UNIVERSITÀ DI PISA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Laurea Triennale in Ingegneria Informatica

Misura delle prestazioni di cifratura omomorfica per servizi di localizzazione crowd

Relatore:

Prof. Pericle Perazzo

Candidato:

Alessandro Corsi

ANNO ACCADEMICO 2021/2022

"Non ho mai agito aspettando Godot
Per tutti i miei giorni aspettando Godot"

Indice

Elenco delle figure

Capitolo 1

Introduzione

La crittografia omomorfica è un tipo di crittografia a chiave pubblica che permette di effettuare, oltre alle azioni di cifratura e decifrazione, anche operazioni sui dati cifrati senza l'utilizzo della chiave privata. La sicurezza della crittografia omomorfica deriva dalla difficoltà di risolvere alcuni problemi definiti su particolari gruppi algebrici, i reticoli. Anche se lo schema usato, NTRU, non è dimostrabilmente sicuro, non sono ancora noti attacchi efficaci, neanche con algoritmi quantistici.
I crittosistemi omomorfici, se sufficientemente performanti, possono essere usati in tutti i casi in cui computazioni pesanti devono essere effettuate in cloud, come l'analisi di dati cifrati, o il loro utilizzo in modelli di machine learning.
Nel seguente lavoro di tesi viene analizzata una delle possibili applicazioni, ovvero la localizzazione di masse di utenti. È stato utilizzato un protocollo privacy-oriented con le funzionalità introdotte dalla libreria open source OpenFHE[1] e sono state realizzate alcune simulazioni per valutare le prestazioni della libreria stessa. La tesi è così strutturata:

- nel capitolo 2 vengono descritte brevemente le principali tecniche di crittografia basata sui reticoli, il tipo di crittografia alla base della crittografia omomorfica;
- nel capitolo 3 viene descritto un prototipo di applicativo utilizzato per effettuare crowd localization usando la crittografia omomorfica;
- nel capitolo 4 vengono descritti i benchmark effettuati sulla libreria OpenFHE;
- nel capitolo 5 vengono vengono tratte le conclusioni.

Tutto il codice usato è reperibile a [2].

Capitolo 2

La crittografia omomorfica

La crittografia omomorfica è una tecnica crittografica che permette di effettuare elaborazioni direttamente sui dati cifrati (*ciphertext*); si divide in due categorie principali: la crittografia completamente omomorfica (*FHE, fully homomorphic encryption*) e la crittografia parzialmente omomorfica (*PHE, partially homomorphic encryption*).
Mentre la prima permette di eseguire tutte le operazioni principali, addizione, moltiplicazione e operazioni booleane, la seconda permette di eseguirne solo una, usualmente la addizione o la moltiplicazione.
La libreria OpenFHE [1] è una libreria che implementa i principali schemi crittografici ad oggi usati per effettuare le operazioni di cifratura omomorfica, tra cui lo schema BFV, del quale sono state misurate le prestazioni.

2.1 Crittografia basata su reticoli

Mentre i primi schemi PHE erano basati sull'aritmetica modulare, come RSA, omomorfico rispetto alla moltiplicazione (infatti in RSA $E(m_1) * E(m_2) = E(m_1 * m_2)$), gli schemi FHE sono basati sulla crittografia su reticoli.
I reticoli sono strutture algebriche, in particolare sottogruppi discreti di $\mathbb{R}^n$. Data una base $\{v_1, ..., v_n\}$ dello spazio $\mathbb{R}^n$, un reticolo Δ è così definito:

$$\Delta = \sum_{i=1}^{n} a_i v_i, a_i \in \mathbb{Z}$$

Una base del reticolo si può rappresentare in forma matriciale: $\mathbf{V} = [v_1, .., v_n]$.
Il reticolo è quindi definito come $\Delta = \{\mathbf{V}x : x \in \mathbb{Z}^n\}$, ed il **determinante del reticolo** come $\det(\Delta) = |\det(\mathbf{V})|$.

2.1.1 Shortest vector problem

Esiste una classe di problemi ritenuti NP-Ardui sui reticoli, tra i quali il più noto è lo *Shortest Vector Problem* (*SVP*), che consiste nel trovare il vettore non nullo più

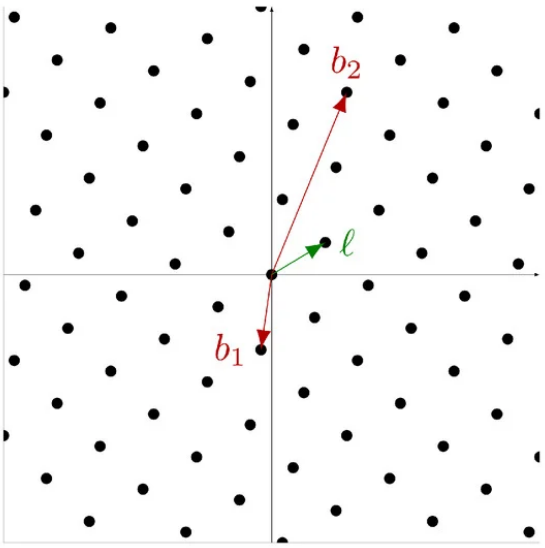

Figura 2.1: Reticolo generato dai vettori b_1 e b_2, in cui il vettore l è la soluzione allo SVP

corto appartenente al reticolo.
La difficoltà del problema dipende dalla base usata per descrivere il reticolo, in particolare con basi corte e ortogonali è facile trovare la soluzione (che spesso coincide con uno dei vettori della base), mentre è difficile con basi lunghe e non ortogonali.
La difficoltà dei problemi di SVP deriva dalla difficoltà nel trasformare una cattiva base in una buona, con una operazione chiamata *lattice reduction*.

Algoritmo LLL

L'algoritmo LLL (Lenstra-Lenstra-Lovask) permette di dare una soluzione approssimata al problema della *lattice reduction*, con un'approssimazione $\mathcal{O}(2^n)$ in un tempo polinomiale. Nessun algoritmo conosciuto permette di avere approssimazioni polinomiali in tempo polinomiale.

Worst-to-average-case connection

Nonostante esistano reticoli nei quali è più facile risolvere lo SVP, il teorema di Ajtai (1996) dimostra che i reticoli generati casualmente sono anche i reticoli più difficili da risolvere: se si sviluppasse una soluzione efficace per i reticoli generati randomicamente (il caso medio), avremmo una soluzione efficace per qualsiasi reticolo, ovvero anche per i casi peggiori.

Il collegamento tra il caso medio e il caso peggiore garantisce che è sufficiente generare una chiave privata in modo casuale per avere uno schema crittografico computazionalmente difficile da forzare.

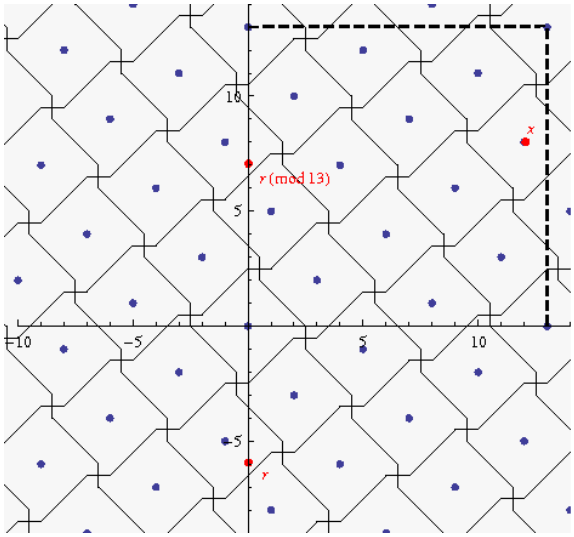

Figura 2.2: Un reticolo q-ario con $q = 13$ in $\mathbb{R}^2$

2.2 Reticoli q-ari

Un reticolo q-ario (Fig. 2.2) è un reticolo intero che contiene vettori con coordinate multiple di q: $q\mathbb{Z}^n \subseteq \Delta$.
I reticoli q-ari si ripetono secondo pattern di ipercubi: è possibile descrivere il reticolo descrivendo solo un ipercubo di lato q, e operare con punti con coordinate in $[0, q-1]$, in modo da risparmiare memoria.

L'utilizzo di reticoli q-ari permette la loro rappresentazione matriciale: un reticolo q-ario è rappresentabile con una matrice $A \in \mathbb{Z}_q^{n \times m}$, con $n < m$.
Data la matrice A, si possono individuare due reticoli: il *reticolo kernel* di A e il *reticolo immagine* di A. Il primo è usato per gli schemi crittografici basati sul problema della *Short Integer Solution* (*SIS*), il secondo per gli schemi basati su *Learning With Errors* (*LWE*).

2.2.1 Problemi usati nella crittografia basata su reticoli

Short Integer Solution problem

Data una matrice random $A \in \mathbb{Z}_q^{n \times m}$, il problema consiste nel trovare il vettore non nullo $y \in \mathbb{Z}_q^m$ tale che $Ay \equiv 0 \pmod{q}$. Il problema è equivalente allo SVP, ed ha una connessione tra worst-case e average-case: generando la matrice A in modo casuale si può ottenere un problema difficile come nel worst-case.

Learning With Errors problem

Data una matrice random $A \in \mathbb{Z}_q^{n \times m}$ e un vettore $y = A^T s + e \pmod{q} \in \mathbb{Z}_q^m$ il problema consiste nel trovare il vettore s, dato e errore gaussiano.

Se l'errore e non è grande, il problema è difficile come il *Closest Vector Problem* (*CVP*), un problema difficile della crittografia basata su reticoli che consiste nel trovare il vettore più vicino a un vettore dato.

2.3 NTRU

I problemi sui reticoli sono resi difficili dall'utilizzo di reticoli definiti in spazi vettoriali di grandi dimensioni. L'utilizzo di tali reticoli porta ad avere matrici, e quindi chiavi, di grandi dimensioni.
Per ovviare a tale problema nel 1996 è stato proposto di usare un crittosistema chiamato NTRU (*N-th degree Truncated polynomial Ring Units* [3]), realizzato usando reticoli di natura particolare.

2.3.1 Reticoli ideali

I reticoli ideali sono descritti da matrici A composte da sottomatrici quadrate $A^{(i)}$, tali che $A^{(i)} = F * a^{(i)}$, dove $*$ è l'*operatore asterisco*, e F una matrice dalla forma

$$F = \left[\begin{array}{ccc|c} 0 & ... & 0 & \pm 1 \\ \hline & & & 0 \\ & I & & ... \\ & & & 0 \end{array}\right]$$

Il vantaggio principale sono le chiavi più corte (è infatti sufficiente memorizzare i vettori $a^{(i)}$) e la velocità delle operazioni può essere velocizzata: si possono descrivere le matrici come polinomi, e si può usare la Fast-Fourier-Transform per e velocizzare le moltiplicazioni tra matrici e vettori.
Ogni polinomio di grado massimo $n-1$ è un anello: sfruttando i polinomi sono stati quindi definiti due problemi analoghi al SIS ed al LWE: il Ring-SIS ed il Ring-LWE.

Ring-SIS

Indicando con R_q il set dei polinomi definiti su $\mathbb{Z}_q$ di grado massimo $n-1$, dato un set di polinomi casuali $a^{(1)}, ..., a^{(m)} \in R_q$, il problema consiste nel trovare $y^{(1)}, ..., y^{(m)}$ tale che $a^{(1)}y^{(1)} + ... + a^{(m)}y^{(m)} = 0$.

Ring-LWE

Dato un set di polinomi casuali $a^{(1)}, ..., a^{(m)} \in R_q$, dato un set di polinomi perturbati $y^{(1)} = a^{(1)}s + e^{(1)}, ..., y^{(m)} = a^{(m)}s + e^{(m)}$, con errori polinomiali $e^{(i)}$ estratti secondo una distribuzione gaussiana, il problema consiste nel trovare il polinomio $s \in R_q$.
Lo schema crittografico BFV è basato sulla difficoltà del problema di Ring-LWE.

2.3.2 Resistenza ai calcolatori quantistici

Attualmente, non sono note vulnerabilità dello schema NTRU agli attacchi con computer quantistici, e non è vulnerabile all'algoritmo di Shor. Lo schema è stato selezionato dal NIST tra gli algoritmi finalisti per la standardizzazione della crittografia post quantistica, e il progetto PQCRYPTO dell'Unione Europea sta valutando l'adozione di una versione dimostrabilmente sicura (ma più lenta) di NTRU come standard, la Stehle–Steinfeld NTRU.

Capitolo 3

Sviluppo di un sistema di localizzazione crowd privacy-oriented

Una delle principali applicazioni dei sistemi di cifratura omomorfica è l'implementazione di servizi privacy-oriented: un server può elaborare dati cifrati provenienti dagli utenti senza la necessità di cifrarli, e preservare perciò la segretezza di tali dati. Una particolare applicazione è quella dei sistemi di localizzazione per una folla: un primo utente (Bob in 3.1) richiede la posizione media di un numero elevato di altri utenti, i quali, invece di inviare direttamente la propria posizione all'utente, la inviano cifrata al server, che calcola la media (cifrata) e la invia all'utente iniziale, il quale può decifrarla con la propria chiave privata.
In questo modo, il server può sommare le coordinate degli utenti senza conoscerle, e l'utente finale può conoscere solamente il centro di massa della folla senza conoscere le singole posizioni degli altri utenti.

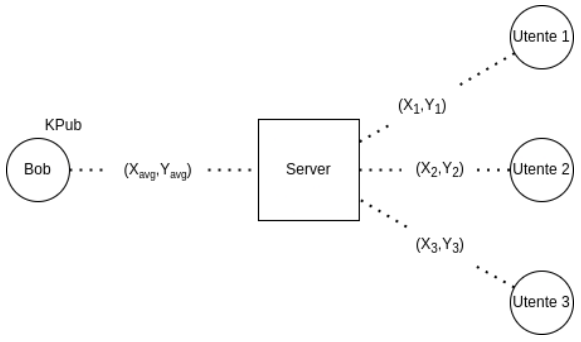

Figura 3.1: Schema dell'invio della posizione

3.1 Implementazione di un prototipo

Per sviluppare un prototipo di applicazione di localizzazione sono state usate le primitive crittografiche fornite dalla libreria OpenFHE. In particolare è stato usato lo schema crittografico *BFV* (dai nomi degli autori, *Brakerski/Fan-Vercauteren*), che permette di effettuare operazioni tra interi.
Le primitive permettono di generare una chiave, cifrare e decifrare vettori di interi, ed effettuare somme e moltiplicazioni tra ciphertext. Lo schema usa una crittografia basata su Ring-LWE.

Per simulare uno scambio di dati tra processi sono state usate le primitive che permettono di serializzare su file chiavi e crittogrammi, gestendo la sincronizzazione con semafori (la cui implementazione è in A.1)

3.1.1 Processo utente richiedente la posizione "*Bob*"

Il primo processo scrive su file la propria chiave pubblica, e attende che il server invii la somma delle coordinate degli altri utenti; infine, calcola la media, dividendo semplicemente la somma ricevuta dal server per il numero degli altri utenti.

```
1  void requestingPosition()
2  {
3      CCParams<CryptoContextBFVRNS> parameters;
4      parameters.SetPlaintextModulus(65537);
5      parameters.SetMultiplicativeDepth(2);
6
7      CryptoContext<DCRTPoly> cryptoContext = GenCryptoContext(
       parameters);
8      // Enable features that you wish to use
9      cryptoContext->Enable(PKE);
10     cryptoContext->Enable(KEYSWITCH);
11     cryptoContext->Enable(LEVELEDSHE);
12
13
14     // Serialize cryptocontext
15     if (!Serial::SerializeToFile(DATAFOLDER + "/cryptocontext.
       txt", cryptoContext, SerType::BINARY)) {
16         std::cerr << "Error writing serialization of the
       crypto context to "
17                      "cryptocontext.txt"
18                   << std::endl;
19         return;
20     }
21
22     // Key Generation
23
24     // Initialize Public Key Containers
25     KeyPair<DCRTPoly> keyPair;
26
```

```
27      // Generate a public/private key pair
28      keyPair = cryptoContext->KeyGen();
29
30
31      // Serialize the public key
32      if (!Serial::SerializeToFile(DATAFOLDER + "/key-public.txt
    ", keyPair.publicKey, SerType::BINARY)) {
33          std::cerr << "Error writing serialization of public
    key to key-public.txt" << std::endl;
34          return;
35      }
36
37      for (int i = 0; i <= numberOfProcesses; i++)
38      {
39          publicKeyExchange.release();
40      }
41      mutexFile.release();
42      mutexCipherTextCounter.release();
43      mutexEncryption.release();
44
45      endOfComputation.acquire();
46
47      Ciphertext<DCRTPoly> result;
48
49      if (Serial::DeserializeFromFile(DATAFOLDER + "/result.txt"
    , result, SerType::BINARY) == false) {
50          std::cerr << "Could not read the ciphertext" << std::
    endl;
51          return;
52      }
53
54      Plaintext ptResult;
55
56      cryptoContext->Decrypt(keyPair.secretKey, result, &
    ptResult);
57
58      // The vector will store all BFV polynomial coefs
59      // We are intersted only in the first two: X and Y sum of
    values
60
61      ptResult->SetLength(2);
62  }
```

Listing 3.1: Thread che richiede la posizione

3.1.2 Processo utente che invia la posizione

Gli altri processi, dopo aver letto la chiave pubblica, cifrano e inviano la propria posizione al server.

```
1
2  void sendingPosition(int i)
3  {
4      std::string indexString = std::to_string(i);
5
6      publicKeyExchange.acquire();
7      mutexFile.acquire();
8
9      CryptoContext<DCRTPoly> cc;
10     if (!Serial::DeserializeFromFile(DATAFOLDER + "/
     cryptocontext.txt", cc, SerType::BINARY)) {
11         std::cerr << "I cannot read serialization from " <<
     DATAFOLDER + "/cryptocontext.txt" << std::endl;
12         return;
13     }
14
15     PublicKey<DCRTPoly> pk;
16     if (Serial::DeserializeFromFile(DATAFOLDER + "/key-public.
     txt", pk, SerType::BINARY) == false) {
17         std::cerr << "Could not read public key" << std::endl;
18         return;
19     }
20
21     mutexFile.release();
22
23     // Generate random coordinates
24
25     int64_t xCoord = (rand() % 201) - 100;
26     int64_t yCoord = (rand() % 201) - 100;
27
28     std::vector<int64_t> coordinates = {xCoord, yCoord};
29     Plaintext coordinatesPlaintext = cc->MakePackedPlaintext(
     coordinates);
30
31     // Encryption benchmark
32
33     mutexEncryption.acquire();
34
35
36     auto coordinatesCiphertext = cc->Encrypt(pk,
     coordinatesPlaintext);
37
38     mutexEncryption.release();
39
40     std::string fileName = "/ciphertext" + indexString + ".txt
     ";
41
42     if (!Serial::SerializeToFile(DATAFOLDER + fileName,
     coordinatesCiphertext, SerType::BINARY)) {
43         std::cerr << "Error writing serialization of
     ciphertext 1 to ciphertext1.txt" << std::endl;
44         return;
```

```
45      }
46
47      mutexCipherTextCounter.acquire();
48
49      cipherTextLoadedInFiles++;
50      // If every user has sent his cipherText, it's possible to
        unlock the server
51      if(cipherTextLoadedInFiles == numberOfProcesses)
52          cipherTextExchange.release();
53
54      mutexCipherTextCounter.release();
55
56  }
```

Listing 3.2: Thread che invia la propria posizione

3.1.3 Processo server

Il processo server aspetta che tutti gli altri processi abbiano inviato le proprie coordinate cifrate, poi le legge e le somma (usando la FHE). Infine, invia la somma all'utente iniziale.

```
1
2  void server(){
3
4      publicKeyExchange.acquire();
5
6      mutexFile.acquire();
7
8      CryptoContext<DCRTPoly> cc;
9      if (!Serial::DeserializeFromFile(DATAFOLDER + "/
       cryptocontext.txt", cc, SerType::BINARY)) {
10         std::cerr << "I cannot read serialization from " <<
       DATAFOLDER + "/cryptocontext.txt" << std::endl;
11         return;
12     }
13
14     mutexFile.release();
15
16
17     cipherTextExchange.acquire();
18
19
20     Ciphertext<DCRTPoly> cipherTextSum;
21     if (Serial::DeserializeFromFile(DATAFOLDER + "/ciphertext1
       .txt", cipherTextSum, SerType::BINARY) == false) {
22         std::cerr << "Could not read the ciphertext" << std::
       endl;
23         return;
```

```
24      }
25
26      for(int i = 1; i < numberOfProcesses; i++){
27          std::string index = std::to_string(i + 1);
28          Ciphertext<DCRTPoly> tmpCipherText;
29          if (Serial::DeserializeFromFile(DATAFOLDER + "/
    ciphertext" + index +".txt", tmpCipherText, SerType::BINARY
    ) == false) {
30              std::cerr << "Could not read the ciphertext" <<
    std::endl;
31              return;
32          }
33
34          cipherTextSum = cc->EvalAdd(cipherTextSum,
    tmpCipherText);
35      }
36
37
38      if (!Serial::SerializeToFile(DATAFOLDER + "/result.txt",
    cipherTextSum, SerType::BINARY)) {
39          std::cerr << "Error writing serialization of
    ciperTextSum to result.txt" << std::endl;
40          return;
41      }
42
43
44      endOfComputation.release();
45  }
```

Listing 3.3: Thread server

Capitolo 4

Misura delle prestazioni di OpenFHE

Per effettuare i benchmark, è stata usata la libreria standard `std::chrono`. I tempi di esecuzione sono stati calcolati in nanosecondi.
È stato realizzato un programma in C++ in cui, viene generata una coppia chiave privata-chiave pubblica, vengono generati casualmente due vettori di coordinate, vengono cifrati e successivamente sommati e moltiplicati. Uno dei vettori risultanti viene anche decifrato, in modo da calcolare il tempo di decifrazione.
Le stesse misurazioni sono state effettuate modificando il parametro `MultiplicativeDepth` dello schema BFV. Il parametro indica il numero massimo di moltiplicazioni che si possono effettuare su un ciphertext: è necessario limitare le moltiplicazioni perché viene usato uno schema Ring-LWE e si deve evitare la crescita eccessiva degli errori gaussiani.

4.1 Raccolta dei campioni

Per raccogliere i campioni è stato utilizzato un ciclo for, in cui, ad ogni iterazione è stato misurato il tempo impiegato per generare la chiave, per cifrare un dato, per effettuare la somma e la moltiplicazione di due dati e per decifrare un dato. Ad ogni esecuzione vengono passati come parametri da linea di comando il numero di iterazioni, il modulo con il quale vengono effettuate le operazioni di somma e moltiplicazione (`PlaintextModulus`), e la `MultiplicativeDepth`. In ogni esperimento è stata usato come modulo il numero primo 65537. Il codice eseguito ad ogni iterazione è il seguente:

```
1 cryptoContext = GenCryptoContext(parameters);
2 cryptoContext->Enable(PKE);
3 cryptoContext->Enable(KEYSWITCH);
4 cryptoContext->Enable(LEVELEDSHE);
5
```

```
6  // Benchmark for key generation
7
8  auto t1 = std::chrono::high_resolution_clock::now();
9
10 KeyPair<DCRTPoly> keyPair;
11 keyPair = cryptoContext->KeyGen();
12 cryptoContext->EvalMultKeyGen(keyPair.secretKey); // Used in
       multiplication
13
14 auto t2 = std::chrono::high_resolution_clock::now();
15
16 auto int_ns_keyGen = std::chrono::duration_cast<std::chrono::
       nanoseconds>(t2 - t1);
17
18 std::ofstream keyGenBenchmark;
19
20 keyGenBenchmark.open("benchmark/keyGenTime_" + std::to_string(
       plaintextModulus) + "_" + std::to_string(
       multiplicativeDepth) + ".txt", std::ios_base::app);
21 keyGenBenchmark << int_ns_keyGen.count() << std::endl;
22
23 // Benchmark for encryption
24 int64_t xCoord = (rand() % 201) - 100;
25 int64_t yCoord = (rand() % 201) - 100;
26
27 std::vector<int64_t> coordVector = {xCoord, yCoord};
28 Plaintext coordPlaintext = cryptoContext->MakePackedPlaintext(
       coordVector);
29
30
31 auto t3 = std::chrono::high_resolution_clock::now();
32
33 auto coordCipherText = cryptoContext->Encrypt(keyPair.
       publicKey, coordPlaintext);
34
35 auto t4 = std::chrono::high_resolution_clock::now();
36
37 auto int_ns_encryption = std::chrono::duration_cast<std::
       chrono::nanoseconds>(t4 - t3);
38
39 std::ofstream encryptionBenchmark;
40
41 encryptionBenchmark.open("benchmark/encryptionTime_" + std::
       to_string(plaintextModulus) + "_" + std::to_string(
       multiplicativeDepth) + ".txt", std::ios_base::app);
42 encryptionBenchmark << int_ns_encryption.count() << std::endl;
43
44
45 int64_t xCoordAux = (rand() % 201) - 100;
46 int64_t yCoordAux = (rand() % 201) - 100;
47
48 std::vector<int64_t> coordVectorAux = {xCoordAux, yCoordAux};
```

```
49
50 Plaintext coordPlaintextAux = cryptoContext->
       MakePackedPlaintext(coordVectorAux);
51 auto coordCipherTextAux = cryptoContext->Encrypt(keyPair.
       publicKey, coordPlaintextAux);
52
53 // Benchmark for sum of two vectors
54
55 auto t5 = std::chrono::high_resolution_clock::now();
56
57 auto ciphertextSum = cryptoContext->EvalAdd(coordCipherText,
       coordCipherTextAux);
58
59 auto t6 = std::chrono::high_resolution_clock::now();
60
61 auto int_ns_sum = std::chrono::duration_cast<std::chrono::
       nanoseconds>(t6 - t5);
62
63 std::ofstream sumBenchmark;
64
65 sumBenchmark.open("benchmark/sumTime_" + std::to_string(
       plaintextModulus) + "_" + std::to_string(
       multiplicativeDepth) + ".txt", std::ios_base::app);
66 sumBenchmark << int_ns_sum.count() << std::endl;
67
68 // Benchmark for multiplication of two vectors
69
70 auto t7 = std::chrono::high_resolution_clock::now();
71
72 auto ciphertextMultiplication = cryptoContext->EvalMult(
       coordCipherText, coordCipherTextAux);
73
74 auto t8 = std::chrono::high_resolution_clock::now();
75
76 auto int_ns_mult = std::chrono::duration_cast<std::chrono::
       nanoseconds>(t8 - t7);
77
78 std::ofstream multBenchmark;
79
80 multBenchmark.open("benchmark/multTime_" + std::to_string(
       plaintextModulus) + "_" + std::to_string(
       multiplicativeDepth) + ".txt", std::ios_base::app);
81 multBenchmark << int_ns_mult.count() << std::endl;
82
83 // Benchmark for decryption
84
85 Plaintext sumPlaintext;
86 Plaintext multPlaintext;
87
88 auto t9 = std::chrono::high_resolution_clock::now();
89
```

```
90 cryptoContext->Decrypt(keyPair.secretKey, ciphertextSum, &
       sumPlaintext);
91
92 auto t10 = std::chrono::high_resolution_clock::now();
93
94 std::ofstream decryptionBenchmark;
95
96 auto int_ns_decryption = std::chrono::duration_cast<std::
       chrono::nanoseconds>(t10 - t9);
97
98 decryptionBenchmark.open("benchmark/decryptionTime_" + std::
       to_string(plaintextModulus) + "_" + std::to_string(
       multiplicativeDepth) + ".txt", std::ios_base::app);
99 decryptionBenchmark << int_ns_decryption.count() << std::endl;
```

4.2 Analisi dei dati

Dopo aver raccolto circa 2000 campioni per ogni valore della `MultiplicativeDepth` da 1 a 12, sono state calcolate alcune statistiche sui dati. Le operazioni sui dati sono state effettuate usando il software R, usato anche per disegnare i grafici.
Prima di calcolare la media e la varianza sono stati rimossi gli *outliers*, dovuti prevalentemente a cause esterne all'esecuzione del programma, come gli swap o altre cause hardware: sono stati infatti tolti i valori superiori al 95° percentile e quelli inferiori al 5° percentile.
Ogni distribuzione è stata poi rappresentata con un istogramma delle frequenze e un boxplot.

4.2.1 Rappresentazione della distribuzione con multiplicative depth uguale a 2

Misurazione con depth uguale a 2	Media (ms)	Deviazione standard (ms)
Generazione della chiave	18.66	0.28
Cifratura	6.23	0.29
Somma	0.0582	0.00604
Moltiplicazione	9.74	0.75
Decifrazione	0.884	0.078

I dati raccolti sono rappresentati nei boxplot 4.1, 4.2, 4.3, 4.4, 4.5 e negli istogrammi 4.6, 4.7, 4.8, 4.9, 4.10.

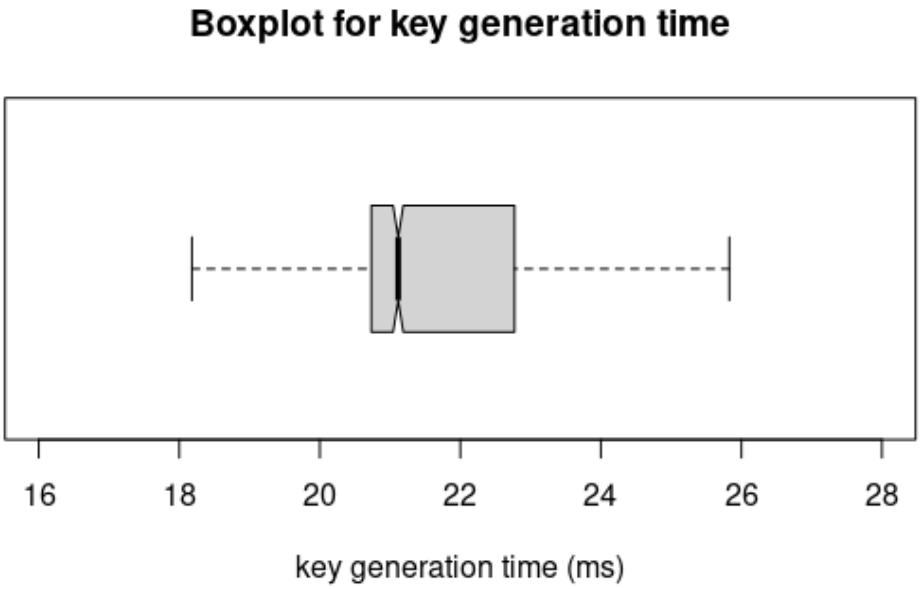

Figura 4.1: Boxplot dei tempi di generazione della chiave

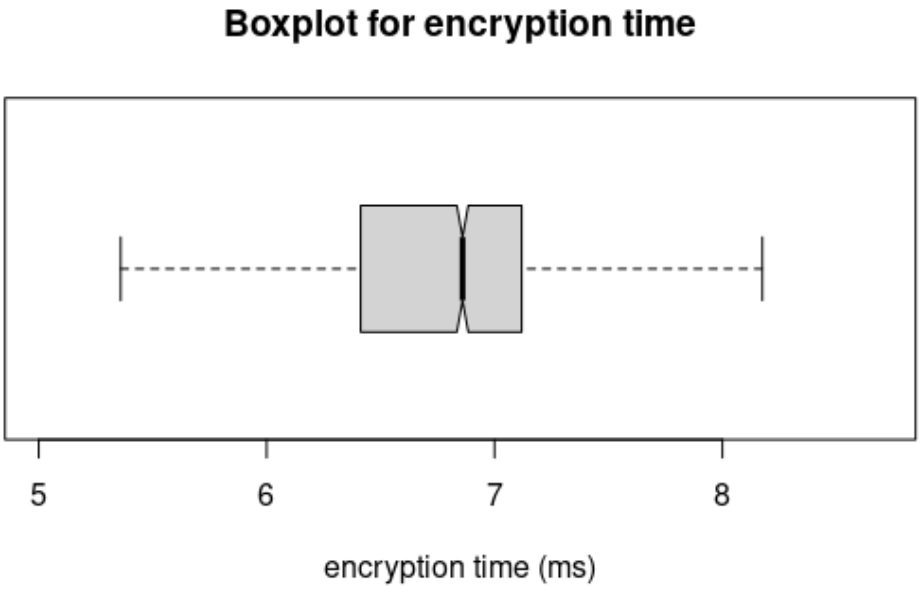

Figura 4.2: Boxplot dei tempi di cifratura

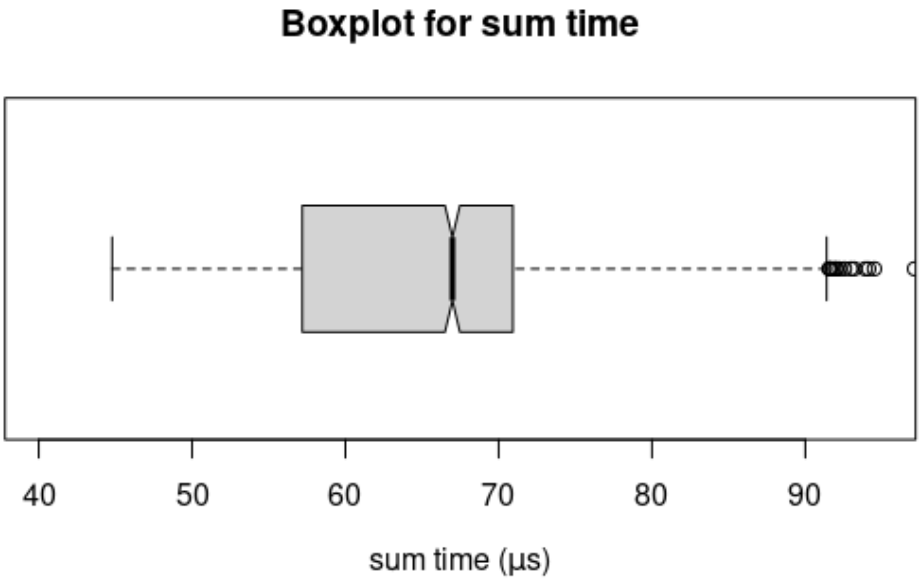

Figura 4.3: Boxplot dei tempi di somma

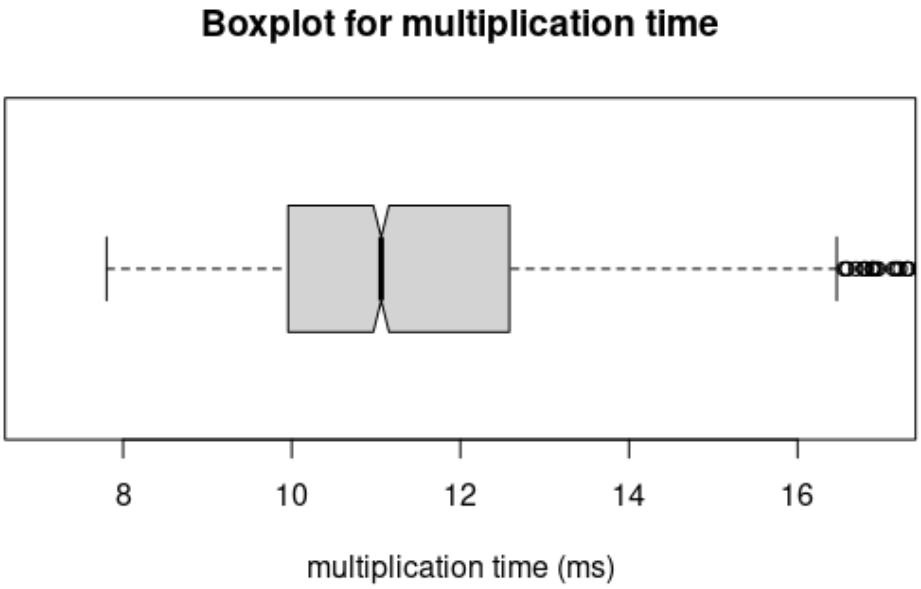

Figura 4.4: Boxplot dei tempi di moltiplicazione

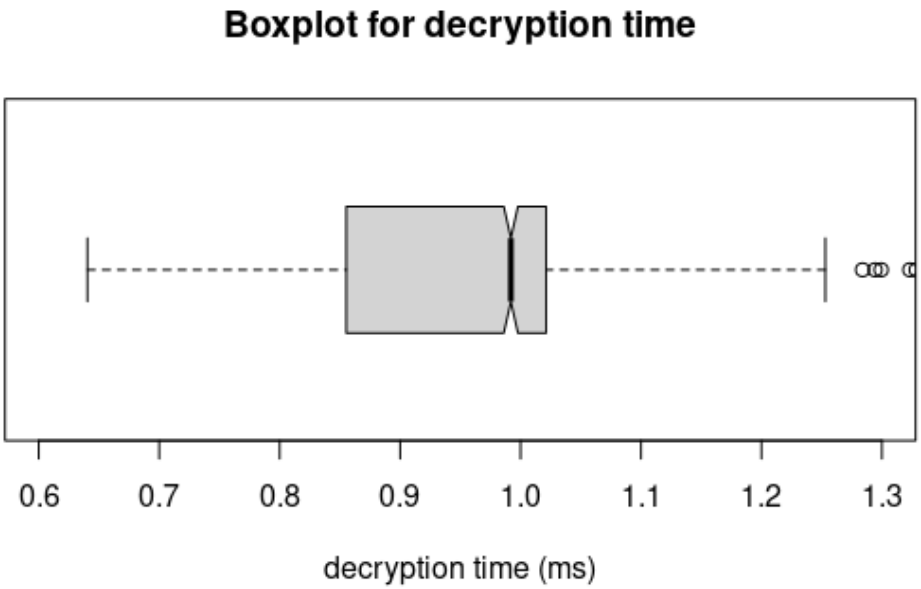

Figura 4.5: Boxplot dei tempi di decifrazione

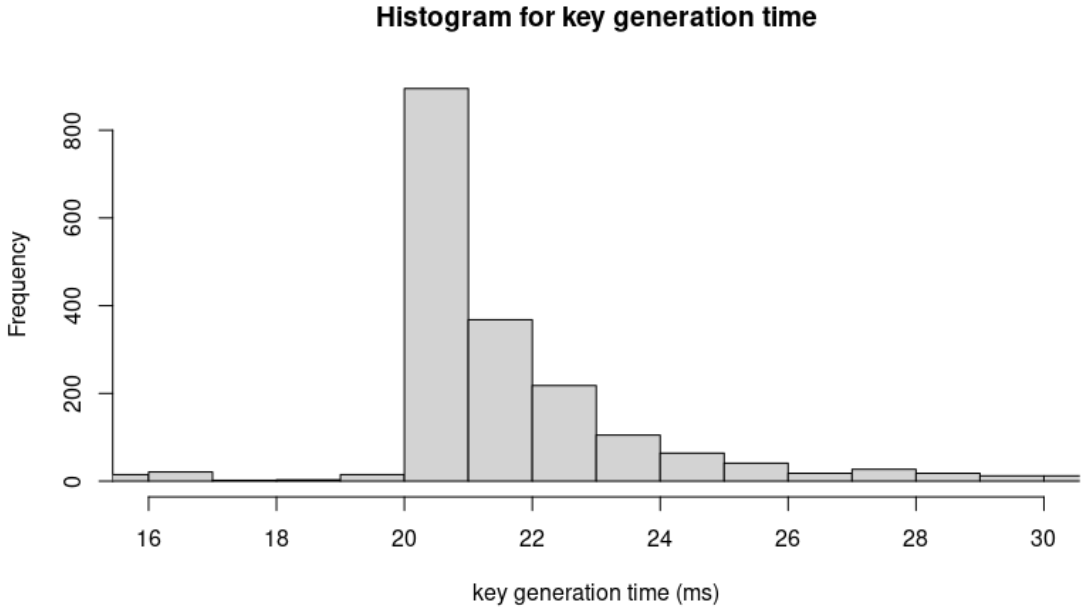

Figura 4.6: Istogramma dei tempi di generazione della chiave

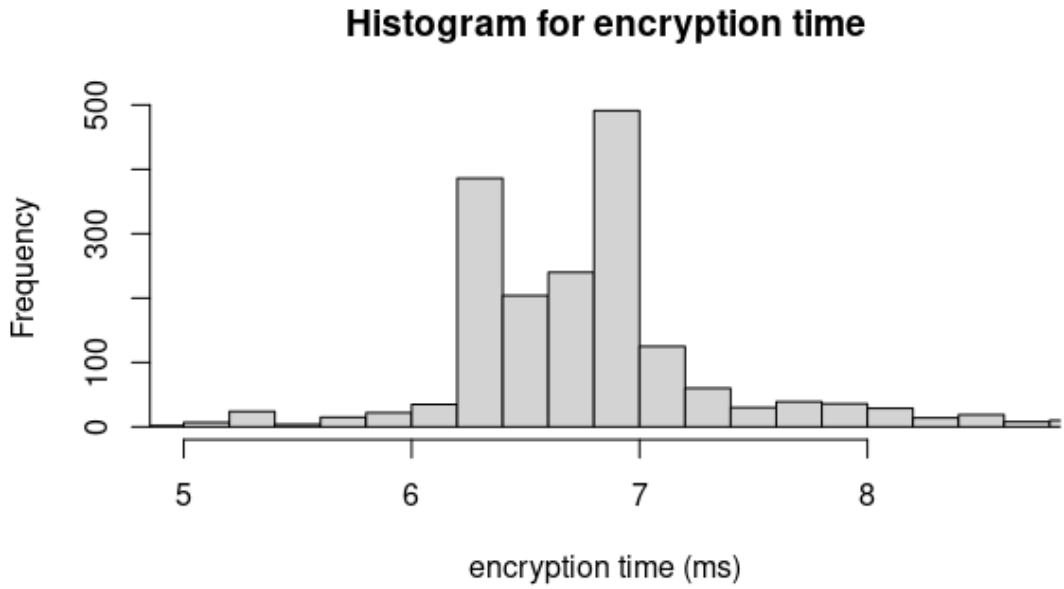

Figura 4.7: Istogramma dei tempi di cifratura

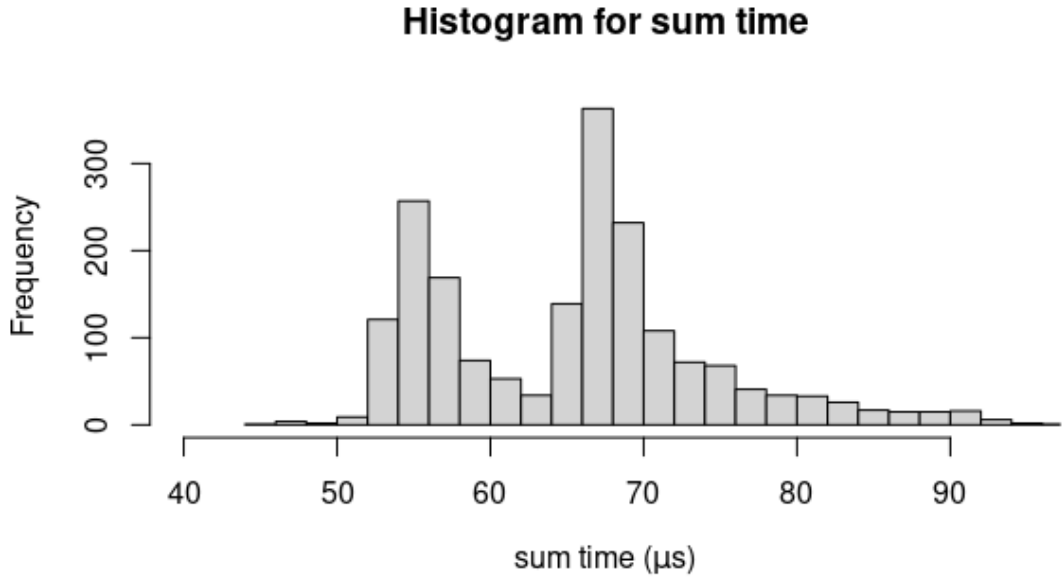

Figura 4.8: Istogramma dei tempi di somma

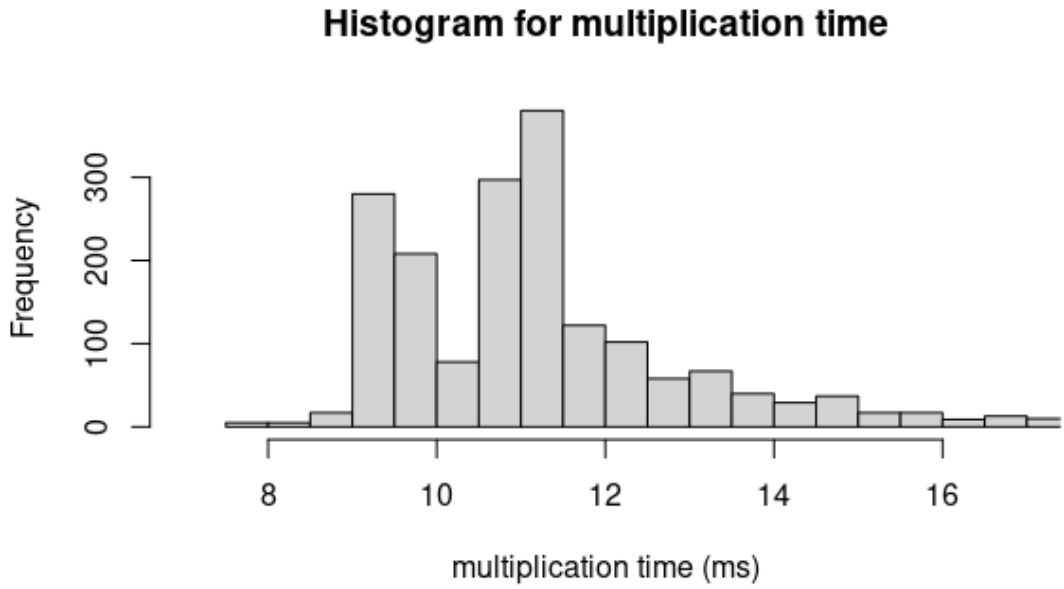

Figura 4.9: Istogramma dei tempi di moltiplicazione

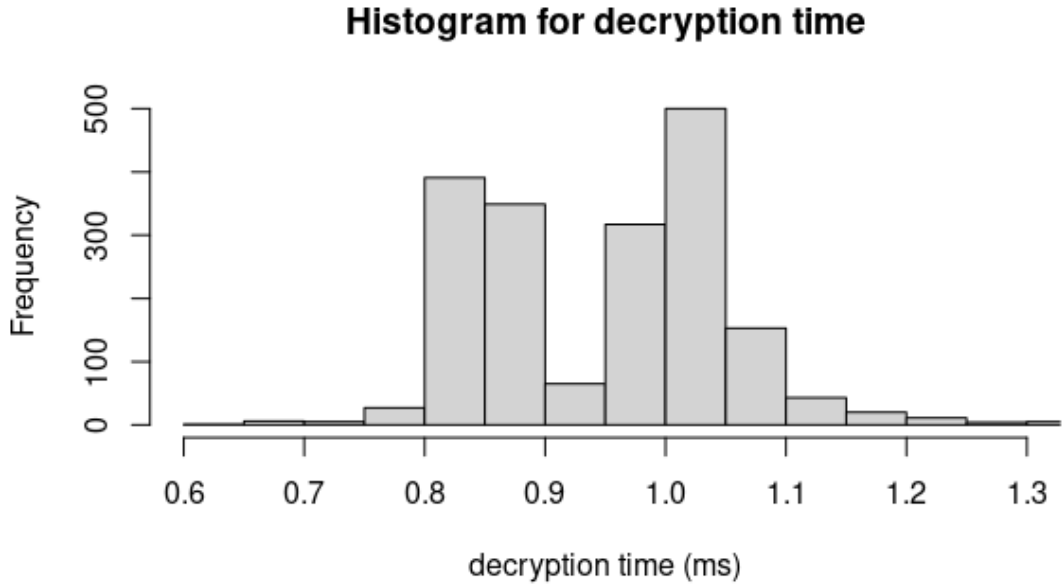

Figura 4.10: Istogramma dei tempi di decifrazione

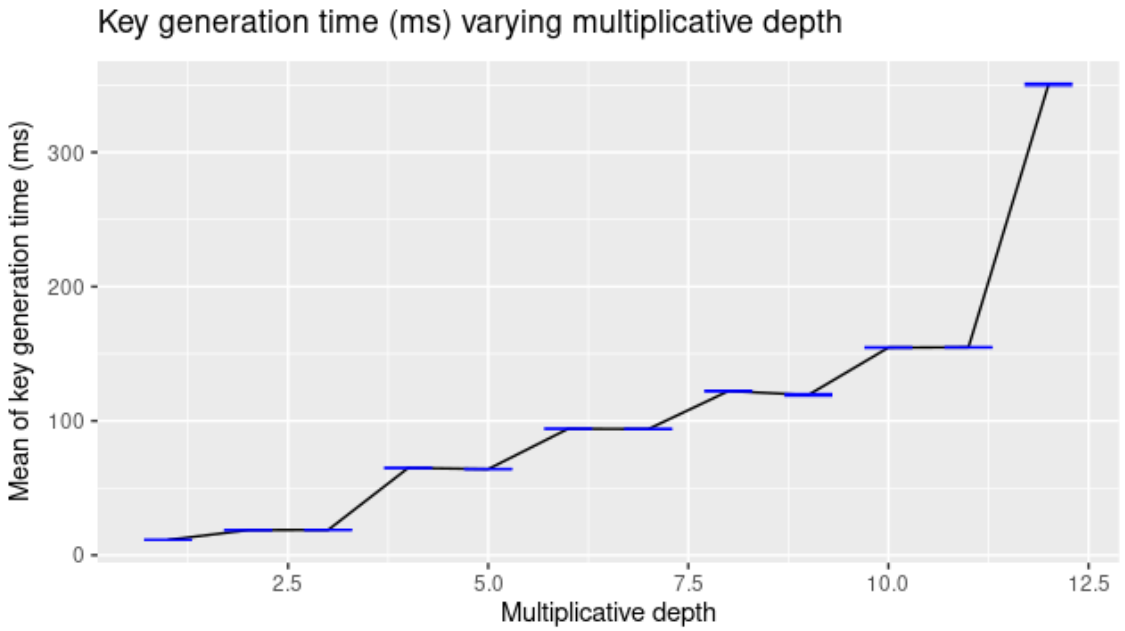

Figura 4.11: Line chart dei tempi di generazione della chiave al variare della multiplicative depth

4.2.2 Calcolo delle statistiche al variare della `MultiplicativeDepth`

Per ogni valore della `MultiplicativeDepth` sono stati calcolati media e deviazione standard su 2000 campioni, e la loro variazione è stata riportata in un grafico a linee, in cui sono stati presi come error bar gli intervalli di confidenza al 95%, calcolati secondo la formula dell'intervallo di confidenza bilaterale ([5]):

$$ErrorBar = 1.96\frac{s}{\sqrt{n}}$$

dove 1.96 è il quantile al 95%, s la deviazione standard e n il numero di campioni. Le line chart sono i grafici 4.11, 4.12, 4.13, 4.14, 4.15.

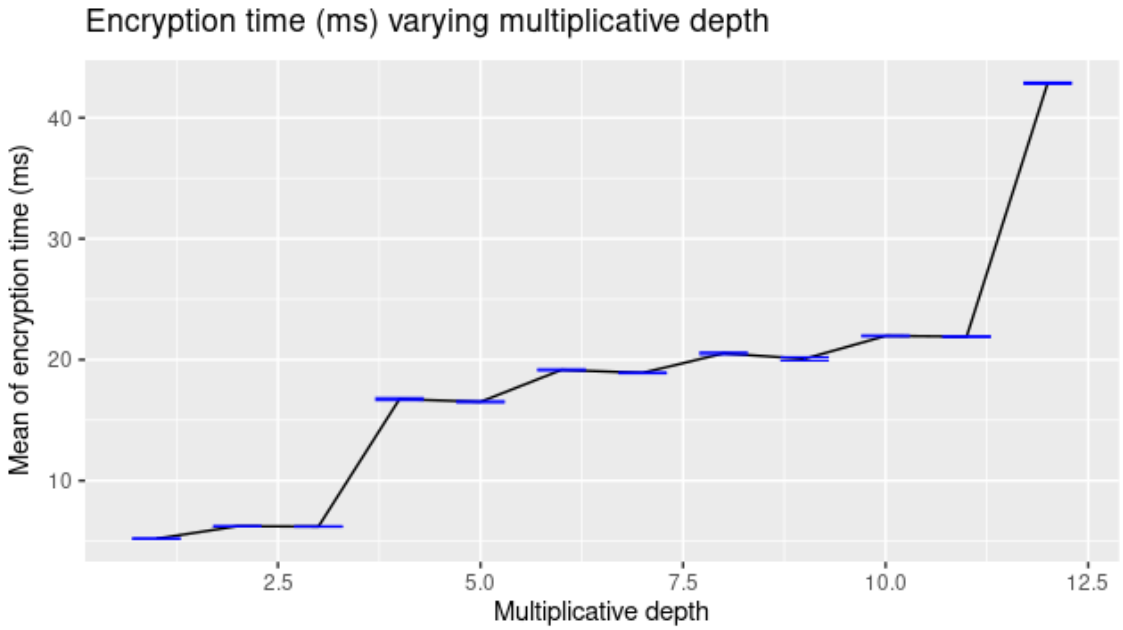

Figura 4.12: Line chart dei tempi di cifratura al variare della multiplicative depth

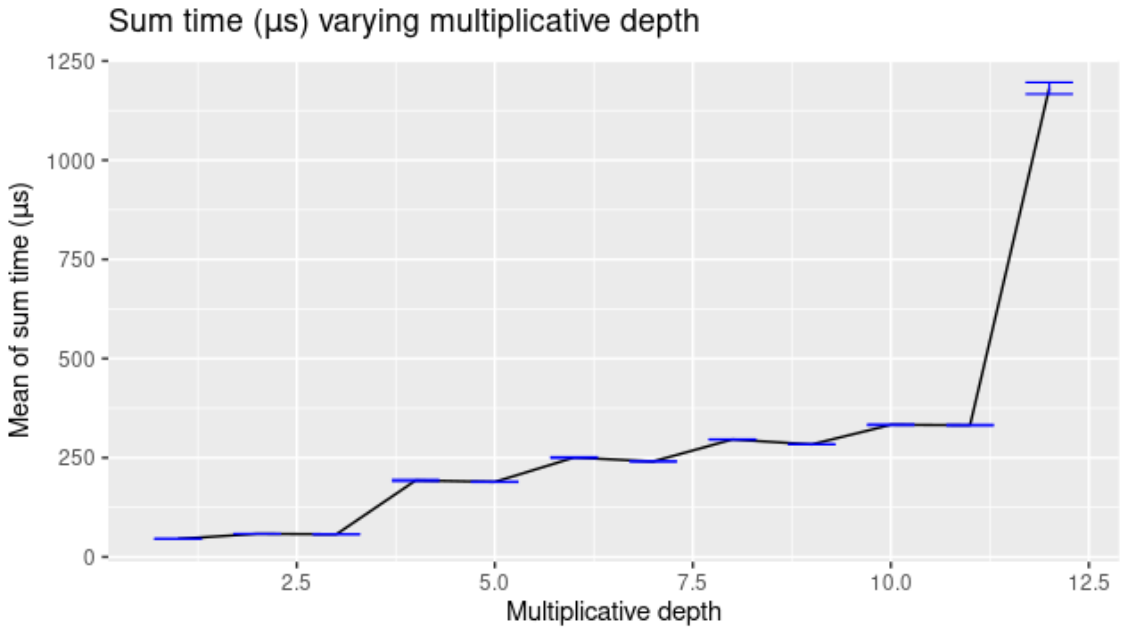

Figura 4.13: Line chart dei tempi di somma al variare della multiplicative depth

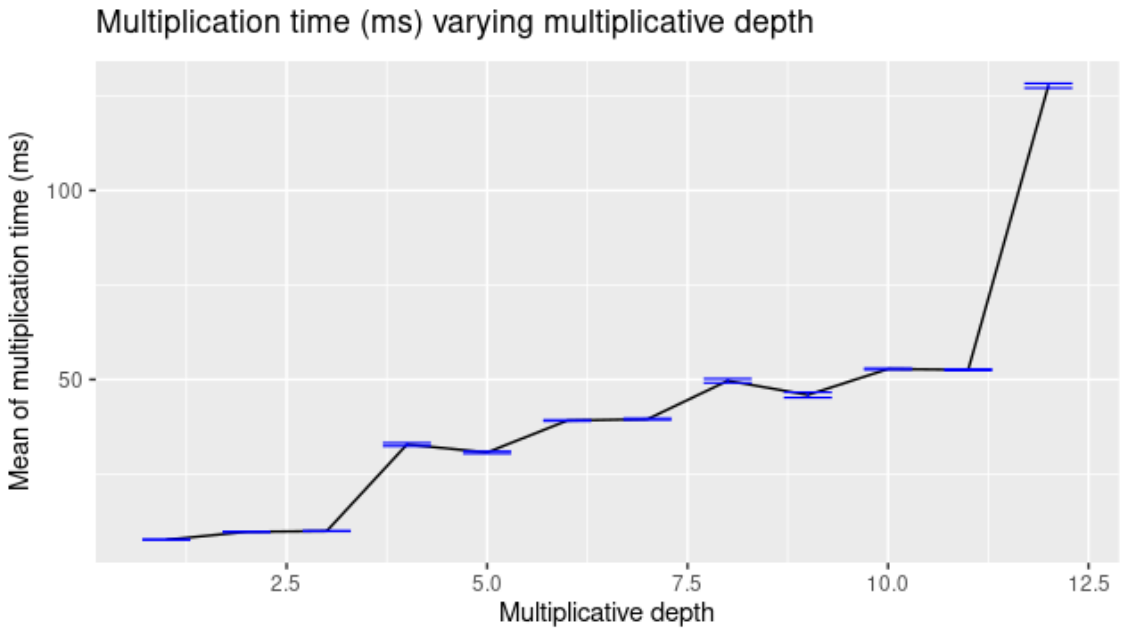

Figura 4.14: Line chart dei tempi di moltiplicazione al variare della multiplicative depth

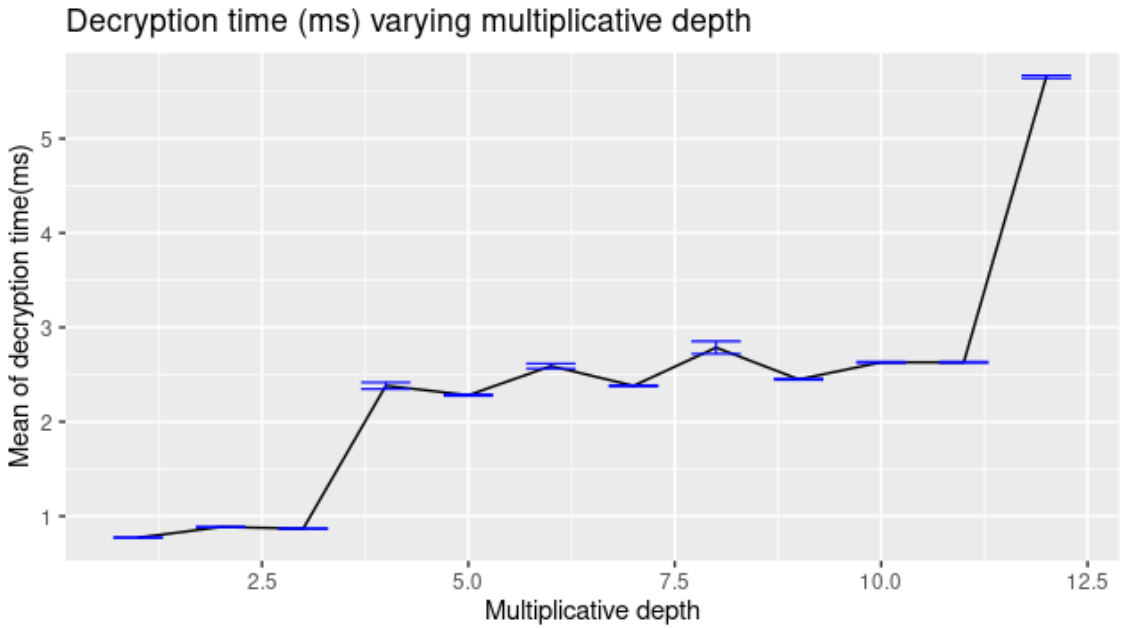

Figura 4.15: Line chart dei tempi di decifrazione al variare della multiplicative depth

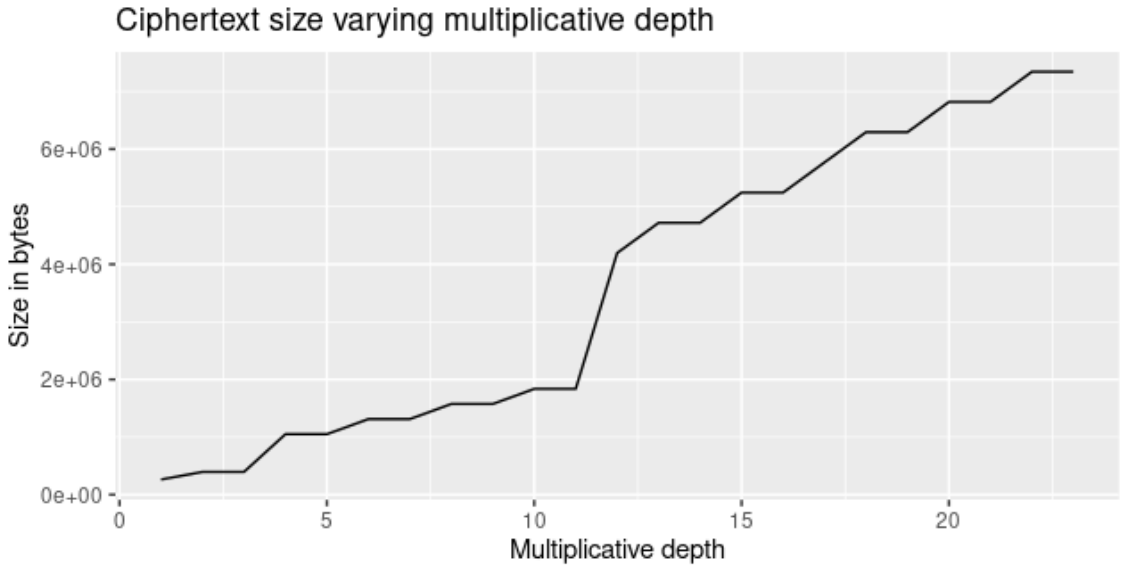

Figura 4.16: Line chart delle dimensioni dei crittogrammi

4.2.3 Dimensione dei crittogrammi al variare della `MultiplicativeDepth`

Per valutare l'usabilità della libreria, sono state calcolate le dimensioni dei crittogrammi al variare della multiplicative depth. All'aumentare del numero di moltiplicazioni possibili, i ciphertext aumentano notevolmente di dimensioni, rappresentate nel grafico 4.16.

Capitolo 5

Conclusioni

L'utilizzo di primitive come quelle fornite da OpenFHE può essere utile per implementare algoritmi privacy oriented, sempre più necessari dal momento in cui la maggior parte delle computazioni viene effettuata su server in cloud: con algoritmi simili può essere infatti possibile effettuare il training di reti neurali e di modelli di machine learning, implementare funzionalità di riconoscimento facciale e fare analisi di dati in ambito sanitario manipolando esclusivamente i dati cifrati, evitando i rischi che derivano dalla conservazione di dati sensibili in chiaro.

OpenFHE permette di realizzare somme e moltiplicazioni tra crittogrammi velocemente in caso di applicativi in cui non è necessario effettuare molte moltiplicazioni. I principali limiti degli schemi attuali basati su Ring-LWE sono legati alla necessità di limitare gli errori introdotti, e quindi la velocità delle operazioni, così come le dimensioni dei dati e delle chiavi usate, aumentano con l'aumentare della *Multiplicative Depth* a causa della dimensione delle matrici usate: nel caso debbano essere effettuate decine di moltiplicazioni, come nel caso delle reti neurali, i calcoli possono risultare lenti e può essere necessario molto spazio, perciò il crittosistema risulta di scarsa applicabilità.

Inoltre uno dei limiti attuali legati all'utilizzo degli schemi di FHE come il BFV è legato alla mancanza di operazioni di divisione efficaci tra crittogrammi. Sono stati proposti infatti schemi di divisione, come in [4], ma necessitano di una `MultiplicativeDepth` lineare nel numero di bit dell'input, ed effettuano un numero di moltiplicazioni e di somme esponenziale.

Appendice A

A.1 Classe Semaphore

La classe `Semaphore` è stata implementata per gestire la sincronizzazione tra thread, usando le librerie `mutex` e `condition_variable`.

```
class Semaphore
{
    std::mutex mutex_;
    std::condition_variable condition_;
    unsigned long count_ = 0; // Initialized as locked.

public:
    void release()
    {
        std::lock_guard<decltype(mutex_)> lock(mutex_);
        ++count_;
        condition_.notify_one();
    }

    void acquire()
    {
        std::unique_lock<decltype(mutex_)> lock(mutex_);
        while (!count_) // Handle spurious wake-ups.
            condition_.wait(lock);
        --count_;
    }

    bool try_acquire()
    {
        std::lock_guard<decltype(mutex_)> lock(mutex_);
        if (count_)
        {
            --count_;
            return true;
        }
        return false;
    }
```

```
33 };
```

Listing A.1: Classe `Semaphore`

Bibliografia

[1] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Ian Quah, Yuriy Polyakov, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. Openfhe: Open-source fully homomorphic encryption library. Cryptology ePrint Archive, Paper 2022/915, 2022. https://eprint.iacr.org/2022/915.

[2] A. Corsi. benchmarking-openfhe. https://github.com/lessandroA/benchmarking-openFHE, 2022.

[3] J. Hoffstein, J. Pipher, and J.H. Silverman. *An Introduction to Mathematical Cryptography*. Undergraduate Texts in Mathematics. Springer, New York, NJ, USA, 2008.

[4] Hiroki Okada, Carlos Cid, Seira Hidano, and Shinsaku Kiyomoto. Linear depth integer-wise homomorphic division. In Olivier Blazy and Chan Yeob Yeun, editors, *Information Security Theory and Practice*, pages 91–106, Cham, 2019. Springer International Publishing.

[5] S.M. Ross. *Introduction to Probability and Statistics for Engineers and Scientists, Student Solutions Manual*. Elsevier Science, fourth edition, 2009.

Ringraziamenti

Vorrei in primis ringraziare il mio relatore, Pericle Perazzo, per l'argomento fornitomi, estremamente interessante e innovativo.

Voglio ringraziare i miei genitori, per avermi sempre supportato, e la Betta, che c'è sempre stata nonostante la distanza.

Infine, voglio ringraziare gli amici: i *fan di Nello* per le boiate sul treno, i *best secondo Bob* per tutte quelle Founders al Chimney, gli amici del *gruppo semplice* per avermi fatto da cavie per la sangria, le *mamme pancine* e gli *NPC* per aver reso le lezioni di elettronica un po' meno pesanti.