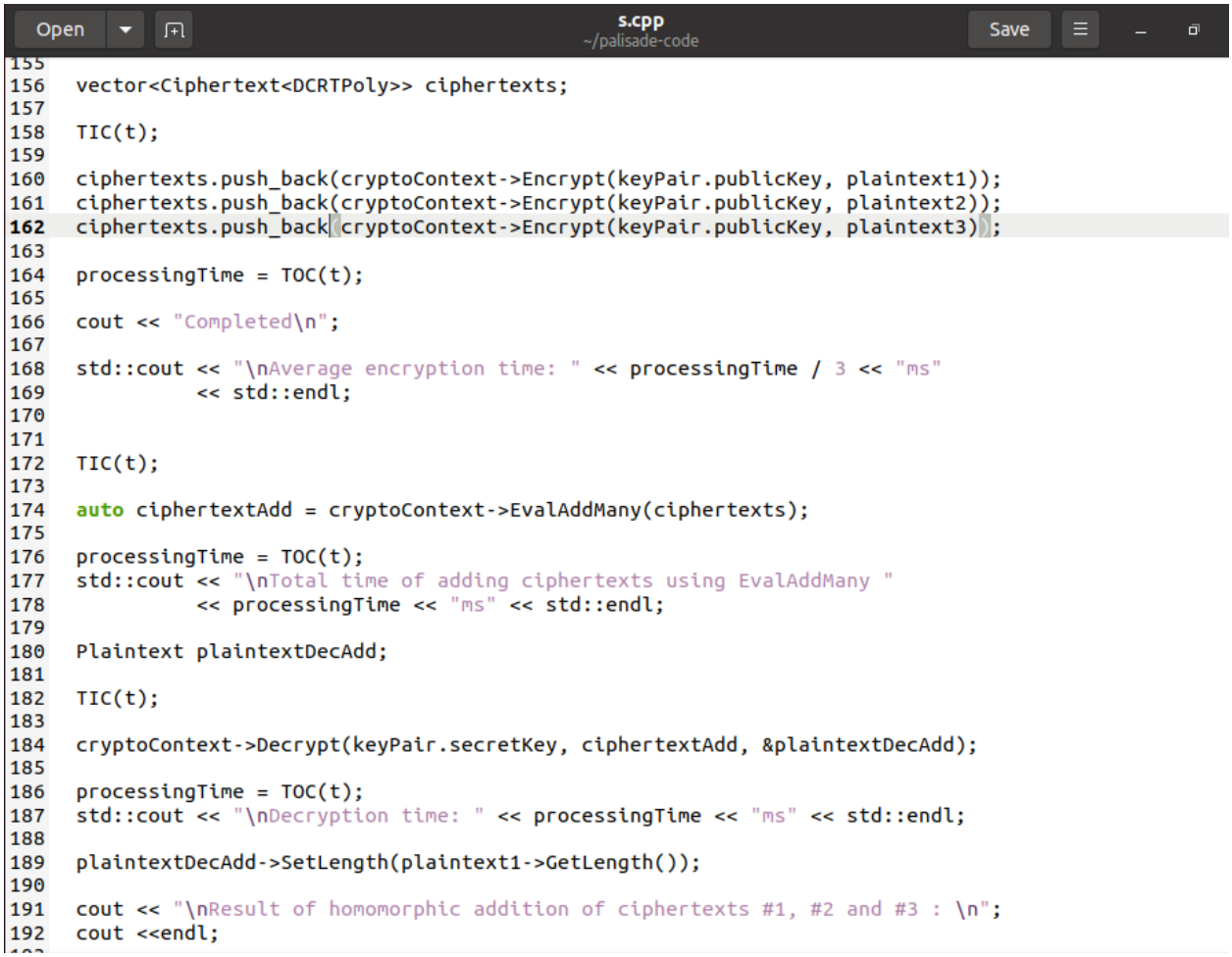
The Fan-Vercauteren (FV) scheme, (aka BFV scheme) is considered as one of the second generation of FHE schemes that is constructed based on the Ring-Learning with Errors (RLWE) problem [LPR13]. BFV is instantiated over two rings: 1) the plaintext ring which includes encodings of unencrypted or intelligible messages and 2) the ciphertext ring which includes encrypted messages. Similar to any other FHE scheme, BFV allows an untrusted party to induce meaningful computation over encrypted data without access to the decryption key. This is possible due to the homomorphism property which offers a map (or function) between the plaintext and ciphertext spaces that preserves the operations in these two spaces.

IV. Implementation and Comparison of Homomorphic Encryption with BFV Scheme Using PALISADE and SEAL Open-Source Libraries

PALISADE

The homomorphic multiplication and addition is performed on the vectors of integers using the BFV scheme. The parameters for the encrypted computation are set, this includes setting Plaintext and Coefficient Modulus used by the BFV scheme. The key generators create the secret key. Initially, the integer vectors are made the plaintexts, which will be used for encryption, thus resulting in the ciphertexts that will be used for computation.


```
131
132   std::vector<int64_t> vectorOfInts1 = {979, 209, 679, 201, 651, 726, 634, 433, 716, 735, 683, 164,
    268, 149, 386, 783, 139, 113, 505, 684, 440, 232, 488, 249, 350, 617, 198, 797, 381, 243, 809, 967,
    267, 236, 531, 920, 399, 948, 675, 468, 996, 512, 610, 174, 794, 710, 829, 426, 362, 127};
133   Plaintext plaintext1 = cryptoContext->MakePackedPlaintext(vectorOfInts1);
134
135   std::vector<int64_t> vectorOfInts2 = {773, 998, 815, 469, 724, 211, 381, 217, 728, 509, 991, 400,
    324, 587, 233, 960, 590, 202, 846, 146, 126, 699, 831, 678, 831, 547, 363, 726, 165, 668, 779, 971,
    237, 825, 183, 687, 424, 422, 900, 328, 165, 412, 294, 384, 349, 342, 674, 374, 601, 888};
136   Plaintext plaintext2 = cryptoContext->MakePackedPlaintext(vectorOfInts2);
137
138   std::vector<int64_t> vectorOfInts3 = {452, 992, 893, 804, 620, 682, 150, 307, 328, 403, 324, 246,
    346, 477, 626, 570, 923, 260, 978, 904, 761, 532, 593, 634, 520, 625, 342, 555, 411, 762, 842, 498,
    736, 416, 673, 261, 548, 835, 718, 253, 925, 119, 386, 333, 418, 503, 495, 865, 439, 825};
139   Plaintext plaintext3 = cryptoContext->MakePackedPlaintext(vectorOfInts3);
140
141
142
143   cout << "\nOriginal Plaintext #1: \n";
144   cout << plaintext1 << endl;
145
146   cout << "\nOriginal Plaintext #2: \n";
147   cout << plaintext2 << endl;
148
149   cout << "\nOriginal Plaintext #3: \n";
150   cout << plaintext3 << endl;
151
152
```




```
155
156  vector<Ciphertext<DCRTPoly>> ciphertexts;
157
158  TIC(t);
159
160  ciphertexts.push_back(cryptoContext->Encrypt(keyPair.publicKey, plaintext1));
161  ciphertexts.push_back(cryptoContext->Encrypt(keyPair.publicKey, plaintext2));
162  ciphertexts.push_back(cryptoContext->Encrypt(keyPair.publicKey, plaintext3));
163
164  processingTime = TOC(t);
165
166  cout << "Completed\n";
167
168  std::cout << "\nAverage encryption time: " << processingTime / 3 << "ms"
169            << std::endl;
170
171
172  TIC(t);
173
174  auto ciphertextAdd = cryptoContext->EvalAddMany(ciphertexts);
175
176  processingTime = TOC(t);
177  std::cout << "\nTotal time of adding ciphertexts using EvalAddMany "
178            << processingTime << "ms" << std::endl;
179
180  Plaintext plaintextDecAdd;
181
182  TIC(t);
183
184  cryptoContext->Decrypt(keyPair.secretKey, ciphertextAdd, &plaintextDecAdd);
185
186  processingTime = TOC(t);
187  std::cout << "\nDecryption time: " << processingTime << "ms" << std::endl;
188
189  plaintextDecAdd->SetLength(plaintext1->GetLength());
190
191  cout << "\nResult of homomorphic addition of ciphertexts #1, #2 and #3 : \n";
192  cout <<endl;
```

The ‘ciphertexts’ contain all the plaintexts encrypted. The Average Time for encryption is calculated when the function *‘EvalAddMany’* is used to add all the ciphertexts, namely ciphertext 1, ciphertext 2 and ciphertext 3 corresponding to the plaintext 1, plaintext 2 and plaintext 3 of their respective integer vectors. The Decryption time is also calculated while decrypting the computed encrypted data using the secret key back to integers.

```
196
197   TIC(t);
198
199   auto ciphertextMult = cryptoContext->EvalMultMany(ciphertexts);
200
201   processingTime = TOC(t);
202
203   cout << "Completed\n";
204
205   std::cout << "\nTotal time of multiplying 3 ciphertexts using EvalMultMany: "
206             << processingTime << "ms" << std::endl;
207
208   Plaintext plaintextDecMult3;
209
210   cryptoContext->Decrypt(keyPair.secretKey, ciphertextMult,
211                          &plaintextDecMult3);
212
213   plaintextDecMult3->SetLength(plaintext1->GetLength());
214
215   cout << "\nResult of 2 homomorphic multiplications: \n";
216   cout << endl;
217
218
219   cout << "\n Depth-3 multiplication with relinearization  ";
220
221   TIC(t);
222
223   auto ciphertextMult12 =
224       cryptoContext->EvalMultAndRelinearize(ciphertexts[0], ciphertexts[1]);
225
226   auto ciphertextMult123 =
227       cryptoContext->EvalMultAndRelinearize(ciphertextMult12, ciphertexts[2]);
228 processingTime = TOC(t);
```

C++ ▾ Tab Width: 8 ▾ Ln 219, Col 16 ▾ INS

The homomorphic multiplication is performed using the '*EvalMultMany*' function which takes in the ciphertexts as the parameter and outputs the encrypted result to which the decryptor decrypts to the plaintext with the secret key and the time is calculated. The *'EvalMultAndRelinearize*' function reduces the length of the ciphertext produced after each multiplication of two ciphertexts, through which the further computation will get lesser length of ciphertext for computation(in this case, multiplication).

```
231
232   std::cout << "Time of multiplying 2 ciphertexts with relinearization: "
233             << processingTime << "ms" << std::endl;
234   Plaintext plaintextDecMult123;
235
236   cryptoContext->Decrypt(keyPair.secretKey, ciphertextMult123,
237                          &plaintextDecMult123);
238
239   plaintextDecMult123->SetLength(plaintext1->GetLength());
240
241   cout << "\nResult of 2 EvalMultAndRelinearize, 3 homomorphic multiplications: \n";
242   cout << endl;
243
244
245   // Homomorphic multiplication without relinearization
246
247   cout << "\nDepth-3 multiplication without relinearization...";
248 TIC(t);
249   ciphertextMult12 =
250       cryptoContext->EvalMultNoRelin(ciphertexts[0], ciphertexts[1]);
251   ciphertextMult123 =
252       cryptoContext->EvalMultNoRelin(ciphertextMult12, ciphertexts[2]);
253 processingTime = TOC(t);
254   cout << "Completed\n";
255
256   cryptoContext->Decrypt(keyPair.secretKey, ciphertextMult123,
257                          &plaintextDecMult123);
258
259   plaintextDecMult123->SetLength(plaintext1->GetLength());
260
261   cout << "\nResult of 2 EvalMultNoRelin, 3 homomorphic multiplications: \n";
262   cout << processingTime << "ms" << std::endl;
263
264
265
266   // Homomorphic multiplication with relinearization after each multiplication
267
268
```

The multiplication is done by availing no relinearization option through the *'EvalMultNoRelin'*. This performs no relinearization, thus the original ciphertext produced will be passed down to the next calculations without manipulating the size, this comes as a user's choice.

```
  // Homomorphic multiplication with relinearization after each multiplication

  cout << "\nDepth-3 multiplication with relinearization after each multiplication...";

  TIC(t);

  ciphertextMult12 = cryptoContext->EvalMult(ciphertexts[0], ciphertexts[1]);


  ciphertextMult123 = cryptoContext->EvalMult(ciphertextMult12, ciphertexts[2]);
 processingTime = TOC(t);
  cout << "Completed\n";
  std::cout << "Time of multiplying 2 ciphertexts with relinearization: "
            << processingTime << "ms" << std::endl;

  cryptoContext->Decrypt(keyPair.secretKey, ciphertextMult123,
                         &plaintextDecMult123);

  plaintextDecMult123->SetLength(plaintext1->GetLength());

  cout << "\nResult of 2 EvalMults, homomorphic multiplications: \n";


  return 0;
}
```

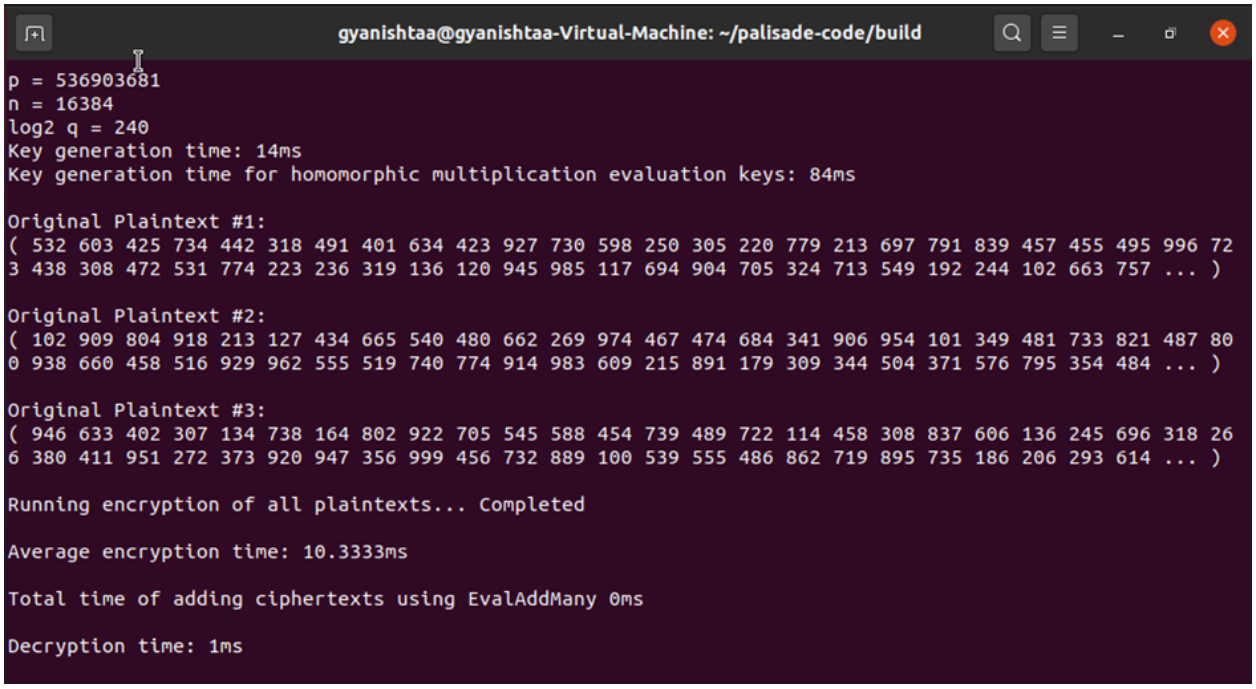
C++ Tab Width: 8 Ln 268, Col 1 INS

The *'EvalMult'* implicitly performs relinearization after every multiplication resulting in ciphertext. Thus, this happens to be one of the ways where relinearization can be done without explicitly using any function specifically.

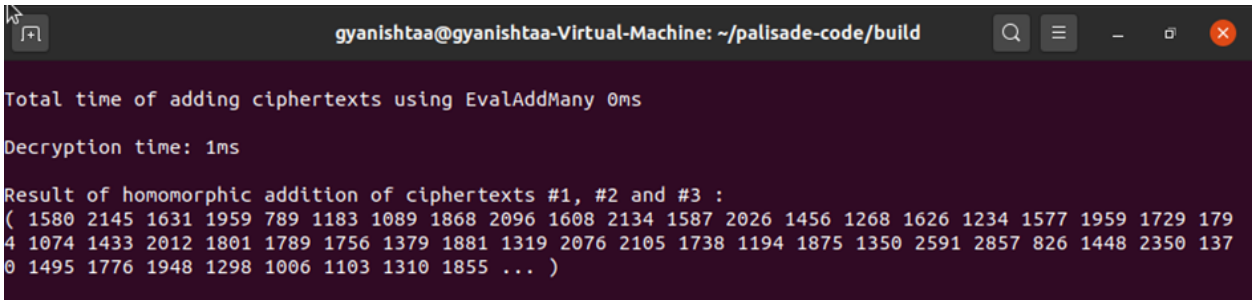
After cloning the library's github repo to the local machine, other system prerequisites/dependencies were downloaded. The binary files are built inside the directory specified by the user and the PALISADE library is built and installed in the system through the commands *cmake, make, make install.*

The

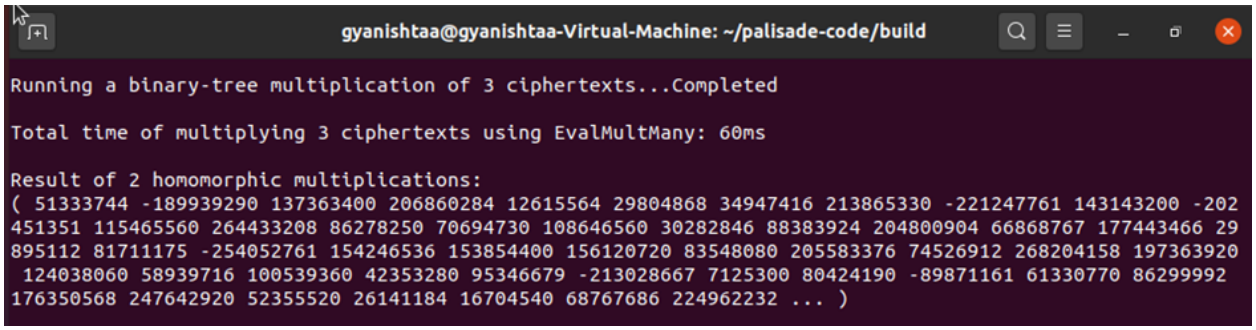
```
gyanishtaa@gyanishtaa-Virtual-Machine: ~/palisade-code/build
176350568 247642920 52355520 26141184 16704540 68767686 224962232 ... )
gyanishtaa@gyanishtaa-Virtual-Machine:~/palisade-code/build$ cd ..
gyanishtaa@gyanishtaa-Virtual-Machine:~/palisade-code$ gedit s.cpp
gyanishtaa@gyanishtaa-Virtual-Machine:~/palisade-code$ cd build
gyanishtaa@gyanishtaa-Virtual-Machine:~/palisade-code/build$ cmake ..
-- Configuring done
-- Generating done
-- Build files have been written to: /home/gyanishtaa/palisade-code/build
gyanishtaa@gyanishtaa-Virtual-Machine:~/palisade-code/build$ make
Scanning dependencies of target test
[ 50%] Building CXX object CMakeFiles/test.dir/s.cpp.o
[100%] Linking CXX executable test
[100%] Built target test
gyanishtaa@gyanishtaa-Virtual-Machine:~/palisade-code/build$ ./test
```



The parameters set are shown above. The three vectors (or lists) of integers are made the Plaintexts and have corresponding integers (in this case, 50 integers in each vector) lists along with the average encryption time.



The time for running the *EvalAddMany* function is calculated along with the decryption time (to convert the ciphertext back to plaintext of integers).



The time needed to run the *EvalMultMany* is observed which performs two homomorphic multiplications that includes all ciphertexts.

```
Running a depth-3 multiplication with relinearization until the very end...Completed
Time of multiplying 2 ciphertexts with    relinearization: 52ms

Result of 2 EvalMultAndRelinearize, 3 homomorphic multiplications:
( 51333744 -189939290 137363400 206860284 12615564 29804868 34947416 213865330 -221247761 143143200 -202
451351 115465560 264433208 86278250 70694730 108646560 30282846 88383924 204800904 66868767 177443466 29
895112 81711175 -254052761 154246536 153854400 156120720 83548080 205583376 74526912 268204158 197363920
 124038060 58939716 100539360 42353280 95346679 -213028667 7125300 80424190 -89871161 61330770 86299992
176350568 247642920 52355520 26141184 16704540 68767686 224962232 ... )
```

The time needed to run the *EvalMultAndRelinearize* is observed which performs two homomorphic multiplications that includes all ciphertexts but are reduced in size.

```
Running a depth-3 multiplication without relinearization...Completed

Result of 2 EvalMultNoRelin, 3 homomorphic multiplications:
( 51333744 -189939290 137363400 206860284 12615564 29804868 34947416 213865330 -221247761 143143200 -202
451351 115465560 264433208 86278250 70694730 108646560 30282846 88383924 204800904 66868767 177443466 29
895112 81711175 -254052761 154246536 153854400 156120720 83548080 205583376 74526912 268204158 197363920
 124038060 58939716 100539360 42353280 95346679 -213028667 7125300 80424190 -89871161 61330770 86299992
176350568 247642920 52355520 26141184 16704540 68767686 224962232 ... )47ms
```

The time needed to run the *EvalMultNoRelin* is observed which performs two homomorphic multiplications that includes all ciphertexts.

```
Running a depth-3 multiplication with relinearization after each multiplication...Completed
Time of multiplying 2 ciphertexts with relinearization: 50ms

Result of 2 EvalMults, homomorphic multiplications:
( 51333744 -189939290 137363400 206860284 12615564 29804868 34947416 213865330 -221247761 143143200 -202
451351 115465560 264433208 86278250 70694730 108646560 30282846 88383924 204800904 66868767 177443466 29
895112 81711175 -254052761 154246536 153854400 156120720 83548080 205583376 74526912 268204158 197363920
 124038060 58939716 100539360 42353280 95346679 -213028667 7125300 80424190 -89871161 61330770 86299992
176350568 247642920 52355520 26141184 16704540 68767686 224962232 ... )
gyanishtaa@gyanishtaa-Virtual-Machine:~/palisade-code/build$
```

The time needed to run the *EvalMult twice* is observed which performs two homomorphic multiplications step-wise that includes the ciphertexts and are also relinearized.
Likewise, for a variable number of integers (500,1500,2500,3500 and 4500) the following values are noted.

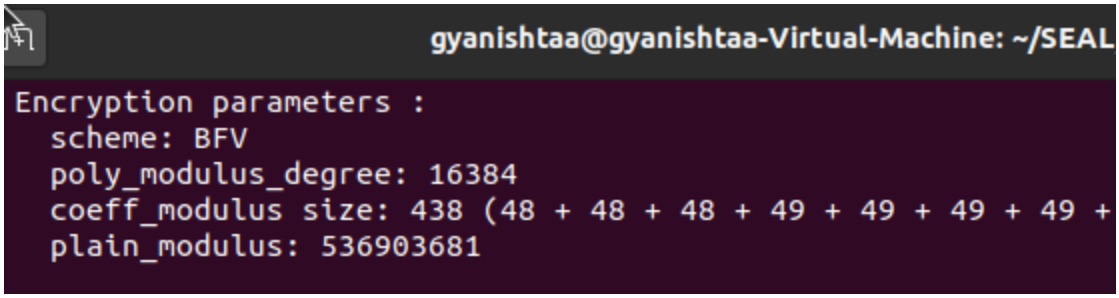
Plaintext Vector Size (Number of elements)	Average Encryption Time (in ms)	Decryption Time (in ms)	Adding Ciphertexts Time	Multiplying with EvalMultMany	Multiplying With Relinearization	Multiplying Without Relinearization	Multiplying step-wise with Relinearization
500	10.33	1	0	49	51	45	50
1500	11.67	1	6	54	65	56	64
2500	12.34	1	7	60	58	49	63
3500	13.33	1	0	56	64	62	60
4500	13.67	1	0	62	59	48	60

SEAL

The SEAL library was developed by the Microsoft Research Group, it implements the BFV scheme as well. The BFV scheme which is used for performing homomorphic arithmetic operations on encrypted data (usually integers).

The github repo was cloned and under the directory in native/src, the *cmake* command was used to build the library and *make -j* and *sudo make install* to install.

The parameters below for the BFV scheme are set as the same as those values used for PALISADE library implementation, the plaintext and coefficient modulus.




```
218     cout << endl;
219     print_line(__LINE__);
220     Plaintext plainmat1,plainmat2,plainmat3,addresult,multresult;
221     vector<uint64_t> addresultvec,multresultvec;
222     Ciphertext ciph1,addciph12,addciph,addcipher,multcipher;
223     Ciphertext ciph2,ciph3;
224
225     vector<uint64_t> vectorOfInts1 = {532, 603, 425, 734, 442, 318, 491, 401, 634, 423, 927, 730,
    598, 250, 305, 220, 779, 213, 697, 791, 839, 457, 455, 495, 996, 723, 438, 308, 472, 531, 774, 223,
    236, 319, 136, 120, 945, 985, 117, 694, 904, 705, 324, 713, 549, 192, 244, 102, 663, 757};
226      batch_encoder.encode(vectorOfInts1,plainmat1);
227
228
229      vector<uint64_t> vectorOfInts2 = {102, 909, 804, 918, 213, 127, 434, 665, 540, 480, 662, 269,
    974, 467, 474, 684, 341, 906, 954, 101, 349, 481, 733, 821, 487, 800, 938, 660, 458, 516, 929, 962,
    555, 519, 740, 774, 914, 983, 609, 215, 891, 179, 309, 344, 504, 371, 576, 795, 354, 484};
230      batch_encoder.encode(vectorOfInts2,plainmat2);
231
232
233      vector<int64_t> vectorOfInts3 = {946, 633, 402, 307, 134, 738, 164, 802, 922, 705, 545, 588,
    454, 739, 489, 722, 114, 458, 308, 837, 606, 136, 245, 696, 318, 266, 380, 411, 951, 272, 373, 920,
    947, 356, 999, 456, 732, 889, 100, 539, 555, 486, 862, 719, 895, 735, 186, 206, 293, 614};
234      batch_encoder.encode(vectorOfInts3,plainmat3);
235
236
237 //    print_matrix(vectorOfInts,1);
238
```

The three vectors containing 50 integers each (for the above instance) is taken and encoded in the plaintext variables.

```
239 auto start1 = high_resolution_clock::now();
240
241 encryptor.encrypt(plainmat1,ciph1);
242 encryptor.encrypt(plainmat2,ciph2);
243 encryptor.encrypt(plainmat3,ciph3);
244
245 auto stop1 = high_resolution_clock::now();
246
247 auto duration1 = duration_cast<microseconds>(stop1 - start1);
248
249 cout << "Average Encryption Time: "
250          << (duration1.count())/3000 << " ms" << endl;
251
252 auto start2 = high_resolution_clock::now();
253
254 evaluator.add_many({ciph1,ciph2,ciph3},addcipher);
255
256 auto stop2 = high_resolution_clock::now();
257
258
259 auto duration2 = duration_cast<microseconds>(stop2 - start2);
260
261 auto start3 = high_resolution_clock::now();
262
263 decryptor.decrypt(addcipher,addresult);
264 auto stop3  = high_resolution_clock::now();
265
266 auto duration3 = duration_cast<microseconds>(stop3- start3);
267 cout << "Decryption Time: "
268          << (duration3.count())/1000 << " ms" << endl;
269 cout << endl;
270 print_line(__LINE__);
271 batch_encoder.decode(addresult,addresultvec);
272 cout<< "Addition result with add_many"<<endl;
273 print_matrix(addresultvec,0);
274 cout << "Time taken by addmany function: "
275          << (duration2.count())/1000 << " ms" << endl;
```

The encryptor object encrypts the plaintexts generated to the ciphertexts, the average time is calculated to encrypt. Likewise, the time taken for *add_many* is noted which adds all the ciphertexts and produces the resulting ciphertext. The decryption time to decrypt the resulting ciphertext to integer is performed using *decrypt*. The time is measured in milliseconds. The decoding gives the integers values after the computations on ciphertext is performed.
Note: The duration.count value is divided by 1000, as a result it will be microseconds/1000=milliseconds.

```
278
279 RelinKeys relin_keys;
280     keygen.create_relin_keys(relin_keys);
281
282 auto start4 = high_resolution_clock::now();
283
284 evaluator.multiply_many({ciph1,ciph2,ciph3},relin_keys,multcipher);
285
286 auto stop4 = high_resolution_clock::now();
287
288 auto duration4 = duration_cast<microseconds>(stop4 - start4);
289
290 decryptor.decrypt(multcipher,multresult);
291 batch_encoder.decode(multresult,multresultvec);
292 cout<<"Multiplication result with relinearization with multiply_many" <<endl;
293 print_matrix(multresultvec,0);
294 cout << "Time taken by multiply_many function with relinearization: "
295          << (duration4.count())/1000 << " ms" << endl;
296
297 cout << endl;
298 print_line(__LINE__);
299
300 auto start5 = high_resolution_clock::now();
301 evaluator.multiply(ciph1,ciph2,multcipher);
302 evaluator.multiply_inplace(multcipher,ciph3);
303 auto stop5 = high_resolution_clock::now();
304
305 auto duration5 = duration_cast<microseconds>(stop5 - start5);
306 decryptor.decrypt(multcipher,multresult);
307 batch_encoder.decode(multresult,multresultvec);
308 cout<<"Multiplication result without relinearization with 2 multiply functions only" <<endl;
309 print_matrix(multresultvec,0);
310 cout << "Time taken by multiply function without relinearization: "
311          << (duration5.count())/1000 << " ms" << endl;
312 cout << endl;
313 print_line(__LINE__);
314 }
                                C++ ▾   Tab Width: 8 ▾      Ln 313, Col 22 ▾
```

The *multiply_many* function is used to multiply all the ciphertexts at once. The parameters passed to this function also includes relinearization keys which are used after every resulting ciphertext is obtained, the relinearization is performed on that resulting ciphertext and this will be used to multiply with the next ciphertext. This is analogous to *EvalMultAndRelinearize* from Palisade library.

On the contrary, the *multiply* function is used to multiply two ciphertexts and produce one resulting ciphertext at a time, which will then be used to multiply with next cipher without any relinearization. This function is analogous to *EvalMultNoRelin* in Palisade library.

For Plaintext Vector Size - **50,**

```
Line 219 --> Average Encryption Time: 13 ms
Decryption Time: 5 ms

Line 265 --> Addition result with add_many

    [1580,2145,1631,1959,789, ...,    [1580,2145,1631,1959,789, ...,
Time taken by addmany function: 6 ms

Line 272 --> Multiplication result with relinearization with multiply_many

    [51333744,346964391,137363400,206860284,12615564, ...,    [51333744,346964391,137363400,206860284,12
615564, ...,
Time taken by multiply_many function with relinearization: 180 ms

Line 293 --> Multiplication result without relinearization with 2 multiply functions only

    [51333744,346964391,137363400,206860284,12615564, ...,    [51333744,346964391,137363400,206860284,12
615564, ...,
Time taken by multiply function without relinearization: 146 ms
```

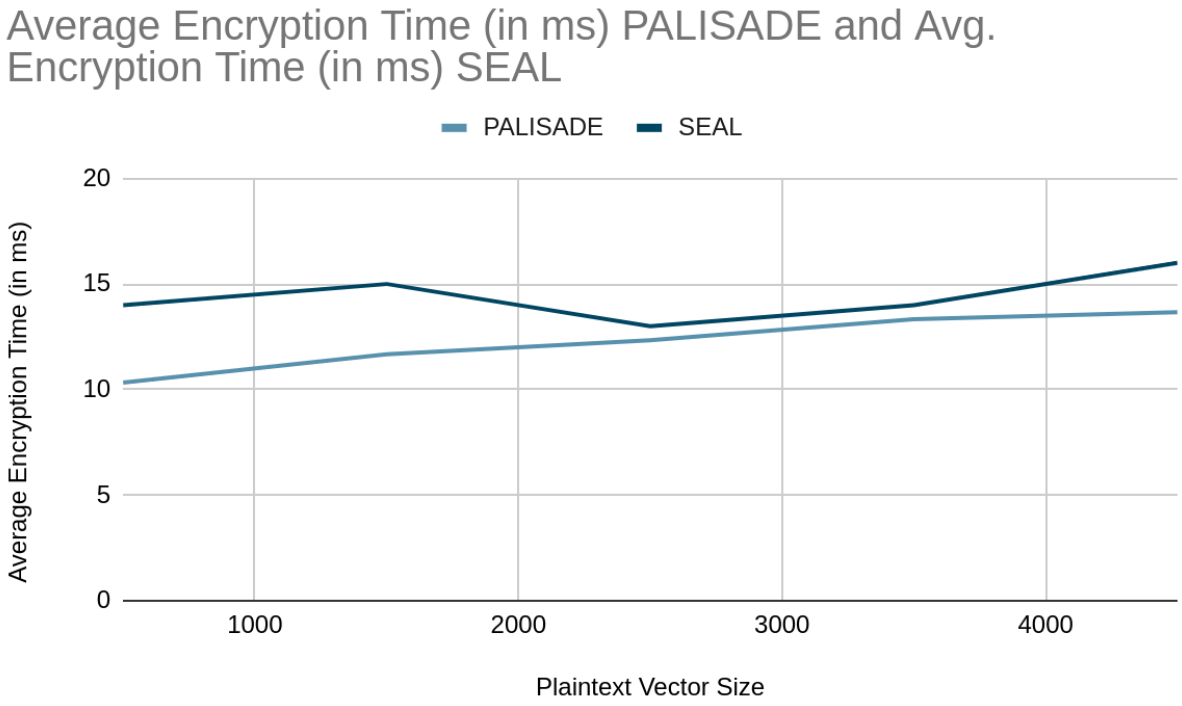
The average encryption time is observed and the decryption time taken by this library and the time taken by the *add_many, multiply_many* and *multiply* are observed and noted.

For Plaintext Vector Size - **3500,** the below observations are seen and noted in the table below for other variable vector sizes.

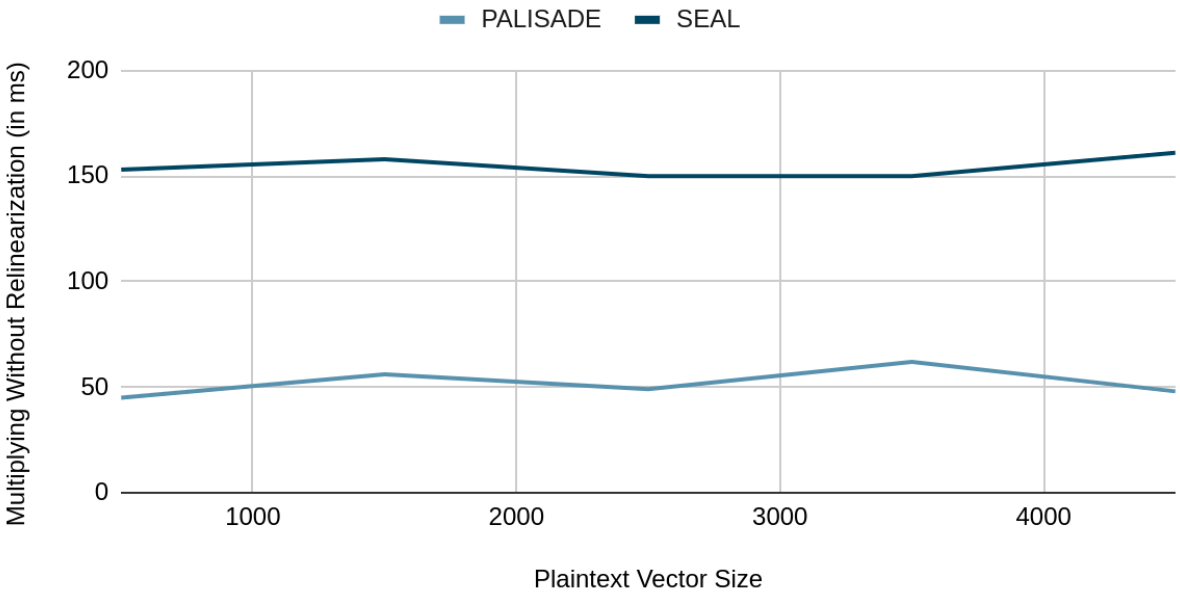
```
Line 219 --> Average Encryption Time: 14 ms
Decryption Time: 7 ms

Line 270 --> Addition result with add_many

    [1914,1449,1411,1016,1261, ...,    [1914,1449,1411,1016,1261, ...,
Time taken by addmany function: 7 ms

Line 277 --> Multiplication result with relinearization with multiply_many

    [88720280,76669504,54454400,37411696,40259700, ...,    [88720280,76669504,54454400,37411696,40259700
, ...,
Time taken by multiply_many function with relinearization: 181 ms

Line 298 --> Multiplication result without relinearization with 2 multiply functions only

    [88720280,76669504,54454400,37411696,40259700, ...,    [88720280,76669504,54454400,37411696,40259700
, ...,
Time taken by multiply function without relinearization: 150 ms
```

Plaintext Vector Size (Number of elements)	Avg. Encryption Time (in ms)	Decryption Time (in ms)	Adding ciphertexts time (in ms)	Multiplying with Relinearization (in ms)	Multiplying without Relinearization (in ms)

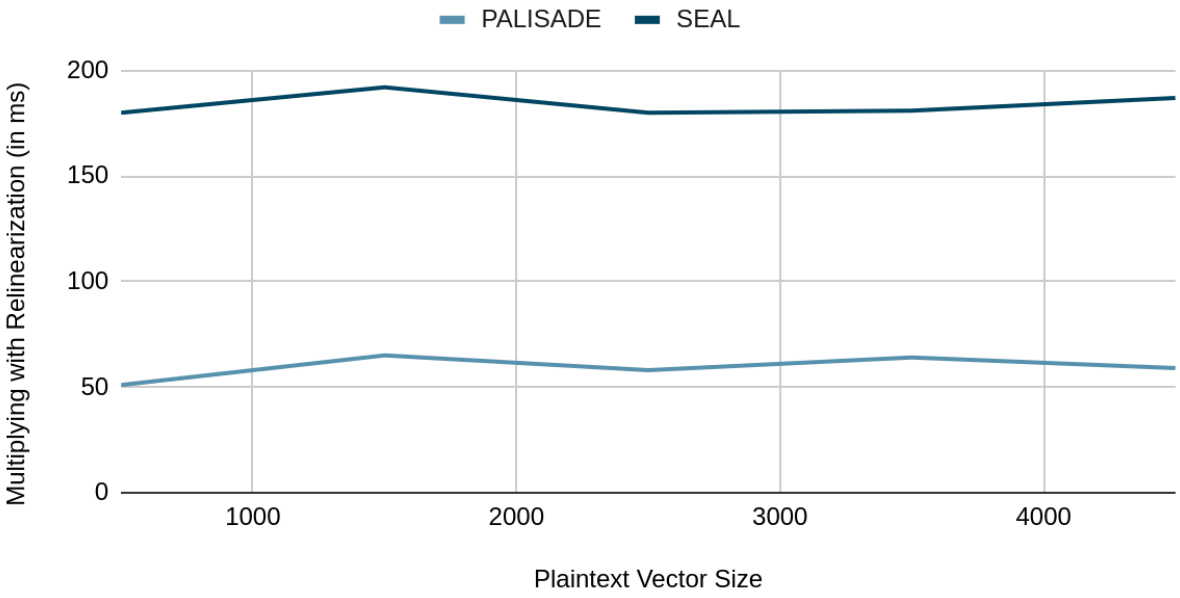
500	14	5	5	180	153
1500	15	6	9	192	158
2500	13	8	7	180	150
3500	14	7	7	181	150
4500	16	8	10	187	161



Multiplying Without Relinearization PALISADE and Multiplying without Relinearization SEAL



Multiplying With Relinearization PALISADE and Multiplying with Relinearization (in ms) SEAL



V. References

[1] R. Rivest, L. Adleman, and M. Dertouzos, “On data banks and privacy homomorphisms,” in Foundations of Secure Computation, pp. 169–177, Academic Press, 1978.

[2] Ayoub Benaissa, “Introduction to Homomorphic Encryption” in Jan 2, 2020, https://www.ayoub-benaissa.com/blog/introduction-to-homomorphic-encryption/

[3] Fontaine, Caroline, and Fabien Galand. "A survey of homomorphic encryption for nonspecialists." EURASIP Journal on Information Security 2007 (2007): 1-10.

[4] Kim Laine, “Simple Encrypted Arithmetic Library 2.3.1”, in https://www.microsoft.com/en-us/research/uploads/prod/2017/11/sealmanual-2-3-1.pdf

[5] https://github.com/microsoft/SEAL/blob/main/native/examples/2_encoders.cpp

[6]https://medium.com/privacy-preserving-natural-language-processing/homomorphic-encryption-for-beginners-a-practical-guide-part-1-b8f26d03a98a

[7] https://scholarworks.uark.edu/cgi/viewcontent.cgi?article=1074&context=csceuht

[8] https://gitlab.com/palisade/palisade-release

[9] I. Damgard and M. Jurik, “A length-flexible threshold cryp- ˚tosystem with applications,” in Proceedings of the 8th Australian Conference on Information Security and Privacy (ACISP’03), vol. 2727 of Lecture Notes in Computer Science, Wollongong, Australia, 2003.

[10] A. Adelsbach, S. Katzenbeisser, and A. Sadeghi, “Cryptology meets watermarking: detecting watermarks with minimal or zero-knowledge disclosures,” in Proceedings of the European Signal Processing Conference (EUSIPCO ’02), Toulouse, France, September 2002.

[11] B. Pfitzmann and W. Waidner, “Anonymous fingerprinting,” in Advances in Cryptology (EUROCRYPT ’97), vol. 1233 of Lecture Notes in Computer Science, pp. 88–102, Springer, New York, NY, USA, 1997.

[12] N. Memon and P. Wong, “A buyer-seller watermarking protocol,” IEEE Transactions on Image Processing, vol. 10, no. 4, pp. 643–649, 2001.

[13] C.-L. Lei, P.-L. Yu, P.-L. Tsai, and M.-H. Chan, “An efficient and anonymous buyer-seller watermarking protocol,” IEEE Transactions on Image Processing, vol. 13, no. 12, pp. 1618–1626, 2004.

[14] M. Kuribayashi and H. Tanaka, “Fingerprinting protocol for
images based on aditive homomorphic property,” IEEE Transactions on Image Processing, vol. 14, no. 12, pp. 2129–2139, 2005.

[15] V. Shoup, A Computational Introduction to Number Theory and Algebra, Cambridge University Press, 2005, http://www.shoup.net/ntb/.

[16] A. Menezes, P. Van Orschot, and S. Vanstone, Handbook of applied cryptography, CRC Press, 1997, http://www.cacr.math.uwaterloo.ca/hac/.

[17] H. Van Tilborg, Ed., Encyclopedia of Cryptography and Security, Springer, New York, NY, USA, 2005.

[18] A. Kerckhoffs, “La cryptographie militaire (part i),” Journal des Sciences Militaires, vol. 9, no. 1, pp. 5–38, 1883.

[19] A. Kerckhoffs, “La cryptographie militaire (part ii),” Journal des Sciences Militaires, vol. 9, no. 2, pp. 161–191, 1883.

[20] J. Daemen and V. Rijmen, “The block cipher RIJNDAEL,” in(CARDIS ’98), vol. 1820 of Lecture Notes in Computer Science, pp. 247–256, Springer, New York, NY, USA, 2000.

[21] J. Daemen and V. Rijmen, “The design of Rijndael,” in AES— the Advanced Encryption Standard, Informtion Security and Cryptography, Springer, New York, NY, USA, 2002.

[22] G. Vernam, “Cipher printing telegraph systems for secret wire and radio telegraphic communications,” Journal of the American Institute of Electrical Engineers, vol. 45, pp. 109–115, 1926.

[23] P. Ekdahl and T. Johansson, “A new version of the stream cipher SNOW,” in Selected Areas in Cryptography (SAC ’02), vol. 2595 of Lecture Notes in Computer Science, pp. 47–61,
Springer, New York, NY, USA, 2002.

[24] R. Rivest, A. Shamir, and L. Adleman, “A method for obtaining digital signatures and public-key cryptosystems,” Communications of the ACM, vol. 21, no. 2, pp. 120–126, 1978.

[25] T. ElGamal, “A prublic key cryptosystem and a signature scheme based on discrete logarithms,” in Advances in Cryptology (CRYPTO ’84), vol. 196 of Lecture Notes in Computer Science, pp. 10–18, Springer, New York, NY, USA, 1985.

[26] C. Shannon, “Communication theory of secrecy systems,” Bell System Technical Journal, vol. 28, pp. 656–715, 1949.

[27] M. Ajtai and C. Dwork, “A public key cryptosystem with worst-case/average-case equivalence,” in Proceedings of the 29th ACM Symposium on Theory of Computing (STOC ’97),
pp. 284–293, 1997.

[28] P. Nguyen and J. Stern, “Cryptanalysis of the Ajtai-Dwork cryptosystem,” in Advances in Cryptology (CRYPTO ’98), vol. 1462 of Lecture Notes in Computer Science, pp. 223–242,
Springer, New York, NY, USA, 1999.

[29] R. Canetti, O. Goldreich, and S. Halevi, “The random oracle model, revisited,” in Proceedings of the 30th ACM Symposium on Theory of Computing (STOC ’98), pp. 209–218, Berkeley, Calif, USA, 1998.

[30] P. Paillier, “Impossibility proofs for RSA signatures in the standard model,” in Proceedings of the RSA Conference 2007, Cryptographers’ (Track), vol. 4377 of Lecture Notes in Computer Science, pp. 31–48, San Fancisco, Calif, USA, 2007.