Dipartimento di informatica
Corso di laurea Magistrale in Informatica

Relazione per il corso di
COMPRESSIONE DATI

A Huffman Code Based Crypto-System

Docente:
Prof. Bruno Carpentieri

Gruppo:
Manuel Flora 0522500962
Paolo Labanca 0522501161
Maria Giovanna Albanese 0522501356

Anno Accademico 2022-2023

Indice

Capitolo 1

Introduzione

La trasmissione di informazioni su una rete di comunicazione è spesso influenzata dalla velocità di elaborazione, dall'aumento del throughput e dalla necessità di proteggere i dati sottostanti. Con il numero crescente di attacchi informatici e minacce, questi obiettivi diventano tutti necessari, sopratutto quando si ha a che fare con enormi quantità di informazioni delicate. La combinazione di metodi di compressione e crittografia può superare queste sfide rappresentando i dati in modo compatto e in un formato sicuro.

1.1 Stato dell'arte

Attualmente, sappiamo che la compressione non può essere applicata su un file crittografato perché esso, di solito, non può essere distinto da uno generato casualmente. Pertanto, quando si desiderano sia la compressione che la crittografia, la compressione deve essere applicata prima della crittografia o contemporaneamente.
La compressione e la crittografia in simultanea formano i cosiddetti Compression Cryptosystems, i quali possono essere ottenute incorporando la compressione negli algoritmi di crittografia o aggiungendo funzionalità crittografiche nello schema

di compressione. Il nostro lavoro si concentra sulla seconda modalità, in particolare sono stati analizzati e proposti miglioramenti all'approccio di Yoav Gross at al. [1]

In questo articolo viene proposto un algoritmo basato sula codifica di Huffman che fornisce crittografia mantenendo lo stesso rapporto di compressione. Il crittosistema di compressione mantiene lo stesso modello e le stesse distribuzioni del metodo di compressione su cui si basa, mentre utilizza la chiave segreta per controllare quale delle possibili traformate scegliere. Infatti, l'impatto cumulativo di un gran numero di tali cambiamenti porta a testi cifrati completamente diversi, che possono essere decodificati solo se si conosce una determinata chiave segreta. In fine, vengono suggerite diverse varianti e i loro risultati vengono testati in vari contesti, in particolare per la sicurezza contro gli attacchi CPA.

Altre metodologie citate nel lavoro di Yoav Gross at al., riguardano Setyaningsih e Wardoyo [2] che forniscono uno studio completo sulla combinazione di tecniche crittografiche e di compressione, considerando sia i metodi di compressione lossy che lossless e concentrandosi principalmente sulla compressione dell'immagine. Gli autori osservano che la maggior parte degli studi si concentra più sulla sicurezza delle immagini che sulla riduzione delle dimensioni dei dati.

Wang [3] applica una mescolanza casuale dell'alfabeto sorgente a diversi algoritmi di compressione secondo una data chiave segreta, in una fase di pre-elaborazione. In particolare, il dizionario iniziale viene mescolato in compressione LZW e l' ordine dei simboli dell'alfabeto viene mescolato, in caso di codifica aritmetica, [4] e, successivamente, l'albero di Huffman viene criptato.

La differenza tra tutti questi lavori e Yoav Gross at al. sta nel fatto che nei la-

vori precedenti l'efficienza della sicurezza non è dimostrata contro attacchi in chiaro scelti (CPA).

1.2 Struttura del documento

Gli argomenti introdotti sono trattati in dettaglio nei seguenti capitoli che compongono questo elaborato.

- Capitolo 2: Progettazione e Implementazione, dove vengono scelte le metodologie applicative e le motivazioni alle base di esse.
- Capitolo 3: Risultati, dove viene testato l'algoritmo e gli esiti vengono confrontati con quelli del paper di riferimento.
- Capitolo 4: Esiti e sviluppi futiri, dove vengono discussi approfondimenti e riflessioni.

Capitolo 2

Progettazione e Implementazione

In questo capitolo verranno descritte le scelte implementative e le motivazioni che ci hanno portato ad utilizzarle. Il nostro processo implementativo è passato attraverso diverse fasi di sviluppo non solo per affrontare separatamente le tecniche proposte dal paper di riferimento ma anche per verificare se esse fossero valide ed applicabili.
Il capitolo si divide in tre sottoparagrafi.
Nel primo verrà descritta l'organizzazione dei file di progetto, nel secondo l'approccio iniziale e l'implementazione di quanto descritto nell'articolo e nel terzo le nostre modifiche.

2.1 Organizzazione del progetto

I linguaggi di programmazione da noi scelti, sono C++ per l'implementazione degli algoritmi e Python per la stesura dei risultati.
Alla base della scelta di C++ abbiamo diverse motivazioni:

- Comodo per l'implementazione di strutture ad albero
- Lavora a basso livello

- Eccellente gestione della memoria

Alla base della scelta di Python abbiamo diverse motivazioni:

- Moduli ben organizzati
- Semplice e veloce
- Librerie potenti per realizzare grafici e analizzare dati

2.2 Algoritmi di cifratura e traformazioni

Dopo aver analizzato l'articolo la nostra attenzione si è focalizzata sulla realizzazione di quanto descritto per verificarne la validità.

2.2.1 Huffman

La codifica di Huffman è un algoritmo per eseguire la compressione dei dati e costituisce l'idea alla base della compressione dei file. Invece di utilizzare una codifica a blocchi k-aria, cioè codificare ogni carattere emesso dalla sorgente con una stringa di lunghezza fissa di simboli dell'alfabeto di codifica, usiamo la codifica di Huffman per codificare ciascun simbolo emesso dalla sorgente con una stringa che avrà un numero variabile di bit.

L'algoritmo di Huffman parte da una foresta di alberi. Inizializziamo questa foresta di alberi in maniera tale che abbia tanti alberi di un singolo nodo quanti sono i caratteri della sorgente S, e per ogni x il peso di quell'albero T_x sarà posto uguale alla probabilità p_x. Il ciclo che costruisce man mano l'albero di codifica opera come segue: finché la foresta ha più di un albero, si prendono i due alberi che hanno peso più basso (e quindi hanno associato il carattere che ha probabilità di emissione più

bassa), si combinano in un unico albero creando una nuova radice r con peso uguale alla somma dei pesi dei due alberi, e che ha come figli i due alberi scelti, etichettati rispettivamente con 0 e con 1. Questo ciclo continua finché non si ottiene un singolo albero binario, che sarà il nostro albero di Huffman. Di seguito, possiamo vedere l'implementazione dell'argoritmo dove il file preso in input è stato partizionato in caratteri e per ogniuno di essi è stata assegnata una distribuzione di probabilità la quale è fissa per tutto il processo di codifica e che rappresenta il peso dell'albero nella foresta.

```
void HuffmanCodes(int size)
{
  struct MinHeapNode *left, *right, *top;
  for (map<char, int>::iterator v = freq.begin();
    v != freq.end(); v++)
    minHeap.push(new MinHeapNode(v->first, v->second));
  while (minHeap.size() != 1) {
    left = minHeap.top();
    minHeap.pop();
    right = minHeap.top();
    minHeap.pop();
    top = new MinHeapNode('$', left->freq + right->freq);
    top->left = left;
    top->right = right;
    minHeap.push(top);
  }
  storeCodes(minHeap.top(), "");
}
```

Risulta facile, dopo ciò, decodificare un file codificato con questo approccio perchè basta accedere all'albero come possiamo vedere nel codice seguente.

Inoltre, abbiamo costruito un codice prefisso e quindi univocamente decifrabile, do-

ve nessuna parola codice sarà prefissa di un'altra, perchè a partire dalle foglie non costruisco altre parole codice.

```
string decode_file(struct MinHeapNode* root, string s)
{
  string ans = "";
  struct MinHeapNode* curr = root;
  for (int i = 0; i < s.size(); i++) {
    if (s[i] == '0')
      curr = curr->left;
    else
      curr = curr->right;
    // reached leaf node
    if (curr->left == NULL and curr->right == NULL) {
      ans += curr->data;
      curr = root;
    }
  }
  return ans;
}
```

2.2.2 Trasformazioni

Prima di applicare le trasformazioni, è stato selezionato un nodo interno del corrispondente albero di Huffman selezionato in base a una chiave segreta generata in modo pseudo-casuale. Inoltre, supponiamo che questa chiave sia stata scambiata tra l'encoder e il decoder prima delle fasi di codifica.

Viene quindi applicata una trasformazione a partire dal sottoalbero radicato dal nodo selezionato v, assicurandosi che tutte le lunghezze rimangono le stesse.

Consideriamo diverse possibilità di trasformazione sul nodo interno v.

Mirror

Applicando questa trasformazione, l'intero albero radicato in v viene sostituito con la sua immagine speculare. Ad esempio, consideriamo la figura 2.1, nella quale sulla sinistra viene raffigurato l'albero originale e in grigio viene evidenziato il nodo v selezionato tramite la chiave privata. Il risultato dell'applicazione della trasformazione mirror è mostrato sulla destra. Inoltre in alto, vengono mostrati i bit associati alla foglia i (foglia dell'albero radicato in v) che verranno intaccati applicando questa trasformazione.

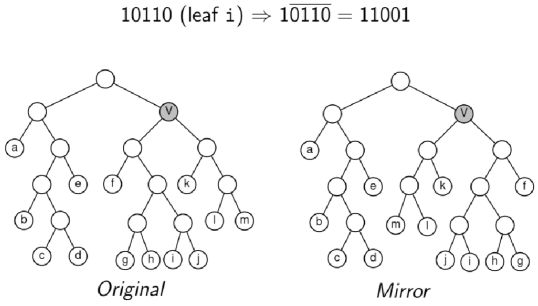

Figura 2.1: Mirror

```
void mirror(struct MinHeapNode* node){
  if (node == NULL)
    return;
  else {
    struct MinHeapNode* temp;
    /* do the subtrees */
    mirror(node->left);
    mirror(node->right);

    /* swap the pointers in this node */
    temp = node->left;
    node->left = node->right;
    node->right = temp;
  }
```

```
15 }
```

Swap

In questo caso, verranno semplicemente scambiati i due figli del nodo selezionato v. In figura 3.3 possiamo vedere in azione questa trasformazione e i bit associati alla foglia i che verranno modificati.

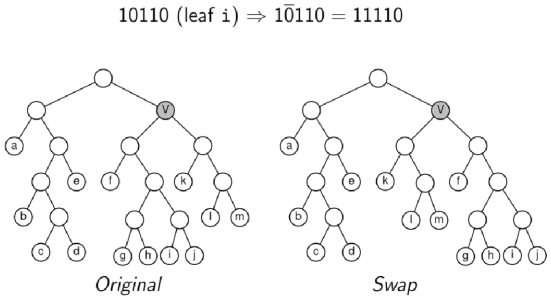

Figura 2.2: Swap

```
1 void swap(MinHeapNode *root, char val) {
2   MinHeapNode *parent = findParent(root, val);
3   if (parent != NULL)
4     simpleSwap(parent);
5 }
```

Bulk-Crypto Huffman

Qui non verrà scelto un nodo, ma la trasformazione verrà fatta sull'intero albero. Come possiamo vedere in figura 2.3, ogni volta che si andrà sulla destra (in Huffman, bit 1), si effettuerà lo scambio dei suoi figli.

```
1 void bulkCryptoHuffman(MinHeapNode *node) {
2   if (node->left != NULL)
3     bulkCryptoHuffman(node->left);
4   if (node->right != NULL) {
```

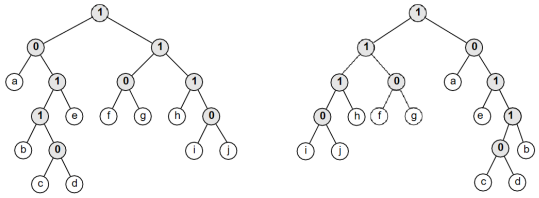

Figura 2.3: Bulk-Crypto Huffman

```
5     bulkCryptoHuffman(node->right);
6     simpleSwap(node->right);
7   }
8 }
```

Level-swap

In tal caso, verrà scelto il nodo v e si effettuerà lo scambio con il suo *right(v)*. La figura 2.4 mostra questa trasformazione e i bit associati alla foglia i.

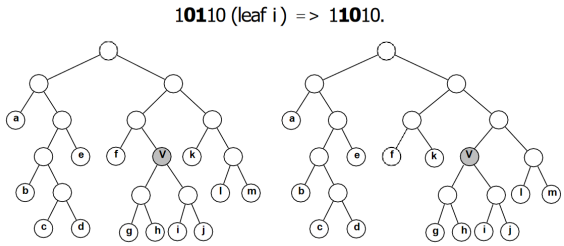

Figura 2.4: Level-swap

```
1 void levelSwap(MinHeapNode* root, char data) {
2   MinHeapNode* right = nextRight(root, data);
3   if (right == NULL)
4     cout << "No next right node found for " << data << endl;
5   else {
6     MinHeapNode* parent1 = findParent(root, data);
7     MinHeapNode* parent2;
8     if (right->data == '$')
9       parent2 = findParent(root, right->freq);
```

```
10      else
11        parent2 = findParent(root, right->data);
12      MinHeapNode* node1;
13      if (parent1->left->data == data)
14        node1 = parent1->left;
15      else
16        node1 = parent1->right;
17      swap(parent1, parent2, node1, right);
18    }
19  }
```

2.3 Intuizione

Yoav Gross at al. applicavano le trasformazioni sull'albero di Huffman singolarmente. Questo semplifica le operazioni di decodifica e rende il testo facile da detrasformare intaccando, però, la sicurezza. Abbiamo cercato, quindi, di migliorare quest'ultima mutando l'output non solo in base alla chiave segreta, ma anche in base alle iterazioni delle traformazioni. Abbiamo generato tre chiavi segrete pseudocasuali, che stabiliscono: la lettera su cui far partire la traformazione, il numero di trasformazioni da iterare e quale traformazione applicare sulla codifica.

Un'altra intuizione è stata quella di aggiungere dei bit sparsi nel testo, in modo da non poter effettuare attacchi che tengono conto della probabilità delle lettere nella lingua. Il nostro suggerimento, riguarda modificare continuamente la forma dell'albero di Huffman applicando scambi o trasformazioni scelte in modo pseudocasuale, che porta a un tentativo alternativo di introdurre abbastanza perturbazioni nel testo cifrato, in modo da trasformare il problema della rottura del codice in un problema difficile.

Capitolo 3

Risultati

Per i nostri risultati sperimentali abbiamo considerato: il Large Corpus tratto dal Canterbury [1] corpora, nello specifico il file bible.txt, la versione King James della Bibbia, di dimensioni 4.047.392 byte. E.coli la versione Complete genome of the E. Coli bacterium, di dimensione 4.638.690 byte.In fine il world192.txt, la versione The CIA world fact book, di dimensione 2.473.400 byte. Attraverso questa scelta siamo riusciti a mettere a confronto non solo la nostra tecnica con quella proposta dal paper, ma siamo riusciti a testare l'efficienza anche su testi che hanno forma e lunghezza diverse.

3.1 Analisi di indistinguibilità

Come primo test di sicurezza abbiamo analizzato l'indistinguibilità. Qualsiasi file ragionevolmente compresso o crittografato dovrebbe consistere in una sequenza di bit che non è distinguibile da una sequenza binaria generata casualmente. Un criterio per tale casualità potrebbe essere la probabilità di occorrenza.

All'interno del file compresso, di qualsiasi sottostringa di lunghezza m bit dovrebbe essere 2^{-m}, per tutti gli $m \geq 1$, cioè le probabilità per 1 o 0 sono entrambe 0.5, le probabilità per 00, 01, 10 e 11 sono 0.25, ecc. Abbiamo ripetuto più volte l'esperimento per verificare che i risultati fossero simili tra loro.
Nella tabella 3.1 sono mostati i risultati, applicando: lo stesso numero di trasformate, le stesse trasformate, lo stesso nodo e senza l'inserimento dei caratteri speciali.
Nella tabella 3.2 invece vengono confrontati con i risultati dell'articolo di riferimento, con le stesse caratteristiche dette prima.

Per misurare la deviazione fra la distribuzione di probabilità 2^m possibili pattern di bit di lunghezza m e la distribuzione di probabilità di lunghezza m, abbiamo usato la divergenza Kullback-Leibler e abbbiamo confrontato i risultati come si vede nella tabella 3.4.

$$D_{KL}(P||Q) = \sum_{x \epsilon X} P(x) \log \frac{P(x)}{Q(x)}$$

Dove *P(x)* è la distribuzione attesa e *Q(x)* la distribuzione ottenuta.

3.1.1 Scelta della chiave che genera il numero di trasformazioni

Per scegliere una soglia minima di volte per iterare le trasfomazioni, abbiamo eseguito il nostro algoritmo più volte e con un numero di iterazioni fisse. Abbiamo notato che al crescere delle iterazioni le trasformazioni hanno una distribuzione che tende ad uniformarsi. Questo significa che un attaccante non ha modo di preferire una trasmorazione rispetto a un'altra. Per ottenere l'indistinguibilità, è necessario che ogni trasformazione abbia la stessa probabilità di essere scelta, in modo che l'avversario non abbia preferenze per uno o per un'altra.

bit-string	bible.txt	e.coli	world192.txt
0	0.50653	0.49020	0.49552
1	0.51347	0.50681	0.51002
00	0.25259	0.25261	0.24272
01	0.25526	0.24533	0.25471
10	0.25526	0.25533	0.25423
11	0.25968	0.24639	0.25921
000	0.12874	0.12966	0.12921
001	0.11317	0.13208	0.13308
010	0.12228	0.11590	0.12400
011	0.13990	0.13859	0.13729
100	0.12978	0.13208	0.13238
101	0.12468	0.12015	0.11039
110	0.12398	0.13859	0.12839
111	0.12525	0.12752	0.12800

Tabella 3.1: Indistinguibilità file

Per un numero maggiore di 50 iterazioni, le probabilità tendono a uniformarsi, abbiamo posto quest'ultimo come limite minimo. L'analisi viene sintetizzara nel grafico 3.2

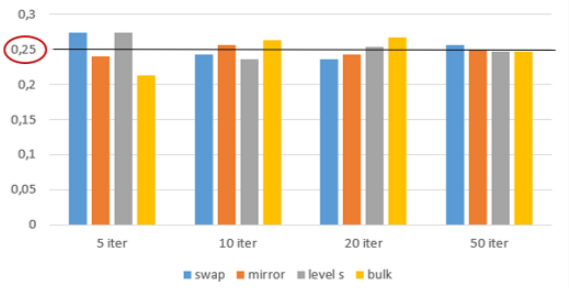

Figura 3.1: Probabilità delle varie iterazioni

bit-string	m-m-s	BCH-BCH-ls	ls-ls-BCH
0	0.50251	0.49232	0.50232
1	0.50749	0.50768	0.50768
00	0.25259	0.25261	0.25272
01	0.25526	0.25533	0.25471
10	0.24526	0.25533	0.25423
11	0.25269	0.25986	0.25921
000	0.12976	0.12966	0.12921
001	0.13147	0.12208	0.12308
010	0.11228	0.12390	0.12400
011	0.12990	0.12859	0.12729
100	0.12147	0.13208	0.12238
101	0.12953	0.11015	0.12039
110	0.13420	0.12859	0.12839
111	0.12525	0.12752	0.12800

Tabella 3.2: Indistinguibilità esecuzioni su Bible.txt.
m = mirror, s = swap, BCH = Bulk-Crypto Huffman, ls = level-swap

3.2 Miglioramenti della sicurezza

Per monitorare la sicurezza su attacchi cpa possiamo simulare un esperimeno e valutarne la probabilità che l'attaccante riesca a individuare la chiave.

- Sia **Knod** la chiave usata per generare il nodo da cui far partire la trasformazione;
- Sia **KTrans** la chiave usata per scegliere quale trasformazione usare;
- Sia **KnTrans** la chiave usata per scegliere quante trasformazioni applicare in un'unica esecuzione;

bit-string	**mix**	L-Swap	Mirror	Swap	Bulk
0	**0.50749**	0.500097	0.500061	0.500138	0.500034
1	**0.49251**	0.499903	0.499939	0.499862	0.499966
00	**0.25259**	0.250076	0.249985	0.250074	0.250060
01	**0.25526**	0.250021	0.250076	0.250074	0.249974
10	**0.25526**	0.250021	0.250076	0.250074	0.249974
11	**0.24269**	0.249882	0.249863	0.249798	0.249992
000	**0.12367**	0.121652	0.124964	0.125050	0.125053
001	**0.13147**	0.128424	0.125022	0.125024	0.125007
010	**0.12228**	0.121522	0.125012	0.124975	0.124968
011	**0.12990**	0.128499	0.125063	0.125089	0.125007
100	**0.12147**	0.128424	0.125022	0.125024	0.125007
101	**0.11953**	0.121597	0.125054	0.125040	0.124967
110	**0.12990**	0.128499	0.125063	0.125089	0.125007
111	**0.12525**	0.121384	0.124800	0.124709	0.124985

Tabella 3.3: Confronto risultati

	m = 1	m = 2	m = 3
D_{kl} (1)	0,000494544	0,286415483	0,206711938
D_{kl} (2)	0,000494467	0,286414104	0,206711524

Tabella 3.4: Divergenza Kullback-Leibler

- Sia **A** un avversario PPT che tenta di scoprire le chiavi.

L'esperimento è strutturato come segue:

1. Eseguo A. Ogni volta che A interroga l'oracolo esso risponde con un certo messaggio.

2. A sceglie unformemente a caso la chiave KnTrans $\in \mathbb{R}$ e sulla base di essa sceglie per KnTrans-volte Knod $\in \mathbb{N}$, kTrans $\in \mathbb{N}$

3. Da in input queste chiavi all'oracolo e ottiene una risposta m.

4. Se La risposta m=m' (m' è la decifratura corretta) allora l'adv ha avuto successo nell esperimento e l'output della simulazione è 1 altrimenti 0.

Che probabilità ha l'avversario di risolvere con successo questo esperimento?
L'obiettivo dell'avversario è quello di indovinare tutte le chiavi, è necessario quindi che l'esperimento abbiamo una probabilità di successo $\leq \frac{1}{2}$. + una funzione trascurabile per dimostrare che la costruzione sia sicura.
Per poter individuare le chiavi è necessario:

- Conoscere quanti simboli sono presenti nella sorgente, supponiamo che questa informazione si conosca e che i simboli sono 64 tra lettere maiuscole, minuscole e punteggiatura.

- Sapere che la seconda chiave è un numero compreso tra 0 e 3.

- Sapere che la terza chiave è un numero ≤ 50.

Avremo quindi 64 combinazioni per knod e 4 combinazioni di KTrans Con un totale di 256 combinazioni per KnTrans uguale a uno. Di conseguenza all'aumentare di knTrans il numero di combinazioni aumenta e la probabilità che l'avversario vince l'esperimento diminuisce.

I tentativi riguardanti la decifratura possono essere basati, inoltre, su statistiche relative alle occorrenze dei caratteri su un determinato alfabeto, che generalmente sono ben note. Queste statistiche possono portare a indovinare le parole in codice con alta probabilità, inducendo così un numero molto basso di possibili partizioni

del testo cifrato nelle parole in codice.

Per far fronte agli attacchi CPA(*Chosen plaintext attack*), definiamo un nuovo simbolo che verrà successivamente inserito all'interno dell'alfabeto. L'obiettivo è produrre testi cifrati diversi, anche nel caso in cui la stessa chiave segreta venga utilizzata per crittografare lo stesso messaggio. Il simbolo definito in precedenze viene inserito ripetutamente nel file di input in posizioni scelte casualmente cercando di inficiare il meno possibile il rapporto di compressione di Huffman.

Nel prossimo paragrafo verranno mostrati i risultati inerenti alle probabilità di occorenza delle lettere basati su un possibile attacco CPA che si basa su un dizionario di lingua inglese e che ha come vettore d'attaco il nostro testo in chiaro. Ad esempio, nella lingua inglese le lettere con maggiore occorenza sono la t e la e, successivamente seguono le vocali. Un possibile attaco è basato sullo studio da parte di un attaccante sulla ricorrenza più alta di un codice, riguardante proprio le lettere t ed e, che collegate tra di loro nella maggior parte delle volte va a generare la frase *the*. Da questa semplice supposizione, l'attaccante incomincia a costruire il file in chiaro. Nel nostro caso con l'inserimento dei caratteri detti in precedenza, andiamo a evidenziare la differenza nella rottura del codice.

I dati inerenti all'attacco descritto in precedenza vengono riportati analizzando la differenzza tra le codifiche su trasformazioni diverse. L'analisi è stata effettuata su piu testi, tramite la distanza di Hamming.

3.3 Distanza di Hamming

Abbiamo inoltre voluto verificare la sensibilità del sistema alle variazioni della chiave segreta. Considerando i file crittografati, possiamo utilizzare come misura la distanza di Hamming normalizzata. La distanza normalizzata di Hamming è così definita

$$NHD(A, B) = \frac{1}{n} \sum_{i=1}^{n} a_i XOR b_i \tag{3.1}$$

Date due stringhe di bit $A = a1...an$ e $B = a1...an$ con $n \geq m$ viene prima esteso da zeri in modo che entrambe le stringhe siano della stessa lunghezza n.
Abbiamo esaminato una coppia di testi cifrati, generati in base a due chiavi casuali scelte in modo indipendente, per ciascuna delle nostre varianti di codifica suggerite. I testi cifrati prodotti erano completamente diversi, con il numero ponderato di differenze nei bit corrispondenti che tende rapidamente al valore atteso $\frac{1}{2}$.
Il valore limite ottenuto dopo l'elaborazione dell'intero file di input è 0.4060802.

Successivamente si determina anche il numero di caratteri speciali da inserire all'interno della nosta codifica per rendere il nostro output indistinguibile, come detto in precendeza nel paragrafo sui Miglioramenti della sicurezza.
Dove si può notare che tutte le trasformazioni tendono ad arrivare al valore ideaale di $\frac{1}{2}$. solo dopo l'aggiunta di 1500 caratteri in maniera pseudo-casuale.

3.4 Complessità computazionale

Successivamente, siamo andati a verificare il tempo d'esecuzione dell'algoritmo, andando ad iterare ripetutamente l'esecuzione dell'algoritmo, in quanto sappiamo che un processore va ad eseguire piu operazioni e quindi il tempo d'esecuzione può essere

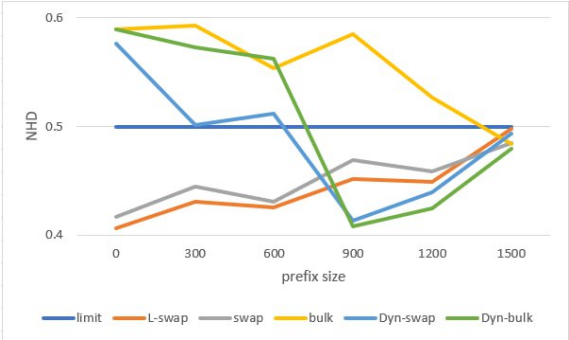

Figura 3.2: Numero di caratteri per avere l'indistinguibilità

legato al contesto.

Le specifiche di sistema utilizzate sono riportate nella tabella 3.5.

Compilatore	MinGW
OS	Windows 11 64 bit
CPU	Intel Core i7-9750H @ 2.60GHz
RAM	16 GB

Tabella 3.5: Specifiche di sistema

Il nostro studio è mirato sia a verificare il tempo d'esecuzione al crescere delle traformazioni e sia a verificare che l'efficienza dell'algoritmo è inalterata al crescere delle trasformazioni.

Il range delle trasformazioni applicate è 5-25, rispettivamente valore minimo e massimo. Ad ogni iterazione, una trasformata viene aggiunta al codice e si calcola il tempo d'esecuzione impiegato. Al termine dello studio, si è notata che la trasformazione Bulk-Crypto Huffman tende a generare dei picchi, tale risultato è legato al numero di scambi maggiori che avviene all'interno dell'albero. Le trasformazioni piu efficienti risultano la Level-Swap e la Swap, dato che si effettuano il minor numero di modifiche all'albero, mentre la trasformata Mirror ha un leggero aumento del tempo d'esecuzione.

Per quanto riguarda la verifica dell'efficienza dell'algoritmo all'aumentare delle trasformazioni, possiamo dire che l'algoritmo ovviamente ha una crescita legata al numero di trasformazioni, quindi al cresce delle trasformazioni cresce anche il tempo d'esecuzione, ma tale crescita non è marcata. Quindi, possiamo dedurre che l'algoritmo ha un'efficienza accettabile.

Il valore medio riguardante tutti i risultati è di 3190 ms, che avviene alla dodicesima iterazione. In questa iterazione, abbiamo la maggior efficienza dell'algoritmo.

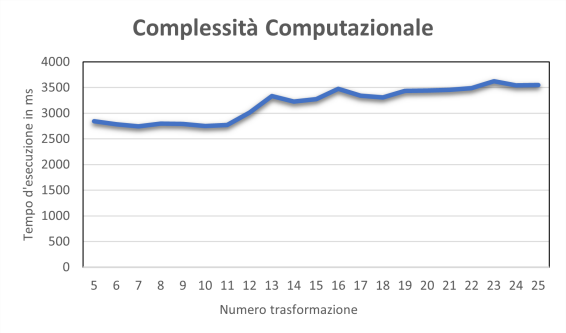

Figura 3.3: Complessità Computazionale

Capitolo 4

Conclusione e sviluppi futuri

Concludiamo dicendo che, la procedura basata mediante i Compression Cryptosystems proposta è estremamente sensibile ad alterazioni anche minime: i file prodotti superano i test di casualità, sono praticamente della stessa dimensione, e sono completamente diversi tra loro, pur conservando l'efficienza della compressione.
Per valutare la resistenza del crittosistema di compressione proposto ad attacchi CPA, abbiamo calcolato la distanza di Hamming normalizzata tra due codifiche dello stesso testo, utilizzando la stessa chiave.

Eventuali sviluppi futuri si potrebbero basare sullo studio della resistenza del sistema su altri tipi di attacchi (non solo CPA) e sull'implementazine di altre trasformate sull'albero di Huffman. Altri studi si potebbero basare sul numero di simboli da inserire per andare a ottenere sia un buon rapporto di compressione che una buona sicurezza.

Bibliografia

[1] Yoav Gross,Yoav Gross, Elina Opalinsky, Rivka Revivo, Dana Shapira, A Huffman Code Based Crypto-System, Ariel University and Bar Ilan University, 2022.

[2] E. Setyaningsih and R. Wardoyo. Review of image compression and encryption techniques. Intern. J. of Advanced Computer Science and Applications, 2017.

[3] C.-E. Wang. Cryptography in data compression. Code-Breakers Journal, 2006.

[4] K. Wong, Q. Lin, and J. Chen. Simultaneous arithmetic coding and encryption using chaotic maps. IEEE Trans. on Circ. and Syst.–II Express Briefs, 2010.

[5] http://corpus.canterbury.ac.nz