Università degli Studi di Salerno
Dipartimento di Informatica

Progetto di
Compressione Dati

Generazione di chiavi simmetriche in base a conoscenza comune nello scenario del Compression Cryptosystem di Huffman

Docente
Prof. Bruno Carpentieri

Studenti
Alfonso Carpentieri
Francesco Chiacchiari
Salvatore Danese
Nicholas Di Pasquo
Edoardo Livornese
Alessio Salzano

Anno Accademico 2023-2024

Indice

1 Introduzione **1**
1.1 Stato dell'Arte 1
1.2 Sicurezza 2
1.3 Struttura e Scenario 2

2 Implementazione **4**
2.1 Organizzazione del progetto 4
2.2 Struttura del progetto 4
2.2.1 Cifratura asimmetrica 5
2.2.2 Sincronizzazione conoscenza comune 9
2.2.3 Generazione chiave condivisa 11
2.3 Sicurezza dello schema 12
2.3.1 Trasmissione degli indici del decompressore 12
2.3.2 Trasmissione della disposizione da utilizzare 13
2.3.3 Cripto-Huffman 15
2.3.4 K-transformations 15
2.3.5 Bulk-transformations 17
2.3.6 Resistenza contro attacchi CPA: inserimento di caratteri DC 18

3 Discussione Risultati **19**
3.1 Analisi degli output 19
3.1.1 Distribuzione sottostringhe binarie 19
3.1.2 Distanza di Hamming normalizzata 20

4 Conclusioni e Sviluppi Futuri **22**

Capitolo 1

Introduzione

La crescente mole di dati generati quotidianamente ha posto una pressione significativa sui sistemi di archiviazione, trasmissione e elaborazione delle informazioni. In risposta a questa sfida, la compressione dati emerge come una strategia essenziale per ridurre la quantità di spazio necessario e migliorare l'efficienza della trasmissione. Nel contesto della compressione dati, l'interazione tra codificatore e decodificatore riveste un ruolo cruciale, poiché determina in modo sostanziale le prestazioni generali del sistema. Inoltre, a causa del maggiore volume di dati trasmesso su internet e i possibili attacchi alla loro relativa confidenzialità hanno determinato l'esigenza di utilizzare tecniche che combinano la compressione con la crittografia.

1.1 Stato dell'Arte

Attualmente, la compressione non può essere applicata a un file crittografato in quanto la sua struttura non è distinguibile da un insieme di dati generati casualmente. Quando si desidera combinare sia la compressione che la crittografia, è necessario applicare la compressione prima della crittografia o contemporaneamente ad essa. I sistemi che integrano compressione e crittografia contemporaneamente sono noti come *Compression Cryptosystems*. Questi possono essere ottenuti incorporando la compressione negli algoritmi di crittografia o aggiungendo funzionalità crittografiche nello schema di compressione utilizzato. Il nostro lavoro si focalizza sulla seconda opzione, concentrandosi in particolare sull'analisi e sulle proposte di miglioramento dell'approccio del *Compression Cryptosystem* basato sulla codifica di Huffman descritto da Yoav Gross at al.[4] .

1.2 Sicurezza

Una problematica rilevante e lasciata ad ulteriori sviluppi durante la presentazione del *Compression Cryptosystem* basato sulla codifica di Huffman è sicuramente la gestione della chiave simmetrica che coordina le trasformazioni. Sebbene il processo di crittografia e compressione sia modellato in maniera lineare, l'elemento critico è la sicurezza della chiave stessa durante le fasi di generazione e di condivisione tra le parti. Lo scopo del seguente lavoro è proporre una soluzione per generare e scambiare in sicurezza una chiave simmetrica a partire da elementi di conoscenza comune tra le parti. Questo approccio ha l'obiettivo di garantire uno schema crittografico robusto e sicuro, preservando la riservatezza delle informazioni trasmesse senza compromettere l'efficienza della compressione.

1.3 Struttura e Scenario

Per garantire la sicurezza e l'efficienza del processo, è fondamentale che nel contesto di utilizzo:

- il **compressore** e il **decompressore** abbiano almeno 5 elementi in comune;
- i file siano indicizzati;
- il **compressore** e il **decompressore** abbiano a disposizione una chiave pubblica e una privata.

In questo scenario non risulta essere necessaria la perfetta sincronizzazione degli elementi delle due parti, ma soltanto l'integrità del numero minimo di elementi in comune.

Il processo di compressione e decompressione in questo scenario segue un intricato ma robusto protocollo di sicurezza per garantire la trasmissione sicura e efficiente dei dati tra le parti.

1. **Inizio Trasmissione**: Il flusso di trasmissione inizia con il **decompressore** che, utilizzando la chiave pubblica del **compressore**, cifra ed invia gli indici degli elementi presenti nella sua memoria. Inoltre, per garantire l'autenticazione, il **decompressore** firma con la sua chiave privata il messaggio.
2. **Calcolo della disposizione**: Il **compressore** riceve il messaggio cifrato e ne verifica l'autenticità controllando la firma del **decompressore**.

Se la firma è verificata, decifra il messaggio utilizzando la propria chiave privata e identifica il sottoinsieme di elementi comuni confrontando gli indici ricevuti con la conoscenza presente nella sua memoria. Una volta identificato il sottoinsieme comune, il **compressore** calcola una disposizione pseudocasuale degli indici comuni e la cifra utilizzando la chiave pubblica del **decompressore**. Inoltre, firma il messaggio utilizzando la propria chiave privata per garantirne l'autenticità.

3. **Ricezione della disposizione**: Il **decompressore** riceve il messaggio e ne verifica la firma. In caso positivo, decifra il messaggio e ottiene la disposizione calcolata dal **compressore**.

4. **Generazione della chiave simmetrica**: In maniera indipendente, il **compressore** e il **decompressore** generano una chiave simmetrica tramite una funzione di derivazione delle chiavi (KDF) utilizzando come *seed* la concatenazione degli elementi individuati dalla disposizione scambiata.

Capitolo 2

Implementazione

2.1 Organizzazione del progetto

La soluzione proposta è stata realizzata utilizzando il linguaggio C++ e per l'analisi dei risultati viene utlizzato il linguaggio Python. Nello specifico, sono state implementate varie soluzioni crittografiche presenti dalla libreria *Crypto++*. La scelta di *Crypto++* è stata motivata dalla sua comprovata capacità di fornire algoritmi crittografici testati, affidabili e ottimizzati, garantendo un livello superiore di sicurezza e prestazioni nelle implementazioni sviluppate.

2.2 Struttura del progetto

Il nucleo del programma è rappresentato dalle classi Compressore.cpp e Decompressore.cpp. Entrambe le classi saranno progettate in modo simile, poiché svolgeranno funzioni analoghe. Prenderemo in considerazione solo la classe Compressore, ma le stesse osservazioni si applicano anche a Decompressore. All'interno di ciascuna classe, troviamo:

- Metodo costruttore che inizializza alcuni attributi della classe;
- Metodi per la gestione del canale comunicazione;
- Metodi di crittografia a chiave pubblica che verranno usati dalle rispettive parti per cifrare, decifrare, firmare e verificare la firma di un messaggio;
- Metodi per la gestione degli indici scambiati;

- Metodo per la creazione della chiave condivisa;
- Altri metodi di utility.

Procediamo ora nell'analisi più approfondita della gestione del flusso degli eventi, ponendo attenzione all'utilizzo delle primitive crittografiche scelte per lo schema presentato.

2.2.1 Cifratura asimmetrica

La cifratura a chiave asimmetrica è un paradigma crittografico che impiega due chiavi distintive ma correlate: una chiave pubblica e una chiave privata. Questo metodo offre diversi vantaggi, tra cui la facilità nella gestione delle chiavi e la sicurezza delle comunicazioni. Nel nostro contesto, il compressore e il decompressore saranno dotati delle proprie chiavi pubbliche e private, consentendo il loro utilizzo durante le comunicazioni. Entrambe le chiavi sono generate all'interno delle rispettive classi: "Compressione.cpp" e "Decompressore.cpp" mediante l'impiego delle seguenti funzioni messe a disposizione dalla libreria *Crypto++*:

1. **params GenerateRandomWithKeySize(rng, 2048)**;
2. **RSA::PrivateKey privateKey(params)**;
3. **RSA::PublicKey publicKey(params)**.

dove **params** è un oggetto che rappresenta i parametri da utilizzare per la creazione delle chiavi crittografiche, **rng** è un oggetto associato che identifica un generatore di numeri casuali e **2048** indica la lunghezza della chiave in bit. La comunicazione e l'utilizzo delle chiavi avviene seguendo la logica descritta:

1. Il decompressore dovrà inviare i propri indici al compressore, cifrandoli utilizzando la chiave pubblica del compressore, in modo che quest'ultimo possa decifrarli con la propria chiave privata. Nel nostro codice, gestiamo questo meccanismo nella seguente funzione:

```
1        std::string Decompressore:: encryptMessageRSA(std::string
     message, RSA::PublicKey destinationPKey) {
2
3        std::string encrypted;
4        RSAES_OAEP_SHA_Encryptor encryptor(destinationPKey);
5
6        StringSource(message, true, new PK_EncryptorFilter(rng,
    encryptor, new StringSink(encrypted)));
7
8        return encrypted;
9
10       }
11
```

Listing 2.1: create-shared-key

L'input della funzione è costituito da:

- **message**: stringa contenente gli indici che il decompressore dovrà inviare al compressore;
- **destinationPKey**: la chiave pubblica del compressore, utilizzata nella cifratura;

Tramite la chiamata a:
StringSource(message, true, new PKEncryptorFilter(rng, encryptor, new StringSink(encrypted))), il messaggio originale viene preso dalla stringa **message**, cifrato utilizzando l'oggetto **encryptor**, e il risultato viene quindi memorizzato nella variabile **encrypted**. La funzione restituisce la variabile **encrypted**, che contiene il messaggio cifrato.

Il metodo appena descritto è presente anche nella classe del compressore, poiché verrà utilizzato nei casi in cui il compressore dovrà comunicare con il decompressore garantendo la confidenzialità del messaggio. L'unica differenza, ovviamente, risiede nel fatto che il compressore cifrerà il messaggio utilizzando la chiave pubblica del decompressore, seguendo una logica simile a quanto descritto precedentemente.

2. Alla ricezione di un cifrato, il destinatario dovrà ricorrere a un meccanismo di decifratura che, tramite la propria chiave privata, consenta di risalire al contenuto del messaggio. Nel nostro codice, gestiamo questo meccanismo nella funzione **decryptMessageRSA()**:

```
1       std::string Compressore::decryptMessageRSA(std::string
    message) {
2           std::string decrypted;
3           RSAES_OAEP_SHA_Decryptor decryptor(getPrivateKey());
4
5           StringSource(message, true, new PK_DecryptorFilter(rng,
    decryptor, new StringSink(decrypted)));
6           return decrypted;
7       }
8
```

Listing 2.2: create-shared-key

L'input della funzione è il seguente:

- **message**: stringa contenente il cifrato ricevuto;

La chiamata a **StringSource(message, true, new PKDecryptorFilter(rng, decryptor, new StringSink(decrypted)))**, viene utilizzata per eseguire la decifratura del messaggio. Il cifrato, preso da **message**, viene decifrato utilizzando l'oggetto **decryptor** e il risultato viene memorizzato nella variabile **decrypted**. La funzione restituisce la variabile **decrypted**, che contiene il messaggio decifrato.

3. Per garantire l'autenticazione durante ogni scambio di messaggi, sia il compressore che il decompressore dovranno firmare il messaggio prima della trasmissione. La firma del messaggio viene creata utilizzando la chiave privata di chi vuole inviare il messaggio. Il ricevente utilizzerà la chiave pubblica del mittente per verificare e garantire che il messaggio sia stato effettivamente inviato dalla suddetta parte. Per generare la firma del messaggio, verrà utilizzata la seguente funzione (L'esempio riporta la funzione presente nella classe Compressore, ma una funzione analoga sarà presente anche nella classe Decompressore):

```
1         std::string Compressore::signMessageRSA(const std::string&
    message) {
2           RSASSA_PKCS1v15_SHA_Signer signer(getPrivateKey());
3
4           std::string signature;
5           StringSource(message, true, new SignerFilter(rng, signer,
     new StringSink(signature)));
6
7
8           return signature;
9       }
10
11
```

Listing 2.3: create-shared-key

In input, la funzione riceve:

- **const std::string& message**: puntatore al messaggio da firmare.

La chiamata a: **StringSource(message, true, newSignerFilter(rng, signer, new StringSink(signature)))**, utilizza la libreria *Crypto++* per generare la firma digitale del messaggio. Il messaggio, puntato da **message**, viene firmato utilizzando l'oggetto **signer** e la firma risultante è memorizzata nella variabile **signature**. La funzione restituisce la variabile **signature**, che contiene la firma digitale del messaggio.

4. Infine, sarà necessario utilizzare una funzione di verifica della firma che permetterà di accertarsi che la firma sia stata effettivamente prodotta dalla parte che invia il messaggio. Questa funzione di verifica utilizza la chiave pubblica del mittente e permette al destinatario di verificare l'autenticità della firma e, quindi, la provenienza del messaggio. Per verificare la firma del messaggio, verrà utilizzata la seguente funzione (L'esempio riporta la funzione presente nella classe Compressore, ma una funzione analoga sarà presente anche nella classe Decompressore):

```
1       bool Compressore::verifySignatureRSA(const std::string&
    message, const std::string& signature, const RSA::PublicKey&
    signerPublicKey) {
2           RSASSA_PKCS1v15_SHA_Verifier verifier(signerPublicKey);
3
4           // Verificare la firma
5           bool result = false;
6           StringSource(signature + message, true, new
    SignatureVerificationFilter(verifier, new ArraySink((byte*)&
    result, sizeof(result))));
7           return result;
8       }
9
```

Listing 2.4: create-shared-key

L'input della funzione è costituito da:

- **string& message**: il messaggio originale che è stato firmato digitalmente;
- **string& signature**: la firma digitale da verificare;
- **PublicKey& signerPublicKey**: la chiave pubblica del firmatario.

La chiamata a: **StringSource(signature + message, true, new SignatureVerificationFilter(verifier, new ArraySink((byte*)&result, sizeof(result))))**, utilizza la libreria *Crypto++* per verificare la firma digitale del messaggio. La concatenazione di **signature** e **message** è necessaria poiché la firma spesso include il digest del messaggio originale. Il risultato della verifica è memorizzato nella variabile booleana **result**. La funzione restituisce la variabile booleana **result**, che indica se la firma è stata verificata correttamente o meno.

2.2.2 Sincronizzazione conoscenza comune

Passiamo ora a descrivere quella che sarà la fase cruciale per la generazione delle chiavi, ovvero la sincronizzazione della conoscenza comune tra il compressore e il decompressore. La procedura di sincronizzazione è avviata dal decompressore, il quale, mediante l'impiego della funzione **checkIndexesString()** estrae gli indici dalla sua memoria e li concatena in una stringa, separandoli tramite virgole. La stringa così ottenuta sarà trasmessa in modo sicuro al compressore, tramite l'utilizzo dei metodi descritti precedentemente.

```
1  std::string Decompressore::checkIndexesString() {
2      std::string indexes = "";
3
4      for (int i = 0; i < std::size(sharedInfo); i++) {
5          if (!(sharedInfo[i].empty())) {
6              indexes = indexes + std::to_string(i) + ",";
7          }
8      }
9      indexes.pop_back();
10     return indexes;
11
12 }
```

Listing 2.5: checkIndexesString()

Successivamente, il compressore riceve la stringa di indici dal decompressore e, utilizzando la funzione **indexesInCommon()**, calcola il sottoinsieme di elementi comuni.

```
std::vector<int> Compressore::indexesInCommon(std::string
    receivedIndexes) {

    std::vector<int> indexes = tokenizeByComma(receivedIndexes);

    std::vector<int> common;

    for (int i = 0; i < indexes.size(); i++) {
        if (i < MAX_SHARED_KNOWLEDGE && !(sharedInfo[indexes[i]].empty
    ())) {
            common.push_back(indexes[i]);
        }

    }

    return common;
}
```

Listing 2.6: indexesInCommon()

Una volta ottenuto tale sottoinsieme, il processo procede con la selezione della disposizione con ripetizione attraverso l'uso della funzione **createDisposition()**. La stringa generata sarà una distribuzione pseudocasuale con ripetizione di 20 elementi,

```
std::string Compressore::createDisposition(std::vector<int>
    commonIndexes) {
    CryptoPP::AutoSeededRandomPool rnd;
    std::string indexDisposition = "";

    for (int i = 0; i < 20; i++) {
        int rand = rnd.GenerateWord32(0, commonIndexes.size() - 1);
        indexDisposition = indexDisposition + std::to_string(
    commonIndexes[rand]) + ",";
    }

    return indexDisposition;
}
```

Listing 2.7: createDisposition()

La disposizione risultante verrà trasmessa in modo sicuro al decompressore, seguendo il protocollo di sicurezza.

A partire da questa disposizione, si potrà procedere, come illustrato nel paragrafo successivo, alla generazione della chiave condivisa.

2.2.3 Generazione chiave condivisa

Una volta che compressore e decompressore sono in possesso della disposizione degli indici degli elementi in comune, si può passare alla fase di generazione della chiave condivisa attraverso l'utilizzo del seguente codice.

```
void createSharedKey(std::string disposition) {
    std::vector<int> indexes = tokenizeByComma(disposition);
    std::string concatenation = "";
    for (int i = 0; i < indexes.size(); i++) {
        concatenation = concatenation + sharedInfo[indexes[i]];
    }

    concatenation = calculateHash(concatenation);

    const byte* ikm = reinterpret_cast<const byte*>(concatenation.data
    ());
    const size_t ikmLength = sizeof(ikm) - 1; // -1 per escludere il
    terminatore null
    hkdf.DeriveKey(sharedKey, derivedKeyLength, ikm, ikmLength, salt,
    saltLength, info, infoLength);
}
```

Listing 2.8: createSharedKey()

La funzione **createSharedKey()** prende come input una stringa **disposition** che contiene gli indici degli elementi separati da una virgola. Gli indici vengono estratti ed inseriti in un array, da cui vengono selezionati gli elementi corrispondenti e concatenati. Una volta ottenuta la concatenazione degli elementi selezionati dalla conoscenza comune, viene applicata una funzione di *hash* su quest'ultima per ottenere una stringa più compatta. A questo punto è necessario effettuare un casting della concatenzaione in un array di byte. Per il corretto funzionamento della *HKDF* è necessario avere la lunghezza dell'array di byte diminuito di uno per escludere il terminatore *null*. Per la generazione della chiave viene utilizzata la funzione *hkdf.DeriveKey* che prende in input i seguenti elementi:

- **sharedKey**: variabile che conterrà l'output della funzione;
- **derivedKeyLenght**:la lunghezza della chiave generata, nel nostro caso 256 byte;
- **ikm** è l'array di byte ottenuto in precedenza;
- **ikmLenght** è la lunghezza dell'array di byte;
- **salt** utilizzato per introdurre variabilità;

- **saltLenght** la lunghezza del salt;
- **info** è usata per aggiungere info addizionali;
- **infoLength** è la lunghezza dell'info;

Il codice mostrato viene utilizzato sia dal compressore che dal decompressore, i parametri utilizzati risultano essere gli stessi, e di conseguenza la chiave sarà uguale per entrambi.

2.3 Sicurezza dello schema

Lo schema di generazione di chiavi simmetriche condivise presentato risulta essere abbastanza articolato ed espone, per costruzione, dei punti cruciali in cui un attaccante attivo, di tipo *man in the middle* (*MITM*), potrebbe cercare di inserirsi. Lo scopo di questo paragrafo è quello di fornire una dimostrazione della robustezza della soluzione proposta ad attacchi di questo tipo, sottolineando l'importanza delle primitive di crittografia scelte durante la fase di progettazione. I punti di attacco per un avversario *MITM* sono individuati, chiaramente, dagli accessi al canale condiviso da parte del compressore e del decompressore, dunque le finestre da analizzare risultano essere:

1. Trasmissione degli indici del decompressore;
2. Trasmissione della disposizione da utilizzare.

2.3.1 Trasmissione degli indici del decompressore

Il flusso d'esecuzione comincia con la comunicazione sul canale condiviso degli indici degli elementi della conoscenza condivisa in possesso del decompressore. Il messaggio costruito dal decompressore viene cifrato con la chiave pubblica del compressore utilizzando lo schema di cifratura asimmetrico RSA-OAEP ed inviato sul canale. La cifratura con RSA-OAEP, utilizzato con chiavi di lunghezza adeguata in base alle raccomandazioni correnti, garantisce il grado di sicurezza CCA ed è stato scelto a discapito dell'implementazione di RSA-PCKS1.5 messa a disposizione in quanto resistente anche ad attacchi di tipo *padding oracle*. Con queste premesse, il messaggio inviato risulta essere indistinguibile da uno randomico ed è dunque computazionalmente troppo oneroso per un avversario polinomiale di tipo CCA ricostruire il testo in chiaro e dunque ricavare informazioni in merito agli indici della conoscenza

comune a disposizione del decompressore. In questo caso, un avversario *MITM* potrebbe reagire nel modo seguente:

1. intercetta e scarta il messaggio inviato dal decompressore sul canale di comunicazione;

2. costruisce un messaggio contenente dei possibili indici della conoscenza comune a cui lui ha accesso;

3. cifra ed invia sul canale il messaggio costruito con la chiave pubblica del compressore.

Sebbene questo tipo di intromissione sia del tutto alla portata di un avversario attivo che ha a disposizione l'accesso ad un numero $n >= 5$ di elementi della conoscenza comune, nello schema di generazione di chiavi simmetriche proposto il decompressore è tenuto anche ad inoltrare una firma del messaggio generata mediante RSA in maniera tale da garantirne l'autenticazione. In questo caso, l'avversario *MITM* non riuscirà a forgiare una firma del decompressore valida per il messaggio da lui costruito grazie alla robustezza della primitiva di autenticazione implementata. L'unica strategia ancora valida per l'attaccante risulta essere quella di modellare un attacco di tipo *REPLY* su una coppia $< cifrato - firma >$ a lui favorevole. La contromisura a questo tipo di attacco è data dalla costruzione dello schema e sarà approfondita nella prossima sezione.

2.3.2 Trasmissione della disposizione da utilizzare

Una volta accertata la confidenzialità e l'autenticazione della comunicazione degli indici in possesso del decompressore, il passo successivo è garantire la sicurezza dell'inoltro della disposizione di indici da utilizzare per la generazione della chiave simmetrica. Come descritto nei paragrafi precedenti, il compressore calcola l'intersezione tra i propri indici di conoscenza comune e quelli ricevuti dal decompressore e su questo sottoinsieme genera in maniera pseudocasuale una disposizione con ripetizione di 20 elementi. La comunicazione della disposizione scelta avviene in maniera analoga al flusso di dati ricevuto dal decompressore, soltanto in direzione inversa. Il compressore dunque cifra il messaggio con la chiave pubblica del decompressore tramite RSA-OAEP ed allega la firma del messaggio ottenuta con RSA. Grazie alle garanzie di sicurezza delle primitive crittografiche utilizzate, il messaggio contenente la disposizione risulta essere indistinguibile da un messaggio casuale all'analisi dell'attaccante il quale, dunque, non riesce ad ottenere informazioni

in merito a ciò che verrà utilizzato come *seed* della *key derivation function*. L'avversario *MITM* potrebbe, come nel flusso precedente, intercettare il messaggio e sostituirne uno da lui costruito da inviare al decompressore. Questo approccio, tuttavia, non garantirebbe nessun vantaggio in merito all'esfiltrazione dell'informazione che il compressore si appresterà ad inviare, in quanto essa sarà trasformata in accordo alla chiave generata in locale dalla *key derivation function*, e sarebbe comunque identificato dal decompressore, in quanto il messaggio non risulterebbe autenticato dal compressore. L'ultimo scenario da considerare, già anticipato nella sezione precedente, è quello di un attacco di tipo *REPLY*.
Si consideri una coppia $< cifrato - firma >$ valida, in cui il cifrato individui un insieme minimale di 5 indici comunicati dal decompressore al compressore al passo precedente e contenga solo indici che sono anche a disposizione dell'avversario *MITM*, il quale ha appositamente selezionato questa coppia. Si consideri inoltre, per il corretto funzionamento dello schema, il caso in cui il compressore sia in possesso di tutti gli indici indicati nella coppia. In questo caso il compressore produrrà in maniera pseudocasuale una disposizione con ripetizione su un sottoinsieme di elementi di cardinalità 5, la quale verrà utilizzata per la derivazione della chiave simmetrica. Dunque, il *seed* generato sarà una delle 5^{20} disposizioni con ripetizione possibili a partire dal sottoinsieme di indici comune. Naturalmente, anche disponendo di tutti e soli gli elementi del sottoinsieme, la probabilità che l'avversario ha di generare lo stesso input per la *key derivation function* è di $1/5^{20}$. Dunque risulta essere computazionalmente troppo oneroso per l'attaccante computare la stessa chiave simmetrica generata da compressore e decompressore anche a partire da un sottoinsieme minimale e a lui noto di informazione condivisa.

2.3.3 Cripto-Huffman

Per testare le performance del sistema, abbiamo implementato una versione del *Compression Cryptosytem* di Huffman che utilizzasse la chiave simmetrica K generata tramite la procedura descritta nei paragrafi precedenti per la scelta delle trasformazioni da applicare. A partire dalle proposte descritte da Y.Gross et al. [4] e dalle soluzioni forniteci dai colleghi che ci hanno preceduto, sono state implementate due modalità operative:

1. **K-transformations:** che applica le trasformazioni a k nodi scelti in base alla chiave;

2. **Bulk-transformations:** che applica le trasformazioni a tutti gli elementi dell'albero ai quali corrisponde un bit a 1 nella chiave.

Di seguito è riportata un'analisi approfondita dell'implementazione delle due modalità operative.

2.3.4 K-transformations

In questa variante sono scelte k foglie presenti nell'albero binario e viene applicata una trasformazione di *level-swap* su ciascuna di esse. Per selezionare le k foglie *target* della trasformazione, si utilizza una parte corrispondente a $k \log \sigma$ bit della chiave simmetrica, dove σ è la dimensione dell'alfabeto Σ e k è un numero compreso tra 1 e σ. Nell'implementazione proposta, l'albero viene visitato in logica *preorder* e, a seconda del bit della chiave preso in considerazione, viene selezionata o meno la foglia corrispondente. Al fine di introdurre ulteriore variabilità, il bit della chiave selezionato viene invertito prima di codificare un nuovo simbolo.

Di seguito è riportata l'implementazione.

```
1  void Huffman::runKTransformations(Node* root, string key, int k) {
2
3      string sKey = getConvertedKey(key, &k);
4      string remKey;
5
6      int diff = sKey.length() - (k * (frequencies.size() / 2));
7
8      if (diff == 0)
9      {
10         remKey = sKey;
11     }
12     else
13     {
14         remKey = sKey.substr(sKey.length() - diff, sKey.length());
15     }
16     vector<Node*> chosenNodes;
17     int counter = 0;        //Numero di foglie da aggiungere.
18     int startingBit = 0;    //Valore iniziale del bit da ricercare
       nella chiave
19
20     for (int i = 0; i < frequencies.size(); i++)
21     {
22         counter = k;
23         chosenNodes.clear();
24         addKLeaves(root, &chosenNodes, &counter, remKey, startingBit);
25
26         if (startingBit == 0)
27         {
28             startingBit = 1;
29         }
30         else
31         {
32             startingBit = 0;
33         }
34         for (auto element : chosenNodes)
35         {
36             transformationCoding(2, root, element->data); //Level-Swap
37             reCalcFreq(root);
38         }
39     }
40 }
```

Listing 2.9: K-transformations

2.3.5 Bulk-transformations

La modalità operativa *Bulk-transformations*, a differenza di quella precedente, applica le trasformazioni soltanto alle foglie corrispondenti ai bit di valore 1 della chiave simmetrica. In questo caso sono necessari soltanto σ - 1 bit della chiave K e la trasformazione applicata alle foglie selezionate è la *level-swap*. Di seguito é riportata l'implementazione.

```
1  void Huffman::runBulkTransformations(Node* root, string key) {
2
3      int k = 1;
4      string sKey = getConvertedKey(key, &k);
5      vector<Node*> chosenNodes;
6      addHuffmanTreeLeaves(root, &chosenNodes);
7
8      int c = 0;
9      for (char ch : sKey)
10     {
11         if (ch == '0')
12         {
13             chosenNodes.erase(chosenNodes.begin() + c);
14         }
15         else c++;
16     }
17
18     for (auto element : chosenNodes)
19     {
20             transformationCoding(2, root, element->data); //Level-Swap
21             reCalcFreq(root);
22     }
23 }
```

Listing 2.10: Bulk-transformations

2.3.6 Resistenza contro attacchi CPA: inserimento di caratteri DC

Come suggerito nella definizione del *Compression Cryptosytem* di Huffman [4], per evitare attacchi di tipo *chosen plaintext* l'implementazione proposta consente di specificare un numero di caratteri *don't care* (DC) da inserire. L'introduzione di questi caratteri in posizioni scelte in maniera pseudocasuale del file da codificare consente di aggiungere ulteriore variabilità all'output generato, in modo tale da rendere qualunque strategia di attacco CPA computazionalmente paragonabile alla distinzione di due file casuali.

Di seguito viene illustrata la modalità di inserimento dei caratteri DC all'interno del file.

```
...
    int currOutput = 0; //Numero di caratteri inseriti
    int currPos = 0;
    int numOutput = numDC; //Numero di caratteri da inserire

    while (currOutput < numOutput)
    {
        int range = (inputString.length() - 1) - currPos + 1;
        cout << range << endl;
        currPos = rand() % range + currPos;
        char previousvalue = inputString[currPos];

        //Verifica se il carattere non ha subito modifiche
        if (int(previousvalue) >= 0) {
            char newvalue = (char) int(previousvalue) - int(0x7f);
            inputString[currPos] = newvalue;
            currOutput++;
        }

        //Riporta il cursore all'inizio se siamo a fine stringa
        if (currPos == inputString.length() - 1)
        {
            currPos = 0;
        }
    }

 ...
```

Listing 2.11: Inserimento DC

Capitolo 3

Discussione Risultati

3.1 Analisi degli output

3.1.1 Distribuzione sottostringhe binarie

In questo studio abbiamo esplorato le dinamiche di due distinte esecuzioni, la prima per la variante *Bulk-transformations* e la seconda per quella *K-transformations*, all'interno di un contesto di analisi delle frequenze delle sottostringhe binarie. L'analisi delle occorrenze di sottostringhe di taglia crescente rappresenta un semplice test per visualizzare la distribuzione globale dei file in output e consente di effettuare un rapido paragone con file generati da sorgenti casuali. Le seguenti tabelle illustrano i risultati ottenuti tramite le due modalità operative implementate.

Concludiamo dicendo che le analisi delle frequenze in *Bulk-transformations* e in *K-transformations* fornisce una visione dettagliata delle caratteristiche dei dati. Una distribuzione equilibrata di 0 e 1, e delle variazioni delle sottostringhe binarie di lunghezza 2 e 3 tra le due esecuzioni mostra che i file codificati prodotti in output sono paragonabili a file generati in maniera randomica.

	Frequenza	Percentuale
'0'	8682766	48.92%
'1'	9067955	51.08%
'00'	2869177	18.95%
'01'	4584342	30.28%
'10'	4584343	30.28%
'11'	3100620	20.48%
'000'	1145696	7.33%
'001'	2417664	15.48%
'010'	1804618	11.55%
'011'	2413441	15.45%
'100'	2417665	15.48%
'101'	1800428	11.53%
'110'	2413442	15.45%
'111'	1208117	7.73%

Dati per Bulk-transformations

	Frequenza	Percentuale
'0'	9048743	50.98%
'1'	8701962	49.02%
'00'	3123333	21.61%
'01'	4200810	29.07%
'10'	4200810	29.07%
'11'	2926700	20.25%
'000'	1358457	9.13%
'001'	2219120	14.92%
'010'	1625537	10.93%
'011'	2215870	14.89%
'100'	2219120	14.92%
'101'	1634841	10.99%
'110'	2215871	14.89%
'111'	1389469	9.34%

Dati per K-transformations

3.1.2 Distanza di Hamming normalizzata

Per valutare la resistenza del file compresso agli attacchi CPA, abbiamo calcolato la distanza di Hamming normalizzata tra due file compressi generati al termine della fase di codifica utilizzando:

- lo stesso file in input;
- due chiavi *k1* e *k2*, generate in maniera indipendente come suggerito da Y.Gross et al. [4];
- la stessa modalità operativa.

L'analisi è stata condotta verificando, a coppie, le perturbazioni generate dall'inserimento di 200, 400 e 600 caratteri DC. I seguenti grafici mostrano i risultati ottenuti dalle varianti *Bulk-transformations* e *K-transformations*.

I risultati ottenuti per entrambe le modalità operative sono soddisfacenti e sono prossimi a 0.50 già a partire dal numero minimo di caratteri DC considerati (200). Pertanto, a partire dai dati ricavati, possiamo concludere che l'implementazione presentata garantisce un'elevata resistenza contro attacchi di tipo CPA.

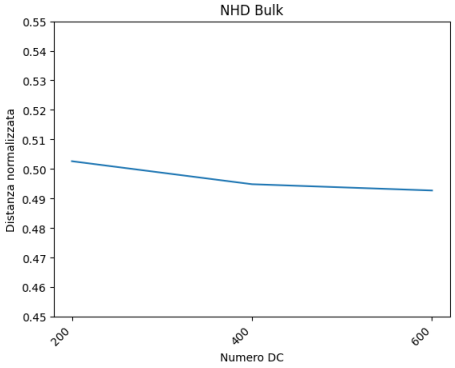

Figura 3.1: NHD Bulk

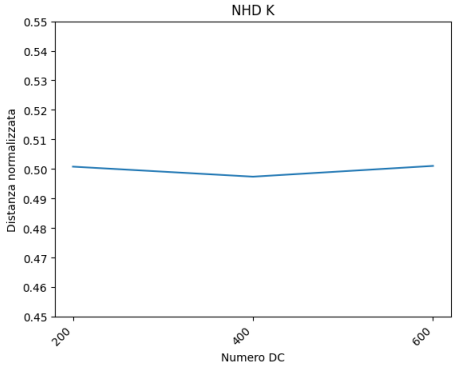

Figura 3.2: NHD K

Capitolo 4

Conclusioni e Sviluppi Futuri

A partire dalle precedenti implementazioni del *Compression Cryptosystem* di Huffman, il lavoro proposto affronta il problema della generazione e della condivisione della chiave simmetrica tramite la quale sono scelte le trasformazioni da applicare all'informazione trasmessa. L'approccio illustrato consente di sfruttare elementi, anche non necessariamente sincronizzati, che le due parti hanno a disposizione in modo tale da evitare ritrasmissioni e introdurre un eccessivo *overhead* di comunicazione. Al contempo, sebbene lo schema sia relativamente leggero dal punto di vista dell'invio delle informazioni preliminari per il calcolo della chiave condivisa, sono state adottate primitive crittografiche in grado di rendere robusta l'interazione tra compressore e decompressore anche sotto nozioni di sicurezza più elevate rispetto alle trattazioni precedenti. In particolare, lo schema proposto è stato progettato per contrastare scenari con attaccanti attivi di tipo *MITM* e le primitive crittografiche introdotte sono state scelte per garantire la robustezza del sistema nei suoi principali punti d'attacco. Per testare gli effetti della chiave generata dallo schema presentato, essa è stata utilizzata nel contesto dell'implementazione del *Compression Cryptosystem* di Huffman e i risultati raggiunti si sono dimostrati soddisfacenti ed in linea a quelli ottenuti mendiante l'utilizzo di una chiave randomica fissata. In conclusione, il sistema presentato rappresenta un tentativo di risolvere il problema della generazione di chiavi simmetriche aggiungendo il minor *overhead* possibile senza introdurre vulnerabilità sfruttabili da un attaccante attivo sul canale di comunicazione.

Possibili sviluppi futuri potrebbero essere quelli di adattare lo schema presentato ad altri sistemi di compressione, quali ad esempio LZW o Garsia-Wachs, confrontando l'efficienza e la sicurezza rispetto all'implementazione di Huffman oppure modellare lo scambio di informazioni preliminari senza l'utilizzo della crittografia a chiave pubblica.

Bibliografia

[1] Elaine Barker, Allen Roginsky e Richard Davis.
≪Recommendation for Cryptographic Key Generation≫.
In: *NIST Special Publication 800-133 Revision 2* (2012).

[2] Bruno Carpentieri.
≪Efficient compression and encryption for digital data transmission≫.
In: *Security and Communication Networks* (2018).

[3] ≪Crypto++≫. In: *https://cryptopp.com/* ().

[4] Yoav Gross et al. ≪A Huffman Code Based Crypto-System≫.
In: *Ariel University and Bar Ilan University* (2022).