



SAPIENZA
UNIVERSITÀ DI ROMA

Studio e Implementazione di Attacchi ai Sistemi di Cifratura

Ingegneria dell'informazione, Informatica e Statistica
Informatica

Andrea Maggiore

Matricola 1947898

Relatore

Prof. Emiliano Casalicchio

Anno Accademico 22/23, V Sessione di laurea

Tesi non ancora discussa

Studio e Implementazione di Attacchi ai Sistemi di Cifratura
Sapienza Università di Roma

© 2023 Andrea Maggiore. Tutti i diritti riservati

Questa tesi è stata composta con L^AT_EX e la classe Sapthesis.

Email dell'autore: maggiore.1947898@studenti.uniroma1.it

*Ai miei genitori, soprattutto a mio padre, per avermi aiutato nel percorso di studio
e per la stesura di questa tesi.*

*A mio fratello Riccardo, per avermi motivato nella vita e nello studio e per essere
il miglior fratello che si potesse mai desiderare.*

A mia nonna e mio zio, per il loro sostegno continuo.

*A Marina, per avermi aiutato con le varie correzioni di questa tesi, ma soprattutto
per riempire ogni giorno che passa la mia vita nel miglior modo che si possa
chiedere, per trovare sempre la soluzione nei momenti più duri, per risollevarmi
dai momenti più difficili e per aver tirato fuori il meglio di me.*

*A Gianmarco, Riccardo ed Umberto, per essere degli amici fantastici e compagni
di vita.*

*A Michela e Stefano, per la loro disponibilità e l'aiuto che mi hanno dato in questi
3 anni.*

*A Lorenzo, per aver condiviso paure e gioie e soprattutto per essere stato un
compagno di studi straordinario.*

Indice

1	Introduzione	7
2	I sistemi di cifratura	9
3	Attacchi ai sistemi di cifratura	17
3.1	Attacco tramite dizionario	19
3.1.1	Aggiunta di un utente	20
3.1.2	Cifratura della password con il salt	21
3.1.3	Implementazione dell'attacco	23
3.2	Brute Force: ciphertext only	25
3.2.1	Cifratura tramite DES	26
3.2.2	Implementazione attacco	28
3.3	Brute Force: attacco plaintext	31
3.3.1	Implementazione attacco	32
3.4	Chosen ciphertext	34
3.4.1	CCA è oneroso sull'algoritmo del DES	34
3.4.2	Cifratura tramite RSA	35
3.4.3	Implementazione attacco	38
4	Conclusioni	41

Capitolo 1

Introduzione

La crittografia e la sicurezza informatica rivestono un ruolo cruciale nel mondo digitale contemporaneo.

In particolare, la crittografia, è il modo di proteggere le informazioni attraverso la trasformazione di dati in un formato che risulta illeggibile senza la chiave appropriata per decifrarli. Questo processo garantisce la riservatezza delle comunicazioni e dei dati, impedendo l'accesso non autorizzato da parte di terze parti. Ne consegue che la crittografia è fondamentale per preservare la privacy degli individui, la riservatezza delle transazioni finanziarie, la sicurezza delle comunicazioni aziendali, se non addirittura la sicurezza nazionale.

La sicurezza informatica, d'altra parte, è un concetto più ampio che incorpora diverse pratiche e tecnologie volte a proteggere i sistemi informatici da minacce quali malware, attacchi di phishing, accessi non autorizzati e altro ancora. La sua importanza è evidente considerando il continuo aumento degli attacchi informatici che mirano a compromettere dati personali, informazioni aziendali sensibili e addirittura infrastrutture critiche.

In un contesto in cui la digitalizzazione è onnipresente, la mancanza di adeguati protocolli di sicurezza potrebbe avere conseguenze devastanti. Di conseguenza, le organizzazioni, le istituzioni e gli individui devono adottare misure efficaci per proteggersi da tali minacce.

Per quello che concerne la crittografia, queste misure sono in sostanza metodi sofisticati per nascondere l'informazione in transito e renderle comprensibili soltanto al destinatario, facendo in modo di rendere il più difficile possibile la traduzione in chiaro delle informazioni.

Tuttavia nessun sistema di crittografia può essere considerato totalmente inespugnabile, poiché la sicurezza è strettamente legata al tempo di calcolo disponibile per decifrare un messaggio crittografato.

L'avvento di computer sempre più potenti e algoritmi avanzati può ridurre significativamente il tempo necessario per rompere una crittografia. Ciò significa che un algoritmo di crittografia considerato sicuro oggi potrebbe diventare vulnerabile in futuro, quando nuove tecnologie renderanno disponibili risorse di calcolo più potenti. (**Languasco e Zaccagnini 2020a**)

Un altro aspetto critico riguarda l'implementazione dell'algoritmo ed in particolare l'aspetto del trasferimento di informazioni per la creazione della chiave di decip-

tazione, o lo scambio della chiave stessa che rappresenta un punto di debolezza secondo il principio di **Kerckhoffs**: "*Un crittosistema deve essere sicuro anche se il suo funzionamento è di pubblico dominio, con l'eccezione della chiave*". (**Kerckhoffs 1883**)

Le tecniche crittografiche possono essere suddivise in diverse categorie, ognuna con le proprie caratteristiche e applicazioni. Di seguito, una sintesi dei metodi considerate in questa tesi:

1. **Hashing**: Non è strettamente una tecnica crittografica, ma svolge un ruolo cruciale nella sicurezza informatica. Produce una stringa di lunghezza fissa (**digest**) da un input di lunghezza variabile. Comunemente utilizzato per verificare l'integrità dei dati e generare firme digitali. Algoritmi di hash noti includono SHA-256 (Secure Hash Algorithm) e MD5 (Message Digest Algorithm 5).
2. **Crittografia Simmetrica**: Questo approccio coinvolge l'utilizzo di una singola chiave per cifrare e decifrare i dati. Algoritmi comuni includono **DES** (Data Encryption Standard) ed **AES** (Advanced Encryption Standard). La sfida principale è la gestione sicura della chiave condivisa tra mittente e destinatario.
3. **Crittografia Asimmetrica (o a chiave pubblica)**: Coinvolge l'uso di una coppia di chiavi: una chiave pubblica utilizzata per cifrare e una chiave privata per decifrare. **RSA (Rivest-Shamir-Adleman)** è un esempio noto di algoritmo asimmetrico. Risolve il problema della gestione delle chiavi condivise, ma può essere computazionalmente più oneroso rispetto alla crittografia simmetrica.

Queste formano un insieme diversificato di strumenti utilizzati per affrontare specifici requisiti di sicurezza in diversi contesti. La loro implementazione e combinazione strategica sono cruciali per garantire una protezione efficace delle informazioni.

Un concetto che si è andato a creare negli anni è quello del "**Password Cracking**", un insieme di metodologie finalizzate al recupero non autorizzato di password, con implicazioni dannose per l'integrità dei dati e la privacy degli individui e non solo. Le pratiche di password cracking spaziano da tentativi diretti e metodici fino a sofisticate strategie d'ingegneria sociale, rappresentando una sfida costante per la sicurezza digitale. Questo fenomeno genera una necessità di continuo miglioramento delle misure di protezione delle informazioni.

Questa tesi si propone di esplorare una gamma di approcci e tecniche impiegate nell'ambito del password cracking, fornendo un'analisi dei metodi più noti. In particolare, saranno esaminati "**attacco tramite dizionario**", attacchi di tipo "**brute force**" ed il "**chosen ciphertext**".

La tesi ha la seguente struttura:

- Nel capitolo due andremo a parlare dei sistemi di cifratura, in particolare dello SHA256, dell'algoritmo DES e del RSA.
- Nel capitolo tre affronteremo degli attacchi sopracitati.
- Il quarto ed ultimo capitolo rappresenta la conclusione della tesi.

Capitolo 2

I sistemi di cifratura

Per crittografia, o cifratura, si intende un procedimento secondo il quale un testo in chiaro viene convertito in una sequenza incomprensibile tramite l'utilizzo di una chiave. Nel migliore dei casi il contenuto del testo segreto è accessibile solo se la cifratura viene resa nulla grazie all'utilizzo della chiave in questione. Oggigiorno i cifrari, ovvero gli algoritmi utilizzati per determinati processi di cifratura o decifratura, vanno però oltre alla codifica di mere comunicazioni testuali, e vengono utilizzati anche per altre informazioni elettroniche come messaggi vocali, immagini o codici di programmazione.

La crittografia viene utilizzata per proteggere file, drive e directory vari da accessi indesiderati o per garantire una trasmissione dati sicura.

Cifratura a trasposizione

CIFRARIO DI GIULIO CESARE																				
A	B	C	D	E	F	G	H	I	L	M	N	O	P	Q	R	S	T	U	V	Z
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21

Esempi: CASA = FDVD - ROMA = URPD

Figura 2.1. Un esempio di cifrario di Cesare. (Rognetta 1999)

Già nell'antichità venivano utilizzati dei cifrari, sebbene ben più semplici di quelli attuali, i quali si limitavano alla codifica delle informazioni da proteggere. In questi casi, dei singoli simboli, parole o intere frasi del testo in chiaro venivano dislocati all'interno del messaggio, tecnica chiamata cifrario a trasposizione, o sostituiti tramite delle combinazioni alternative dei simboli, cifrario a sostituzione. Per poter decodificare un testo criptato in questo modo, il destinatario doveva solitamente conoscere la regola con la quale il testo era stato criptato in primo luogo. Le permutazioni riguardanti il cifrario a trasposizione avvenivano solitamente grazie all'utilizzo di una matrice, ovvero al fine di rielaborare un testo segreto a trasposizione nel testo in chiaro, deve dunque essere conosciuta o ricostruita la matrice

utilizzata per la trasposizione.

Il cifrario di Giulio Cesare è di tipo *A sostituzione per traslazione*, quindi con un numero di chiavi, pari al numero delle lettere dell'alfabeto, per questo appare assai debole come sistema, una volta scoperto che esso si basa su una traslazione di caratteri, è facile indovinare la chiave con pochi tentativi.

Cifratura a sostituzione

CIFRARIO A SOSTITUZIONE PER CORRISPONDENZA DIRETTA																									
A	B	C	D	E	F	G	H	I	L	M	N	O	P	Q	R	S	T	U	V	Z					
D	G	Q	V	Z	T	R	I	M	O	P	S	A	C	B	U	Z	F	H	N	E					
3	7	4	11	15	21	6	9	10	13	14	16	17	20	19	1	5	2	8	12	18					

CASA = QDZD - ROMA = UAPD

CASA = 4-3-5-3 ROMA = 1-17-14-3

Figura 2.2. Un esempio di cifratura a sostituzione. (Rognetta 1999)

Per rendere più sicura la cifratura, si iniziò ad elaborare il testo da cifrare in modo non più lineare le lettere dell'alfabeto chiaro a quello cifrante.

Con il passare del tempo sono stati impiegati i più disparati sistemi di cifratura che si fondavano su metodi in sintonia con le nuove conoscenze matematiche e tecnologiche.

La consapevolezza delle debolezze di questi sistemi di cifratura del passato hanno determinato un'evoluzione della crittografia moderna nel senso di fondare la sua efficacia sulla segretezza della chiave e non del metodo di cifratura usato, ovvero l'algoritmo, che può addirittura essere conosciuto: "*Un sistema dovrebbe essere progettato secondo l'assunzione che il nemico acquisirà immediatamente familiarità con esso*". (**Shannon 1949**)

Infatti, non è possibile compiere la decifrazione a causa della segretezza e soprattutto della solidità della chiave.

La crittografia moderna, in particolare, inizia nel XX secolo ad avvalersi di processi automatici di cifratura, derivanti dall'impiego di macchine automatiche. Sono esempio i nastri perforati di Gilbert Vernam, oppure Enigma, usata dai tedeschi nella seconda guerra mondiale per cifrare i propri messaggi. Lo sforzo di decifrare Enigma stimolò la costruzione di macchine sempre più sofisticate per decrittare i codici nemici. Tale esigenza fu soddisfatta dal sistema Ultra progettato sugli studi del matematico Alan Turing al fine di violare, con successo, Enigma. (**Rognetta 1999**)

Con l'avvento del computer, in seguito, si realizzano cifrari sempre più complessi, grazie alla potenza dei computer; tale potenza, però, è utilizzata anche per la crittanalisi, per cui la crittografia diventa una storia di rincorse tra sistemi di crit-

tografia e relativi attacchi di crittanalisi. D'altro canto, la progressiva espansione delle comunicazioni elettroniche determina la preferenza dei sistemi di crittografia asimmetrica rispetto a quelli di crittografia simmetrica. Per questi ultimi, infatti, non è possibile trovare canali sicuri di trasmissione dell'unica chiave che serve a cifrare e decifrare; tale problema, viceversa, è irrilevante nei sistemi asimmetrici. Nel vasto panorama della sicurezza informatica, i sistemi di cifratura giocano un ruolo cruciale nel garantire la riservatezza e l'integrità delle informazioni digitali. Tra gli algoritmi crittografici adottati per questo scopo, spiccano tre protagonisti che hanno segnato significativamente il mondo della crittografia: SHA256, DES (Data Encryption Standard), e RSA (Rivest-Shamir-Adleman), che sono proprio gli algoritmi che sono stati utilizzati.

Funzioni di Hash

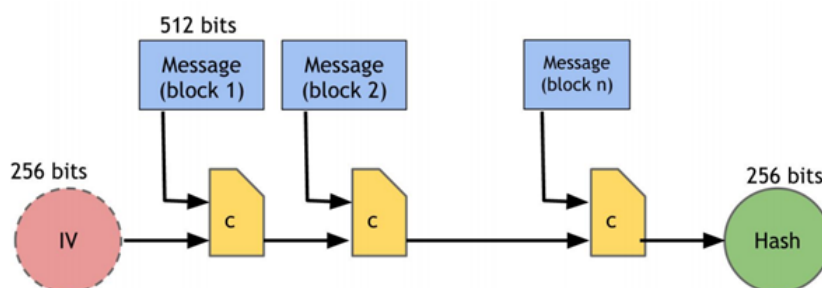


Figura 2.3. La funzione di compressione di SHA-256 opera su blocchi di messaggio di 512 bit e su valori intermedi di 256 bit. Essa è essenzialmente un algoritmo cipher block di 256 bit, con valore di input il valore hash intermedio e avente come chiave il blocco $M(\text{block } i)$ del messaggio.

Ci sono altri aspetti importanti che riguardano la trasmissione sicura delle informazioni, che non sono relazionati soltanto alla cifratura di queste. Le funzioni di Hash risolvono questi aspetti accessori, infatti oltre alla segretezza, i moderni sistemi di crittografia devono garantire:

- *L'integrità* del messaggio inviato in modo da evitare il rischio che per errori di trasmissione o per intrusione, un messaggio prevenga al destinatario corrotto rispetto all'originale.
- *L'autenticazione* dell'invio, in modo che il mittente sia riconoscibile come autore del messaggio.
- *L'Impossibilità di ripudio*, ovvero il mittente non deve poter negare la produzione del messaggio che è avvenuta per sua volontà in un certo istante.
- *L'Identificazione* del destinatario, dove il mittente deve essere sicuro dell'associazione della chiave pubblica con il destinatario.

Il termine "hash" si riferisce a una funzione che prende un input (o messaggio) e produce una stringa di caratteri alfanumerici di lunghezza fissa, "digest", che è un

rappresentante univoco del messaggio originale. In altre parole, il valore di hash è una rappresentazione sintetica e crittograficamente sicura dei dati in input.

Gli algoritmi **SHA** (Secure Hash Algorithm), indicano una famiglia di cinque diverse funzioni crittografiche hash sviluppate a partire dal 1993 dalla NSA (National Security Agency) e pubblicate dal NIST (National Institute of Standards and Technology) nel 1995 come standard federale del governo degli Stati Uniti d'America. Come ogni algoritmo hash, l'SHA, partendo da un messaggio di lunghezza variabile, produce un message digest di lunghezza fissa dal quale è impossibile risalire al messaggio originale. La versione originale, spesso denominata SHA0, conteneva una debolezza che fu in seguito scoperta dalla NSA e che portò ad un documento di revisione dello standard, l'SHA1. Gli algoritmi della famiglia SHA sono denominati SHA1, SHA224, SHA256, SHA384 e SHA512: le ultime 4 varianti sono spesso indicate genericamente con SHA2, per distinguerle dal primo.

SHA-1 produce un digest del messaggio di soli 160 bit, mentre gli altri producono digest di lunghezza in bit pari al numero indicato nella sigla. L'SHA1 è l'algoritmo più diffuso della famiglia ed è utilizzato in numerose applicazioni e protocolli; la sua sicurezza è stata in parte compromessa, come detto, dai crittoanalisti per cui si usano normalmente le versioni SHA256 e SHA512. La sicurezza dell'algoritmo dipende interamente dalla capacità di produrre un valore diverso per ogni specifico input. Quando una funzione hash produce lo stesso digest per due diversi dati, allora si dice che c'è una collisione, perciò maggiore è la resistenza alle collisioni e più efficiente sarà l'algoritmo. Una delle caratteristiche del SHA256 è la sua resistenza alle collisioni, il che significa che risulta estremamente improbabile che due insiemi di dati producano lo stesso digest. Questa proprietà è essenziale per garantire l'integrità dei dati e la sicurezza nelle applicazioni in cui l'algoritmo viene utilizzato. Infatti l'algoritmo è progettato in modo che anche un minimo cambiamento del testo in input generi un "effetto valanga" che modifica in modo completamente diverso il valore di hash.

Cifratura a chiave simmetrica

La crittografia a chiave simmetrica è caratterizzato dall'uso di una singola chiave condivisa tra le parti autorizzate per cifrare e decifrare i dati. Questa chiave segreta è cruciale per garantire la riservatezza delle informazioni e deve essere mantenuta in modo sicuro.

Nel processo di cifratura, l'algoritmo utilizza la chiave segreta per trasformare il testo in chiaro in una forma cifrata. La stessa chiave è successivamente impiegata nella decifratura per riportare il testo cifrato al suo stato originale. La scelta di algoritmi come AES (Advanced Encryption Standard) è comune in implementazioni di crittografia a chiave simmetrica, poiché offrono una buona combinazione di sicurezza ed efficienza.

La principale forza di questo approccio risiede nella sua velocità ed efficienza, rendendolo adatto alla cifratura di grandi volumi di dati. Tuttavia, la distribuzione sicura delle chiavi costituisce una sfida significativa. Per garantire la sicurezza del sistema, è essenziale proteggere la chiave segreta da accessi non autorizzati, poiché una compromissione della chiave potrebbe mettere a rischio l'intera sicurezza del sistema.

La prima fase coinvolge la "permutazione iniziale" (**IP**), in cui il blocco di dati in ingresso, di solito di 64 bit, viene riorganizzato secondo una tabella prestabilita. Questo processo prepara il terreno per una serie di iterazioni chiamate "rounds", che costituiscono il cuore dell'algoritmo.

Durante ciascun round, il blocco di dati è diviso in due parti: sinistra (**L**) e destra (**R**). La parte destra diventa la parte sinistra nel round successivo, creando un flusso continuo di trasformazioni.

Una componente chiave è la "funzione di espansione" (**E**), che allarga la metà destra del blocco a 48 bit attraverso una specifica tabella di espansione. Ogni round è associato a una sottochiave derivata dalla chiave principale, precedentemente permutata e suddivisa in sottochiavi per ciascun round.

L'uso combinato di operazioni XOR (OR esclusivo) e S-Box (scatole di sostituzione) aggiunge una componente di non linearità all'algoritmo, complicando ulteriormente il processo. Queste operazioni si ripetono in ogni round, portando il blocco di dati attraverso una serie di trasformazioni.

In particolare queste due operazioni proteggono efficacemente l'algoritmo di cifratura DES dagli attacchi **Chosen Ciphertext**.

Dopo i 16 rounds, il blocco permutato subisce una "permutazione finale" (IP-1), che è l'inverso della permutazione iniziale. Questo fornisce l'output cifrato o decifrato, a seconda dell'applicazione dell'algoritmo.

Tuttavia, a causa dell'aumento della potenza computazionale e delle tecniche di attacco, il DES è diventato gradualmente meno sicuro, e nel 2001 è stato sostituito dall'AES (Advanced Encryption Standard) come standard crittografico ufficiale negli Stati Uniti. Il DES utilizza una chiave di 56 bit per cifrare e decifrare i dati, il che lo rende vulnerabile agli attacchi di forza bruta, in cui un aggressore cerca tutte le possibili combinazioni di chiavi per rompere la crittografia.

Cifratura a chiave asimmetrica

La crittografia asimmetrica si basa su una coppia di chiavi: una segreta o privata, di cui solo il titolare ha conoscenza, e una pubblica, che è invece conoscibile da chiunque sia interessato. Le due chiavi sono:

- *complementari*, perché se una è usata per cifrare, l'altra deve essere usata per decifrare e viceversa (se si usa la stessa chiave per cifrare e decifrare la crittografia, come già detto, è simmetrica);
- *indipendenti*, perché la conoscenza della chiave pubblica non consente di risalire a quella privata.

Ciò consente di superare il problema della trasmissione insicura della chiave simmetrica: essendo il sistema fondato su una coppia di chiavi, quella pubblica può, anzi deve essere resa conoscibile, senza alcun problema di trovare canali sicuri di trasmissione. Per comprendere come mediante un sistema di chiavi asimmetriche si possa giungere a realizzare requisiti giuridici equivalenti a quelli di un documento cartaceo e della sua sottoscrizione, in riferimento al documento informatico e alla firma digitale, è opportuno esaminare tre distinte applicazioni di un sistema di crittografia asimmetrica.

L'algoritmo di **RSA** (Rivest-Shamir-Adleman) è un algoritmo di crittografia asimmetrica ampiamente utilizzato per scopi di crittografia e firma digitale. È stato sviluppato da Ron Rivest, Adi Shamir e Leonard Adleman nel 1977 ed è basato su problemi matematici difficili da risolvere, noti come il problema della fattorizzazione dei numeri interi.

Ecco una semplice rappresentazione dell'RSA:

1. **Generazione delle chiavi:** Una coppia di chiavi viene generata per l'utente: una chiave pubblica (n, e) e una chiave privata (n, d) . La chiave pubblica può essere condivisa liberamente con chiunque, mentre la chiave privata deve essere mantenuta segreta.
2. **Crittografia:** Per cifrare un messaggio da inviare a un destinatario, si utilizza la sua chiave pubblica. Il mittente converte ogni carattere di un messaggio in un numero intero e applica l'operazione di cifratura con la chiave pubblica. Questo processo rende il messaggio illeggibile a chiunque tranne al destinatario, che possiede la chiave privata corrispondente.
3. **Decrittografia:** Il destinatario utilizza la sua chiave privata per decifrare il messaggio cifrato e recuperare il testo originale.

La chiave pubblica e privata sono invertite nel caso il mittente voglia certificare il messaggio con la sua firma. La sicurezza dell'RSA si basa sulla difficoltà di fattorizzare il numero n che è il risultato della moltiplicazione di due numeri primi p e q molto grandi nella chiave. Finora, non esiste un algoritmo efficiente noto che possa fattorizzare grandi numeri composti in tempi ragionevoli.

Se qualcuno conosce solo la chiave pubblica (n, e) , la sfida è estrarre i fattori primi p e q di n per calcolare la chiave privata (n, d) . Questo è un problema computazionalmente difficile noto come il problema della fattorizzazione di numeri grandi.

RSA è stato uno dei primi algoritmi di crittografia asimmetrica ed è ampiamente utilizzato per scopi di sicurezza informatica, compresa la crittografia dei dati, la firma digitale e altri protocolli di sicurezza su Internet.

Capitolo 3

Attacchi ai sistemi di cifratura

L'esistenza e il continuo sviluppo di sistemi crittografici sono evidentemente giustificati dall'esistenza di minacce che hanno come fine quello di impossessarsi informazioni riservate per i più diversi scopi. I possibili metodi con cui queste minacce si realizzano sono essenzialmente due:

- **Passivi:** queste minacce hacker sono tipologie di attacchi informatici in cui l'attaccante monitora o intercetta il flusso di dati senza alterare attivamente o danneggiare il sistema o le informazioni. Questi attacchi si concentrano principalmente sull'ascolto, l'osservazione e la raccolta di informazioni senza alterare lo stato o la funzionalità del sistema target.
- **Attivi:** l'attaccante può intervenire sul comportamento del sistema; come l'induzione di determinate azioni, prendendo possesso di una parte di sistema e tendandone la forzatura, oppure inducendo l'uso di un sistema o un sottosistema di cui detiene il controllo. In questo tipo di attacco i bersagli possono essere:
 - *Gli algoritmi di cifratura*, per i quali si usano tecniche crittoanalitiche per invertirli e risalire al testo in chiaro.
 - *I protocolli* e le modalità operative con i quali si cerca di attaccare attraverso il modo in cui gli algoritmi crittografici vengono utilizzati, ad esempio durante lo scambio della chiave.
 - *I sistemi informatici* nei quali vengono minacciati gli ambienti operativi e le condizioni in cui operano algoritmi e protocolli (attraverso attacchi fisici a dispositivi, manomissione del software, ecc.)

Se da una parte le minacce tendono ad aggredire un metodo di cifratura inteso come insieme di algoritmo e modalità operative queste si misurano con quello che viene definito "robustezza" di un sistema di cifratura ed in particolare dell'aspetto che riguarda l'algoritmo in se.

I fattori chiave che contribuiscono alla robustezza di un algoritmo di cifratura: (Zimuel 2002)

- **Proprietà matematiche:** La sicurezza di un algoritmo può basarsi su proprietà matematiche intrinseche. Ad esempio, la difficoltà computazionale di

problemi matematici specifici può essere utilizzata per garantire la sicurezza dell'algoritmo.

Un algoritmo crittografico ben progettato dovrebbe presentare una complessità di cifratura polinomiale e al contempo una complessità di forzatura esponenziale. Questo comporta che l'aumento della capacità di calcolo, che permette codifiche con chiavi più complesse, agisce allora a favore e in modo definitivo della robustezza.

- **Resistenza agli attacchi di forza bruta:** L'algoritmo dovrebbe rendere impraticabile l'attacco di forza bruta, che coinvolge la prova di tutte le possibili chiavi fino a trovare quella corretta. Questo può essere ottenuto utilizzando chiavi sufficientemente lunghe e complesse rendendo questo tipo di resistenza un corollario della precedente.
- **Verifica e analisi pubblica:** Durante il suo uso il sistema di cifratura è spesso soggetto a una rigorosa revisione pubblica e analisi crittografica. Un algoritmo che può resistere all'analisi da parte della comunità crittografica è più probabile che possa essere considerato resistente agli attacchi.
- **Resistenza agli attacchi di crittanalisi:** La resistenza agli attacchi di crittanalisi, che cercano di sfruttare debolezze nell'algoritmo stesso o nelle sue implementazioni è un buon indicatore di resilienza. La crittanalisi può includere attacchi come l'analisi differenziale, l'analisi lineare e l'attacco Man-in-the-Middle.
- **Proprietà di diffusione e confusione:** Gli algoritmi di cifratura dovrebbero incorporare proprietà di diffusione e confusione. La diffusione distribuisce l'influenza di un singolo bit del messaggio su molti bit dell'output, mentre la confusione rende complesso il legame tra la chiave e l'output.
- **Resistenza agli attacchi di crittanalisi laterale:** La robustezza dell'algoritmo dovrebbe estendersi anche alla resistenza contro attacchi di crittanalisi laterale, che cercano di sfruttare informazioni ausiliarie come tempi di esecuzione per dedurre informazioni sulla chiave.
- **Adattabilità e resistenza ai progressi tecnologici:** Gli algoritmi di cifratura devono essere adattabili ai progressi tecnologici. Ciò significa che dovrebbero resistere a nuovi attacchi e alle capacità di calcolo sempre crescenti, garantendo al contempo la sicurezza a lungo termine.
- **Aggiornamenti e flessibilità:** Sono necessari per essere in grado di adattarsi a nuove minacce e vulnerabilità. In caso di scoperta di debolezze o attacchi, l'algoritmo dovrebbe essere aggiornato o sostituito con una nuova versione che risolva le debolezze.

La robustezza di un algoritmo di cifratura è cruciale per garantire la sicurezza delle comunicazioni e la protezione dei dati. Una progettazione attenta, un'analisi critica e una verifica costante sono elementi chiave per mantenere l'efficacia e la robustezza degli algoritmi crittografici nel tempo. È fondamentale rimanere al passo con gli sviluppi nella crittografia e adottare le migliori pratiche di sicurezza, come l'uso di

algoritmi robusti e l'implementazione corretta dei protocolli di sicurezza. Inoltre, la gestione sicura delle chiavi è cruciale per garantire l'efficacia della cifratura ed evitare ogni tipo di attacco. (Languasco e Zaccagnini 2020b)

3.1 Attacco tramite dizionario

Un attacco a dizionario è un metodo sistematico per identificare una password. Si basa sull'osservazione che le password normalmente utilizzate sono stringhe di caratteri, numeri e simboli che hanno generalmente un significato per l'utilizzatore. Infatti ricordare, scrivere o immettere una stringa di caratteri sconnessi risulta decisamente più difficile. Si privilegiano allora generalmente parole comuni, al più con dei caratteri sostituiti da numeri o simboli con qualche maiuscolo. Quindi l'insieme delle parole che potenzialmente rappresenterebbero delle password in quanto di senso compiuto possono essere intese alla stregua di un comune dizionario.

Gli aggressori quindi usano elenchi estesi di password deboli, facilmente indovinabili, come "**password**", "**123456**", o nomi comuni, modificando a volte qualche carattere, come "**p@ssw0rd**".

Nel corso degli anni sono stati fonte di studio vari metodi di contrasto a questo attacco, il più riuscito è sicuramente la cifratura tramite **hash** più il **salt**.

Il termine "*salt*" nel contesto delle password si riferisce ad una tecnica che consiste nell'aggiungere alla password una stringa di caratteri casuali di opportuna lunghezza, prima di eseguire la funzione di hash. L'hashing è un processo crittografico che converte la password in una stringa di lunghezza fissa, che è tipicamente una sequenza di caratteri alfanumerici, per cui se due utenti hanno la stessa password il processo di hashing genererebbe lo stesso digest. In questo modo si rende molto più difficile l'uso di tecniche di attacco in cui si pre-calcolano gli hash per molte password e cercando corrispondenza nel database.

Di seguito è mostrato un codice che simula un attacco tramite dizionario, suddiviso in 3 file python, rispettivamente per l'aggiunta di un nuovo utente soggetto dell'attacco, per la creazione del digest della password ed in ultimo per l'emulazione dell'attacco. Il programma **main** invece realizza l'interfaccia di utilizzo del programma.

3.1.1 Aggiunta di un utente

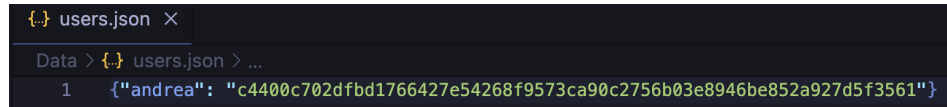
```
1 import json as js
2
3 def aggUtente(utente : str, passwd : str) -> None:
4     """
5     questa funzione serve per aggiungere un utente
6     all'interno del dataset
7     """
8     with open("Data/users.json", "r") as f:
9         dati = js.load(f)
10        dati[utente] = passwd
11    with open("Data/users.json", "w") as f:
12        js.dump(dati,f)
```

La sopracitata implementazione fornisce un metodo di aggiunta di utenti, per i quali si ricerca la password, su un file json.

Inizialmente l'algoritmo richiede l'importazione della libreria json. La funzione "aggUtente" prende in input due valori di tipo stringa, il nome dell'utente e la password cyptata con salt e SHA256, utilizzando la funzione "cripto" del file cryptografia.py, descritto più avanti.

I dati vengono inseriti nel file come una coppia "**utente : password**".

Il costo computazionale del seguente algoritmo è $O(n)$ dove n è il numero di elementi da leggere all'interno del file json prima dell'aggiunta dell'ultimo.



```
{
  "andrea": "c4400c702dfbd1766427e54268f9573ca90c2756b03e8946be852a927d5f3561"
}
```

Figura 3.1. Un esempio utente:password nel file utenti.json.

3.1.2 Cifratura della password con il salt

L'obiettivo del seguente codice è quella di crittografare la password presa in input dall'utente che si sta registrando nel file `aggUtente.py`. Vengono utilizzate due librerie python, la libreria di hashing "hashlib" e la libreria "time", questa serve per generare un Salt basato sull'orario e sulla data.

```
1 from hashlib import sha256
2 import datetime as dt
3
4 def cripto(passwd : str) -> str:
5     """
6     Questa funzione python serve per crittografare delle
7     password prese in input,
8     ritornando la stringa crittografata e con un salt
9     """
10    salt = genSalt()
11    salt = salt[::-1]
12    with open("Data/Salt.txt", "a") as f:
13        f.write(salt + "\n")
14    return sha256(passwd.encode() + salt.encode()).hexdigest()
15
16 def genSalt() -> str:
17     """
18     Questa funzione python serve per generare un salt da
19     usare per la crittografia
20     questo valore prende il giorno e l'orario in cui e' stata
21     creata la password
22     """
23    return str(dt.datetime.now()).strip().replace(" ",
24        "").replace(":", "").replace("-", "").replace(".", "")
```

Hashlib è una libreria standard di Python che fornisce un'interfaccia per calcolare le funzioni di hash crittografiche e non crittografiche. Nello specifico andremo ad utilizzare il sopracitato SHA256 (acronimo di Secure Hash Algorithm 256-bit), un algoritmo di hash crittografico che fa parte della famiglia di algoritmi di hash **SHA-2**.

Un algoritmo di hash come SHA-256 prende in input dati di lunghezza variabile e restituisce un valore hash di lunghezza fissa di 256 bit (32 byte). L'obiettivo principale di SHA-256 è quello di produrre un hash che sia unico per ogni diverso input e che sia estremamente difficile da invertire o da forzare per ottenere l'input originale. In altre parole, è progettato per essere unidirezionale e resistente alle collisioni (cioè, situazioni in cui due input diversi producono lo stesso hash).

Il codice è suddiviso in due funzioni, "**cripto**", che serve per la cifratura della password passata in input, mentre "**genSalt**" serve a generare un Salt pseudocasuale:

1. **Input:** La funzione prende in input una stringa, in questo caso contenente la password inserita dall'utente tramite la funzione **aggUtente** del file `aggUtente.py` visto in precedenza.

2. **Creazione del Salt:** Il passo successivo è quello di generare un Salt, questo viene generato dalla funzione apposita "**genSalt**" che ritorna una stringa pseudocasuale che si basa sull'orario e la data esatta della richiesta della sua generazione ma levando ogni tipo di spazio e punteggiatura (**es.** Se la funzione `datetime.now()` mi genera la seguent stringa "2023-09-19 17:30:15.430300", avremmo "20230919173015430300"). Per avere una stringa più sicura questa verrà invertita ("00303451037191903202").
3. **Memorizzazione del Salt:** Lo step successivo è quello di memorizzare, su un file apposito, tutti i Salt generati, questo avverrà tramite la funzione `write`, con modalità "a" (append).
4. **Return della password:** come ultimo passo avremmo il return della password cryptata.

Il costo di questo algoritmo è $O(1)$, ogni riga dell'algoritmo ha un costo unitario.

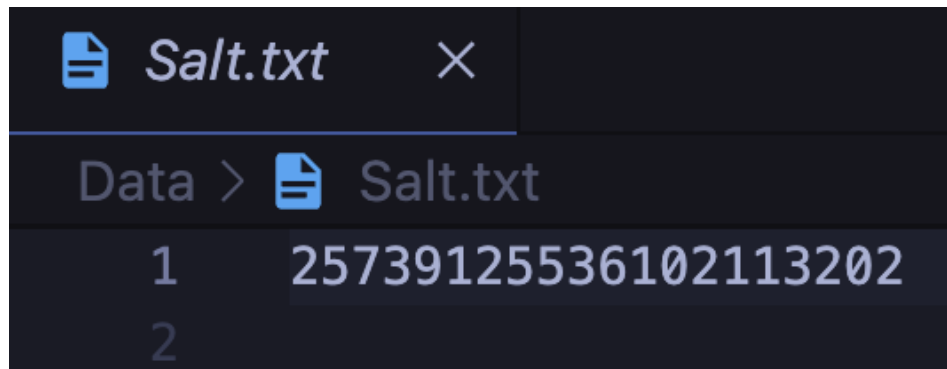


Figura 3.2. Un esempio di Salt nel file Salt.txt.

3.1.3 Implementazione dell'attacco

```

1 from hashlib import sha256
2 import json
3
4 def attaccoDiz() -> str:
5     """
6     Questa funzione apre il file json contenente la password
7     criptografata,
8     ed il percorso del file del salt
9     """
10    dicPass = {}
11    dicPassDec = {}
12    strFin = "Le Password decriptate sono:\n"
13    with open("Data/users.json", "r") as f:
14        dati = json.load(f)
15        for i in dati.keys():
16            dicPass[i] = dati[i]
17    """
18    apro e leggo il dataset contenente le varie password
19    """
20    tentativi = 0
21    with open("Data/DataSet.txt", "r") as f:
22        dataSet = f.readlines()
23        for line in dataSet:
24            line = line.strip()
25            with open("Data/Salt.txt", "r") as u:
26                salt = u.readlines()
27                for i in salt:
28                    tentativi += 1
29                    i = i.strip()
30                    for ident in dicPass.keys():
31                        if sha256(line.encode() +
32                                i.encode()).hexdigest() ==
33                            dicPass[ident]:
34                            dicPassDec[ident] = [line]
35                            dicPassDec[ident]
36                                .append(tentativi)
37
38    for nome, passwTemp in dicPassDec.items():
39        strFin += f"\t-{nome}: {passwTemp[0]} in
40        {passwTemp[1]} tentativi\n"
41    return strFin

```

Adesso passiamo all'attacco. Verranno usati, come riportato nel codice sopracitato, due librerie viste già in precedenza, **SHA256** della libreria **hashlib** e la libreria **json**.

1. **Creazione dizionari:** Il codice parte con la creazione di due dizionari vuoti; in Python ogni elemento in un dizionario è costituito da una coppia chiave-valore, dove la chiave è unica e viene utilizzata per accedere al valore corrispondente. Andremmo a salvare all'interno gli utenti come chiave e le password dei seguenti. Il primo dizionario ha come obiettivo quello di salvare le password

criptografate, mentre il secondo è il dizionario di arrivo **utente - password decifrata**.

2. **Aprire il file JSON:** Successivamente dovremmo copiare sul primo dizionario tutte le coppie utente-password del file json.
3. **Aprire il DataSet:** Passiamo alla scansione del DataSet, nel nostro caso useremo **RockYou** (Vedi Cap. 3.1).
4. **Tentare il Salt su ogni valore:** Per ogni riga del nostro DataSet, dopo averla letta, verrà aperto il file contenente tutti i Salt per creare una stringa **[password+salt]**.
5. **Confronto con ogni password:** Per ogni stringa del tipo **[password+salt]** eseguiremo un Hash tramite la funzione SHA256 per creare una stringa criptata. In questo caso avremo due possibilità:
 - Se La stringa cifrata è uguale ad una delle password del file JSON, allora abbiamo trovato la password e in tal caso aggiungeremo al secondo dizionario la coppia utente-password (questa sarebbe la riga dove ci troviamo del DataSet, quindi aggiungeremo quest'ultima come valore).
 - Se la stringa alterata non combacia, proviamo con un nuovo Salt, fino a quando non saranno finiti, se arrivati all'ultimo Salt non abbiamo ancora trovato la password, allora significa che nessun utente ha utilizzato quel valore del DataSet e quindi andiamo alla prossima riga per ripetere ogni passaggio dal **punto 3**.

Dopo aver finito tutti questi passaggi, ed eseguito l'algoritmo con costo $O(n \cdot k \cdot j)$ dove n sono il numero di righe del file "RockYou", k il numero di righe del file "Salt" e j il numero di identità nel primo dizionario, il nostro programma ritornerà una stringa contenente il nome degli utenti, le password e i tentativi effettuati per trovare quella determinata password.

```
Salve, cosa si vuole fare?
  1) Aggiungere un nuovo utente
  2) Emulare un attacco tramite dizionario
2
Procedo con la decriptazione delle password...
Le Password decriptate sono:
  -andrea: shefferd3 in 11450445 tentativi

Tempo impiegato: 108.34911799430847 secondi, 1.8058186332384745 minuti
```

Figura 3.3. Risultati dell'attacco tramite Dizionario.

3.2 Brute Force: ciphertext only

In crittografia, un **Ciphertext Only Attack** (COA), noto anche come attacco con testo cifrato noto, è un modello di attacco utilizzato nella crittoanalisi in cui si presume che l'attaccante disponga solo di un insieme di testi cifrati.

Il malintensionato potrebbe essere soltanto a conoscenza della lingua in cui il testo in chiaro è redatto o potrebbe aver familiarità con la distribuzione statistica prevista dei caratteri nel testo in chiaro. Ne consegue che, anche in assenza di informazioni dirette sul testo sfruttando queste informazioni per compiere progressi nell'analisi crittografica.

In molti sistemi distribuiti, i dati e i messaggi del protocollo standard costituiscono tipicamente una parte del testo in chiaro. Questi elementi sono spesso suscettibili ad essere estratti o conosciuti ed utilizzati in modo efficace come parte di un attacco basato solo sul ciphertext. Questa vulnerabilità pone in evidenza l'importanza di considerare non solo la robustezza della crittografia in sé, ma anche la protezione dei dati e dei messaggi durante la loro trasmissione e archiviazione.

La procedura di attacco realizzata si compone di 2 moduli distinti:

- **Cifratura:** prende in ingresso un file di testo in chiaro da cifrare con l'algoritmo DES. Viene richiesta una chiave di cifratura che per ragioni di complessità computazionale legate al successivo processo di attacco viene ristretta alla scelta dei seguenti caratteri: "abcdefABCDEF01234" con un totale di 17^6 valori, invece dei 2^{64} che renderebbe computazionalmente estremamente lungo.
- **Attacco:** In questa fase il testo viene decodificato per ogni possibile password creata dall'alfabeto dei caratteri della password, il processo termina quando la successiva codifica con la stessa password produce il testo codificato iniziale.

3.2.1 Cifratura tramite DES

```

1 import subprocess
2
3 def eseguiCripto(path : str) -> str:
4     """
5     La seguente funzione serve a crittografare, con un valore
6     preso in input
7     da utente, un file di un determinato percorso.
8     """
9     crypto = ""
10    while len(crypto) != 6:
11        print("La password deve essere di 6 caratteri")
12        crypto = input("Inserire il valore di password che si vuole dare, "
13                        "utilizzare solo questi simboli: abcdefABCDEF01234:")
14
15    with open("../DES/keyfile.key", "w") as f:
16        f.write(crypto)
17    process = subprocess.Popen(f"../DES/run_des.o -e ../DES/keyfile.key {path} Data/sample.enc", shell=
18                                True, stdout=subprocess.PIPE, stderr=subprocess.PIPE)
19    output, error = process.communicate()
20
21    if process.returncode == 0:
22        return f"File criptato con successo: {output.decode()}"
23    else:
24        return f"Errore {error.decode()}"

```

```

Benvenuto, cosa si vuole fare?
1) Criptare un file
2) Decriptare un file
3) Esci
Inserisci il numero corrispondente: 1
Ora ci troviamo nel seguente percorso: /Users/andrea/Developer/Tirocinio/BruteForce/Ciphertext
Inserire il percorso del file che si vuole criptare: Data/Test.txt
La password deve essere di 6 caratteri
Inserire il valore di password che si vuole dare, utilizzare solo questi simboli: abcdefABCDEF01234: aab
bcc

```

Figura 3.4. Un esempio di cifratura di un file.

Prima di iniziare a parlare della parte della decifratura, dobbiamo parlare della fase di cifratura, l'algoritmo sopracitato ha questa struttura:

1. **Importare Subprocess:** Il modulo **subprocess** in Python offre un modo per avviare nuovi processi, collegarsi ai loro canali di input/output/errore e ottenere i loro codici di ritorno. Fondamentalmente, consente di avviare nuovi processi, interagire con essi e recuperare i risultati. in questo caso lo utilizziamo per richiamare il DES tramite linea di comando.
2. **Scelta della chiave:** Verrà chiesto all'utente di scegliere una chiave composta da 6 valori usando un alfabeto composto da "abcdefABCDEF01234", se la lunghezza del valore non corrisponde a 6 questa non verrà registrata.

3. **Scrittura della chiave:** La chiave verrà scritta nel file "keyfile.key".
4. **Cifratura:** Utilizzando la chiave andremmo a cifrare il file del quale il programma ci chiederà il percorso come input.
Se tutto verrà eseguito correttamente il file verrà salvato nella cartella Data, altrimenti verrà stampato un messaggio di errore.

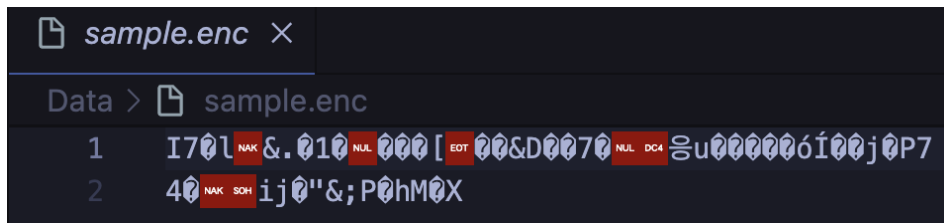


Figura 3.5. Un esempio di file criptato con il DES, dove il messsaggio era "Questo è un messaggio di prova. This is a test message.".

3.2.2 Implementazione attacco

```

1 import subprocess
2 import time as tm
3 import os
4 import platform
5
6 def eseguiDecripto(path : str) -> str:
7     """
8     La seguente funzione serve a decrittografare con un
9     valore preso in input
10    da utente un file di un determinato percorso.
11    """
12    tm1 = tm.time()
13    with open("../key.txt","r") as f:
14        x = 0
15        chiavi = f.readlines()
16
17        for chi in chiavi:
18            chi = chi.strip()
19            with open("Data/keyAppoggio.key", "w") as f:
20                f.write(chi)
21            x += 1
22
23            v = subprocess.Popen([f"../../DES/run_des.o -d
24            Data/keyAppoggio.key Data/sample.enc
25            Data/dec.txt"], shell = True,
26            stdout=subprocess.PIPE,
27            stderr=subprocess.PIPE).wait()
28
29            if v == 0:
30                subprocess.Popen([f"../../DES/run_des.o -e
31                Data/keyAppoggio.key Data/dec.txt
32                Data/test.enc"], shell = True,
33                stdout=subprocess.PIPE,
34                stderr=subprocess.PIPE)
35
36            if x == 1:
37                tm.sleep(0.2)
38            if confronta_file(path, "Data/test.enc"):
39                file = "Data/test.enc"
40                if os.path.exists(file):
41                    os.remove(file)
42                if os.path.exists("Data/keyAppoggio.key"):
43                    os.remove("Data/keyAppoggio.key")
44
45            try:
46                with open("Data/dec.txt", "r") as f:
47                    file = f.readlines()
48                    tm2 = tm.time()
49                    print(f"File decriptato con
50                    successo, chiave: {chi}, dopo
51                    {x} tentativi, con tempo di
52                    esecuzione: {tm2-tm1}")

```

```

secondi.\nIl_file_si_trova_in_
Data/dec.txt")
40         return
41     except:
42         continue
43     print("File non decriptato, nessuna chiave
corrispondente")
44     return
45
46
47 def confronta_file(secret : str, samples : str) -> bool:
48     """
49     La seguente funzione serve a confrontare due file, se
50     sono uguali ritorna True
51     altrimenti False. Il confronto avviene tramite lo sha512
52     dei file.
53     """
54     piattaforma = platform.system()
55     match piattaforma:
56         case "Windows":
57             t = subprocess.Popen(f"sha512sum_{samples}",
58                                 shell = True, stdout=subprocess.PIPE,
59                                 stderr=subprocess.PIPE).communicate()[0]
60             v = subprocess.Popen(f"sha512sum_{secret}", shell
61                                 = True, stdout=subprocess.PIPE,
62                                 stderr=subprocess.PIPE).communicate()[0]
63         case "Linux":
64             t = subprocess.Popen(f"sha512sum_{samples}",
65                                 shell = True, stdout=subprocess.PIPE,
66                                 stderr=subprocess.PIPE).communicate()[0]
67             v = subprocess.Popen(f"sha512sum_{secret}", shell
68                                 = True, stdout=subprocess.PIPE,
69                                 stderr=subprocess.PIPE).communicate()[0]
70         case _:
71             t = subprocess.Popen(f"shasum -a 512_{samples}",
72                                 shell = True, stdout=subprocess.PIPE,
73                                 stderr=subprocess.PIPE).communicate()[0]
74             v = subprocess.Popen(f"shasum -a 512_{secret}",
75                                 shell = True, stdout=subprocess.PIPE,
76                                 stderr=subprocess.PIPE).communicate()[0]
77
78     t = t.decode("utf-8").split("_")[0]
79     v = v.decode("utf-8").split("_")[0]
80     if t == v: return True
81     else: return False

```

Il funzionamento di questo attacco è simile al funzionamento dell'attacco tramite dizionario: avremmo un dataset contenente tutte le possibili combinazioni di chiavi composte dall'alfabeto: "abcdefABCDEF01234" di lunghezza 6; Avremmo un totale di 24.000.000+ di chiavi da testare. **Funzionamento:**

1. **Input:** La funzione prende il percorso del ciphertext a noi interessato, per poi aprirlo.

2. **Inizio scansione:** Apriamo il file contenente tutte le chiavi e scorriamo ogni stringa.
3. **Prova della chiave:** Per ogni chiave che viene scansionata verrà creato un file "keyfile.key" di appoggio, che ci servirà per testare l'apertura del file. Per verificare che la password sia corretta, se non verrà restituito nessun errore, il programma proverà a re-criptare il file ottenuto.
4. **Confronto:** Verrà effettuato un confronto tramite lo SHA512, i due file vengono trasformati due stringhe cifrate, se le due stringhe sono uguali, allora il file è stato decriptato correttamente, altrimenti no, e verrà testata un'altra chiave. Se il confronto dovesse ritornare "True", tutti i file creati in appoggio (KeyAppoggio.key, sample.enc, ecc.), verranno eliminati e verrà stampata una stringa che afferma il corretto funzionamento dell'algoritmo.

```
Benvenuto, cosa si vuole fare?
  1) Criptare un file
  2) Decriptare un file
  3) Esci
Inserisci il numero corrispondente: 2
Ora ci troviamo nel seguente percorso: /Users/andrea/Developer/Tirocinio/BruteForce/Ciphertext
Inserire il percorso del file che si vuole decriptare: Data/sample.enc
File decriptato con successo, chiave: aaaabbbb, dopo 5221 tentativi, con tempo di esecuzione: 183.54256
010055542 secondi.
Il file si trova in Data/dec.txt
```

Figura 3.6. Risultati dell'attacco tramite Only Ciphertext.

3.3 Brute Force: attacco plaintext

Nel campo della crittografia, il concetto di "plaintext" denota i dati privi di cifratura o codifica. L'espressione "**attacco plaintext**" si riferisce a un approccio in cui un potenziale aggressore dispone contemporaneamente del file originale in chiaro e della sua controparte cifrata, con l'intento di individuare la chiave utilizzata per la trasformazione del file.

Questo tipo di attacco si basa sull'idea di esaminare il testo in chiaro e il corrispondente testo cifrato, cercando di stabilire una relazione che riveli la chiave di cifratura impiegata. L'obiettivo finale è determinare il meccanismo con cui il file è stato alterato, allo scopo di compromettere la sicurezza del sistema crittografico in uso.

L'attaccante, avendo a disposizione entrambe le versioni del file, sfrutta questa conoscenza per sottoporre il sistema a un processo iterativo, provando diverse chiavi fino a trovare quella corretta. L'efficacia di tale metodologia dipende essenzialmente dalla robustezza della chiave utilizzata.

In sintesi, un attacco plaintext rappresenta un tentativo di violazione della sicurezza crittografica sfruttando la disponibilità simultanea del testo in chiaro e del suo corrispondente cifrato, con l'obiettivo di trovare la chiave impiegata.

Si omette la procedura di cifratura del testo in chiaro e per convenienza si utilizza l'algoritmo DES nell'emulazione realizzata. L'attacco realizzato si compone di 2 moduli distinti:

- **Cifratura:** prende in ingresso un file di testo in chiaro da cifrare con l'algoritmo DES. Viene richiesta una chiave di cifratura che per ragioni di complessità computazionale legate al successivo processo di attacco viene ristretta alla scelta dei seguenti caratteri: "abcdefABCDEF01234" con un totale di 17^6 valori, invece dei 2^{64} che renderebbe computazionalmente estremamente lungo.
- **Attacco:** In questa fase il testo viene decodificato per ogni possibile password creata dall'alfabeto dei caratteri della password, il processo termina quando la successiva codifica con la stessa password produce il testo codificato iniziale, a differenza del COA, faremo un confronto con il plaintext.

3.3.1 Implementazione attacco

```

1 import subprocess
2 import time as tm
3 import os
4 import platform
5
6 def plainText(path : str, crypto : str) -> str:
7     """
8     Questa funzione prende in input il path del file da
9     decifrare e il path del file cifrato.
10    Serve per trovare la chiave con cui viene cifrato il File.
11    """
12    with open("../key.txt", "r") as f:
13        t1 = tm.time()
14        x = 0
15        chiavi = f.readlines()
16        for key in chiavi:
17            x+=1
18            key = key.strip()
19            with open("Data/keyAppoggio.key", "w") as f:
20                f.write(key)
21
22            subprocess.Popen(f"../DES/run_des.o -e
23                Data/keyAppoggio.key {path}
24                Data/test.enc", shell= True,
25                stdout=subprocess.PIPE, stderr=subprocess.PIPE)
26            if x == 1:
27                tm.sleep(0.2)
28            if confronta_file("Data/test.enc", crypto):
29                subprocess.Popen([f"../DES/run_des.o -d
30                    Data/keyAppoggio.key Data/sample.enc
31                    Data/dec.txt"], shell = True,
32                    stdout=subprocess.PIPE,
33                    stderr=subprocess.PIPE).wait()
34                t2 = tm.time()
35                if os.path.exists("Data/keyAppoggio.key"):
36                    os.remove("Data/keyAppoggio.key")
37                if os.path.exists("Data/test.enc"):
38                    os.remove("Data/test.enc")
39                return f"La password '{key}', trovata dopo
40                    {x} tentativi, in {t2-t1} secondi, trovi
41                    il file decriptato in Data/dec.txt"
42
43    return "Chiave non trovata"
44
45
46 def confronta_file(secret : str, samples : str) -> bool:
47     """
48     La seguente funzione serve a confrontare due file, se
49     sono uguali ritorna True
50     """

```



```

40     altrimenti False. Il confronto avviene tramite lo sha512
      dei file.
41     """
42     piattaforma = platform.system()
43     t = ""
44     match piattaforma:
45         case "Windows":
46             t = subprocess.Popen(f"sha512sum_{samples}",
                                   shell = True, stdout=subprocess.PIPE,
                                   stderr=subprocess.PIPE).communicate()[0]
47             v = subprocess.Popen(f"sha512sum_{secret}", shell
                                   = True, stdout=subprocess.PIPE,
                                   stderr=subprocess.PIPE).communicate()[0]
48         case "Linux":
49             t = subprocess.Popen(f"sha512sum_{samples}",
                                   shell = True, stdout=subprocess.PIPE,
                                   stderr=subprocess.PIPE).communicate()[0]
50             v = subprocess.Popen(f"sha512sum_{secret}", shell
                                   = True, stdout=subprocess.PIPE,
                                   stderr=subprocess.PIPE).communicate()[0]
51         case _:
52             t = subprocess.Popen(f"shasum -a_{512}_{samples}",
                                   shell = True, stdout=subprocess.PIPE,
                                   stderr=subprocess.PIPE).communicate()[0]
53             v = subprocess.Popen(f"shasum -a_{512}_{secret}",
                                   shell = True, stdout=subprocess.PIPE,
                                   stderr=subprocess.PIPE).communicate()[0]
54
55     t = t.decode("utf-8").split("_")[0]
56     v = v.decode("utf-8").split("_")[0]
57     if t == v: return True
58     else: return False

```

Il codice implementato è simile a quello dell'attacco tramite dizionario; **Funzionamento:**

1. **Input:** La funzione prende in input due percorsi, per l'esattezza il percorso del plaintext e del ciphertext a disposizione.
2. **Inizio scansione:** Iniziamo a scansionare, come per l'attacco precedente, il file "key.txt", contenente tutte le 24M+ di password contenute nel file.
3. **Cifratura:** Adesso, per ogni chiave, a differenza del "ciphertext only", andremo a cifrare il file in chiaro tramite l'algoritmo di DES.
4. **Confronto:** Manderemo i due file criptati nella funzione che effettua il confronto, trasformando i due file in due stringhe SHA512, per velocizzare il procedimento, poiché il confronto fra stringhe è sensibilmente più veloce. Se il confronto va a buon fine la funzione restituirà "True", "False" altrimenti. Se il confronto dovesse avere un riscontro positivo, tutti i file creati in appoggio (KeyAppoggio.key, sample.enc, ecc.), verranno eliminati e verrà stampata una stringa che afferma il corretto funzionamento dell'algoritmo.

```

Cosa si vuole fare:
  1)Cifrare un file
  2)Decifrare un file
  3)Esci dal programma
Inserire il numero corrispondente alla scelta: 2
Inserire il path del file decifrato: Data/plaintext.txt
Inserire il path del file del file cifrato: Data/sample.enc
La password è aaaabddd, trovata dopo 5257 tentativi, in 175.68620800971985 secondi, trovi il file decri
ptato in Data/dec.txt

```

Figura 3.7. Risultati dell'attacco tramite Plaintext.

3.4 Chosen ciphertext

Un attacco "**chosen ciphertext**" (CCA), o "**attacco a testo cifrato scelto**", è un tipo di attacco crittografico in cui un attaccante ha la capacità di ottenere la decifrazione di specifici testi cifrati di sua scelta. Questo tipo di attacco si concentra sulla manipolazione dei testi cifrati in modo da ottenere informazioni utili o indebolire la sicurezza del sistema crittografico.

Nell'ambito di un attacco chosen ciphertext, un avversario può avere il controllo su parte del testo cifrato inviato al sistema di decifrazione e osservare la risposta risultante. L'obiettivo principale è capire come il sistema di cifratura risponde a determinate modifiche al testo cifrato e utilizzare queste informazioni per dedurre la chiave segreta o ottenere accesso non autorizzato ai dati.

Per proteggersi dagli attacchi chosen ciphertext, è importante implementare algoritmi crittografici robusti e assicurarsi che il sistema crittografico sia resistente a varie forme di manipolazione dei dati cifrati. Inoltre, è fondamentale adottare buone pratiche di sicurezza, come l'uso di algoritmi di firma digitale e la verifica accurata dei dati in ingresso, per prevenire eventuali manipolazioni dannose dei testi cifrati.

3.4.1 CCA è oneroso sull'algoritmo del DES

Nel caso del Data Encryption Standard (DES), alcune caratteristiche della sua progettazione contribuiscono a rendere impossibile un attacco di tipo CCA.

Il DES utilizza una struttura iterativa che coinvolge 16 fasi di sostituzione e permutazione, chiamate "rounds". Una delle caratteristiche chiave che contribuisce alla resistenza agli attacchi CCA è l'uso di S-boxes (scatole di sostituzione) durante le fasi di sostituzione. Queste S-boxes introducono una non linearità che rende impossibile stabilire una relazione tra il testo cifrato e il testo in chiaro. Modifiche minime nei dati cifrati portano a cambiamenti significativi nei dati decifrati a causa dell'effetto amplificante delle S-boxes.

Inoltre, la chiave segreta di 56 bit utilizzata da DES contribuisce all'impossibilità di un attacco CCA. La chiave è espansa durante il processo di cifratura, ma la sua lunghezza relativamente corta la rende vulnerabile agli attacchi di forza bruta. Tuttavia, senza la conoscenza della chiave, un attaccante deve affrontare la sfida di dedurre la relazione tra il testo cifrato e la chiave segreta. (Biryukov 2011)

3.4.2 Cifratura tramite RSA

La cifratura RSA è un algoritmo di crittografia a chiave pubblica che utilizza due chiavi: una chiave pubblica per cifrare i dati e una chiave privata per decifrarli.

```

1 from rsa import *
2 from chosen import *
3
4 if __name__ == '__main__':
5     c = input("Cosa si vuole fare?\n1) Usare l'RSA\n2) Simulare un chosen ciphertext?\nInserire uno dei valori: ")
6     if c == '1':
7         print("\nEncrypt e Decrypt con RSA")
8         print("Il prodotto di due numeri primi deve essere maggiore di 256")
9         p = int(input("Inserisci il primo numero primo (17, 19, 23, etc): "))
10        q = int(input("Inserisci il secondo numero primo (Non come prima): "))
11        public, private = generate_keypair(p, q)
12        print(f"La tua chiave pubblica è {public} e la tua chiave privata è {private}")
13
14        choice = input("Cosa si vuole fare?\n[E]ncrypt\n[D]ecrypt\n[S]top:\nInserire uno dei valori: ").upper()
15        while not choice.startswith('S'):
16            if choice.startswith('E'):
17                message = input("Inserisci il percorso del file che si vuole Cryptare: ")
18                encrypted_msg = encrypt(public, message)
19                print(' '.join(map(lambda x: str(x), encrypted_msg)))
20            elif choice.startswith('D'):
21                encrypted_msg = input("Inserisci il percorso del file che si vuole Decryptare: ")
22                encrypted_msg = list(map(int, encrypted_msg.split("_")))
23                print(f"Sblocco il file con la chiave: {private}")
24                plain = decrypt(private, encrypted_msg)
25                print("Questo è il tuo messaggio:")
26                print(' '.join(list(map(str, plain))))
27                print("Questo è il tuo messaggio in ASCII: ")
28                print(''.join(list(map(chr, plain))))
29                choice = input("Cosa vuoi fare ora?\n[E]ncrypt\n[D]ecrypt\n[S]top:\nInserire uno dei valori: ").upper()
30            if choice.startswith('S'):
31                print("Grazie per aver usato il programma.")
32                exit(0)
33        elif c == '2':
34            chosen = input("Inserisci il percorso del file che si

```

```

35         vuole_decriptare: ")
36     x = attack(chosen)
37     dec(x, chosen)
38     plain = input("Inserisci il percorso del file
39         decriptato (Quello a disposizione, se non ci sta
40         premere INVIO): ")
41     if plain == "":
42         print("Il file decriptato e' nella cartella Data,
43         come decrypted.txt")
44     elif confronto(plain, "Data/decrypted.txt"):
45         print("Il file e' stato decriptato correttamente.
46         Trovi il file nella cartella Data, come
47         decrypted.txt")
48     else:
49         print("Il file non e' stato decriptato
50         correttamente. Riprovare.")

```

```

Cosa si vuole fare?
1) Usare l'RSA
2) Simulare un chosen ciphertext?
Inserire uno dei valori: 1

Encrypt e Decrypt con RSA
Il prodotto di due numeri primi deve essere maggiore di 256
Inserisci il primo numero primo (17, 19, 23, etc): 23
Inserisci il secondo numero primo (Non come prima): 29
Vuoi avere una chiave pubblica specifica? [Y]/N N
La tua chiave pubblica è (551, 667) e la tua chiave privata è (199, 667)
Cosa si vuole fare?
[E]ncrypt
[D]ecrypt
[S]top:
Inserire uno dei valori: E
Inserisci il percorso del file che si vuole Criptare: Data/Test.txt
Fatto, trovi il file cipher.enc nella cartella Data.

```

Figura 3.8. Esempio di cifratura tramite RSA.

Funzionamento:

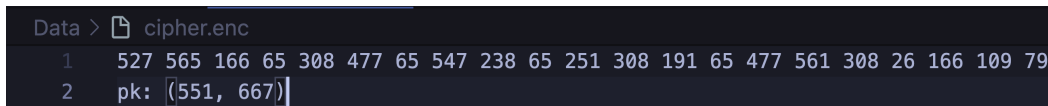
1. Generazione delle chiavi:

- Si scelgono due numeri primi molto grandi, spesso indicati come **p** e **q**.
- Calcolare **n**, che sarà la moltiplicazione fra i due numeri primi.
- Si calcola la funzione di Eulero $\phi(n) = (p - 1) \cdot (q - 1)$, per trovare il numero di tutti i numeri coprimi con **n**.
- Sceglie un numero intero **e** (esponente di cifratura) in maniera casuale o scelto dall'utente, che sia però coprimo con $\phi(n)$.
- Si calcola il valore di **d** (esponente di decifrazione) tale che $d \cdot e \equiv 1 \pmod{\phi(n)}$
- La chiave pubblica è costituita dalla coppia (n, e) e la chiave privata dalla coppia (n, d) .

2. Cifratura:

- Il mittente ottiene la chiave pubblica del destinatario, che è costituita da **n** ed **e**.

- Il mittente converte il testo in chiaro in un numero m che rappresenta il messaggio. Tipicamente, viene utilizzato uno schema di padding per garantire che il messaggio abbia una lunghezza appropriata.
- Il mittente calcola il valore cifrato di ogni carattere del file con questa formula: $c \equiv m^e \pmod{n}$.
- Il mittente salverà tutto nel file .enc.



```
Data > cipher.enc
1 527 565 166 65 308 477 65 547 238 65 251 308 191 65 477 561 308 26 166 109 79
2 pk: [551, 667]
```

Figura 3.9. Esempio di file cifrato con RSA.

3.4.3 Implementazione attacco

L'attacco tramite "**Chosen Ciphertext**" rappresenta una minaccia significativa nel contesto della crittografia, in particolare quando si tratta di metodi di cifratura. Questo tipo di attacco si basa sull'abilità dell'attaccante di influenzare attivamente il processo di decifrazione, fornendo input specificamente progettati per rivelare informazioni sensibili. Nel corso di questa sezione, esploreremo il funzionamento di tale attacco, concentrandoci sull'inversione dell'algoritmo di cifratura come strategia chiave. Attraverso questa analisi, acquisiremo una comprensione più approfondita delle vulnerabilità intrinseche e delle contromisure necessarie per preservare l'integrità dei sistemi crittografici.

```
1 import math
2
3 def attack(ciphertext : str) -> int:
4     """
5     Questa funzione simula l'attacco di chosen ciphertext, in
6     cui l'attaccante ha
7     a disposizione uil file criptato e il file decriptato.
8     Useremo un RSA semplificato.
9     """
10    p = 0
11    q = 0
12    d = 0
13    with open(ciphertext, "r") as f:
14        for ele in f:
15            if ele.startswith("pk"):
16                pk = eval(ele[4:])
17                d = pk[0]
18                break
19            c = ele.replace("\n", "")
20
21    for i in range(2, pk[1]-1):
22        for j in range(2, pk[1]-1):
23            if i*j == pk[1]:
24                p = i
25                q = j
26                break
27
28    phi = (p-1) * (q-1)
29    return multiplicative_inverse(d, phi), pk[1]
30
31 def multiplicative_inverse(e : int, phi : int) -> int:
32     """
33     Algoritmo euclideo per trovare l'inverso moltiplicativo
34     fra due numeri
35     """
36    r0 = phi
37    r1 = e
38    t0 = 0
```

```

37     t1 = 1
38     while r1 > 0:
39         q = math.floor(r0/r1)
40         temp = r0 - q * r1
41         r0 = r1
42         r1 = temp
43
44         temp = t0 - q * t1
45         t0 = t1
46         t1 = temp
47     if r1 == 1:
48         return t1 % phi
49
50 def dec(pk : tuple, ciphertext : str) -> None:
51     key, n = pk
52     with open(ciphertext, "r") as f:
53         c = f.readlines()[0].replace("\n", "")
54         plain = [chr(pow(int(char), key, n)) for char in
55                  c.split("_")]
56     with open("Data/decrypted.txt", "w") as f:
57         f.write(''.join(plain))
58
59 def confronto(cipher : str, plain : str) -> bool:
60     with open(cipher, "r") as f:
61         c = f.readlines()[0]
62     with open(plain, "r") as f:
63         p = f.readlines()[0]
64     return c == p

```

```

Cosa si vuole fare?
  1) Usare l'RSA
  2) Simulare un chosen ciphertext?
Inserire uno dei valori: 2
Inserisci il percorso del file che si vuole decryptare: Data/cipher.enc
Inserisci il percorso del file decryptato (Quello a disposizione, se non ci sta premere INVIO):
Il file decryptato è nella cartella Data, come decrypted.txt
La chiave è: (199, 667)

```

Figura 3.10. Risultati dell'attacco tramite Chosen Ciphertext con RSA.

Spiegazione:

1. **Apertura file:** Iniziamo con l'apertura del file criptato, contenente la chiave pubblica, come coppia di valori, per poi salvare nella variabile *d* il primo valore della chiave.
2. **Scovare i numeri primi:** Poiché sappiamo che il secondo valore della chiave pubblica è la moltiplicazione di due valori primi, andremo ad iterare, con due cicli for con un range che va da 2 al valore *n*-1, per scovare tramite moltiplicazione quali sono i due valori che restituiscono la chiave.
3. **Moltiplicativo inverso:** Questa funzione ci serve per trovare il valore inverso del numero *d* mod *phi*, ovvero per trovare il valore della chiave privata.

4. **Decifratura:** Ora che abbiamo ottenuto il valore della chiave privata, possiamo passare alla decifratura del file, ogni valore verrà sbloccato in maniera numerica, per poi associare a questo numero un valore alfabetico.
5. **Confronto finale:** Per verificare che il file sia stato sbloccato in maniera corretta, verrà effettuato un confronto fra file, fra testo in chiaro e testo decifrato, questo passaggio è opzionale è verrà chiesto se si vorrà effettuare.

Capitolo 4

Conclusioni

Questa tesi ha preso in esame alcune metodologie di attacco ai sistemi di cifratura, attraverso l'utilizzo di diverse tecniche. Le sperimentazioni hanno dimostrato come la vulnerabilità di diversi sistemi di cifratura presi in esame sono in funzione diretta della complessità della chiave. È evidente comunque la necessità di una costante attenzione e miglioramento delle misure di sicurezza informatica, così come l'importanza di adottare politiche di sicurezza robuste, implementando cifrature avanzate e promuovere una consapevolezza continua sulla sicurezza informatica.

Risultati

- **Attacco tramite Dizionario:** È stato utilizzato un dizionario di 8.800.000 circa di password con lunghezza variabili, il tempo di elaborazione si basa sulla posizione della password dentro al Dataset, il tempo medio su 10 tentativi è di 2.42 minuti.
- **Attacco Only Ciphertext e plaintext:** È stato utilizzato un DataSet di chiavi lunghe 8 caratteri ognuna, per un totale di 24.500.000 circa di valori, per un tempo di elaborazione di 3.11 minuti, dopo 5221 tentativi.
- **Chosen Ciphertext:** La procedura realizzata ha un mero significato didattico in quanto la fattorizzazione del modulo n ha operato su un numero con limitate cifre.

Una delle difficoltà è stata la ricerca di documentazione approfondita e affidabile su questo argomento. Trovare risorse informative di alta qualità si è dimostrato un compito impegnativo, ma la ricerca di fonti autorevoli è stata perseverante per ampliare la comprensione.

Bibliografia

- Biryukov, Alex (2011). *DES-X (or DESX)*.
- Kerckhoffs, Auguste (1883). *La cryptographie militaire, ou, Des chiffres usités en temps de guerre: avec un nouveau procédé de déchiffrement applicable aux systèmes à double clef*. Librairie militaire de L. Baudoin.
- Languasco, Alessandro e Alessandro Zaccagnini (2020a). *Manuale di crittografia: teoria, algoritmi e protocolli*. Hoepli Editore.
- (2020b). *Manuale di crittografia: teoria, algoritmi e protocolli*. Hoepli Editore.
- Rognetta, Giorgio (1999). “Crittografia asimmetrica: dal cifrario di Giulio Cesare alla firma digitale”. In: *Informatica e diritto* 8.1, pp. 59–75.
- Shannon, Claude (1949). Bell System Technical Journal, Vol. 28, 4 Ottobre 1949.
- Zimuel, Enrico (2002). “Introduzione alla crittografia Introduzione alla crittografia ed alla crittoanalisi”. In: *Sikurezza.org*.