Crittografia

Alessio Gjergji

Indice

Capitolo 1

Tecniche crittografiche classiche

1.1 Modelli di crittografia simmetica

La crittografia simmetrica è formata da cinque elementi:

- **Plaintext**: si tratta del testo in chiaro, quindi interpretabile.
- **Algoritmo di cifratura**: l'algoritmo di cifratura esegue varie sostituzioni e trasformazioni del testo in chiaro.
- **Chiave segreta**: la chiave è anch'essa argomento dell'algoritmo di cifratura. La chiave è indipendente dal testo in chiaro e dall'algoritmo. L'algoritmo produrrà un risultato diverso a seconda della specifica chiave utilizzata.
- **Testo cifrato**: Si tratta del messaggio prodotto in output. Dipenderà dal testo in chiaro e dalla chiave segreta. Date due chiavi diverse il risultato in output sarà diverso, genererà quindi due testi cifrati differenti. Il testo cifrato è apparentemente un flusso di dati casuale, sarà quindi illegibile.
- **Algoritmo di decifratura** Si tratta essenzialmente dell'algoritmo di cifratura eseguito inversamente. Prende in input il testo cifrato e la chiave, producendo il testo originale.

Ci sono due requisiti per per l'uso sicuro della crittografia convenzionale:

1. Abbiamo bisogno di un'algoritmo di cifratura forte. Ciò significa che che chi possiede testi cifrati non sia in grado di trovare facilmente la chiave per poterli decifrare.
2. Il mittente e il destinatario devono aver ricevuto le copie delle chiavi segrete mediante un canale sicuro. Se qualcuno trovasse la chiave conoscendo l'algoritmo, l'intera comunicazione diventerebbe leggibile.

Nella cifratura simmetrica, l'algoritmo di cifratura non deve rimanere segreto, ma solo la chiave. Questa caratteristica la rende pratica per un uso diffuso. I chip di cifratura a basso costo sono ampiamente disponibili e incorporati in vari prodotti. La principale preoccupazione di sicurezza è mantenere segreta la chiave.

1.1.1 Elementi essenziali di una cifratura simmetrica

- Una sorgente produce un messaggio in testo in chiaro $X = [X_1, X_2, \ldots, X_M]$.
- Viene generata una chiave di cifratura $K = [K_1, K_2, \ldots, K_J]$, che deve essere mantenuta segreta.
- Con il messaggio X e la chiave K come input, l'algoritmo di cifratura genera il testo cifrato $Y = [Y_1, Y_2, \ldots, Y_N]$, rappresentato come $Y = E(K, X)$.
- Il destinatario inteso, in possesso della chiave, può decifrare il messaggio: $X = D(K, Y)$.

1.1.2 Sicurezza

Un avversario che osserva Y senza conoscere K o X può tentare di recuperare X o K. È presumibile che l'avversario conosca gli algoritmi di cifratura (E) e decifratura (D). Se l'avversario è interessato solo a un messaggio specifico, si concentrerà sul recupero di X. Tuttavia, se desidera leggere futuri messaggi, cercherà di recuperare K.

1.1.3 Crittografia

I sistemi crittografici possono essere caratterizzati lungo tre dimensioni indipendenti:

1. **Il tipo di operazioni utilizzate per trasformare il testo in chiaro in testo cifrato**. Tutti gli algoritmi di cifratura si basano su due principi generali: la sostituzione, in cui ciascun elemento nel testo in chiaro (*bit, lettera, gruppo di bit o lettere*) è mappato in un altro elemento, e la trasposizione, in cui gli elementi nel testo in chiaro vengono riarrangiati. Il requisito fondamentale è che non venga persa alcuna informazione (*ossia, che tutte le operazioni siano reversibili*). La maggior parte dei sistemi, chiamati sistemi a prodotto, coinvolge multiple fasi di sostituzioni e trasposizioni.
2. **Il numero di chiavi utilizzate**. Se mittente e destinatario utilizzano la stessa chiave, il sistema è chiamato simmetrico, a chiave singola, a chiave segreta o cifratura convenzionale. Se mittente e destinatario utilizzano chiavi diverse, il sistema è chiamato asimmetrico, a due chiavi o cifratura a chiave pubblica.
3. **Il modo in cui il testo in chiaro viene processato**. Una cifra a blocchi processa l'input un blocco di elementi alla volta, producendo un blocco di output per ciascun blocco in input. Una cifra a flusso processa gli elementi in input in modo continuo, producendo l'output elemento per elemento man mano che procede.

Solitamente, l'obiettivo nell'attaccare un sistema di crittografia è di recuperare la chiave in uso anziché semplicemente ottenere il testo in chiaro di un singolo testo cifrato. Esistono due approcci generali per attaccare uno schema di crittografia convenzionale:

1. **Criptoanalisi:** Gli attacchi crittoanalitici si basano sulla natura dell'algoritmo e talvolta su qualche conoscenza delle caratteristiche generali del testo in chiaro o anche su alcune coppie di testo in chiaro - testo cifrato di esempio. Questo tipo di attacco sfrutta le caratteristiche dell'algoritmo per cercare di dedurre un testo in chiaro specifico o la chiave in uso.

2. **Attacco a Forza Bruta:** L'attaccante prova ogni possibile chiave su un testo cifrato finché non ottiene una traduzione intelligibile in testo in chiaro. In media, è necessario provare la metà di tutte le chiavi possibili per avere successo.

Se uno dei due tipi di attacco riesce a dedurre la chiave, l'effetto è catastrofico: tutti i messaggi futuri e passati crittografati con quella chiave sono compromessi.

Il primo tipo di attacco, la criptoanalisi, si basa sulla conoscenza dell'algoritmo e delle caratteristiche del testo in chiaro o su coppie di testo in chiaro - testo cifrato di esempio. L'obiettivo è dedurre il testo in chiaro o la chiave in uso. L'altro tipo di attacco è basato sulla forza bruta, dove vengono provate tutte le possibili chiavi fino a trovare una traduzione intelligibile del testo cifrato.

L'attacco basato solo sul testo cifrato è il più facile da difendere perché l'avversario ha la minore quantità di informazioni con cui lavorare. Tuttavia, in molti casi, l'analista dispone di più informazioni. L'analista potrebbe essere in grado di catturare uno o più messaggi in testo in chiaro insieme alle loro cifrature. Oppure l'analista potrebbe sapere che certi modelli di testo in chiaro appariranno in un messaggio. Ad esempio, un file codificato in formato PostScript inizia sempre con lo stesso modello, o potrebbe esserci un'intestazione o un banner standardizzato in un messaggio di trasferimento di fondi e così via. Tutti questi sono esempi di testo in chiaro conosciuti. Con questa conoscenza, l'analista potrebbe essere in grado di dedurre la chiave in base al modo in cui il testo in chiaro noto viene trasformato.

Strettamente correlato all'attacco basato sul testo in chiaro conosciuto è quello che potrebbe essere definito come un attacco basato su parole probabili. Se l'avversario sta lavorando con la crittografia di un messaggio di prosa generale, potrebbe avere poca conoscenza di ciò che è nel messaggio. Tuttavia, se l'avversario sta cercando informazioni molto specifiche, potrebbero essere noti alcuni pezzi del messaggio. Ad esempio, se viene trasmesso un intero file contabile, l'avversario potrebbe conoscere la posizione di alcune parole chiave nell'intestazione del file. Come altro esempio, il codice sorgente di un programma sviluppato dalla Corporation X potrebbe includere una dichiarazione di copyright in una posizione standardizzata. Se l'analista è in grado in qualche modo di far inserire al sistema sorgente un messaggio scelto dall'analista, allora è possibile un attacco basato sul testo in chiaro scelto. Un esempio di questa strategia è la crittoanalisi differenziale. In generale, se l'analista è in grado di scegliere i messaggi da cifrare, potrebbe deliberatamente selezionare modelli che possono essere previsti per rivelare la struttura della chiave.

Due altri tipi di attacco elencati sono testo cifrato scelto e testo scelto, che sono meno comuni ma possibili. Solo algoritmi relativamente deboli non resistono a un attacco basato solo sul testo cifrato. In generale, un algoritmo di crittografia è progettato per resistere a un attacco basato sul testo in chiaro conosciuto. Uno schema di crittografia è incondizionatamente sicuro se il testo cifrato generato dallo schema non contiene informazioni sufficienti per determinare univocamente il testo in chiaro corrispondente, indipendentemente dalla quantità di testo cifrato disponibile. Cioè, non importa quanto tempo abbia un avversario, è impossibile per lui o lei decifrare il testo cifrato semplicemente perché le informazioni necessarie non ci sono. Con l'eccezione di uno schema noto come "one-time pad", non esiste un algoritmo di crittografia che sia incondizionatamente sicuro. Pertanto, tutto ciò a cui gli utenti di un algoritmo di crittografia possono aspirare è un algoritmo che soddisfi una o entrambe delle seguenti criteri:

1. Il costo per rompere la cifra supera il valore delle informazioni crittografate.
2. Il tempo richiesto per rompere la cifra supera la vita utile delle informazioni.

Uno schema di crittografia è considerato sicuro computazionalmente se soddisfa una qualsiasi delle due precedenti criteri. Sfortunatamente, è molto difficile stimare la quantità di sforzo necessaria per crittoanalizzare con successo il testo cifrato.

Tutte le forme di crittoanalisi per gli schemi di crittografia simmetrica sono progettate per sfruttare il fatto che tracce di struttura o modello nel testo in chiaro possono sopravvivere alla crittografia e possono essere discernibili nel testo cifrato.

1.2 Tecniche di sostituzione

I due elementi base di tutte le tecniche di crittografia sono la sostituzione e la trasposizione.

Una tecnica di sostituzione è una tecnica in cui le lettere del testo in chiaro vengono sostituite da altre lettere o da numeri o simboli. Se il testo in chiaro viene visto come una sequenza di bit, allora la sostituzione comporta la sostituzione di modelli di bit del testo in chiaro con modelli di bit di testo cifrato.

1.2.1 Cifrario di cesare

Il cifrario di Cesare è noto come il primo e più semplice esempio di cifrario a sostituzione. Fu utilizzato da Giulio Cesare e coinvolge la sostituzione di ogni lettera dell'alfabeto con la lettera situata tre posizioni più in basso nell'alfabeto. Ad esempio:

Plain	**Ciphertext**
a	D
b	E
c	F
d	G
e	H
f	I
g	J
h	K
i	L
j	M
k	N
l	O
m	P

Plain	**Ciphertext**
n	Q
o	R
p	S
q	T
r	U
s	V
t	W
u	X
v	Y
w	Z
x	A
y	B
z	C

È importante notare che l'alfabeto è avvolto in modo che la lettera successiva a Z sia A. Ogni lettera dell'alfabeto viene quindi sostituita dalla lettera che si trova a tre posizioni più in basso.

Il cifrario di Cesare è un esempio semplice ma storico di crittografia a sostituzione. Può essere utilizzato per crittografare un messaggio spostando ogni lettera di tre posizioni nell'alfabeto.

L'algoritmo utilizzato è il seguente:

$$C = E(3, p) = (p + 3) \mod 26$$

Lo shift potrebbe essere un valore generico k, quindi l'agoritmo generalizzato è:

$$C = D(k, p) = (p + k) \mod 26$$

ove k prende un valore nel compreso tra 1 e 25. L'algoritmo di decifrazione è simile:

$$p = D(k, C) = (C - k) \mod 26$$

Se è noto che un certo testo cifrato è un cifrario di Cesare, allora una crittoanalisi a forza bruta è facilmente eseguibile: basta provare tutte e 25 le possibili chiavi. La Figura 2.3 mostra i risultati di questa strategia applicata all'esempio di ciphertext. In questo caso, il plaintext salta fuori occupando la terza linea.

Tre importanti caratteristiche di questo problema ci hanno permesso di utilizzare una crittoanalisi a forza bruta:

1. Gli algoritmi di cifratura e decifratura sono noti.
2. Ci sono solo 25 chiavi da provare.
3. La lingua del plaintext è nota ed è facilmente riconoscibile.

La crittoanalisi a forza bruta è un metodo efficace quando si tratta di cifrari di Cesare, in quanto le limitate possibilità di chiavi e la conoscenza dell'algoritmo semplificano notevolmente il processo di decrittografia.

Nella maggior parte delle situazioni di networking, possiamo presumere che gli algoritmi siano noti. Quello che rende generalmente impraticabile la crittoanalisi a forza bruta è l'uso di un algoritmo che impiega un grande numero di chiavi. Ad esempio, l'algoritmo Triple DES, esaminato nel Capitolo 6, utilizza una chiave di 168 bit, che crea uno spazio delle chiavi di 2^{168} o più di 3.7×10^{50} possibili chiavi.

La terza caratteristica è anche significativa. Se la lingua del plaintext è sconosciuta, allora l'output del plaintext potrebbe non essere riconoscibile. Inoltre, l'input potrebbe essere abbreviato o compresso in qualche modo, rendendo di nuovo difficile il riconoscimento.

1.2.2 Cifrario monoalfabetico

Con solo 25 chiavi possibili, il cifrario di Cesare è molto lontano dall'essere sicuro. Un aumento drammatico dello spazio delle chiavi può essere ottenuto consentendo una sostituzione arbitraria. Prima di procedere, definiamo il termine "permutazione".

Permutazione

Una permutazione di un insieme finito di elementi S è una sequenza ordinata di tutti gli elementi di S, con ciascun elemento che appare esattamente una volta. Ad esempio, se $S = \{a, b, c\}$, ci sono sei permutazioni di S:

$$abc, acb, bac, bca, cab, cba$$

In generale, ci sono $n!$ permutazioni di un insieme di n elementi, poiché il primo elemento può essere scelto in uno dei modi n possibili, il secondo in $n-1$ modi, il terzo in $n-2$ modi e così via.

Ricordiamo l'assegnazione per il cifrario di Cesare:

Plain	**Ciphertext**
a	D
b	E
c	F
d	G
e	H
f	I
g	J
h	K
i	L
j	M
k	N
l	O
m	P

Se invece la linea "ciphertext" può essere qualsiasi permutazione dei 26 caratteri alfabetici, allora ci sono 26! o più di 4×10^{26} possibili chiavi. Questo è 10 ordini di grandezza superiore all spazio delle chiavi per `DES` e sembrerebbe eliminare le tecniche di crittoanalisi a forza bruta. Un approccio del genere è chiamato cifrario di sostituzione monoalfabetica, perché viene utilizzato un singolo alfabeto cifrato (*mappatura dall'alfabeto in chiaro all'alfabeto cifrato*) per ogni messaggio.

C'è, tuttavia, un'altra linea di attacco. Se il crittoanalista conosce la natura del testo in chiaro (*ad esempio, testo inglese non compresso*), può sfruttare le regolarità della lingua. Per vedere come potrebbe procedere tale crittoanalisi, diamo qui un esempio parziale adattato da uno in `[SINK09]`. Il testo cifrato da risolvere è il seguente:

```
UzqSovUoHxmoPvgPozPevSgzWSzoPfPeSxUDBmeTSxaIz
vUePHzHmDzSHzoWSfPaPPDTSvPqUzWymxUzUHSx
ePyePoPDzSzUfPomBzWPfUPzHmDJUDTmoHmq
```

Come primo passo, può essere determinata la frequenza relativa delle lettere e confrontata con una distribuzione di frequenza standard per l'inglese. Se il messaggio fosse abbastanza lungo, questa tecnica da sola potrebbe essere sufficiente, ma poiché questo è un messaggio relativamente breve, non possiamo aspettarci una corrispondenza esatta. In ogni caso, le frequenze relative delle lettere nel testo cifrato (*in percentuale*) sono le seguenti:

P 13.33	Z 11.67	S 8.33	U 8.33
O 7.50	M 6.67	H 5.83	D 5.00
E 5.00	V 4.17	X 4.17	F 3.33
W 3.33	Q 2.50	T 2.50	A 1.67
B 1.67	G 1.67	Y 1.67	I 0.83
J 0.83	C 0.00	K 0.00	L 0.00
N 0.00	R 0.00		

confrontando questa suddivisione con la Figura 1.2.1, sembra probabile che le lettere cifrate P e Z siano equivalenti alle lettere in chiaro e e t, ma non è certo quale sia quale. Le lettere S, U, O, M e H sono tutte di frequenza relativamente alta e probabilmente corrispondono alle lettere in chiaro dell'insieme $\{a, h, i, n, o, r, s\}$. Le lettere con le frequenze più basse (*ovvero A, B, G, Y, I, J*) sono probabilmente incluse nell'insieme $\{b, j, k, q, v, x, z\}$.

Ci sono diverse modalità per procedere in questo punto. Potremmo fare alcune assegnazioni provvisorie e iniziare a completare il testo in chiaro per vedere se assomiglia a uno scheletro ragionevole di un messaggio. Un approccio più sistematico è cercare altre regolarità. Ad esempio, potrebbero essere noti alcuni termini nel testo. Oppure potremmo cercare sequenze ripetute di lettere cifrate e cercare di dedurne le corrispondenti lettere in chiaro.

Un potente strumento è rappresentato dalla frequenza delle combinazioni di due lettere, note come digrammi. Potrebbe essere compilata una tabella simile alla Figura 1.2.1 che mostri la frequenza relativa dei digrammi. Il digramma più comune è th. Nel nostro testo cifrato, il digramma più comune è ZW, che appare tre volte. Quindi facciamo corrispondere Z a t e W a h. Quindi, in base alla nostra ipotesi precedente, possiamo equiparare P a e. Ora notiamo che la sequenza ZWP appare nel testo cifrato, e possiamo tradurre quella sequenza come "the". Questo è il trigramma (*combinazione di tre lettere*) più frequente in inglese, il che sembra indicare che siamo sulla strada giusta.

Successivamente, notiamo la sequenza ZWSZ nella prima riga. Non sappiamo se queste quattro lettere formano una parola completa, ma se lo fanno, è del tipo *th_t*. In tal caso, S corrisponde ad a.

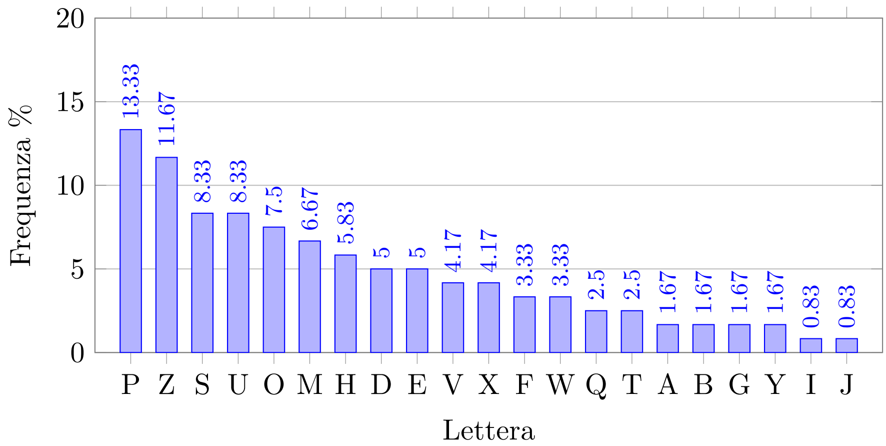


Figura 1.2.1: Frequenze relative alle lettere nei testi inglesi

Finora abbiamo identificato solo quattro lettere, ma già abbiamo una buona parte del messaggio. Continuando l'analisi delle frequenze e sperimentando, dovremmo facilmente trovare una soluzione da questo punto. Il testo completo, con spazi aggiunti tra le parole, è il seguente:

```
it was disclosed yesterday that several informal but
direct contacts have been made with political
representatives of the viet cong in moscow
```

I cifrari monoalfabetici sono facili da decifrare perché riflettono i dati di frequenza dell'alfabeto originale. Una contromisura consiste nel fornire più sostituti, noti come omofoni, per una singola lettera. Ad esempio, la lettera "e" potrebbe essere assegnata a diversi simboli cifrati, come 16, 74, 35 e 21, con ciascun omofono assegnato a una lettera in rotazione o casualmente. Se il numero di simboli assegnati a ciascuna lettera è proporzionale alla frequenza relativa di quella lettera, le informazioni sulla frequenza delle singole lettere vengono completamente oscurate. Il grande matematico Carl Friedrich Gauss credeva di aver ideato un cifrario infrangibile utilizzando gli omofoni. Tuttavia, anche con gli omofoni, ogni elemento del testo in chiaro influenza solo un elemento del testo cifrato, e i modelli di lettere multiple (*ad esempio, le frequenze dei digrammi*) sopravvivono comunque nel testo cifrato, rendendo la crittoanalisi relativamente semplice.

1.2.3 Cifrario di Playfair

Il cifrario di crittografia a più lettere più conosciuto è il cifrario di *Playfair*, che tratta i digrammi nel testo in chiaro come unità singole e li traduce in digrammi nel testo cifrato. L'algoritmo di *Playfair* si basa sull'uso di una matrice 5×5 di lettere costruita utilizzando una parola chiave. Ecco un esempio, risolto da Lord Peter Wimsey nel romanzo "Have His Carcase" di *Dorothy Sayers*:

M	O	N	A	R
H	Y	B	C	D
E	F	G	I	K
L	P	Q	S	T
U	V	W	X	Z

In questo caso, la parola chiave è "monarchia". La matrice viene costruita riempiendo le lettere della parola chiave (senza duplicati) da sinistra a destra e dall'alto verso il basso, e poi riempiendo il resto della matrice con le lettere rimanenti in ordine alfabetico. Le lettere I e J contano come una sola lettera. Il testo in chiaro viene crittografato due lettere alla volta, secondo le seguenti regole:

1. Le lettere ripetute nel testo in chiaro che si trovano nella stessa coppia vengono separate da una lettera di riempimento, come ad esempio "x", quindi "balloon" verrebbe trattato come "ba lx lo on".
2. Due lettere nel testo in chiaro che si trovano nella stessa riga della matrice vengono ciascuna sostituite dalla lettera a destra, con il primo elemento della riga che segue ciclicamente l'ultimo. Ad esempio, "ar" viene crittografato come ¡¡RM".
3. Due lettere nel testo in chiaro che si trovano nella stessa colonna vengono ciascuna sostituite dalla lettera sottostante, con l'elemento superiore della colonna che segue ciclicamente l'ultimo. Ad esempio, "mu" viene crittografato come "CM".
4. In caso contrario, ogni lettera nel testo in chiaro nella coppia viene sostituita dalla lettera che si trova nella stessa riga e nella colonna occupata dall'altra lettera nel testo in chiaro. Quindi, "hs" diventa "BP" e "ea" diventa "IM" (o "JM", a discrezione dell'incifratore).

Il cifrario Playfair rappresenta un grande passo avanti rispetto ai semplici cifrari monoalfabetici. Per una cosa, mentre ci sono solo 26 lettere, ci sono $26 \cdot 26 = 676$ digrammi, rendendo più difficile l'identificazione dei digrammi individuali. Inoltre, le frequenze relative delle singole lettere mostrano una gamma molto più ampia rispetto a quella dei digrammi, rendendo l'analisi delle frequenze molto più difficile. Per queste ragioni, il cifrario Playfair è stato a lungo considerato indistruttibile. È stato utilizzato come sistema standard sul campo dall'Esercito Britannico durante la Prima Guerra Mondiale e ha ancora goduto di un notevole utilizzo da parte dell'Esercito degli Stati Uniti e altre forze alleate durante la Seconda Guerra Mondiale. Nonostante questo livello di fiducia nella sua sicurezza, il cifrario Playfair è relativamente facile da decifrare, perché lascia comunque gran parte della struttura della lingua in chiaro intatta. Di solito, poche centinaia di lettere del testo cifrato sono sufficienti. Un modo per rivelare l'efficacia del cifrario Playfair e di altri cifrari è mostrato nella Figura 1.2.2. La linea etichettata "testo in chiaro" rappresenta una tipica distribuzione di frequenza dei 26 caratteri alfabetici (*senza distinzione tra maiuscole e minuscole*) in un testo normale. Questa è anche la distribuzione di frequenza di qualsiasi cifrario di sostituzione monoalfabetica, perché i valori di frequenza per le singole lettere sono gli stessi, solo con lettere diverse sostituite alle lettere originali. Il grafico è sviluppato nel seguente modo: il numero di occorrenze di ciascuna lettera nel testo viene conteggiato e diviso per il numero di occorrenze della lettera più frequentemente utilizzata. Utilizzando i risultati della Figura 1.2.1, vediamo che la lettera "e" è quella più frequentemente utilizzata. Di conseguenza, "e" ha una frequenza relativa di 1, "t" di $9.056/12.702 \approx 0,72$, e così via. I punti sull'asse orizzontale corrispondono alle lettere in ordine decrescente di frequenza. La Figura 1.2.2 mostra anche la distribuzione di frequenza che si ottiene quando il testo viene crittografato utilizzando il cifrario Playfair. Per normalizzare il grafico, il numero di occorrenze di ciascuna lettera nel testo cifrato è stato nuovamente diviso per il numero di occorrenze della lettera "e" nel testo in chiaro. Il grafico risultante mostra quindi in che misura la distribuzione di frequenza delle lettere, che rende facile risolvere i cifrari di sostituzione, sia mascherata dalla crittografia. Se l'informazione sulla distribuzione di frequenza fosse totalmente nascosta nel processo di crittografia, il grafico delle frequenze nel testo cifrato sarebbe piatto e l'analisi crittografica utilizzando solo il testo cifrato sarebbe effettivamente impossibile. Come mostra la figura, il cifrario Playfair ha una distribuzione più piatta rispetto al testo in chiaro, ma comunque rivela molta struttura su cui un crittoanalista può lavorare. Il grafico mostra anche il cifrario Vigenère, discusso successivamente. Le curve di Hill e Vigenère nel grafico si basano su risultati riportati in `[SIMM93]`.

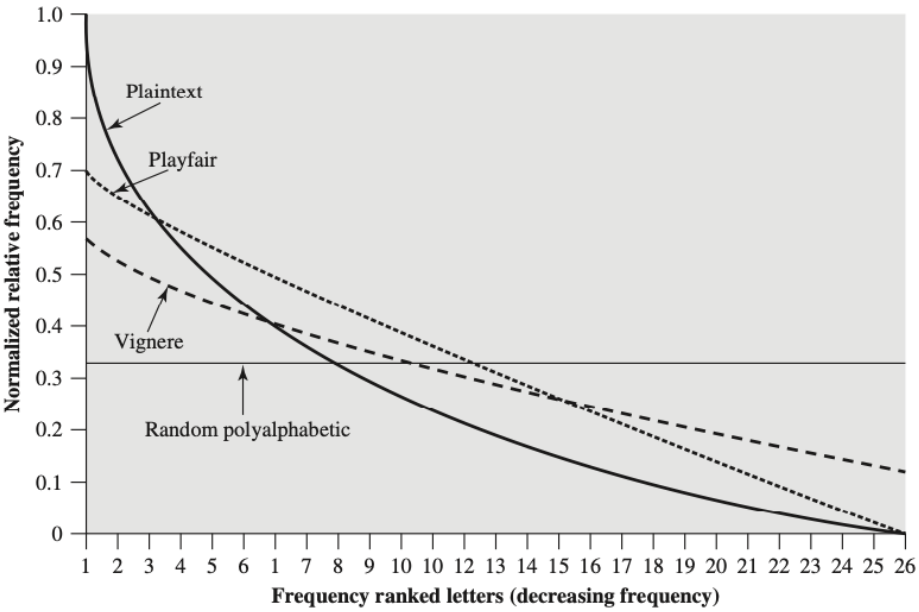


Figura 1.2.2: Frequenza delle occorrenze delle lettere nel testo cifrato

1.2.4 Cifrario poliafabetico

Un altro modo per migliorare la semplice tecnica monoalfabetica è utilizzare diverse sostituzioni monoalfabetiche man mano che si procede attraverso il messaggio in chiaro. Il nome generico per questo approccio è cifrario di sostituzione polialfabetica. Tutte queste tecniche hanno le seguenti caratteristiche in comune:

1. Un insieme di regole di sostituzione monoalfabetica correlate viene utilizzato.
2. Una chiave determina quale particolare regola viene scelta per una data trasformazione.

Cifrario di Vigenère

Il cifrario di Vigenère è uno dei più noti e semplici cifrari polialfabetici. In questo schema, l'insieme di regole di sostituzione monoalfabetica correlate consiste nei 26 cifrari di Cesare con spostamenti da 0 a 25. Ogni cifrario è indicato da una chiave lettera, che è la lettera cifrata che sostituisce la lettera in chiaro "a". Quindi, un cifrario di Cesare con uno spostamento di 3 è indicato dalla chiave "3".

Il cifrario di Vigenère può essere espresso nel seguente modo. Supponiamo una sequenza di lettere in chiaro $P = p_0, p_1, p_2, \ldots, p_{n-1}$ e una chiave costituita dalla sequenza di lettere $K = k_0, k_1, k_2, \ldots, k_{m-1}$, dove tipicamente $m \leq n$. La sequenza di lettere cifrate $C = C_0, C_1, C_2, \ldots, C_{n-1}$ viene calcolata come segue:

$$C_i = (p_i + k_{i \mod m}) \mod 26$$

Dove $i \mod m$ indica l'operazione di modulo. In altre parole, si somma la lettera in chiaro p_i con la corrispondente lettera chiave $k_{i \mod m}$ e il risultato è ridotto modulo 26 per ottenere la lettera cifrata C_i.

La decrittografia è effettuata in modo simile:

$$p_i = (C_i - k_{i \mod m}) \mod 26$$

Per crittografare un messaggio, è necessaria una chiave lunga quanto il messaggio stesso. Di solito, la chiave è una parola chiave ripetuta. Ad esempio, se la parola chiave è "deceptive" e il messaggio è "we are discovered save yourself", la cifratura procede come segue:

chiave:	deceptivedeceptivedeceptive
in chiaro:	wearediscoveredsaveyourself
cifrato:	zIcvTWqngRzgvTWavzHcqyglmgJ

Espresso in forma di enumerazione alfabetica, il cifrario di Vigenère è il seguente:

key	3	4	2	4	15	19	8	21	4	3	4	2	4	15
plaintext	22	4	0	17	4	3	8	18	2	14	21	4	17	4
ciphertext	25	8	2	21	19	22	16	13	6	17	25	6	21	19

key	19	8	21	4	3	4	2	4	15	19	8	21	4
plaintext	3	8 18	0	21	4	24	14	20	17	18	4	11	5
ciphertext	22	0	21	25	7	2	16	24	6	11	12	6	9

La forza di questo cifrario è che ci sono molte lettere cifrate per ciascuna lettera in chiaro, una per ciascuna lettera unica della parola chiave. Pertanto, le informazioni sulla frequenza delle lettere sono oscurate. Tuttavia, non viene persa tutta la conoscenza della struttura del testo in chiaro.

Innanzitutto, supponiamo che l'avversario ritenga che il cifrato sia stato crittografato utilizzando una sostituzione monoalfabetica o un cifrario di Vigenère. Si può effettuare un semplice test per effettuare una determinazione. Se si utilizza una sostituzione monoalfabetica, le proprietà statistiche del cifrato dovrebbero essere le stesse del linguaggio del testo in chiaro. Quindi, facendo riferimento alla Figura 1.2.1, dovrebbe esserci una lettera del cifrato con una frequenza relativa di circa il $12,7\%$, una con circa il $9,06\%$, e così via. Se è disponibile solo un singolo messaggio per l'analisi, non ci si aspetterebbe una corrispondenza esatta di questo piccolo campione con il profilo statistico del linguaggio del testo in chiaro. Tuttavia, se la corrispondenza è vicina, possiamo assumere una sostituzione monoalfabetica.

Se, d'altra parte, si sospetta un cifrario di Vigenère, il progresso dipende dalla determinazione della lunghezza della chiave, come verrà visto tra un momento. Per ora, concentriamoci su come può essere determinata la lunghezza della chiave. L'importante intuizione che porta a una soluzione è la seguente: se due sequenze identiche di lettere in chiaro si verificano a una distanza che è un multiplo intero della lunghezza della chiave, genereranno sequenze di cifrati identiche. Nell'esempio precedente, due istanze della sequenza "red" sono separate da nove posizioni dei caratteri. Di conseguenza, in entrambi i casi, la "r" è cifrata usando la lettera chiave "e", la "e" è cifrata usando la lettera chiave "p", e la "d" è cifrata usando la lettera chiave "t". Quindi, in entrambi i casi, la sequenza di cifrati è "VTW". Indichiamo ciò evidenziando le lettere pertinenti del cifrato e sfumando i numeri di cifrato rilevanti.

Un analista che osserva solo il cifrato rileverebbe le sequenze ripetute "VTW" con uno spostamento di 9 e farebbe l'assunzione che la chiave sia lunga tre o nove lettere. L'apparizione di "VTW" due volte potrebbe essere casuale e potrebbe non riflettere lettere in chiaro identiche crittografate con lettere chiave identiche. Tuttavia, se il messaggio è abbastanza lungo, ci saranno diverse sequenze di cifrati ripetuti. Cercando fattori comuni negli spostamenti delle diverse sequenze, l'analista dovrebbe essere in grado di fare una buona congettura sulla lunghezza della chiave.

La soluzione del cifrario ora dipende da un'importante intuizione. Se la lunghezza della chiave è "m", il cifrario, in effetti, consiste di "m" sostituzioni monoalfabetiche separate. Ad esempio, con la chiave "DECEPTIVE", le lettere nelle posizioni $1, 10, 19$ e così via sono tutte crittografate con lo stesso cifrario monoalfabetico. Quindi, possiamo utilizzare le conosciute caratteristiche di frequenza del linguaggio del testo in chiaro per attaccare separatamente ciascuna delle sostituzioni monoalfabetiche.

La natura periodica della chiave può essere eliminata utilizzando una chiave non ripetitiva lunga quanto il messaggio stesso. Vigenère ha proposto quello che viene chiamato un sistema autokey, in cui una chiave è concatenata al testo in chiaro stesso per fornire una chiave in esecuzione. Nel nostro esempio:

Chiave	deceptivewearediscoveredsav
Testo in chiaro	wearediscoveredsaveyourself
Cifrato	zIcvTWqngKzeIIgaSxSTSlvvWla

Anche questo schema è vulnerabile all'analisi crittografica. Poiché la chiave e il testo in chiaro condividono la stessa distribuzione di frequenza delle lettere, è possibile applicare una tecnica statistica. Ad esempio, la lettera "e" cifrata con "e", come indicato in Figura 1.2.1, ci si aspetta che si verifichi con una frequenza di $(0,127)^2 \approx 0,016$, mentre la lettera "t" cifrata con "t" si verificherebbe solo circa la metà delle volte. Queste regolarità possono essere sfruttate per raggiungere un'analisi crittografica di successo.

Cifraro di Vernam

La difesa definitiva contro una crittoanalisi di questo tipo consiste nel scegliere una chiave lunga quanto il testo in chiaro e che non abbia alcuna relazione statistica con esso. Un sistema del genere fu introdotto da un ingegnere AT&T di nome Gilbert Vernam nel 1918.

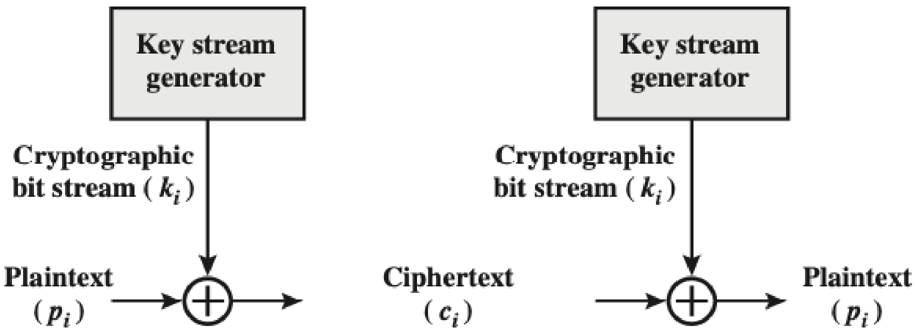


Figura 1.2.3: Schema di Vernam

Il suo sistema lavora con i dati binari (*bit*) e utilizza l'operazione di `XOR` logico:

$$c_i = p_i \oplus k_i$$

dove:

- c_i è l'i-esimo bit del testo cifrato

- p_i è l'i-esimo bit del testo in chiaro
- k_i è l'i-esimo bit della chiave
- $\oplus$ è l'operatore `XOR` logico

L'operazione di `XOR` logico è definita come segue:

Input	**Output**	**Descrizione**
0	0	$0 \oplus 0 = 0$
0	1	$0 \oplus 1 = 1$
1	0	$1 \oplus 0 = 1$
1	1	$1 \oplus 1 = 0$

L'essenza di questa tecnica risiede nel modo in cui viene costruita la chiave. Vernam ha proposto l'uso di una lunga striscia di nastro che alla fine ripeteva la chiave, quindi di fatto il sistema funzionava con una chiave molto lunga ma ripetitiva. Sebbene uno schema del genere, con una chiave lunga, presenti notevoli difficoltà crittografiche, può essere violato con una quantità sufficiente di testo cifrato, l'uso di sequenze di testo in chiaro conosciute o probabili, o entrambe.

1.2.5 One-Time Pad

Il cifrario monouso, o *one-time pad*, è un sistema di crittografia che utilizza una chiave casuale di lunghezza uguale o maggiore del messaggio da crittografare. È un sistema di crittografia perfetta, nel senso che il messaggio cifrato non può essere decifrato o violato senza conoscere la chiave.

- **Chiave Casuale**: La chiave utilizzata nel cifrario monouso è una sequenza casuale di bit o caratteri, lunga quanto il messaggio da crittografare. Essendo completamente casuale, non contiene alcuna struttura o pattern riconoscibile.
- **Lunghezza della Chiave**: La chiave deve essere della stessa lunghezza del messaggio in chiaro. Questo significa che ogni messaggio richiede una chiave diversa e della stessa lunghezza.
- **Unicità**: Ogni chiave è utilizzata una sola volta. Dopo essere stata usata per crittografare o decrittografare un messaggio, la chiave viene scartata e non viene mai riutilizzata.
- **Sicurezza Statistica**: La sicurezza del cifrario monouso deriva dalla sua totale casualità. Poiché la chiave è una sequenza casuale e unica per ogni messaggio, non esiste alcuna relazione statistica tra il testo cifrato e il testo in chiaro. Questo significa che il testo cifrato non fornisce alcuna informazione utile per violare il cifrario, rendendolo teoricamente indistruttibile.

Il cifrario monouso, noto come "one-time pad" è considerato perfetto dal punto di vista statistico e crittografico per due ragioni principali:

1. **Casualità della Chiave:** La chiave nel cifrario monouso è una sequenza casuale di bit o caratteri. La casualità è fondamentale dal punto di vista statistico. In termini di probabilità, ogni bit o carattere nella chiave ha una probabilità del 50% di essere 0 o 1 (*in caso di bit*) o di essere una qualsiasi lettera nell'alfabeto (*in caso di caratteri*). Questo fatto è rappresentato dalla distribuzione di probabilità uniforme.

   Formula della distribuzione uniforme per bit:

   $$P(X = 0) = P(X = 1) = \frac{1}{2}$$

   Formula della distribuzione uniforme per caratteri:

   $$P(X = x_i) = \frac{1}{n} \text{ per ogni } x_i \text{ nell'alfabeto di lunghezza } n$$

   Ad esempio, in un alfabeto di 26 lettere, la probabilità di ciascuna lettera è $1/26$.

   Nel caso di una chiave di lunghezza n, la probabilità di una particolare sequenza di n bit o caratteri è $1/2^n$ o $1/n^n$ rispettivamente. Poiché non vi è alcuna relazione nei bit o caratteri della chiave.

2. **Unicità della Chiave:** Ogni chiave viene utilizzata una sola volta per crittografare o decrittografare un messaggio specifico e viene scartata dopo l'uso. Questo significa che non c'è alcuna relazione statistica tra il testo cifrato e il testo in chiaro. L'assenza di qualsiasi pattern o relazione è fondamentale dal punto di vista della teoria della probabilità.

1.3 Concatenazione di crittosistemi

Una permutazione è un mapping iniettivo e suriettivo di un insieme in se stesso. Una permutazione è una sostituzione che mappa ogni lettera dell'alfabeto in un'altra lettera. Quindi:

$$\pi : \mathcal{A} \to \mathcal{A}$$

sappiamo che è vulnerabile all'analisi delle frequenze, quindi possiamo applicare un sistema di concatenazione:

$$\pi_1 \circ \pi_2 \circ \pi_3 \circ \pi_4 \circ \pi_5$$

il problema è che la composizione di permutazioni è ancora una permutazione, quindi è la stessa cosa chè l'eseguire un'unica permutazione π. Varia quindi solo la rappresentazione della permutazione. Per risolvere il problema serve un elemento aggiuntivo che non sia una permutazione, ad esempio una trasposizione.

1.4 Macchina a Rotori

Una macchina a rotori è una macchina crittografica che sfrutta la crittografia a sostituzione **polialfabetica**. La macchina è composta da cilindri rotanti, ognuno con 26 pin di input e 26 pin di output, ciascuno con connessioni interne che collegano input e output in modo univoco. Un singolo cilindro crea una sostituzione monoalfabetica, ruotando dopo ogni input, il che crea una sostituzione polialfabetica con un periodo di 26 caratteri.

La vera forza delle macchine a rotori emerge quando vengono utilizzati più cilindri collegati in serie. Quando si preme un tasto, il cilindro più vicino all'input ruota di una posizione, influenzando il successivo e così via. Questa configurazione crea una vasta varietà di sostituzioni alfabetiche, con un'enorme quantità di possibilità quando si utilizzano più cilindri.

Questo schema crittografico rappresenta una sfida significativa per i crittoanalisti poiché richiede un'enorme quantità di dati cifrati per essere decifrato in modo significativo, rendendo molto difficile l'analisi crittografica basata sulla frequenza delle lettere.

Tale sistema protegge dall'analisi delle frequenze poiché per 26^3 permutazioni non è possibile fare un'analisi delle frequenze.

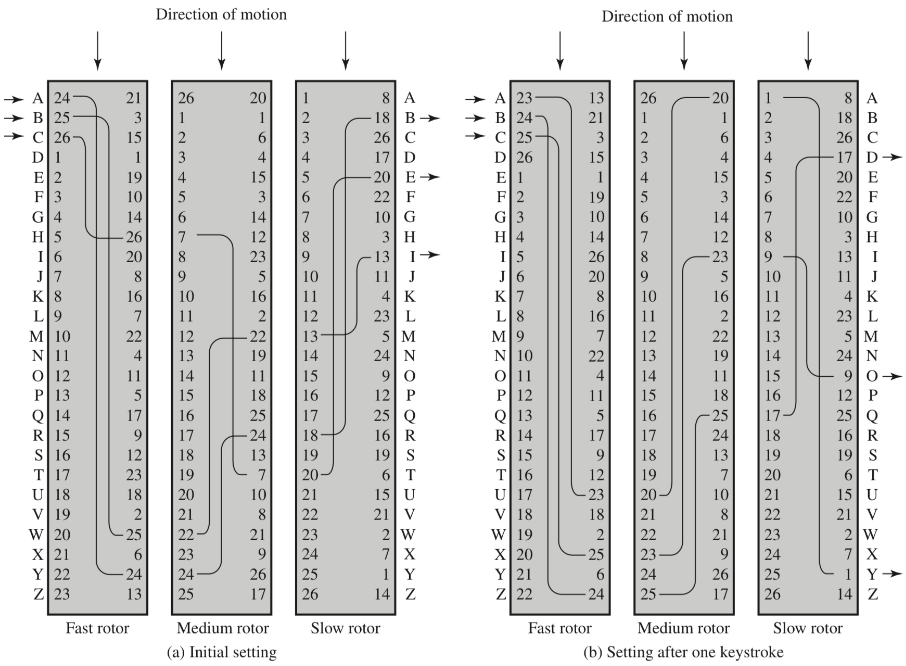


Figura 1.4.1: Macchina a rotori

Enigma

La macchina Enigma è una macchina elettromeccanica portatile utilizzata per cifrare e decifrare messaggi segreti. È stata utilizzata in Germania durante la seconda guerra mondiale. La decodifica dei messaggi è molto complessa, vista la grande quantità di combinazioni possibili.

Se ho a una macchina che per essere decifrata ha bisogno di un tempo più alto della validità dei dati, allora posso dire che è sicura.

Il modo per decodificare i messaggi è stato fondamentale sapere che il messaggio iniziava con una parola chiave, che era sempre la stessa. Sapendo questo, si poteva decodificare il messaggio riducendo notevolmente l'insieme delle chiavi disponibili per la decodifica. Conoscevano quindi il **plaintext**.

La concatenzione di crittosistemi è molto sicura, ma non è sicura contro gli attacchi dove si conosce il plaintext.

1.5 Classificazione dei livelli di sicurezza

La classificazione si basa sulla difficoltà di violare il sistema, quindi sul tipo di attacchi a cui resiste.

- **Known Cipher Text Attack**: l'attaccante conosce il testo cifrato.
- **Known Plaintext Attack**: l'attaccante vede il testo in chiaro e il corrispondente testo cifrato.
- **Chosen Plaintext Attack**: l'attaccante sceglie il testo in chiaro e conosce il corrispondente testo cifrato.
- **Adaptive Chosen Ciphertext Attack**: l'attaccante può continuamente scegliere il testo in chiaro e vedere il corrispondente testo cifrato.

La classifica è in base alla potenza che gli do nell'attaccarmi. L'obiettivo è costruire un sistema che sia sicuro contro gli attacchi Adaptive Chosen Ciphertext Attack.

1.6 Data Encryption Standard (*DES*)

Il DES è un cifrario a blocchi che è stato uno dei primi algoritmi crittografici adottati su larga scala ed è stato ampiamente utilizzato fino a quando è stato sostituito dall'**Advanced Encryption Standard** (*AES*).

Lunghezza della Chiave: Il DES utilizza una chiave di 56 bit. Questo significa che ci sono 2^{56} possibili chiavi differenti che possono essere utilizzate per cifrare e decifrare dati.

Lunghezza del Blocco: Il DES opera su blocchi di dati di 64 bit. Questo significa che ogni blocco di testo in chiaro da cifrare o testo cifrato da decifrare deve essere di 64 bit.

1.6.1 Algoritmo di Cifratura

Il processo di cifratura DES coinvolge una serie di passaggi iterativi noti come "rounds". Di seguito vengono spiegati i passaggi chiave dell'algoritmo di cifratura DES:

- **Permutazione Iniziale (IP)**: Il blocco di testo in chiaro di 64 bit viene permutato secondo una tabella specifica.
- **Divisione in Blocchi Sinistro e Destro**: Il blocco permutato viene diviso in due parti uguali, ciascuna di 32 bit, note come "sinistro" e "destro".

- **Round di Fiestel**: Il DES utilizza una struttura chiamata "round di Fiestel", in cui i blocchi subiscono diverse trasformazioni, inclusa l'applicazione di una funzione di espansione, una funzione di sostituzione (*S-box*), una permutazione e l'operazione `XOR` con una sottochiave derivata dalla chiave principale.
- **Iterazioni (16 Rounds)**: L'intero processo di round di Fiestel viene iterato 16 volte, con l'uso di diverse sottochiavi derivate dalla chiave principale.
- **Permutazione Finale (FP)**: Alla fine delle 16 iterazioni, i blocchi sinistro e destro vengono combinati e permutati nuovamente secondo una tabella specifica, ottenendo così il testo cifrato.

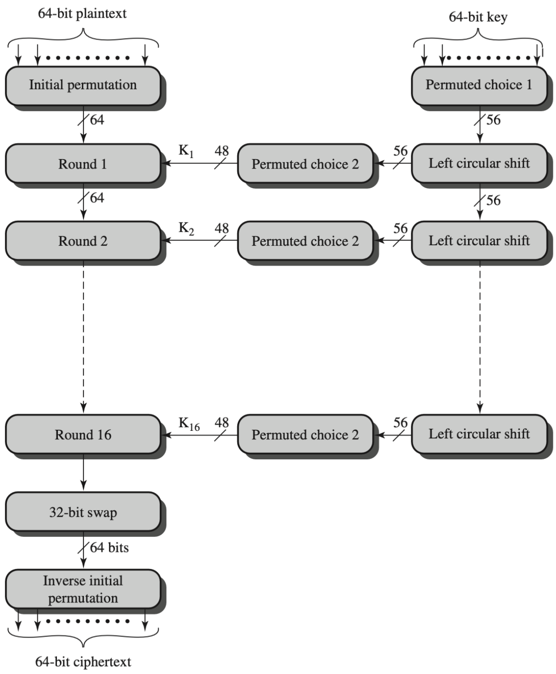


Figura 1.6.1: DES

1.6.2 Decifratura DES

Il processo di decifratura DES è essenzialmente l'operazione inversa della cifratura. Le sottochiavi vengono utilizzate in ordine inverso rispetto alla cifratura per decifrare il testo cifrato e ottenere il testo in chiaro originale.

1.7 Cifrari a flusso e a blocchi

Un **cifrario a flusso** crittografa un flusso di dati digitali un bit o un byte alla volta. Esempi di cifrari a flusso classici sono il cifrario di Vigenère con autochiave e il cifrario di Vernam. Nell'ideale, si utilizzerebbe una versione del cifrario di Vernam con one-time pad, in cui lo stream di chiavi ha la stessa lunghezza dello stream di bit in chiaro. Tuttavia, perché questo sia praticamente realizzabile, lo stream di chiavi deve essere fornito in anticipo ad entrambi gli utenti attraverso un canale indipendente e sicuro, il che può rappresentare una sfida logistica se il traffico dati previsto è di grandi dimensioni.

Di conseguenza, per ragioni pratiche, il generatore di stream di bit deve essere implementato come una procedura algoritmica, in modo che lo stream di bit crittografico possa essere prodotto da entrambi gli utenti. In questo approccio, il generatore di stream di bit è un algoritmo controllato dalla chiave e deve produrre uno stream di bit crittograficamente robusto. I due utenti devono condividere solo la chiave di generazione, e ognuno può generare lo stream di chiavi.

Un **cifrario a blocchi** tratta un blocco di testo in chiaro come un'entità unica e produce un blocco di testo cifrato della stessa lunghezza. Di solito, si utilizza una dimensione di blocco di 64 o 128 bit. Allo stesso modo del cifrario a flusso, i due utenti condividono una chiave di crittografia simmetrica. Utilizzando alcune delle modalità di funzionamento spiegate in precedenza, un cifrario a blocchi può essere usato per ottenere lo stesso effetto di un cifrario a flusso.

Molto più sforzo è stato dedicato all'analisi dei cifrari a blocchi, poiché sembrano essere applicabili a una gamma più ampia di applicazioni rispetto ai cifrari a flusso. La maggior parte delle applicazioni crittografiche simmetriche basate su rete fa uso di cifrari a blocchi. Pertanto, le discussioni in questo contesto si concentreranno principalmente su di essi.

1.7.1 Electronic Code Book

Il "Electronic Code Book" (`ECB`) è una delle modalità di funzionamento di un cifrario a blocchi, utilizzato per crittografare un blocco di testo di lunghezza fissa. In questa modalità, ogni blocco di testo in chiaro viene crittografato separatamente utilizzando la stessa chiave. Non c'è alcuna dipendenza tra i blocchi di testo in chiaro durante il processo di crittografia. Pertanto, gli stessi blocchi di testo in chiaro generano gli stessi blocchi di testo cifrato. Tuttavia, questo comporta il rischio di sicurezza in quanto pattern di testo in chiaro simili generano pattern di testo cifrato simili, rendendo il sistema vulnerabile a un'analisi statistica. Nonostante questa debolezza, l'`ECB` è ancora utilizzato in alcuni scenari in cui la semplicità e la velocità sono prioritarie rispetto alla sicurezza, come per la crittografia di dati non sensibili o per applicazioni specifiche in cui la perdita di alcuni blocchi non compromette la sicurezza complessiva del sistema.

La modalità più semplice è la modalità di codifica elettronica (`ECB`), in cui il testo in chiaro viene gestito un blocco alla volta e ogni blocco di testo in chiaro viene crittografato utilizzando la stessa chiave. Il termine "codice" è utilizzato perché, per una data chiave, esiste un testo cifrato univoco per ogni blocco di testo in chiaro di b bit. Pertanto, possiamo immaginare un'enorme tabella di corrispondenza in cui vi è una voce per ogni possibile modello di testo in chiaro di b bit che mostra il relativo testo cifrato corrispondente.

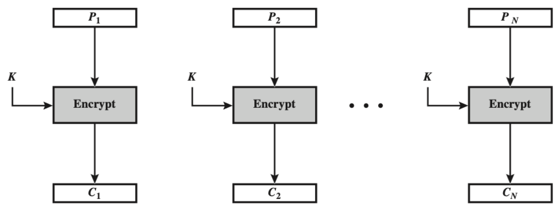


Figura 1.7.1: Electronic Code Book

1.7.2 Cipher Block Chaining

La modalità di funzionamento "Cipher Block Chaining" (CBC) è un metodo di crittografia a blocchi che introduce un certo grado di indipendenza tra i blocchi di testo in chiaro durante il processo di crittografia. Funziona come segue:

1. Prima di crittografare, viene generato un vettore di inizializzazione casuale noto come vettore di inizializzazione (IV). Questo vettore è combinato con il primo blocco di testo in chiaro tramite un'operazione di XOR.
2. Il risultato di questa operazione XOR viene quindi crittografato utilizzando l'algoritmo di cifratura a blocchi insieme alla chiave.
3. Il blocco di testo cifrato risultante viene poi combinato con il blocco di testo successivo prima della crittografia. Questo collegamento tra i blocchi di testo in chiaro aiuta a rompere la correlazione tra i blocchi di testo in chiaro, migliorando la sicurezza rispetto alla modalità di Electronic Code Book (ECB).
4. Questo processo continua per tutti i blocchi di testo in chiaro successivi, garantendo che ciascun blocco di testo cifrato dipenda dal blocco di testo in chiaro precedente, oltre che dalla chiave.

La decodifica avviene seguendo lo stesso processo in ordine inverso, utilizzando il vettore di inizializzazione e la chiave per ottenere il testo in chiaro originale.

La modalità CBC è considerata più sicura dell'ECB poiché introduce una dipendenza tra i blocchi di testo in chiaro, rendendo più complessa l'analisi statistica e aumentando la resistenza agli attacchi crittoanalitici. Tuttavia, è importante gestire correttamente il vettore di inizializzazione per garantire la sicurezza e l'integrità del sistema di crittografia.

$$C_j = E(K, [C_{j-1} \oplus P_j])$$

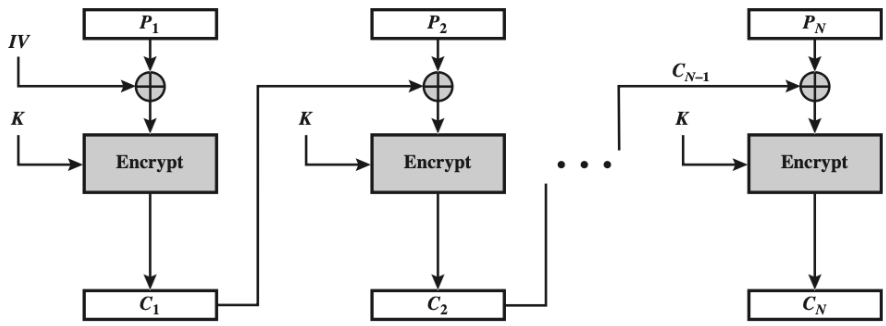


Figura 1.7.2: Cipher Block Chaining

1.7.3 Cipher Feedback

Per `AES`, `DES` o qualsiasi altro cifrario a blocchi, la crittografia viene eseguita su un blocco di b bit. Nel caso di `DES`, $b = 64$, e nel caso di `AES`, $b = 128$. Tuttavia, è possibile convertire un cifrario a blocchi in un cifrario a flusso utilizzando una delle modalità di funzionamento: la modalità di Feedback di Cifratura (`CFB`) e la modalità di Feedback di Output (`OFB`).

Un cifrario a flusso elimina la necessità di aggiungere padding a un messaggio per renderlo un numero intero di blocchi. Inoltre, può funzionare in tempo reale. Pertanto, se viene trasmesso uno stream di caratteri, ogni carattere può essere cifrato e trasmesso immediatamente utilizzando un cifrario a flusso orientato ai caratteri.

Una proprietà desiderabile di un cifrario a flusso è che il testo cifrato abbia la stessa lunghezza del testo in chiaro. Pertanto, se vengono trasmessi caratteri di 8 bit, ogni carattere dovrebbe essere cifrato per produrre un output di testo cifrato di 8 bit. Se vengono prodotti più di 8 bit, la capacità di trasmissione viene sprecata.

La modalità `CFB` è illustrata nello schema. In questa modalità, il testo in chiaro è diviso in segmenti di s bit, dove s è la dimensione dell'unità di trasmissione. Durante la crittografia, viene utilizzato un registro a scorrimento di b bit inizializzato con un vettore di inizializzazione (`IV`). I primi s bit più significativi dell'output della funzione di crittografia vengono combinati con il primo segmento di testo in chiaro P_1 tramite l'operazione XOR per produrre l'unità di testo cifrato C_1, che viene quindi trasmessa. Il contenuto del registro a scorrimento viene spostato a sinistra di s bit e C_1 viene inserito nei s bit meno significativi del registro a scorrimento. Questo processo continua fino a quando tutti i segmenti di testo in chiaro sono stati crittografati.

Per la decifratura, viene utilizzato lo stesso schema, ad eccezione che l'unità di testo cifrato ricevuta viene combinata tramite `XOR` con l'output della funzione di crittografia per produrre l'unità di testo in chiaro. È importante notare che viene utilizzata la funzione di crittografia e non quella di decifratura. Questo è facilmente spiegabile. Sia $MSBs(X)$ definita come i s bit più significativi di X. Quindi

$$C_1 = P_1 \oplus MSBs[E(K, IV)]$$

Di conseguenza, riarrangiando i termini:

$$P_1 = C_1 \oplus MSBs[E(K, IV)]$$

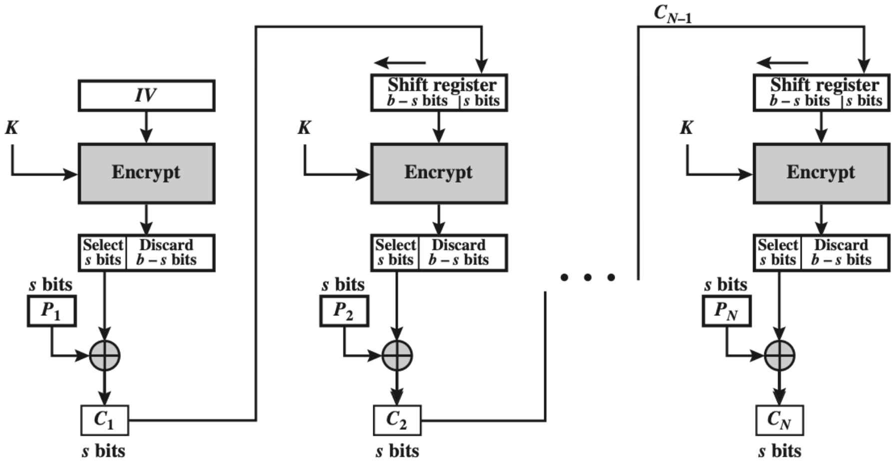


Figura 1.7.3: Cipher Feedback

1.7.4 Output Feedback

La modalità di Feedback di Output (`OFB`) ha una struttura simile a quella di `CFB`. Per `OFB`, l'output della funzione di crittografia viene retroalimentato e diventa l'input per crittografare il blocco successivo di testo in chiaro. In `CFB`, l'output dell'unità `XOR` viene retroalimentato e diventa l'input per crittografare il blocco successivo. L'altra differenza è che la modalità `OFB` opera su blocchi completi di testo in chiaro e testo cifrato, mentre `CFB` opera su un sottoinsieme di s bit. La crittografia `OFB` può essere espressa come:

$$C_j = P_j \oplus E(K, O_{j-1})$$
$$O_{j-1} = E(K, O_{j-2})$$

Un po' di riflessione dovrebbe convincerti che possiamo riscrivere l'espressione di crittografia come:

$$C_j = P_j \oplus E(K, [C_{j-1} \oplus P_{j-1}])$$

Riarrangiando i termini, possiamo dimostrare che la decifratura funziona:

$$P_j = C_j \oplus E(K, [C_{j-1} \oplus P_{j-1}])$$

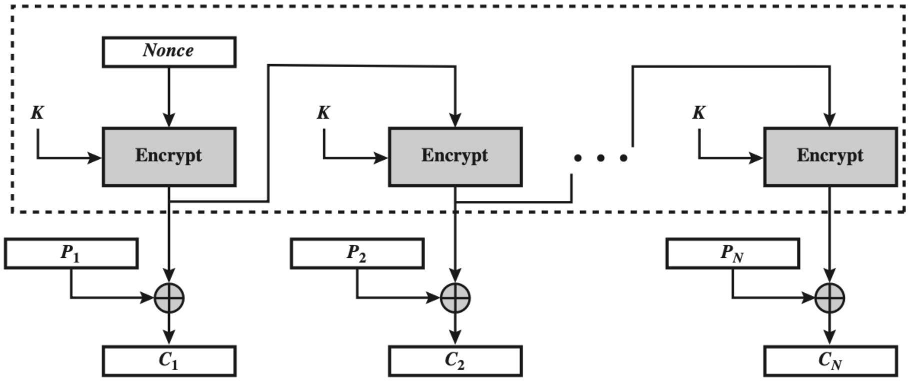


Figura 1.7.4: Output Feedback

I blocchi di output della cifratura, O_i, dipendono solo dalla chiave e dal vettore di inizializzazione (`IV`) e non dipendono dal testo in chiaro. Pertanto, per una data chiave e `IV`, lo stream di bit di output utilizzato per l'operazione XOR con lo stream di bit in chiaro è fisso. Se due messaggi diversi avessero un blocco identico di testo in chiaro nella stessa posizione, un attaccante sarebbe in grado di determinare quella parte dello stream O_i.

Un vantaggio del metodo `OFB` è che gli errori di bit nella trasmissione non si propagano. Ad esempio, se si verifica un errore di bit in C_1, solo il valore ripristinato di P_1 viene influenzato; i blocchi di testo in chiaro successivi non vengono corrotti. Con `CFB`, C_1 serve anche come input al registro a scorrimento e quindi causa corruzioni aggiuntive a valle.

Lo svantaggio di `OFB` è che è più vulnerabile a un attacco di modifica dello stream di messaggi rispetto a `CFB`. Considera che complementare un bit nel testo cifrato complementa il bit corrispondente nel testo in chiaro ripristinato. Pertanto, possono essere effettuate modifiche controllate al testo in chiaro ripristinato. Questo potrebbe rendere possibile per un avversario, apportando le modifiche necessarie alla parte di checksum del messaggio e alla parte di dati, modificare il testo cifrato in modo che non sia rilevato da un codice di correzione degli errori.

`OFB` ha la struttura di un tipico cifrario a flusso, poiché il cifrario genera uno stream di bit come funzione di un valore iniziale e di una chiave, e questo stream di bit viene operato con `XOR` con i bit del testo in chiaro. Lo stream generato che viene operato con `XOR` con il testo in chiaro è indipendente dal testo in chiaro stesso; questo è evidenziato da riquadri tratteggiati in Figura (1.7.4). Una differenza rispetto ai cifrari a flusso è che `OFB` crittografa il testo in chiaro un blocco intero alla volta, dove tipicamente un blocco è di 64 o 128 bit. Molti cifrari a flusso crittografano un byte alla volta.

1.8 Feistel

L'algoritmo di Feistel è una tecnica di cifratura a blocchi che opera dividendo il testo in chiaro in due metà e applicando una serie di round che modificano iterativamente il testo in chiaro. È progettato per essere efficiente e facilmente invertibile per la decifratura. Il suo funzionamento può essere descritto nei seguenti passaggi:

1. **Inizializzazione**: Il testo in chiaro di lunghezza $2n$ viene diviso in due metà di lunghezza n ciascuna, solitamente denotate come L e R (per sinistra e destra). Questi blocchi vengono inizialmente caricati come input nell'algoritmo.

2. **Rounds di Cifratura**: L'operazione di cifratura si compone di più round, ognuno dei quali esegue le seguenti operazioni:

   - La metà destra R del testo in chiaro passa attraverso una funzione di trasformazione che dipende da una sottochiave specifica del round. La funzione è progettata per introdurre una complessità tale da rendere il processo di crittoanalisi complesso e costoso.

   - L'output della funzione di trasformazione viene poi combinato con la metà sinistra L del testo in chiaro utilizzando l'operazione XOR (eXclusive OR). L'output di questa operazione diventa la nuova metà destra R per il round successivo.

   - Nel frattempo, la vecchia metà destra R diventa la nuova metà sinistra L per il prossimo round.

3. **Conclusione**: Dopo un numero prefissato di round, il processo si conclude scambiando le due metà. Quindi, l'output finale dell'ultimo round diventa il testo cifrato.

L'algoritmo di Feistel è considerato sicuro a causa della sua struttura iterativa e dell'uso di funzioni di trasformazione complesse all'interno di ogni round. La sua reversibilità semplifica anche il processo di decifratura, rendendolo altrettanto efficiente. Questa struttura offre un equilibrio tra sicurezza e efficienza, che lo rende adatto per un'ampia gamma di applicazioni di crittografia. È stato ampiamente utilizzato come base per diversi algoritmi di cifratura di successo, come il Data Encryption Standard (`DES`).

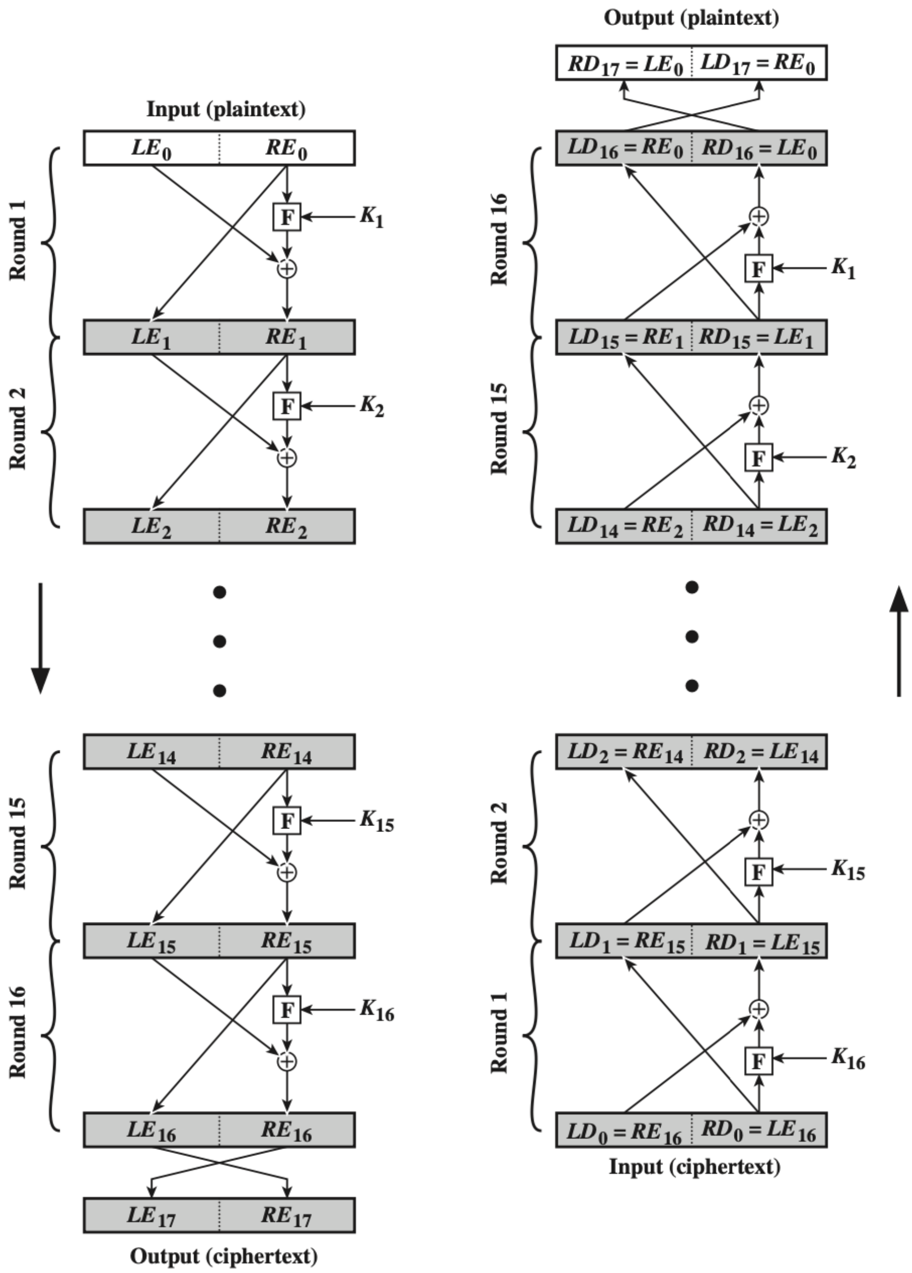


Figura 1.8.1: Funzionamento di Feistel

L'efficacia dell'algoritmo di Feistel con una funzione F arbitraria deriva dalla sua capacità di confondere e diffondere il testo in chiaro attraverso l'uso di operazioni matematiche e logiche complesse. Anche se F può essere scelta in modo flessibile per svolgere una vasta gamma di operazioni crittografiche, ci sono alcune proprietà fondamentali che permettono a questo tipo di cifrario di essere robusto e sicuro:

1. **Confusione**: La funzione F introduce confusione nel testo in chiaro, in modo che la relazione tra il testo cifrato e la chiave sia complessa e non lineare. Questo rende difficile per un crittoanalista estrarre informazioni significative sulla chiave o sul testo in chiaro anche se conoscono la relazione tra il testo cifrato e la chiave.

2. **Diffusione**: Le operazioni all'interno dell'algoritmo di Feistel assicurano che anche piccoli cambiamenti nel testo in chiaro si propaghino in modo significativo attraverso i vari round, garantendo che piccole modifiche nel testo in chiaro producano cambiamenti significativi nel testo cifrato.

3. **Reversibilità**: L'architettura di Feistel consente di effettuare facilmente l'operazione inversa (*decifratura*) con gli stessi componenti dell'algoritmo di cifratura. Questa proprietà di reversibilità semplifica notevolmente il processo di decifratura senza compromettere la sicurezza del sistema.

4. **Complessità computazionale**: La scelta di una F complessa e sufficientemente caotica rende la crittoanalisi computazionalmente costosa e difficile, richiedendo risorse computazionali significative per eseguire con successo un attacco di crittoanalisi.

Tuttavia, è importante notare che l'efficacia del cifrario di Feistel dipende fortemente anche dalla scelta di una F robusta e ben progettata. Una funzione debole o prevedibile potrebbe compromettere la sicurezza complessiva del sistema, pertanto la scelta di una buona funzione di trasformazione è fondamentale per garantire la sicurezza del cifrario di Feistel.

Capitolo 2

Teoria dei numeri

2.1 Proprietà dei numeri

La teoria dei numeri è una branca fondamentale della matematica che studia le proprietà degli interi e delle loro relazioni. Nel contesto della teoria dei numeri, diversi concetti chiave emergono dall'analisi delle operazioni e degli insiemi numerici.

Operazioni chiuse

Un'operazione si dice chiusa se l'operazione applicata a due numeri naturali restituisce un numero naturale.

$$\mathbb{N} \,\mathsf{op}\, \mathbb{N} \to \mathbb{N}$$

Ad esempio, l'addizione e la moltiplicazione sono operazioni chiuse sugli interi.

Quando un'operazione non è chiusa? La divisione, non produce sempre un numero naturale. Gli insiemi sono inoltre caratterizzati da proprietà che li rendono unici.

Proprietà commutativa

$$a \,\mathsf{op}\, b = b \,\mathsf{op}\, a$$

L'addizione e la moltiplicazione sono esempi di operazioni che soddisfano la proprietà commutativa.

Proprietà associativa

$$\forall a, b, c \in A \qquad a \,\mathsf{op}\, (b \,\mathsf{op}\, c) = (a \,\mathsf{op}\, b) \,\mathsf{op}\, c$$

La proprietà associativa è verificata dall'addizione e dalla moltiplicazione su diversi insiemi numerici.

Elemento neutro

$$a \,\texttt{op}\, e = a$$

quindi

$$\exists e \in A \texttt{ t.c. } \quad \forall a \in A\, a \,\texttt{op}\, e = e \,\texttt{op}\, a = a$$

L'elemento neutro per l'addizione è lo zero, mentre per la moltiplicazione è l'unità.

Elemento inverso

$$\forall a \in A\, \exists a^{-1} \in A \texttt{ t.c. } \quad a \,\texttt{op}\, a^{-1} = a^{-1} \,\texttt{op}\, a = e$$

Alcuni esempi di elementi inversi includono l'opposto di un numero per l'addizione e il reciproco di un numero non nullo per la moltiplicazione.

Ogni volta che definiamo delle strutture matematiche che soddisfano tali proprietà, si parla di gruppi. Un **gruppo** è una struttura algebrica che rispetta determinate regole, tra cui chiusura, associatività, presenza di un elemento neutro e di un elemento inverso.

Prendiamo in considerazione i numeri naturali rispetto all'operazione di somma. Hanno l'inverso? No, quindi non è un gruppo. Tuttavia, se aggiungessimo altri elementi e arrivassimo ai numeri interi, allora avremmo un gruppo rispetto alla somma. Allo stesso modo, se considerassimo i numeri interi rispetto alla moltiplicazione (*senza lo zero*), non avremmo l'inverso. In questo caso, dovremmo aggiungere gli inversi e arrivare ai numeri razionali, che costituiscono un gruppo rispetto alla moltiplicazione.

Capita spesso di avere strutture che non sono gruppi. In questi casi, si possono seguire due approcci: ridurre la struttura eliminando gli elementi che non soddisfano le proprietà del gruppo o espandere la struttura aggiungendo elementi in modo da soddisfare tali proprietà.

Vorremmo lavorare su un'algebra diversa, possibilmente con gruppi finiti. In particolare, ci concentreremo su gruppi che costituiscono l'algebra alla base dei nostri algoritmi crittografici. Utilizzeremo funzioni che sono facili da calcolare ma difficili da invertire, comunemente conosciute come **one-way function**. Queste funzioni si basano sull'algebra moltiplicativa.

Successivamente, esamineremo anche la crittografia ellittica, che si basa su un'algebra diversa, operando su curve ellittiche. Tuttavia, gli algoritmi fondamentali rimarranno gli stessi, utilizzando però un'algebra additiva.

2.2 Classe di equivalenza $\mathbb{Z}_n$

Possiamo dire che:

$$\mathbb{Z}_n \equiv \mathbb{Z}_n \qquad a \equiv b \pmod n \iff (a \mod n) = (b \mod n)$$

Quindi due numeri sono equivalenti se hanno lo stesso resto nella divisione per n. Ad esempio, $5 \equiv 11 \pmod 3$, perché entrambi hanno resto 2 nella divisione per 3.

Quando abbiamo una relazione di equivalenza possiamo costruire l'insieme degli oggetti equivalenti tra loro. L'insieme degli insiemi di oggetti equivalenti forma una **partizione**, gli elementi di tale partizione sono detti **classi di equivalenza** e si denotano mediante parentesi quadre di un elemento di tale classe. Ogni singola classe di equivalenza può essere rappresentata da un qualsiasi elemento della classe stessa.

$$\mathbb{Z}_n = \{0, 1, 2, 3, \ldots, n-1\}$$

Un'analogia può essere fatta con le ore. Se sono le 10:00, allora sono anche le 22:00, perché entrambe sono equivalenti a 10:00 $\pmod{12}$, o con le frazioni. È l'insieme delle classi di equivalenza tra numeri naturali dove due coppie sono equivalenti se hanno lo stesso prodotto incrociato, quindi il prodotto dei medi è uguale al prodotto degli estremi. Ad esempio, $2 \times 3 = 1 \times 6$.

$$\frac{1}{2} \equiv \frac{3}{6}$$

La rappresentazione canonica di una classe di equivalenza in $\mathbb{Z}_n$ è il suo rappresentante minimo, ovvero un numero naturale compreso tra 0 e $n-1$.

Su tale insieme voglio definire delle operazioni.

2.2.1 Somma

La somma di due classi di equivalenza è definita come la classe di equivalenza della somma dei rappresentanti. Ad esempio, se voglio calcolare $[2]+[3]$, allora calcolo $2+3=5$ e la classe di equivalenza di 5 è $[5]$. In generale, la somma di due classi di equivalenza è la classe di equivalenza della somma dei rappresentanti.

$$[a] + [b] \triangleq [a+b]$$

Tale rappresentazione è buona solo se il risultato è indipendente dagli elementi delle due classi originali.

Proprietà

- **Commutativa**: $\forall a, b\, [a] + [b] = [b] + [a]$.
- **Associativa**: $\forall a, b, c\, [a] + ([b] + [c]) = ([a] + [b]) + [c]$.
- **Elemento neutro**: $\exists e \in \mathbb{Z}_n \, \forall a\, [a] + [a] = [a] + [a] = [a] \qquad e = 0$.
- **Elemento inverso**: $\forall a\, \exists a^{-1} \in \mathbb{Z}_n$ t.c. $[a] + [a^{-1}] = [a^{-1}] + [a] = [e] \qquad a^{-1} = -a$.

L'insieme $\mathbb{Z}_n$ con l'operazione di somma forma un **gruppo abeliano**, (*abeliano* perché commutativo).

2.2.2 Moltiplicazione

La moltiplicazione di due classi di equivalenza è definita come la classe di equivalenza del prodotto dei rappresentanti. Ad esempio, se voglio calcolare $[2] \cdot [3]$, allora calcolo $2 \cdot 3 = 6$ e la classe di equivalenza di 6 è $[6]$. In generale, la moltiplicazione di due classi di equivalenza è la classe di equivalenza del prodotto dei rappresentanti.

$$[a] \cdot [b] \triangleq [a \cdot b]$$

Tale rappresentazione è buona solo se il risultato è indipendente dagli elementi delle due classi originali.

Proprietà

- **Commutativa**: $\forall a, b \, [a] \cdot [b] = [b] \cdot [a]$.
- **Associativa**: $\forall a, b, c \, [a] \cdot ([b] \cdot [c]) = ([a] \cdot [b]) \cdot [c]$.
- **Elemento neutro**: $\exists e \in \mathbb{Z}_n \, \forall a \, [a] \cdot [a] = [a] \cdot [a] = [a] \qquad e = 1$.

L'insieme $\mathbb{Z}_n$ con l'operazione di moltiplicazione forma un **semigruppo**, ovvero un gruppo senza l'elemento inverso. Per ottenere un gruppo, devo aggiungere l'elemento inverso. Per ottenere tale elemento, quindi passare da un semigruppo ad un gruppo, posso seguire diverse strade; arricchire l'insieme con nuovi elementi, oppure eliminare elementi.

Consideriamo l'insieme $\mathbb{Z}_n - \{0\}$ con l'operazione di moltiplicazione e consideriamo un esempio con $n = 15$.

$$\mathbb{Z}_{15} - \{0\} = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14\}$$

- L'inverso moltiplicativo di 1 è 1.
- L'inverso moltiplicativo di 2 è 8 ($2 \cdot 8 = 16 \equiv 1 \pmod{15}$).
- L'inverso moltiplicativo di 3 non c'è.
- ...

Notiamo che non tutti gli elementi hanno un inverso moltiplicativo, quindi non posso costruire un gruppo. Per ottenere un gruppo, devo eliminare gli elementi che non hanno un inverso moltiplicativo. L'insieme $\mathbb{Z}_{15} - \{0, 3, 5, 6, 9, 10, 12\}$ che sarà quindi:

$$\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$$

Abbiamo quindi definito l'insieme $\mathbb{Z}_n^*$:

$$\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \,|\, \mathtt{mcd}(a, n) = 1\}$$

Ovvero l'insieme dei numeri che sono relativi primi con n. Moltiplicando due numeri relativamente primi con n, ottengo un numero relativamente primo con n. L'operazione di moltiplicazione è chiusa in $\mathbb{Z}_n^*$.

Teorema 2.2.1 **Teorema di Eulero**
Per ogni $a, b\, \exists x, y\; ax + by = \texttt{mcd}(a, b)$

Se $a \in \mathbb{Z}_n^*$ allora $\texttt{mcd}(a, n) = 1$ per definizione e quindi

$$ax + ny = 1$$

$$ax = 1 - ny$$

$$ax \equiv 1 \pmod{n}$$

Quindi la classe di equivalenza di x è l'inverso moltiplicativo di a. La necessità di lavorare con gruppi nasce dal fatto che in informatica è necessario lavorare con insiemi finiti, in questo caso algebre su insiemi finiti, in particolare sui gruppi, in modo da manipolare gli elementi in base alle proprietà.

2.3 Gruppi e generatori

Sia $\mathcal{G}$ un gruppo, un operatore $\otimes$. Sia g un elemento di $\mathcal{G}$.

$$g = g^1 \qquad g \otimes g = g^2 \qquad g \otimes g \otimes g = g^3 \qquad \dots \qquad g \otimes g \otimes g \otimes \cdots \otimes g = g^n$$

Continuando a moltiplicare g per se stesso, non arriverò ad un qualsiasi n generico, poiché il gruppo è finito. Quindi, supponiamo che arrivi a $g \otimes g \otimes g \otimes \cdots \otimes g = g^{|\mathcal{G}|}$ e che l'esponente successivo sia $g^{|\mathcal{G}|+1}$. Per il **pumping lemma** avrò sicuramente almeno un elemento ripetuto.

Teorema 2.3.1 Per ogni gruppo $\mathcal{G}$ finito, per ogni $a \in \mathcal{G}$,

$$a^{|\mathcal{G}|} = 1$$

Da questo teorema segue che se sicuramente $a^{|\mathcal{G}|} = 1$ ovvero l'elemento neutro, ma per un gruppo $\mathcal{G}$ finito, potrei avere anche che per un qualche $a^i = 1$.

Se prendo tutte le potenze di g ottengo un sottogruppo dell'insieme $\mathcal{G}$, un sottogruppo continua ad essere un gruppo. Se quello che ottengo è un sottogruppo proprio, ovvero un sottogruppo che non è tutto l'insieme $\mathcal{G}$, allora G è ciclico e g è un **generatore** di $\mathcal{G}$.

Teorema 2.3.2 Sia $\mathcal{G}'$ è sottogruppo di $\mathcal{G}$, allora $|\mathcal{G}'| \mid |\mathcal{G}|$.

Esempio

$$\mathbb{Z}_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$$

- 1
- $2 \cdot 2 = 4 \cdot 2 = 8 \cdot 2 = 16$ quindi $2^4 \equiv 1 \pmod{15}$
- $4 \cdot 4 = 16 \equiv 1 \pmod{15}$
- $7 \cdot 7 = 49 \equiv 4, 4 \cdot 7 = 28 \equiv 13, 13 \cdot 7 = 91 \equiv 1 \pmod{15}$

- $8 \cdot 8 = 64 \equiv 4, 4 \cdot 8 = 32 \equiv 2, 2 \cdot 8 = 16 \equiv 1 \pmod{15}$
- $11 \cdot 11 = 121 \equiv 1 \pmod{15}$
- $13 \cdot 13 = 169 \equiv 4, 4 \cdot 13 = 52 \equiv 7, 7 \cdot 13 = 91 \equiv 1 \pmod{15}$
- $14 \cdot 14 = 196 \equiv 1 \pmod{15}$

La cardinalità di $\mathbb{Z}_{15}^*$ è 8, ma questo gruppo non è ciclico, poiché non esiste un elemento generatore che generi tutto il gruppo.

2.3.1 Generatori primi

Se prendo $\mathbb{Z}_n^*$ con n primo, allora $\mathbb{Z}_n^*$ è ciclico, poiché ogni elemento di $\mathbb{Z}_n^*$ è un generatore di $\mathbb{Z}_n^*$. In generale un numero primo non può essere scomposto in fattori, quindi:

$$\mathbb{Z}_p^* = \{1, 2, 3, \dots, p - 1\}$$

Esempio

$$\mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$$

- 1
- $2 \cdot 2 = 4 \cdot 2 = 8 \equiv 1 \pmod 7$
- $3 \cdot 3 = 9 \equiv 2, 2 \cdot 3 = 6, 6 \cdot 3 = 18 \equiv 4, 4 \cdot 3 = 12 \equiv 5, 5 \cdot 3 = 15 \equiv 1 \pmod 7$
- $4 \cdot 4 = 16 \equiv 2, 2 \cdot 4 = 8 \equiv 1 \pmod 7$
- $5 \cdot 5 = 25 \equiv 4, 4 \cdot 5 = 20 \equiv 6, 6 \cdot 5 = 30 \equiv 2, 2 \cdot 5 = 10 \equiv 3, 3 \cdot 5 = 15 \equiv 1 \pmod 7$
- $6 \cdot 6 = 36 \equiv 1 \pmod 7$

Abbiamo che $\mathbb{Z}_7^*$ è ciclico, poiché esiste un generatore che genera tutto il gruppo, in questo caso 3 e 5.

2.3.2 Probabilità dei numeri primi

Teorema 2.3.3 **Densità dei numeri primi**
La densità dei numeri primi è proporzionale al numero di bit che compongono il numero.

Suppongo di avere un numero casuale n di k bit, allora la probabilità che n sia primo è $\frac{1}{k}$. Supponiamo di voler comporre un numero n di k bit, utilizzo un algoritmo che mi generi tale numero.

Per verificare che tale numero sia primo, utilizzo una algoritmo casuale che sceglie casualmente un numero a tra 1 e $n - 1$ e verifica che il numero scelto sia primo. Statisticamente circa la metà dei numeri scelti sono testimoni del fatto che un numero non sia primo. Se il test fallisce e mi dice che il numero non è primo, allora termino. Se il test ha esito positivo allora scelgo un altro numero a e ripeto il test di primalità. Se tutte le volte che scelgo un numero a il test ha esito positivo, allora la probabilità di accettare la primalità di n è $\frac{1}{2^t}$, dove t è il numero di volte che ho ripetuto il test (*eventi indipendenti*), abbassando la probabilità di errore notevolmente.

Sulla base di questo ragionamento, siamo in grado di generare numeri primi molto grandi.

2.4 Logaritmo discreto

Supponiamo di avere a disposizione un gruppo $\mathbb{Z}_p^*$ un generatore g e un elemento di tale gruppo $a \in \mathbb{Z}_p^*$, visto che le potenze del generatore enumerano l'intero gruppo, ci sarà una potenza x che mi permette di ottenere a:

$$g^x = a = g^{x+k(p-1)}$$

L'oggetto x è detto **logaritmo discreto** di a in base g. Trovare il logaritmo discreto di un numero è un problema difficile, non possiamo dire che non esistano algoritmi efficienti, ma non ne conosciamo nessuno. Visto che non conosciamo algoritmi efficienti, possiamo utilizzare il logaritmo discreto come funzione di *one-way*. Al crescere della dimensione del gruppo, il problema diventa sempre più difficile, in maniera più che polinomiale.

2.5 Numero quadrato

Se ho un gruppo $\mathcal{G}$ con un'operazione binaria $\otimes$, un elemento $a \in \mathcal{G}$ è detto **quadrato** se e solo se esiste un $x \in \mathcal{G}$ tale che:

$$x \otimes x = a$$

Quindi esiste una radice quadrata di a in $\mathcal{G}$. Gli elementi in $\mathbb{Z}_p^*$ sono $p-1$, ma solo la metà di questi sono quadrati $\frac{p-1}{2}$. Perché un numero sia un quadrato devo trovare un numero del gruppo che elevato al quadrato mi dia il numero dell'insieme, il numero risultante sarà un quadrato.

Esempio

$$\mathbb{Z}_7^* = \{1, 2, 3, 4, 5, 6\}$$

- $1 \cdot 1 = 1$ quindi 1 è un quadrato;
- $2 \cdot 2 = 4$ quindi 4 è un quadrato;
- $3 \cdot 3 = 9 \equiv 2$ quindi 2 è un quadrato;
- $4 \cdot 4 = 16 \equiv 2$ quindi 2 è un quadrato;
- $5 \cdot 5 = 25 \equiv 4$ quindi 4 è un quadrato;
- $6 \cdot 6 = 36 \equiv 1$ quindi 1 è un quadrato;

Gli elementi che non sono quadrati sono quindi $\{3, 5, 6\}$. Quindi esattamente la metà degli elementi del gruppo sono quadrati.

Teorema 2.5.1 Tutti e soli gli elementi g^{2i} sono quadrati in $\mathbb{Z}_p^*$.

Se consideriamo un generatore g di $\mathbb{Z}_p^*$ e generiamo l'intero gruppo, è chiaro che tutti gli elementi che sono potenze pari di g sono quadrati. Di conseguenza tutti gli elementi che sono potenze dispari di g non sono quadrati.

L'elemento g^{2i} ha due radici quadrate, g^{2i} e $g^{2i+\frac{p-1}{2}}$.

$$g^{2i} = \begin{cases} g^{2i} \\ g^{2i+\frac{p-1}{2}} \end{cases}$$

Infatti

$$\left(g^{i+\frac{p-1}{2}}\right) = g^{2i} \cdot g^{p-1} = g^{2i} \cdot 1 = g^{2i}$$

So che le radici quadrate sono anche

$$g^{2i} = \begin{cases} g^{i} \\ g^{-i} \end{cases}$$

Quindi elevare un generatore alla cardinalità del gruppo mi dà 1, ma elevarlo alla metà della cardinalità del gruppo mi dà -1.

Con questa osservazione possiamo costruire un algoritmo che mi permette di distinguere gli elementi che sono quadrati da quelli che non lo sono. Se conosciamo il logaritmo discreto di un numero a in base g. Il fatto che non siamo in grado di calcolare il logaritmo discreto in maniera efficiente non preclude la possibilità che esistano altri algoritmi.

2.6 Simbolo di Legendre

Il simbolo di Legendre

$$\left(\frac{a}{p}\right) \triangleq a^{\frac{p-1}{a}} \pmod p$$

Sia $a = g^{2i}$ ovvero un quadrato, allora

$$a^{\frac{p-1}{2}} = (g^{2i})^{\frac{p-1}{2}} = (g^{i})^{p-1} = 1^{p-1} = 1 \pmod p$$

Sia $a = g^{2i+1}$ ovvero un non quadrato, allora

$$a^{\frac{p-1}{2}} = (g^{2i+1})^{\frac{p-1}{2}} = (g^{2i})^{\frac{p-1}{2}} \cdot g^{\frac{p-1}{2}} = (g^{i})^{p-1} \cdot g^{\frac{p-1}{2}} = 1 \cdot (-1) = -1 \pmod p$$

Quindi applicando il simbolo di Legendre ad un numero a in base g otteniamo

$$\left(\frac{a}{p}\right) = \begin{cases} 1 & \text{se } a \text{ è un quadrato} \\ -1 & \text{se } a \text{ non è un quadrato} \end{cases}$$

Quindi il simbolo di Legendre mi permette di distinguere gli elementi che sono quadrati da quelli che non lo sono.

Sappiamo che nessuno conosce algoritmi efficienti per calcolare il logaritmo discreto, so però dire che un numero è un quadrato o meno osservando l'ultimo bit del numero.

Con il simbolo di Legendre posso calcolare il bit meno significativo di un numero in maniera efficiente, nonostante non siamo capaci di calcolare il logaritmo discreto.

Il fatto che non siamo capaci di calcolare invertire una funzione non implica che non siamo capaci di calcolare qualche bit della funzione. Il nostro obiettivo però è quello di non ricavare informazioni da nulla, neanche un bit. Quindi l'elevamento a potenza in $\mathbb{Z}_p^*$ non potrà essere usato per codificare.

2.6.1 Iterative Squaring

Il simbolo di Legende possiamo calcolarle in maniera polinomiale, ma non in maniera efficiente. Se eleviamo un numero a per b non operiamo con un algoritmo efficiente. In $\mathbb{Z}_p^*$ è possibile calcolare in tempo polinomiale un'esponenziazione a^b con $a, b \in \mathbb{Z}_p^*$.

Rappresentiamo b in base 2:

$$b = \sum_{i=0}^{k} b_i \cdot 2^i$$

Quindi

$$a^b = a^{\sum_{i=0}^{k} b_i \cdot 2^i} = \prod_{i=0}^{k} a^{b_i \cdot 2^i}$$

Con il sistema di numerazione binario possiamo calcolare in maniera efficiente l'esponenziazione, in un numero di moltiplicazioni pari al numero di bit di b.

procedure Iterative Squaring(a, b)
    $y \leftarrow 1$
    $t \leftarrow a$
    while $b \neq 0$ **do**
        if b is odd **then**
            $y \leftarrow y \cdot t$
        end if
        $b \leftarrow b/2$
        $t \leftarrow t \cdot t$
    end while
    return r
end procedure

Se prendiamo però due numeri a 10 cifre e li moltiplichiamo, il risultato sarà un numero nel caso pessimo un numero a 20 cifre. Nel caso pessimo nella moltiplicazione iterativa il risultato finale sarà un numero a 2^k cifre, dove k è il numero di bit di b.

Per contrastare tale problema modifico tale algoritmo in questo modo:

procedure Iterative Squaring(a, b, n)
    $y \leftarrow 1$
    $t \leftarrow a$
    while $b \neq 0$ **do**
        if b is odd **then**

```
            y ← y · t (mod n)
        end if
        b ← b/2
        t ← t · t (mod n)
    end while
    return r
end procedure
```

A questo punto non ho più il problema di avere numeri con crescita esponenziale del risultato, i risultati parziali rimangono sempre nella stessa quantità di bit.

2.6.2 Il gruppo $\mathbb{Z}_n^*$ con $n = p \cdot q$

In questo gruppo è possibile il numero di elementi saranno pari a

$$|\mathbb{Z}_n^*| = |\mathbb{Z}_p^*| \cdot |\mathbb{Z}_q^*| = (p-1) \cdot (q-1)$$

Calcolando quindi gli elementi co-primi con n.

Esiste una funzione $\varphi(n)$ che mi fornisce la cardinalità di $\mathbb{Z}_n^*$, chiama **funzione di Eulero**.

$$\varphi(n) \triangleq |\mathbb{Z}_n^*|$$

Sia x un numero casuale minore di n:

$$\mathbb{P}[x \in \mathbb{Z}_n^*] = \frac{(p-1)(q-1)}{p \cdot q} = \frac{p-1}{p} \cdot \frac{q-1}{q} > \frac{1}{4}$$

Ma se p e q sono grandi, allora $\mathbb{P}[x \in \mathbb{Z}_n^*] \approx 1$.

Ma quanti elementi di $\mathbb{Z}_n^*$ sono quadrati?

$$\frac{p-1}{2} \cdot \frac{q-1}{2} = \frac{\varphi(n)}{4}$$

Un elemento di $\mathbb{Z}_n^*$ è un elemento di $\mathbb{Z}_p^*$ e di $\mathbb{Z}_q^*$, quindi se è un quadrato in $\mathbb{Z}_n^*$ è un quadrato in $\mathbb{Z}_p^*$ e in $\mathbb{Z}_q^*$. Quindi:

- $\frac{1}{4}$ degli elementi di $\mathbb{Z}_n^*$ sono quadrati in $\mathbb{Z}_p^*$ e in $\mathbb{Z}_q^*$.
- $\frac{1}{4}$ degli elementi di $\mathbb{Z}_n^*$ sono quadrati in $\mathbb{Z}_p^*$ e non in $\mathbb{Z}_q^*$.
- $\frac{1}{4}$ degli elementi di $\mathbb{Z}_n^*$ non sono quadrati in $\mathbb{Z}_p^*$ e sono quadrati in $\mathbb{Z}_q^*$.
- $\frac{1}{4}$ degli elementi di $\mathbb{Z}_n^*$ non sono quadrati in $\mathbb{Z}_p^*$ e in $\mathbb{Z}_q^*$.

Per capire se un numero è un quadrato in $\mathbb{Z}_n^*$ in maniera semplice è necessario conoscere la fattorizzazione d n. Basta calcolare il simbolo di Legendre rispetto a p e q e so il risultato. Se non conosco la fattorizzazione di n non conosciamo algoritmi efficienti per stabilire se un numero è un quadrato in $\mathbb{Z}_n^*$ non disponendo della fattorizzazione di n.

Quindi anche la quadraticità di un numero è un problema difficile.

2.7 Simbolo di Jacobi

Il simbolo di Jacobi è una generalizzazione del simbolo di Legendre.

$$\left(\frac{a}{n_1 \cdot n_2}\right) = \left(\frac{a}{n_1}\right) \cdot \left(\frac{a}{n_2}\right)$$

$$\left(\frac{a_1 \cdot a_2}{n}\right) = \left(\frac{a_1}{n}\right) \cdot \left(\frac{a_2}{n_2}\right)$$

Tale simbolo potrà avere valore -1 o 1.

Ad oggi stabilire se un numero è un quadrato in $\mathbb{Z}_n^*$ quando il simbolo di Jacobi è 1 è un problema difficile. Se il simbolo di Jacobi è -1 allora il numero non è un quadrato in $\mathbb{Z}_n^*$ per definizione.

Sia a un elemento di $\mathbb{Z}_n^*$ con $\left(\frac{a}{n}\right) = 1$, vorrei ottenere un oggetto con la stessa quadraticità di a distribuito uniformemente ad oggetti con la stessa quadraticità di a.

2.8 Generazione di numeri casuali con la stessa quadraticità di a

Sia x un elemento casuale di $\mathbb{Z}_n^*$, allora:

$$x \in_R \mathbb{Z}_n^* \Rightarrow x^2 \cdot a$$

ha la stessa quadraticità di a e sarà distribuito uniformemente ad oggetti con la stessa quadraticità di a.

Quadrati diversi vengono mappati con la moltiplicazione di a in quadrati diversi e quindi la distribuzione è uniforme. Perché ho una funzione iniettiva e suriettiva (una biezione), quindi la distribuzione del risultato sarà la stessa della sorgente.

In questo modo se non conosciamo la natura casuale degli oggetti possiamo generare oggetti con la stessa natura, ma casuali (*con quadraticità*).

2.9 Turing riduzione

Lemma 2.9.1 *Siano x e y due radici quadrate di uno stesso quadrato di $\mathbb{Z}_n^*$, tali che $x \not\equiv \pm y$. Allora il* `mcd` *tra $x + y$ e n è un fattore di n.*

Dimostrazione. Sia $n = p \cdot q$, visto che x e y sono radici quadrate di uno stesso numero sappiamo che $x^2 \equiv y^2 \mod n$. Quindi:

$$x^2 \equiv y^2 \mod n \Rightarrow x^2 - y^2 \equiv 0 \mod n$$

$$\Rightarrow (x - y)(x + y) = k \cdot n \qquad \text{per qualche } k$$

Supponiamo che p divida $x+y$, ma è possibile che q divida $x-y$? Se fosse possibile allora n divide $x+y$, ma ciò vorrebbe dire che $x+y \equiv 0 \mod n$, ovvero $x \equiv -y \mod n$ e ciò è assurdo perché $x \not\equiv \pm y$. Quindi $\mathtt{mcd}(x+y,n)=p$.

Supponiamo che q divida $x+y$, ma è possibile che p divida $x-y$? Se fosse possibile allora n divide $x+y$, ma ciò vorrebbe dire che $x+y \equiv 0 \mod n$, ovvero $x \equiv -y \mod n$ e ciò è assurdo perché $x \not\equiv \pm y$. Quindi $\mathtt{mcd}(x+y,n)=q$.

Supponiamo che p non divida $x+y$ e che q non divida $x+y$, ma p e q sono fattori del prodotto $(x-y)(x+y)$, quindi p e q devono essere fattori di almeno uno dei due fattori. Se p non divide $x+y$ allora q deve dividere $x+y$, allora p e q sono fattori di $x-y$, ma allora $x \equiv y \mod n$ e ciò è assurdo.

□

Se abbiamo due radici quadrate distinte allora riusciamo a trovare la fattorizzazione di n. Il problema ora è come faccio a trovare due radici distinte di un numero se mi viene fornito l'algoritmo per il calcolo della radice quadrata? Supponiamo che esista un algoritmo $A \in \mathtt{PPT}$ (*probabilistic polynomial time*) che calcoli la radice quadrata di un numero in $\mathbb{Z}_n^*$. Allora:

procedure FACTORIAL(n)
    $x \in_R \mathbb{Z}_n^*$
    $y \leftarrow A(x^2)$
    $z \leftarrow \mathtt{mcd}(x-y,n)$
    if $z \neq n$ **then return** z
    else
      return `Factorial`(n)
    end if
end procedure

Prendo un quadrato a caso, di questo quadrato conosco una radice scelta uniformemente tra le quattro possibili. L'algoritmo A mi restituisce una radice quadrata di x^2, l'algoritmo sceglierà la radice quadrata in qualche modo, sicuramente indipendente dalla scelta fatta su x. La probabilità che la radice scelta sia x o l'opposto di x è $\frac{1}{2}$, quindi il test che verifica se z è un fattore di n avrà successo con probabilità $\frac{1}{2}$, quindi ripeto l'algoritmo A un numero costante di volte poiché la probabilità di successo è costante. Quindi l'algoritmo è polinomiale.

Il numero di esperimenti da eseguire per aver successo è data dalla distribuzione geometrica, e il valore atteso è il reciproco della ragione di successo, quindi in media devo eseguire due volte l'algoritmo A per avere successo.

Ciò ci porta a dire che calcolare la radice quadrata è verosimilmente difficile, perché se qualcuno ci riuscisse allora potremmo fattorizzare in tempo polinomiale.

L'idea di dimostrare la sicurezza di un crittosistema è quella di dimostrare che esista un algoritmo che utilizzi come sottoprocedura un algoritmo che risolve un problema difficile. I problemi difficili sono quelli che non si riescono a risolvere in tempo polinomiale, quindi il calcolo della radice quadrata, il logaritmo discreto e la fattorizzazione di numeri primi.

Capitolo 3

Crittografia a chiave pubblica

3.1 Crittografia a chiave pubblica

L'idea di crittografia a chiave pubblica è quella di avere due chiavi, una pubblica P_k e una privata S_k. La chiave pubblica è nota a tutti, mentre quella privata è nota solo al proprietario. Con codifica avviene con la chiave pubblica, mentre la decodifica avviene con la chiave privata.

Ovviamente l'idea di fondo è avendo in mano il testo cifrato non si riesce a risalire al testo in chiaro senza la chiave privata. Chiunque può cifrare un messaggio, ma solo il proprietario della chiave privata può decifrarlo.

In un sistema a chiave pubblica abbiamo i seguenti algoritmi:

- Un algoritmo di generazione delle chiavi:

$$\mathcal{G} : 1^k \to (P_k, S_k)$$

  Dove 1^k è un parametro che indica la lunghezza della chiave, ovvero il *security parameter*.

- Un algoritmo di encription:

$$\mathcal{E} : m, P_k \to \mathcal{E}(m, P_k)$$

- Un algoritmo di decription:

$$\mathcal{D} : C, S_k \to D(C, P_k)$$

Ovviamente vale la seguente relazione:

$$\forall m \quad \mathcal{D}(\mathcal{E}(m, P_k), S_k) = m \tag{3.1}$$

Oltre al fatto che decriptare un messaggio a partire dal testo cifrato senza la chiave privata è computazionalmente intrattabile. L'idea è che più la chiave è lunga più è difficile rompere il sistema. L'idea è che il security parameter k è proporzionale alla lunghezza della chiave e al crescere di k cresce la sicurezza del sistema.

Gli algoritmi citati sono algoritmi probabilistici polinomiali. Se voglio affermare che tali algoritmi sono polinomiali l'input che rappresenta la lunghezza della chiave non può essere rappresentato in binario, perché la lunghezza della chiave sarebbe esponenziale nel numero di bit utilizzati per rappresentare il numero. Sulle macchine di Turing la dimensione del problema è data dal numero di celle del nastro di input. Utilizzando la teoria della complessità basata su tali macchine, o in ogni caso su sistemi dove la dimensione dell'input è lo spazio che occupa nella nostra rappresentazione. Per voler dire che un algoritmo è polinomiale nel **valore del security parameter** e non nel modo in cui è rappresentato, imponiamo che il security parameter sia rappresentato in **unario**, ovvero tanti uni quanti la lunghezza della chiave.

3.2 Diffie-Hellman

Il problema di Diffie-Hellman è il seguente: Alice e Bob vogliono scambiarsi un segreto senza che Eve lo possa intercettare. Lo strumento utilizzato per risolvere il problema è del logaritmo discreto.

Si fissa a priori un numero primo p e un generatore g di $\mathbb{Z}_p^*$. Un agente A sceglie un numero $x \in_R \{1, \dots, p-1\}$ e calcola $g^x \mod p$. Un agente B sceglie un numero $y \in_R \{1, \dots, p-1\}$ e calcola $g^y \mod p$.

	Public	**Private**
A	$g^x \mod p$	x
B	$g^y \mod p$	y

A questo punto A e B possono calcolare $g^{xy} \mod p$ e $g^{yx} \mod p$, che coincidono.

Il sistema è sicuro perché calcolare $g^{xy} \mod p$ è computazionalmente intrattabile. Avendo a disposizione g^x e g^y non è possibile calcolare g^{xy}. Se sappiamo rispondere al problema del logaritmo discreto, allora possiamo risolvere il problema di Diffie-Hellman, tale algoritmo potrebbe esistere, ma non è stato ancora trovato.

Sia $\mathcal{A} \in \texttt{PPT}$ che calcola $g^{xy} \mod p$ a partire da $g^x \mod p$ e $g^y \mod p$. Usiamo $\mathcal{A}$ per costruire un algoritmo $\mathcal{B}$ che risolve il problema del logaritmo discreto, ma tale dimostrazione non è ancora stata data, perciò non l'algoritmo di Diffie-Hellman non è dimostrabilmente sicuro. non siamo in grado di dire che rompere Diffie-Hellman è almeno difficile quanto risolvere il problema del logaritmo discreto.

3.2.1 Ipotesi di Diffie-Hellman

Qualcuno potrebbe costruire un algoritmo che si basa sull'ipotesi che l'algoritmo di Diffie-Hellman sia sicuro.

Ipotesi di Diffie-Hellman

Siano x, y, z dei numeri causali scelti in $\{1, \ldots, p-1\}$, allora è difficile distinguere (g^x, g^y, g^{xy}) da (g^x, g^y, g^z).

Il concetto di distinguibilità è un concetto probabilistico, ovvero che la possibilità di poter distinguere due insiemi di elementi è trascurabile. Ovvero che la probabilità di distinguere sia inferiore a $1/2 + \epsilon$, dove ϵ è trascurabile. Quindi l'attaccante non abbia alcun vantaggio rispetto ad un attaccante che non ha alcuna informazione. Se un attaccante avesse anche un minimo vantaggio, allora potrebbe utilizzare tale vantaggio per ottenere informazioni sul segreto. Tale vantaggio può essere utilizzato per ottenere l'informazione totale mediante esperimenti ripetuti.

3.3 Rivest Shamir Adleman - RSA

g^x è una funzione che è facile da calcolare, ma è difficile da invertire, ovvero calcolare x a partire da g^x. Una funzione con questa proprietà è detta **one-way function**. Il protocollo di Diffie-Hellman è sicuro se e solo se esiste una one-way function. Vorremmo che la one-way function sia anche **trapdoor**, ovvero che esista un algoritmo efficiente che permetta di invertire la funzione. Tale algoritmo è detto **trapdoor algorithm**. La trapdoor è una informazione aggiuntiva che permette di invertire la funzione.

Siano p, q due numeri primi molto grandi, $n = pq$. Sia e un numero casuale tale che $\texttt{mcd}(e, \varphi(n)) = 1$, dove $\varphi(n) = (p-1)(q-1)$, ovvero il numero di Eulero. Scegliamo une elemento d co-primo con $\varphi(n)$ tale che $de \equiv 1 \mod \varphi(n)$. La chiave pubblica è la coppia (n, e), mentre la chiave privata è la coppia (n, d).

Algoritmo di cifratura

$$\mathcal{E} : m, (n, e) \mapsto m^e \mod n$$

Algoritmo di decifratura

$$\mathcal{D} : c, (n, d) \mapsto c^d \mod n$$

3.3.1 Funzionamento

Proviamo a prendere un messaggio m e a cifrarlo con la chiave pubblica (n, e), e proviamo a decodificarlo con la chiave privata (n, d).

$$(m^e)^d = m^{ed} = m^{k\varphi(n)+1} = m \cdot (m^{\varphi(n)})^k \equiv m \mod n$$

Infatti $d \cdot e$ è congruo a 1 modulo $\varphi(n)$, quindi $d \cdot e = k\varphi(n) + 1$. Per il teorema del resto cinese, $m^{k\varphi(n)+1} \equiv m$, e non solo per gli elementi di $\mathbb{Z}_n^*$.

La funzione one way è la funzione di codifica, quindi m^e, l'inversa di m^e è la radice e-esima, ovvero $m = c^{1/e}$, ma per calcolare la radice e-esima ad oggi non esiste un algoritmo efficiente.

L'unico modo per calcolare la radice e-esima è calcolare d e decifrare il messaggio, quindi d è l'informazione trapdoor.

Ci piacerebbe dimostrare che se fossimo in grado di invertire la radice e-esima di un numero allora saremmo in grado di fattorizzare n, o di calcolare il residuo quadratico. Se fosse vero, allora RSA sarebbe sicuro, ma non è stato ancora dimostrato ad oggi. L'unica sicurezza di RSA è l'esistenza dell'ipotesi di RSA.

3.3.2 Attacchi a RSA

Ci sono casi in cui RSA è stato attaccato, il motivo però era legato alla cattiva implementazione dell'algoritmo, e non all'algoritmo in sé. Quando parliamo di un crittosistema in realtà parliamo di un insieme di algoritmi, e non di un singolo algoritmo. L'algoritmo di generazione delle chiave impone la scelta **casuale** di e in modo uniforme con gli elementi primi con $\varphi(n)$, ma se non fosse casuale, allora potremmo avere dei problemi.

Attacco sulla base di messaggi piccoli

Inoltre, in caso di messaggi piccoli, quindi se $m^e < n$, vuol dire che non applico nemmeno l'operazione di modulo, e vuol dire in particolare che la radice e-esima di m^e è una normale radice e-esima nell'aritmetica dei numeri interi, e quindi è facile da calcolare. Lavorando con messaggi che a livello numerico sono piccoli, allora invertiamo tutto facilmente, più è piccola e, più è facile avere messaggi che elevati a e sono più piccoli di n. Bisogna quindi star attenti a scenari in cui $m^e < n$.

Attacco sulla base di messaggi sparsi

Altri problemi che potrebbe avere RSA sono legati allo spargimento dei messaggi. Immaginiamo di avere un messaggio:

```
Buongiorno, il suo voto è 30L
Buongiorno, il suo voto è 30
Buongiorno, il suo voto è 29
...
Buongiorno, il suo voto è 0
```

Uno che vuole decifrare il messaggio può prendere i 32 messaggi e cifrarli tutti, per poi distinguerli. Se con RSA devo codificare messaggi che sono presi da un insieme piccolo, devo far attenzione perché potrei venir attaccato da qualcuno che utilizza la stessa chiave pubblica.

Attacco sulla informazione parziale

Siamo sicuri che tutti i bit di questa inversa siano difficili da calcolare? Non è che sulla radice e-esima di m^e ci sono dei bit che sono più facili da calcolare? Ai fini di dire che il sistema è sicuro, non è sufficiente dire che il cypertext non si sappia ricavare il plaintext, vorremmo dire che dal cypertext non si riesca a ricavare nessuna informazione binaria sul plaintext. Ma come possiamo definire tale proprietà?

Attacco sulla base di messaggi ripetuti

Per difendersi da tale problema, aggiungo un po' di rumore al messaggio, ovvero aggiungo un po' di bit casuali al messaggio, in modo tale che con alta probabilità il messaggio non sia mai uguale. In questo modo, anche se il messaggio è sempre lo stesso, il cypertext è sempre diverso, utilizzando quindi la probabilistic encryption.

3.4 Sicurezza di un crittosistema

RSA è sicuro perché non conosciamo un algoritmo probabilistico polinomiale per calcolare la radice e-esima di un numero. Ma cosa vuol dire? Se esistesse un algoritmo probabilistico polinomiale per calcolare la radice e-esima di un numero con probabilità $\frac{1}{k}$, sarebbe un problema, perché reiterando l'algoritmo k volte, avrei una probabilità di successo.

Se fossimo nello scenario in cui con un $a \in_R \mathbb{Z}_n^*$, l'algoritmo mi dia risposta corretta con probabilità $\frac{1}{k}$, potremmo dichiararci tranquilli? No, perché potrebbero attaccare sempre.

Fissando a e scegliendo $r \in_R \mathbb{Z}_n^*$, calcolo $(a \cdot r)^e \mod n$, se prendo un elemento casuale di $\mathbb{Z}_n^*$ e lo elevo ad e, ottengo l'oggetto che è distribuito uniformemente in $\mathbb{Z}_n^*$. Sappiamo che l'elevamento di r alla e-esima è distribuito uniformemente in $\mathbb{Z}_n^*$, perché e è stata scelta in maniera tale che la radice e-esima dia esattamente r. Quindi r^e è una funzione invertibile, di conseguenza la funzione che mappa r in r^e è una funzione biettiva, quindi un elemento scelto uniformemente in $\mathbb{Z}_n^*$ viene mappato da r^e in un elemento scelto uniformemente scelto in $\mathbb{Z}_n^*$. Ogni volta che prendiamo un elemento e creiamo una suriezione dello stesso insieme, se l'elemento di partenza è scelto uniformemente, il risultato della suriezione, che nella sostanza è una permutazione, è scelto uniformemente.

Se prendiamo r^e e lo moltiplichiamo per a, otteniamo un elemento che è distribuito uniformemente in $\mathbb{Z}_n^*$, perché a è scelto uniformemente in $\mathbb{Z}_n^*$, poiché la moltiplicazione per a è una funzione biettiva, perché a ammette inverso.

Siamo partiti da un elemento fissato e abbiamo costruito un elemento distribuito uniformemente e causale, se a quell'elemento applichiamo l'algoritmo della radice e-esima, otteniamo $\frac{1}{k}$ di probabilità di successo, ma se ripetiamo l'algoritmo k volte, abbiamo una probabilità di successo di 1, rendendo quindi l'algoritmo indipendente dalla a di partenza.

$$ar^e \rightarrow \sqrt[e]{ar^e} = r\sqrt[e]{a}$$

Quindi se prendo il risultato e lo divido per r, ottengo $\sqrt[e]{a}$, che è distribuito uniformemente in $\mathbb{Z}_n^*$, perché r è distribuito uniformemente.

Ed ecco che abbiamo un algoritmo che a partire da una blackbox che con a causale calcola correttamente la radice e-esima di a una volta su k, abbiamo una macchina che con a fissato calcola la radice e-esima di a con probabilità 1.

La macchina che calcola la radice e-esima di a darà una sequenza di bit, che potrebbe essere la radice e-esima di a, oppure no. Bisognerebbe riconoscere la risposta corretta, rielevando il risultato ad e, se ottengo l'input allora la risposta è corretta, altrimenti no.

Quindi abbiamo trasformato un algoritmo che funziona una volta su k in un algoritmo che funziona in un tempo medio di k.

Definizione di sicurezza

Diciamo che un sistema è attaccabile se il tempo medio per attaccarlo è polinomiale.

Se esistesse un qualsiasi algoritmo in grado di attaccare la radice e-esima di a, con una probabilità polinomiale in k, riusciamo a costruire un algoritmo che calcola la stessa cosa con un tempo medio polinomiale in k.

Visto che partiamo dall'idea che non esista un algoritmo probabilistico polinomiale in grado di calcolare la radice e-esima di a, allora non esiste un algoritmo che sia in grado di calcolarlo con una probabilità che sia polinomialmente piccola. Quindi la **probabilità di successo è più piccola di qualsiasi polinomio**, dove per polinomio si intende:

$$\mathbb{P}[\texttt{attacco}] < \frac{1}{k} \quad \forall c$$

Fissando un polinomio, con chiavi corte, però, la possibilità di trovare un polinomio esiste, perciò bisogna correggere tale definizione.

$$\forall c \exists \bar{k} \forall k > \bar{k} \quad \mathbb{P}[\texttt{attacco}] < k^{-c} \tag{3.2}$$

Per attacco non intendiamo solo il fatto di non poter essere in grado di poter calcolare la radice e-esima di a, ma anche il fatto di non essere in grado di capire **informazioni binarie**.

L'algoritmo che calcola la radice e-esima di a è l'algoritmo che calcola la la fattorizzazione di n, la fattorizzazione di n è l'informazione binaria che vogliamo proteggere, ovvero la **trapdoor** che permette di risolvere il problema.

Fattorizzazione di n

Si pensa che non esista un algoritmo `PPT` che dati n, e e a, calcola $\sqrt[e]{a} \in \mathbb{Z}_n^*$ con probabilità polinomiale.

3.4.1 Utilizzo pratico di RSA

Nel caso pratico il costo computazionale di RSA è molto alto, infatti codificare un blocco di k bit con RSA richiede k esponenziazioni modulari, ovvero k^3, un costo computazionale molto alto. Per questo motivo RSA viene utilizzato per codificare una chiave di sessione, che viene utilizzata per codificare il messaggio con un algoritmo simmetrico, che è molto più veloce di RSA. Tipicamente l'algoritmo utilizzato è l'algoritmo simmetrico `AES` (1.6).

Tra l'altro vi è una notevole differenza con Diffie-Hellman, infatti in Diffie-Hellman riesco a scambiarmi un'unica chiave, a meno che non faccia un nuovo scambio di chiavi, ogni volta.

Se si parte dall'idea che la crittografia simmetrica sia meno sicura della crittografia asimmetrica, allora si può pensare di utilizzare una chiave di sessione diversa dopo un certo periodo.

Con RSA è possibile fare questo, perché è possibile scambiarsi chiavi diverse, mentre con Diffie-Hellman non è possibile, perché si dovrebbero scambiare chiavi diverse ogni volta, e questo è molto costoso. La generazione della chiave di sessione per Diffie-Hellman è molto costosa, per via delle Certification Authority, che devono essere coinvolte nel processo di generazione della chiave di sessione per certificare le chiavi pubbliche.

3.5 Crittosistema di Micali per la codifica di un singolo bit

Algoritmo di generazione delle chiavi

Si sceglie un numero primo p_1 e un numero p_2 tale che moltiplicati tra loro diano un numero n tale che $n = p_1 p_2$. I due numeri devono essere scelti in modo casuale con $\frac{k}{2}$ bit ciascuno, dove k è la lunghezza della chiave. Sia $y \in_R$ ai non quadrati con simbolo di Jacobi 1 modulo n. Ricordiamo che per costruire un numero non quadrato causale con simbolo di Jacobi 1 basta scegliere un numero casuale e verificare che il simbolo di Jacobi sia 1, ovvero che appartenga a $\mathbb{Z}_n^*$, se non lo è si sceglie un altro numero casuale e si ripete il procedimento. A questo punto verifico che il simbolo di Legendre rispetto a p_1 e q_1 sia -1.

La chiave pubblica è:

$$P_k = (n, y)$$

La chiave privata è:

$$S_k = (p_1, p_2)$$

L'ipotesi di base è che sia difficile fattorizzare n, ma il problema di riferimento sarà il problema del residuo quadratico, ovvero il problema di calcolare la radice quadrata di un numero modulo n.

Algoritmo di codifica

L'algoritmo di codifica prende un bit b, sia $x \in_R \mathbb{Z}_n^*$, se b è 0 allora $c = x^2 \mod n$, altrimenti $c = xy^2 \mod n$. x^2 è un quadrato casuale di $\mathbb{Z}_n^*$, mentre xy^2 è un non quadrato con simbolo di Jacobi 1. Se prendo un quadrato con simbolo di Jacobi 1 e lo moltiplico per un non quadrato con simbolo di Jacobi 1 ottengo un non quadrato con simbolo di Jacobi 1. Se il quadrato è casuale, allora ottengo un non quadrato casuale con simbolo di Jacobi 1 distribuito uniformemente tra tutti i non quadrati con simbolo di Jacobi 1.

Il risultato è che la codifica di 0 è un quadrato a caso, mentre la codifica di 1 è un non quadrato a caso con simbolo di Jacobi 1.

$$\mathcal{E} : \{0, 1\} \to x \in_R \mathbb{Z}_n^*$$

$$f(x) = \begin{cases} x^2 \mod n & \text{se } b = 0 \\ xy^2 \mod n & \text{se } b = 1 \end{cases}$$

Algoritmo di decodifica

L'algoritmo di verifica prende in input c e verifica se c è un quadrato rispetto a p_1 e p_2, ovvero se c è un residuo quadratico modulo p_1 e modulo p_2. Se c è un quadrato rispetto a p_1 e p_2

allora $b = 0$, se entrambe le verifiche falliscono allora $b = 1$, in altri casi non siamo in presenza di un cypertext valido.

$$\left(\frac{c}{p_1}\right) = \left(\frac{c}{p_2}\right) = 1 \qquad \text{allora } b = 0$$

$$\left(\frac{c}{p_1}\right) = \left(\frac{c}{p_2}\right) = -1 \qquad \text{allora } b = 1$$

3.5.1 Rompere il crittosistema di Micali

Rompere tale protocollo significherebbe disporre di un algoritmo $\mathcal{A}$ che preso in input in cyphertext c e la chiave pubblica P_k restituisce b con probabilità diversa da $\frac{1}{2}$, poiché siamo in un contesto binario.

Un attaccante quindi dovrebbe essere in grado di ottenere un vantaggio rispetto a qualcuno che non conosce nulla, ovvero che indovini con probabilità $\frac{1}{2}$. Il vantaggio consiste nell'allontanarsi da $\frac{1}{2}$, sia in positivo che in negativo, poiché se si allontana in negativo basta invertire il risultato per ottenere un vantaggio positivo.

Il numero di esperimenti deve essere tale che la differenza delle probabilità sia maggiore di $\frac{1}{2}$, ma il numero di esperimenti deve essere un numero polinomiale, in modo tale da poter osservare tali esperimenti in tempo polinomiale.

Sicurezza

$$\forall c\, \exists \bar{k}\, \forall k \geq \bar{k} \qquad |\, \mathbb{P}[\texttt{Successo}] - \frac{1}{2}\, | > k^{-c} \tag{3.3}$$

Supponiamo che la probabilità di successo sia maggiore di $\frac{1}{2} + \epsilon$ e vorrei che la probabilità di successo sia quindi prossima a 1. Per farlo eseguo due tipologie di esperimenti, il primo ripete l'esperimento k volte e mediante l'algoritmo che ha a disposizione il vantaggio è $\frac{1}{2}+\epsilon$, mentre il secondo esperimento ripete l'esperimento k volte e mediante esperimenti casuali, ovvero senza l'algoritmo, ottiene un vantaggio di $\frac{1}{2}$. Ripetendo l'esperimento un numero abbastanza grande di volte si ottiene il risultato desiderato, poiché basterebbe visualizzare le due distribuzioni per vedere in cosa differiscono. L'algoritmo quindi indovina con probabilità 1. Più ϵ è piccolo, più esperimenti sono necessari per ottenere il risultato desiderato. Servirebbe quindi stimare, dato un ϵ fissato, il numero di esperimenti necessari per ottenere il risultato desiderato.

Limite di Chernoff

Limite di Chernoff

Siano $X_1, \dots, X_n$ variabili casuali e binarie indipendenti con probabilità di successo $\mathbb{P}[X_i = 1] > \frac{1}{2}$ e $\mathbb{P}[X_i = 0] = 1 - \mathbb{P}[X_i = 1]$. La probabilità che più della metà delle variabili casuali siano 1 è:

$$\mathcal{P} = \sum_{i=\frac{n}{2}+1}^{n} \binom{n}{i} \mathbb{P}[X_i = 1]^i \mathbb{P}[X_i = 0]^{n-i}$$

$$\mathcal{P} \geq 1 - e^{-2n\left(p-\frac{1}{2}\right)^2}$$

Tale formula dice che la probabilità che più della metà degli eventi dia 1 è esponenzialmente vicina a 1, dove l'eponenzialmente è in funzione di n.

$$\mathbb{P}[\texttt{errore}] = e^{-2\epsilon n}$$

Dove ϵ è il vantaggio.

Supponiamo di volere $e^{-2\epsilon n} < \frac{1}{2^k}$, quindi:

$$e^{-2c\epsilon^2 n} < 2^{-k}$$

$$-2\epsilon^2 n < c' - k$$

$$n > \frac{c' - k}{2\epsilon^2}$$

Se ϵ è polinomiale in k allora n è polinomiale in k. Di conseguenza, se il vantaggio è polinomiale in qualche security parameter, allora si riesce ad ottenere una quantità di errore nel security parameter che è esponenzialmente piccola, scegliendo una quantità di esperimenti polinomiale in esso.

Un sistema è attaccato nel momento in cui esiste un algoritmo polinomiale in grado di romperlo.

Nel momento in cui ϵ è un k^{-c}, allora n (*dove n è il numero di esperimenti*) è polinomiale in k.

Visto che l'ipotesi di partenza è che non esistano algoritmi probabilistici polinomiali in grado di rompere il sistema (*vero*) e visto che abbiamo dimostrato che esiste tale algoritmo (*falso*), allora l'algoritmo di Micali è sicuro.

Costruzione degli esperimenti indipendenti

Una volta capito che la costruzione di n esperimenti indipendenti funziona, bisogna capire come costruirli. Supponiamo che la probabilità di successo sia $\frac{1}{2} + \epsilon$ e di disporre di un algoritmo $\mathcal{A}$ che prende in input un numero z con $\left(\frac{z}{n}\right) = 1$, in output restituisce che z è quadrato oppure no.

Per farlo si sceglie $r_1, \ldots, r_n \in_R \mathbb{Z}_n^*$ e si calcola $w_i = z \cdot r_i^2$. L'algoritmo quindi prende in input w_i e restituisce b_i.

$$b_i = \mathcal{A}(w_i)$$

In sostanza si prende in input un numero (*cyphertext*) e l'algoritmo lo moltiplica per una quadrato a caso, quindi l'algoritmo restituisce 1 se il risultato è un quadrato casuale e 0 altrimenti.

Tale algoritmo però ha un difetto, ovvero che se z è un quadrato, allora w_i è un quadrato sempre, quindi l'algoritmo restituisce sempre 1, se invece z non è un quadrato, allora w_i è un quadrato con probabilità $\frac{1}{2} + \epsilon$.

Costruzione di un controesempio

Supponiamo che $\mathcal{A}$ dica correttamente che il 40% dei numeri quadrati sono quadrati e che il 62% dei non quadrati sono non quadrati. Sostanzialmente l'algoritmo $\mathcal{A}$ ha una probabilità di successo a seconda dell'input che gli viene dato.

$$\mathbb{P}[\mathcal{A}(\mathtt{n})\ \ \mathtt{successo}] = \frac{1}{2} \cdot \frac{40}{100} + \frac{1}{2} \cdot \frac{62}{100} = \frac{51}{100} = 51\%$$

L'approccio di costruzione degli esperimenti indipendenti non è corretto, perché non fornisce all'algoritmo $\mathcal{A}$ un input secondo la misura di probabilità che $\mathcal{A}$ si aspetta, quando diciamo che ha una certa probabilità di successo. L'esperimento corretto avviene solamente quando ad $\mathcal{A}$ viene dato un input un oggetto che sia distribuito uniformemente tra gli oggetti con simbolo di Jacobi 1.

Costruzione degli esperimenti indipendenti corretta

L'idea di base è quella di lanciare una moneta per decidere se invertire o meno la quadraticità di z in modo da ottenere un input che sia distribuito uniformemente. Siano $r_1, \ldots, r_n \in_R \mathbb{Z}_n^*$ e siano $s_1, \ldots, s_n \in_R \{0, 1\}$.

$$\forall i \quad w_i = \begin{cases} z \cdot r_i^2 & \text{se } s_i = 0 \\ y \cdot z \cdot r_i^2 & \text{se } s_i = 1 \end{cases}$$

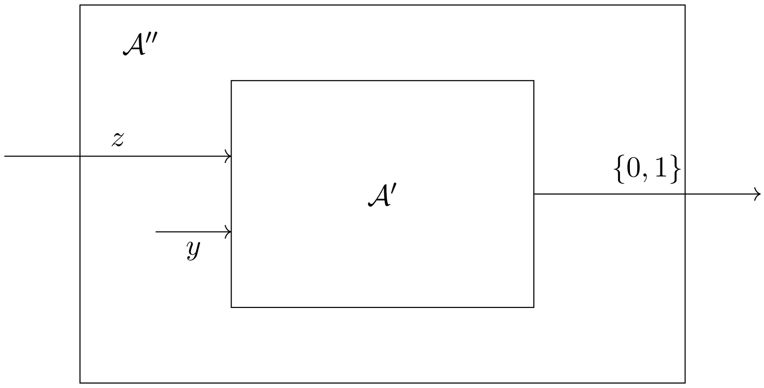
Sia $b_i = \mathcal{A}(w_i)$. In questo modo lasciamo una moneta s_i che decide se mantenere la quadraticità di z oppure no. Quindi $z \cdot r_i^2$ è un oggetto a caso tra gli oggetti con stessa quadraticità di z, mentre $y \cdot z \cdot r_i^2$ è un oggetto a caso tra gli oggetti con quadraticità opposta di z, di conseguenza w_i è un oggetto a caso distribuito uniformemente tra gli oggetti con simbolo di Jacobi 1. In questo caso quindi $\mathcal{A}$ ha una probabilità di successo di $\frac{1}{2} + \epsilon$, ma b_i è la risposta al problema trasformato, ma non è la stessa del problema originale.

$$\forall i \qquad b_i' = \begin{cases} b_i & \text{se } s_i = 0 \\ \bar{b_i} & \text{se } s_i = 1 \end{cases}$$

perché se $s_i = 1$ allora nell'input ho invertito la quadraticità di z, di conseguenza la risposta deve essere a sua volta invertita per avere una risposta corretta nei confronti di z.

Tale costruzione funziona, poiché è possibile passare dall'algoritmo $\mathcal{A}'$ all'algoritmo $\mathcal{A}$, semplicemente invertendo la risposta quando $s_i = 1$, ma non sempre ciò è attuabile, perché non sempre è possibile invertire la risposta.

L'algoritmo che calcola il residuo quadratico prende in input z e restituisce 1 se z è un quadrato e 0 altrimenti, ma a tale algoritmo abbiamo dato in input y, ovvero un non quadrato con simbolo di Jacobi 1. Ma siamo davvero capaci di costruire un algoritmo che calcola un non quadrato con simbolo di Jacobi 1 senza usare la fattorizzazione di n? La risposta è no, perché se fosse possibile allora sarebbe possibile fattorizzare n.



Sappiamo quindi che la macchina funziona correttamente dato y, ma non disponiamo di tale valore. Perché non prendere un y a caso e verificare se è un quadrato?

Sia $x \in_R \mathbb{Z}_n^*$ e sia $s \in_R \{0,1\}$, allora

$$w = \begin{cases} x^2 & \text{se } s = 0 \\ y \cdot x^2 & \text{se } s = 1 \end{cases}$$

Sia b la risposta da utilizzare per l'algoritmo. Se per puro caso però y fosse un quadrato, non riuscirei a cambiare la quadraticità di w nel caso $s = 1$. Quindi nel caso in cui s fosse uguale a 1 e y fosse un quadrato, allora w sarebbe un quadrato, quindi la risposta dell'algoritmo sarebbe sempre sbagliata, poiché verrebbe complementata la risposta pensando che w non sia un quadrato. In ogni caso però è possibile sorvolare tale problema, poiché basta vedere come si comporta statisticamente la macchina in presenza di non quadrati e di quadrati. Il comportamento della macchina è per forza diverso di fronte alle due situazioni, perché altrimenti non sarebbe capace di distinguere i due input.

3.6 Distinguisher

Codificare $b_1, \ldots, b_l$ in $E(b_1), \ldots, E(b_l)$ può essere fatto facendo si che dal testo cifrato non si risalga al testo in chiaro. Dimenticandoci del problema della malleabilità, vorremmo che dal cyphertext non si possa risalire ad alcuna informazione binaria sul plaintext, ma dobbiamo capire cosa vuol dire non poter risalire ad alcuna informazione binaria.

Distinguisher

Un **distinguisher** è un algoritmo binario $\mathcal{D} \in \texttt{PPT}$ che restituisce 0 o 1. Il distinguisher prende in input un evento che soddisfa o meno una certa proprietà e un altro evento che soddisfa o meno la stessa proprietà. Il distinguisher deve essere in grado di distinguere se i due eventi si comportano in modo diverso o meno. Statisticamente il distinguisher verifica che i due eventi si comportino in modo polinomialmente diverso (3.5.1), altrimenti tale differenza non sarebbe distinguibile.

Sia $\mathbb{P}_k^{\mathcal{D},m}$ la probabilità che il distinguisher $\mathcal{D}$ restituisca 1 su input, una codifica di m quando il security parameter vale k. Un sistema di codifica nascone m_1 e m_2 a $\mathcal{D}$ quando:

$$\forall c\, \exists \bar{k}\, \forall k \geq \bar{k} \qquad |\mathbb{P}_k^{\mathcal{D},m_1} - \mathbb{P}_k^{\mathcal{D},m_2}| < k^{-c} \tag{3.4}$$

Ovvero:

$$|\mathbb{P}_k^{\mathcal{D},m_1} - \mathbb{P}_k^{\mathcal{D},m_2}| < k^{-\omega(1)} \tag{3.5}$$

E nasconde informazioni a $\mathcal{D}$ se per ogni m_1 e m_2 E nasconde m_1 e m_2 a $\mathcal{D}$. E nasconde se per ogni $\mathcal{D} \in \texttt{PPT}$ E nasconde m_1 e m_2 a $\mathcal{D}$.

Dimostrazione. Supponiamo per assurdo che $\exists \mathcal{D} \in \texttt{PPT}$ tale che

$$\exists m_1, m_2, \forall \bar{k}\, \exists k \geq \bar{k} \qquad |\mathbb{P}_k^{\mathcal{D},m_1} - \mathbb{P}_k^{\mathcal{D},m_2}| \geq k^{-c}$$

Allora $\mathcal{D}$ può distinguere su due messaggi m_1 e m_2 che differiscono di un solo bit.

$$m_1 = \alpha^1\alpha^2 \dots \alpha^l \qquad m_2 = \beta^1\beta^2 \dots \beta^l$$

Definisco $\forall i \in \{0, \dots, l\}\, m(i) = \beta^1 \dots \beta^{i-1}\beta^i\alpha^{i+1} \dots \alpha^l$, ovvero una serie di bit intermedi per poter passare da m_1 a m_2 variando un solo bit per volta. Quindi $m(0) = m_1$, $m(l) = m_2$, di conseguenza $\mathbb{P}_k^{\mathcal{D},m(i)} = \mathbb{P}(i)$, ovvero la probabilità che il distinguisher restituisca 1 se viene data in input una codifica del messaggio m_i.

$$\mathbb{P}(0) = \mathbb{P}_k^{\mathcal{D},m_1} \qquad \mathbb{P}(l) = \mathbb{P}_k^{\mathcal{D},m_2}$$

Scopriamo quindi che l'ipotesi diventa:

$$k^{-c} \leq |\mathbb{P}(0) - \mathbb{P}(l)| = |\sum_{i=0}^{l-1} \mathbb{P}(i) - \mathbb{P}(i+1)| \leq \sum_{i=0}^{l-1} |\mathbb{P}(i) - \mathbb{P}(i+1)|$$

Poiché sappiamo che $\mathbb{P}(0) - \mathbb{P}(1) + \mathbb{P}(1) - \mathbb{P}(2) + \dots + \mathbb{P}(l-1) - \mathbb{P}(l) = \mathbb{P}(0) - \mathbb{P}(l)$. Quindi:

$$\sum_{i=0}^{l-1} |\mathbb{P}(i) - \mathbb{P}(i+1)| \geq k^{-c}$$

Avendo la somma di numeri che eccede un determinato valore allora so che esiste almeno un elemento che è maggiore o uguale della media.

$$\exists i \in \{0, \dots, l-1\} \qquad t.c. \qquad |\mathbb{P}(i) - \mathbb{P}(i+1)| \geq \frac{k^{-c}}{l} \geq k^{-(c+1)} = k^{-c'}$$

Abbiamo quindi trovato un polinomio c' tale per cui il distinguisher è in grado di distinguere due messaggi che differiscono di un solo bit. I due messaggi m_1 e m_2 differiscono di un solo bit, quindi:

$$\begin{array}{rcllllllll} m(i) & = & \beta^1 & \dots & \beta^{i-1} & \beta^i & \alpha^{i+1} & \alpha^{i+2} & \dots & \alpha^l \\ m(i+1) & = & \beta^1 & \dots & \beta^{i-1} & \beta^i & \beta^{i+1} & \alpha^{i+2} & \dots & \alpha^l \end{array}$$

A questo punto supponiamo senza perdita di generalità che $\alpha^{i+1} = 0$ e $\beta^{i+1} = 1$. Sia z un elemento di $\mathbb{Z}_n^*$ con $\left(\frac{z}{n}\right) = 1$, quindi z potrebbe essere la codifica di uno 0 o di un 1.

Dato z viene costruito $E(\beta_1)E(\beta_2)\dots E(\beta_i)zE(\alpha_{i+2})\dots E(\alpha_l)$ e viene lanciato l'algoritmo $\mathcal{D}$ sul risultato. La probabilità con cui $\mathcal{D}$ restituisce 1 è $\mathbb{P}$(i) se z codifica 0 e $\mathbb{P}(i+1)$ se z codifica 1 se in input viene data la codifica di $m(i)$ secondo l'algoritmo per codificare il messaggio m_i, ovvero applicando E a tutti i bit del messaggio. Avendo però aggiunto z in mezzo al messaggio non ho codificato secondo l'algoritmo che il distinguisher si aspetta, quindi bisogna codificare il messaggio $m(i)$ in maniera corretta:

$$E(\beta_1)E(\beta_2)\dots E(\beta_i)\, z \cdot x^2\, E(\alpha_{i+2})\dots E(\alpha_l) \qquad \text{con } x \in \mathbb{Z}_n^*$$

A questo punto il distinguisher restituisce 1 con probabilità $\mathbb{P}(i)$ se z codifica 0 e $\mathbb{P}(i+1)$ se z.

Chiamiamo $\mathcal{P}_0$ la probabilità che il distinguisher restituisca 0 (*quindi* $\mathbb{P}(i)$) e $\mathcal{P}_1$ la probabilità che il distinguisher restituisca 1 (*ovvero* $\mathbb{P}(i+1)$) se in input dato è costruito come sopra.

Se si riceve la codifica di un bit scelto a caso, con quale probabilità si riesce a distinguere se è 0 o 1? Si utilizza il risultato del distinguisher $\mathcal{D}$ come tentativo.

$$\mathbb{P}[\texttt{indovinare}] = \frac{1}{2} \cdot (1 - \mathcal{P}_0) + \frac{1}{2} \cdot \mathcal{P}_1 = \frac{1}{2} + \frac{1}{2} \cdot (\mathcal{P}_1 - \mathcal{P}_0)$$

Se sottraiamo $\frac{1}{2}$ otteniamo:

$$\begin{aligned} \mathbb{P}[\texttt{non indovinare}] &= \frac{1}{2} + \frac{1}{2} \cdot (\mathcal{P}_1 - \mathcal{P}_0) - \frac{1}{2} \\ &= \frac{1}{2} \cdot (\mathcal{P}_1 - \mathcal{P}_0) \\ &= \frac{1}{2} \cdot |\mathbb{P}(i) - \mathbb{P}(i+1)| \\ &\geq \frac{1}{2} \cdot k^{-c'} \end{aligned}$$

L'algoritmo ha quindi un vantaggio di almeno $\frac{1}{2} \cdot k^{-c'}$, quindi tale algoritmo è un attaccante.

□

3.7 Lancio della moneta

Il sistema descritto in precedenza ha un problema, infatti sappiamo che la codifica di un singolo bit richiede una quantità molto alta di bit. Infatti implicherebbe un enorme spreco di banda, di risorse. Ci piacerebbe giungere ad uno scenario in cui un singolo bit viene codificato con un solo bit.

Per arrivare a questo punto impareremo a costruire numeri **pseudo-casuali crittograficamente sicuri** da utilizzare come base per simulare un one-time pad, dove la chiave non è realmente casuale, ma è una chiave generata da noi, ma nessuno con potenza di calcolo polinomiale possa realmente notare la differenza.

Il lancio di monete in rete è un problema molto importante, infatti simulare un reale lancio di monete è molto difficile, in quanto non si può avere la certezza che uno dei due agenti non stia mentendo. Disponiamo dei due agenti A e B che vogliono simulare un lancio di moneta e che comunicano attraverso un canale di comunicazione e vorremmo che il lancio avvenga in modo simile a quello reale, ovvero che entrambi gli agenti non sappiano il risultato del lancio prima che questo avvenga e che il risultato sia frutto del lancio di una moneta.

Il fatto che A lanci la moneta e che comunichi il risultato a B non va bene, poiché il risultato può essere manipolato da A affinché sia di suo gradimento, e l'altro agente non può avere la certezza che il risultato sia reale. Si potrebbe provare a comunicare il risultato in contemporanea, ma anche in questo caso si potrebbe simulare un ritardo di rete in modo da sfruttare il risultato a proprio vantaggio. Ci si potrebbe affidare ad un terzo agente, ma anche in questo caso l'agente esterno deve essere fidato.

Abbiamo quindi dato due risultati al lancio della moneta, ovvero $Coin_a$ e $Coin_b$. $Coin_a$ è una variabile del processo locale dell'agente A e $Coin_b$ è una variabile del processo locale dell'agente B, ed entrambe conterranno il risultato del lancio della moneta. Se A e B seguono il protocollo (*sono onesti*), allora $Coin_a = Coin_b$, con

$$\mathbb{P}[Coin_a = 0] = \frac{1}{2}$$

Potrebbero però non seguire il protocollo e indurre l'altro agente in errore, ottenendo un risultato iniquo, nel nostro caso A segue il protocollo, mentre B no, non possiamo fornire alcuna garanzia su B, possiamo però tutelare A. Per tutelarlo, A deve essere in grado di osservare il risultato di un lancio di una moneta.

$$|\mathbb{P}[Coin_a = 0] - \frac{1}{2}| < k^{-\omega(1)}$$

Ammettiamo che B possa alterare tale probabilità, ma non la può alterare di troppo, quindi la differenza tra la probabilità reale e quella che B può alterare deve essere minore di qualsiasi polinomio. Per ogni esponente esiste un valore minimo della lunghezza della chiave tale per cui con chiavi sufficientemente lunghe, la probabilità si discosta da $\frac{1}{2}$ di un valore minore di $k^{-\omega(1)}$.

3.7.1 Lancio della moneta con il residuo quadratico

Ci basiamo sulla difficoltà di stabilire se un numero è un quadrato o meno in $\mathbb{Z}_p^*$.

Alice

Bob

Sceglie $p, q \in_R primi$

Calcola $n = p \cdot q$

Calcola $z \in \mathbb{Z}_n^*$ con $\left(\frac{z}{n}\right) = 1$

$$\xrightarrow{n, z}$$

Sceglie $b \in_R \{0, 1\}$

$$\xleftarrow{b}$$

$$\xrightarrow{p, q}$$

Il risultato finale sarà $b \oplus \texttt{isSquare}(z)$. Sostanzialmente A lancia una moneta (*sceglie a caso fra un quadrato o un non quadrato, poiché scegliendo uniformemente fra gli elementi di* $\mathbb{Z}_n^*$ *con simbolo di Jacobi* 1*, si ottiene un quadrato con probabilità* $\frac{1}{2}$) e B sceglie un bit b a caso e lo invia ad A. Quando B riceve z non sa qual è il risultato del lancio della moneta, perché non sa risolvere il problema del residuo quadratico, di conseguenza è vero che A invia prima il risultato del proprio lancio della moneta, ma nella condizione in cui B non vede il risultato, ma B non invia il bit b in funzione di z.

Se entrambi gli agenti sono onesti, z è stato campionato secondo una misura causale, quindi z è un quadrato con probabilità $\frac{1}{2}$ e un non quadrato con probabilità $\frac{1}{2}$. Anche b è stato campionato secondo una misura causale, quindi b è 0 con probabilità $\frac{1}{2}$ e 1 con probabilità $\frac{1}{2}$. Alla fine del protocollo, tutti sono in grado di calcolare il risultato perché tutti conoscono gli interi parametri dell'algoritmo, che sarà uguale per entrambi seguendo la formula: $b \oplus \texttt{isSquare}(z)$.

Se B è disonesto, allora può calcolare il bit b non casuale, distribuendolo in maniera differente. Se B calcola b **indipendentemente** da z, allora qualunque sia la regola di calcolo di b, sarà sempre uguale alla quadraticità di z con probabilità $\frac{1}{2}$. Un altro modo per imbrogliare è quello di calcolare b in funzione di z, ma questo richiede la conoscenza dell'algoritmo della quadraticità di z. B ha il vantaggio di allontanarsi da $\frac{1}{2}$, ma è il vantaggio pari a quello della risoluzione del problema del residuo quadratico, quindi più piccolo di qualsiasi polinomio.

Se A è disonesto, può calcolare p e q non primi, ma B, alla fine del protocollo se ne accorgerebbe ricevendo p e q e quindi potrebbe lanciare una propria moneta per decidere il proprio risultato o, se in conoscenza del risultato che gli è favorevole, può decidere come risultato un valore che gli fornirebbe tale vantaggio. A potrebbe non inviare p e q, in questo caso B ha un enorme svantaggio, perché dopo il secondo messaggio A conosce $Coin_A$, ma B non conosce $Coin_B$, se A invia il proprio messaggio a B, B è in grado di calcolare $Coin_B$ e quindi il risultato finale, ma se A sa il risultato vincente e il risultato ottenuto è sfavorevole, A non spedisce il terzo messaggio e B lancia la propria moneta che con probabilità $\frac{1}{2}$ è favorevole, ma con probabilità $\frac{1}{2}$ è sfavorevole, ma si crea una situazione sfavorevole a B

$$\mathbb{P}[Coin_B = 0] = \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} = \frac{3}{4}$$

Il comportamento di A ha alterato la misura di probabilità di successo di B, a favore di A.

Il protocollo appena descritto non soddisfa le proprietà di un protocollo di lancio della moneta. Ci sono casi in cui il protocollo però può essere seguito, ovvero quando chi impersona A non conosce il risultato vincente e B può o non può conoscere il risultato vincente.

> Un protocollo per funzionare deve essere tale che gli agenti possano conoscere il risultato anche senza l'ultimo messaggio. Se l'ultimo messaggio è quello significativo, allora il protocollo non è corretto.

Sia n il numero minimo di messaggi di un protocollo che funziona, quindi gli agenti conoscono il risultato del lancio della moneta anche senza l'invio dell'ultimo messaggio. ciò vuol dire che l'ultimo messaggio non serve, ma se non serve allora il penultimo messaggio è quello che permette ad A di conoscere il risultato, ma se B decide di non inviare tale messaggio si innesca una reazione a catena che porta A e B a non inviare nulla.

> Non esiste il protocollo per il lancio della moneta che soddisfi le condizioni dettate da noi. Il protocollo funziona solo se, con n agenti, ad imbrogliare sono meno di $\frac{n}{2}$ agenti.

Indeboliamo la soluzione in modo che funzioni.

Lancio di moneta nel pozzo

L'idea è quella di utilizzare un pozzo, in cui finirà la moneta. A riesce a vedere il risultato del lancio della moneta, ma B no, perché si trova di intralcio un cane tenuto al guinzaglio da A. A può decidere far si che B possa vedere il risultato, facendo rientrare il cane che si trova nei pressi del pozzo, ma A può anche decidere di non far rientrare il cane.

A non può variare il risultato del lancio della moneta, ha solamente la facoltà di impedire o meno a B di vedere il risultato. Sa a B decidere se partecipare al processo o meno, non gli interessa il risultato del lancio della moneta, ma interessa solamente che la probabilità del lancio sia di almeno $\frac{1}{2}$.

Oblivious transfer

Supponiamo che B sappiamo il risultato della borsa del prossimo anno e che chieda a ad A di avere in cambio del denaro per i risultati della borsa. B non vuole dare tutto il denaro ad A, ma vuole dare solamente una parte del denaro. A questo punto A decide di far il lancio della moneta per decidere se mostrare o meno il risultato della borsa a B. B a questo punto deve trasferire l'informazione ad A, ma **non gli interessa se l'informazione è stata trasferita o meno**, ma solamente che la probabilità che l'informazione sia stata trasferita sia di almeno $\frac{1}{2}$.

Alice

Bob

Alice		Bob
		Sceglie $p, q \in_R$ *primi* Calcola $n = p \cdot q$ Calcola e, d chiave p e p per `RSA` Codifica il segreto con `RSA` ottenendo un messaggio c
	$\xleftarrow{n, e, c}$	
Sceglie $x \in_R \mathbb{Z}_n^*$		
	$\xrightarrow{x^2}$	
		Calcola una radice quadrata di x, che sarà y
	$\xleftarrow{y}$	

Quindi A invia un quadrato a caso e B invia una delle quattro radici quadrate di x. Con probabilità $\frac{1}{2}$ tale radice sarà diversa da $\pm x$. Quando un agente possiede due radici quadrate di uno stesso numero **che non sono una l'opposta dell'altro**, allora è in grado di fattorizzare n. Con probabilità $\frac{1}{2}$, B invierà ad A una radice diversa da $\pm x$, metterà quindi A nelle condizioni di calcolare p e q e quindi d per decodificare il messaggio c. B non ha alcun interesse nel risultato, che otterrà con probabilità $\frac{1}{2}$, ma interessa che A abbia sia riuscita a decifrare c con probabilità $\frac{1}{2}$.

Il vantaggio di A è implicito nel risultato.

3.7.2 Lancio della moneta con il logaritmo discreto

Il problema è che fino ad ora abbiamo lavorato con la quadraticità di un numero in $\mathbb{Z}_n^*$, ma vorremmo lavorare con singoli bit. Quindi non con l'inversa di una funzione one way, ma con singoli bit, ovvero con l'informazione **binaria** dell'inversa di una funzione one way.

Disponiamo di un numero p e un generatore g di $\mathbb{Z}_p^*$, quindi g è un numero che genera tutti gli elementi di $\mathbb{Z}_p^*$, ovvero $\{g^0, g^1, \ldots, g^{p-1}\}$. I due valori p e g sono condivisi tra A e B, prima dell'inizio del protocollo.

Alice

Bob

Sceglie $x \in_R \{0, \ldots, p-2\}$
Calcola $z = g^x$

$$\xrightarrow{\quad z \quad}$$

Sceglie $b \in_R \{0, 1\}$

$$\xleftarrow{\quad b \quad}$$

$$\xrightarrow{\quad x \quad}$$

Il risultato è quindi $b \oplus \left(x < \frac{p-1}{2}\right)$. L'idea di fondo è che B quando sceglie b non è in grado di dedurre il risultato del lancio della moneta, perché dovrebbe verificare se x è minore o meno di $\frac{p-1}{2}$, facendo quindi un test binario su il **logaritmo discreto** di z. Nel momento in cui A invia x, B è in grado di verificare la correttezza del risultato e a calcolare il risultato.

> B non è in grado di calcolare il predicato binario $x < \frac{p-1}{2}$, a meno di un vantaggio più piccolo di qualsiasi polinomio.

Tale algoritmo utilizza un **predicato binario sull'inversa di una funzione one way**. Data una funzione one way f, un predicato binario P sull'inversa di f, non siamo certi che il predicato sia difficile da calcolare.

Per il logaritmo discreto, il predicato binario facile da calcolare è il **bit meno significativo** di x. Tale bit vale zero se e solo se z è un quadrato in $\mathbb{Z}_p^*$. Ma tale predicato dice qualcosa di differente, ovvero ci chiediamo se il logaritmo discreto di z sta nella prima metà o nella seconda metà dei logaritmi discreti di $\mathbb{Z}_p^*$ o nella seconda, ovvero una sorta di **bit più significativo** di x.

3.8 Hardcore predicate del logaritmo discreto

Hardcore predicate

Si tratta di un predicato binario sull'inversa di una funzione one way, che è difficile da calcolare. Se scegliessimo a caso il valore di tale predicato binario, la probabilità con cui un algoritmo riesce a calcolare il predicato conoscendo z è $\frac{1}{2} + \epsilon$, con ϵ più piccolo di qualsiasi polinomio.

Sia $y = g^x$ e sia il predicato in considerazione $x < \frac{p-1}{2}$. Se y è un quadrato, allora ammette due radici quadrate, z_1 e z_2, di queste due radici, una ha logaritmo discreto minore di $\frac{p-1}{2}$ e l'altra maggiore o uguale a $\frac{p-1}{2}$. Questo perché le rispettive radici quadrate sono nella forma g^i e $g^{i+\frac{p-1}{2}}$. La radice quadrata nella forma g^i è chiamata **radice quadrata principale**.

Di base se abbiamo due radici quadrate di y, che sappiamo calcolare aritmeticamente, non sappiamo come capire quale sia la radice quadrata principale, ma se avessimo a disposizione l'algoritmo per calcolare se il logaritmo discreto di un numero è minore di $\frac{p-1}{2}$, allora potremmo calcolare la radice quadrata principale di un quadrato.

Supponiamo di avere un algoritmo che date le due radici quadrate di un numero distingue la radice quadrata principale. In questo caso possiamo costruire un algoritmo che calcola il predicato binario $x < \frac{p-1}{2}$.

Teorema 3.8.1 Qualunque funzione one-way ammette un hardcore predicate.

Ne deriva quindi che:

Teorema 3.8.2 Se esiste un algoritmo per il calcolo della radice principale di un quadrato in $\mathbb{Z}_p^*$, allora esiste un algoritmo efficiente per il calcolo del logaritmo discreto in $\mathbb{Z}_p^*$.

Teorema 3.8.3 Se esiste che riesce a calcolare il predicato binario con vantaggio polinomiale rispetto al caso casuale, allora esiste un algoritmo probabilistico polinomiale per il calcolo del logaritmo discreto.

Per dimostrare il teorema dovremmo seguire tre passaggi:

1. Mostrare che siamo in grado di calcolare il logaritmo discreto di un numero avendo in mano un algoritmo per il calcolo della radice quadrata principale di un quadrato.
2. Mostrare che tale algoritmo funziona anche se la radice quadrata principale è calcolata con probabilità esponenzialmente vicina a 1.
3. Mostrare che partendo da un algoritmo che funziona con vantaggio polinomiale rispetto al caso casuale, siamo in grado di costruire un algoritmo che funziona con probabilità esponenzialmente vicina a 1.

3.8.1 Calcolare il logaritmo discreto avendo a disposizione l'algoritmo per il calcolo della radice quadrata principale (PSQR)

Avremo a disposizione l'algoritmo `PSQR` che calcolerà la radice quadrata principale di un quadrato in $\mathbb{Z}_p^*$, avendo a disposizione il predicato binario. Per farlo prende in input y ne calcola la radice quadrata, verifica il predicato binario per capire se è la radice quadrata principale, se la verifica fallisce, calcola la radice principale opposta e restituisce il risultato.

```
procedure LSB(y)
    if y/p = 0 then
        return 0
    else
        return 1
    end if
end procedure
```

Ricordiamo che la procedura `LSB` verifica se il bit meno significativo di un numero è pari o dispari. Se è pari restituisce 0, altrimenti restituisce 1.

```
procedure DiscreteLogarithm(y)
    if y = 1 then
        return 0
    end if
    b ← LSB(DiscreteLogarithm(y))
    if b = 1 then
        return y ← y · g^{-1}          ▷ imposta a zero il bit meno significativo
    end if
    y ← PSQR(y)                        ▷ scorri a destra di un bit
    return 2 · DiscreteLogarithm(y) + b
end procedure
```

Per calcolare la radice quadrata abbiamo necessariamente bisogno del bit meno significativo a zero.

L'algoritmo ricorsivo permette di calcolare il logaritmo discreto di un numero calcolando il bit meno significativo e riconducendo il calcolo dello stesso problema in una situazione in cui si ha un bit in meno.

> Se abbiamo un algoritmo che funziona per il calcolo della radice quadrata principale allora possiamo costruire un algoritmo che calcola il logaritmo discreto.

Supponiamo di avere un algoritmo B per `PSQR` che funziona con probabilità esponenzialmente vicina a 1, ovvero $1 - \epsilon$. La probabilità che l'algoritmo `DiscreteLogarithm` invocato k volte fornisca sempre la risposta corretta è $(1 - \epsilon)^k$. L'algoritmo `DiscreteLogarithm` può non funzionare, ma abbiamo modo di accorgerci:

- Se il numero di passi è maggiore del numero di bit di y, allora siamo sicuri che il `PSQR` non ha funzionato.

- Sul risultato finale possiamo verificare che $g^x = y$. Se non è così allora `PSQR` non ha funzionato.

In media se funziona con probabilità $(1-\epsilon)^k$, allora il numero di volte che funziona in cui si dovrà lanciare l'algoritmo sarà $\frac{1}{(1-\epsilon)^k}$. Se $\epsilon \leq \frac{1}{2}$, allora $(1-\epsilon)^k \geq \frac{1}{2^k}$.

> Se abbiamo un algoritmo che funziona con probabilità esponenzialmente vicina a 1, allora possiamo costruire un algoritmo che calcola il logaritmo discreto con probabilità almeno $\frac{1}{2}$, quindi se reiteriamo in media due volte l'algoritmo otteniamo la risposta corretta. Questo perché possiamo verificare che il risultato sia corretto.

Riusciamo ora a mostrare che se ci viene dato un algoritmo che funziona con vantaggio polinomiale rispetto a $\frac{1}{2}$, allora possiamo costruire un algoritmo che funziona con probabilità esponenzialmente vicina a 1? Assolutamente si, per farlo dobbiamo costruire tante istanze indipendenti dello stesso problema e prendendo il risultato osservato la maggior parte delle volte. Come costruiamo gli esperimenti indipendenti, tali per cui nel momento in cui conosciamo la risposta al problema costruito, allora troviamo la risposta al problema originale?

3.8.2 Costruzione dell'esperimento indipendente y' tale per cui dalla risposta y' otteniamo la risposta a y

Teorema 3.8.4 Sia y un quadrato in $\mathbb{Z}_p^*$ e sia $r \in_R \left[0, \dots \frac{p-1}{2}\right]$ (*esponente a caso per una possibile radice quadrata di un numero, presente nella prima metà*). Sia $y' = y \cdot g^{2x}$, dove y' sarà un'altra istanza indipendente del problema dei residui quadratici, distribuita uniformemente in $\mathbb{Z}_p^*$, di cui conosciamo il logaritmo discreto. Se $2x + 2r < p - 1$ allora $z \cdot g^r$ è `PSQR` di $y \cdot g^{2r} \iff z$ è `PSQR` di y.

$$\textit{Sappiamo che } \sqrt{g^{2x} g^{2r}} = \begin{cases} g^x g^r \\ g^x g^r g^{\frac{p-1}{2}} \end{cases}$$

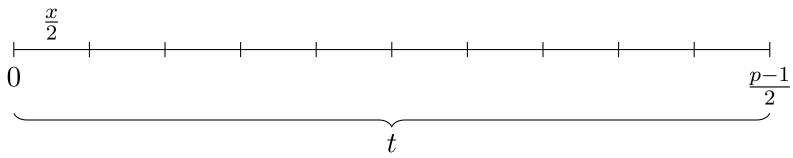
Con l'ipotesi $2x + 2r < p - 1$ allora g^{x+r} è la radice quadrata principale.

Questo perché moltiplicando $y \cdot g^{2r}$ abbiamo un'operazione **aritmetica** poiché gli esponenti sono sommati e non si ricade nella radice quadrata successiva, ovvero quella dopo la prima metà (*l'esponente di y è $2x$*). Moltiplicare quindi fa si che non si vada oltre $p-1$, quindi il logaritmo discreto del prodotto è la somma dei logaritmi discreti; andar oltre $p-1$ significa tornare indietro, siccome siamo in un gruppo ciclico.

Dalla risposta al problema trasformato, sappiamo quindi calcolare la risposta al problema originale.

Ma come faccio a far si che r scelta casualmente soddisfi le proprietà del teorema?

> Più x è piccolo e più è probabile trovare un r che soddisfa la proprietà del teorema.



Supponiamo che $\frac{x}{2}$ sia nel primo intervallo, la probabilità di trovare un r dove $\frac{x}{2} + r$ non vada oltre $\frac{p-1}{2}$, ovvero $\frac{t-1}{t}$. La probabilità che la risposta sia corretta è la combinazione della probabilità di soddisfacibilità del teorema, ovvero $\left(\frac{t-1}{t}\right)$ e la probabilità della risposta al problema trasformato, ovvero $\left(\frac{1}{2} + k^{-c}\right)$.

Questo perché qualche volta, dalla risposta al problema trasformato, non si riesce a trovare la risposta al problema originale.

$$\mathbb{P}[\texttt{Esperimento corretto}] = \left(\frac{t-1}{t}\right)\left(\frac{1}{2} + k^{-c}\right) \geq \frac{1}{2} + k^{-2c} \tag{3.6}$$

Risolvendo la disequazione in t riusciamo a capire qual è il limite inferiore a gli intervalli da utilizzare.

Dalla teoria dell'algebra sappiamo che si tratta di una disequazione polinomiale in t, come tale ha una soluzione polinomiale nelle costanti interne. La soluzione è quindi polinomiale nelle costanti presenti nell'equazione. Il limite inferiore è quindi polinomiale nelle costanti presenti nell'equazione.

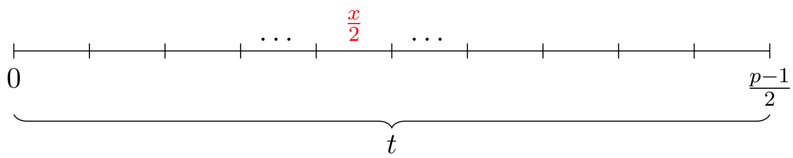
$$\frac{t-1}{t} \geq \frac{1}{2} + k^{-c} \implies t \geq \frac{2}{1 + 2k^{-c}} \tag{3.7}$$

Dividendo in una quantità di intervalli che è polinomiale in k otteniamo un sistema tale per cui, se x è nel primo intervallo, la probabilità di successo, ovvero di fornire una radice quadrata principale è polinomialmente distante da $\frac{1}{2}$, di conseguenza abbiamo un algoritmo che risolve il problema in tempo polinomiale.

Abbiamo quindi costruito un algoritmo che con probabilità $\frac{1}{2}$ calcola il logaritmo discreto di un numero, a patto che la metà di tale logaritmo stia nel primo intervallo.

Generalizzazione dell'intervallo

Supponiamo di sapere che $\frac{x}{2}$ sia nell'intervallo $i - esimo$.



Sappiamo che $y = g^x$ e che $i\frac{p-1}{2t} \leq \frac{x}{2} < (i+1)\frac{p-1}{2t}$

$$g^{x'} = y \cdot g^{-\frac{p-1}{t}i} \implies g^{x'} = g^x \cdot g^{-\frac{p-1}{t}i} \implies x' = x - \frac{p-1}{t}i$$

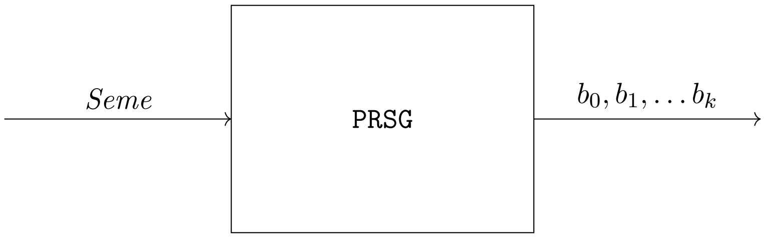
Quindi x' sta nel primo intervallo, infatti da $\frac{x}{2}$ abbiamo tolto il punto d'inizio dell'intervallo $i-$*esimo*, togliendo tale quantità abbiamo ottenuto il risultato corrispondente al primo intervallo. Ci siamo quindi ricondotti alla risoluzione di un problema relativo al calcolo della radice quadrata principale, ovvero il problema che abbiamo già risolto. Dopo aver risolto il problema ci riconduciamo nuovamente al problema originale moltiplicando per $g^{\frac{p-1}{t}i}$.

Non conoscendo l'intervallo in cui si trova $\frac{x}{2}$, possiamo provare a risolvere il problema per tutti gli intervalli. Essendo quantità polinomiali in k.

3.9 Bit Pseudocasuali

Vogliamo costruire una sequenza di bit che sia equivalente ad un lancio di moneta. Se volessimo utilizzare all'interno di un programma dei bit che sono casuali, avremmo bisogno di un processo di lancio di moneta all'interno del nostro calcolatore. All'interno del processo di lancio di moneta utilizzato c'è sempre un punto in cui ci sono scelte casuali da dover effettuare e tali scelte possono essere fatte da un processo realmente casuale. All'interno di un calcolatore non abbiamo un processo realmente casuale, poiché i calcoli e le scelte sono deterministiche. Il processo non è casuale, ma deterministico; ma appare come processo casuale agli occhi di chi lo osserva.

L'algoritmo deterministico esegue un'operazione sul **seme** e produce un output. La generazione non può essere tale che la distribuzione di probabilità degli elementi sia uniforme, ma deve essere tale che produca una sequenza di bit tale per cui nessuno con potenza di calcolo probabilistica polinomiale sia in grado di capirne la distribuzione. Chiunque non si deve accorgere del fatto che non stiamo lavorando con sequenze casuali, ma con sequenze pseudocasuali.



Il seme s viene scelto in maniera realmente casuale ed è corto ed ha k bit e l'output ha l bit, normalmente con $l > k$. Il generatore di numeri pseudocasuali possiamo vederlo come un **moltiplicatore di casualità**. Nessun algoritmo probabilistico polinomiale sarà in grado di trovare una regolarità all'interno dell'output generato, nonostante ci sia.

3.9.1 Generare bit pseudocasuali

Avendo a disposizione la sequenza $b_0 b_1 ... b_l$, facciamo fatica ad indovinare il bit b_{l+1}, lo indoviniamo con un vantaggio $\frac{1}{2} + \epsilon$, dove ϵ è più piccolo di qualsiasi polinomio.

$$\left|\mathbb{P}[s \in_R \{0,1\}^k; b_0b_1...b_l \leftarrow \mathcal{G}(s); b \leftarrow \mathcal{A}(b_0b_1...b_l); b = b_{l+1}] - \frac{1}{2}\right| \leq k^{-\omega(1)} \tag{3.8}$$

La sequenza a disposizione la vedo come sequenza realmente casuale quando ogni singolo bit è risultato di un lancio di moneta.

Una sequenza di bit è pseudocasuale se nessun algoritmo probabilistico polinomiale è in grado di distinguere la sequenza generata da una sequenza realmente casuale.

La definizione è un po' strana, perché dice che non siamo capaci di indovinare il bit successivo. Ci piacerebbe di più una definizione che dicesse che stiamo fornendo bit casuali nel momento in cui dalla sequenza di bit generata non siamo capaci di distinguerle da una sequenza realmente casuale.

Per definire un concetto simile dobbiamo essere in grado di dire che non c'è un algoritmo che data la sequenza veramente casuale e la sequenza pseudocasuale, non è in grado di distinguere le due sequenze. Quindi descriviamo la probabilità che l'algoritmo $\mathcal{D}$ dia come risultato 1, ovvero che abbia riconosciuto la sequenza pseudocasuale come tale:

$$\mathcal{P}_k^{\mathcal{D},\mathcal{G}} = \mathbb{P}[s \in_R \{0,1\}^k; b_0b_1...b_l \leftarrow \mathcal{G}(s); \mathcal{D}(b_0b_1...b_l) = 1]$$

Descriviamo la probabilità che l'algoritmo $\mathcal{D}$ dia come risultato 1, se prende in input una sequenza di bit realmente casuale:

$$\mathcal{P}_k^{\mathcal{D},\mathcal{R}} = \mathbb{P}[b_0b_1...b_l \in_R \{0,1\}^{l+1}; \mathcal{D}(b_0b_1...b_l) = 1]$$

Quindi la probabilità che il distinguisher restituisca 1 nei due casi differisce di una quantità più piccola di qualsiasi polinomio.

$$\forall_{\mathcal{D}\in\mathtt{PPT}} \quad \left|\mathcal{P}_k^{\mathcal{D},\mathcal{G}} - \mathcal{P}_k^{\mathcal{D},\mathcal{R}}\right| < k^{-\omega(1)} \tag{3.9}$$

La formula appena descritta potrebbe essere una nuova definizione di generazione di bit pseudocasuali, le due equazioni 3.8 e 3.9 sono quindi equivalenti. Disporre di definizioni equivalenti è utile perché ci permette di scegliere quella che ci è più comoda per il problema che stiamo affrontando.

3.9.2 Generazione di bit pseudocasuali basati sul problema del logaritmo discreto

Fissiamo un numero primo p e un generatore g del gruppo $\mathbb{Z}_p^*$. Prendiamo il seme $x \in_R \{0,\dots,p-1\}$, e definiamo:

$$\begin{array}{cccccc} a_0 & a_1 & a_2 & a_3 & \dots & a_l \\ x & g^x & g^{g^x} & g^{g^{g^x}} & \dots & g^{g^{g^{\cdot^{\cdot^{\cdot^{g^x}}}}}} \\ b_0 & b_1 & b_2 & b_3 & \dots & b_l \end{array}$$

In maniera compatta:

$$\begin{cases} a_{i+1} = g^{a_i} \\ a_0 = x \end{cases}$$

Generiamo elementi di $\mathbb{Z}_p^*$, dove il primo elemento è il seme e gli altri elementi sono ottenuti elevando g all'elemento precedente.

A questo punto definisco i bit pseudocasuali come:

$$b_i = \begin{cases} 0 & \text{se } a_i < \frac{p}{2} \\ 1 & \text{se } altrimenti \end{cases}$$

Quindi l'hardcore predicate è $a_i < \frac{p-1}{2}$ La sequenza pseudocasuale di ritorno sarà:

$$b_l \quad b_{l-1} \quad b_{l-2} \quad b_{l-3} \quad \dots \quad b_0$$

Utilizzando la definizione 3.8, supponiamo per assurdo che esista un algoritmo in grado di predire il bit successivo con un vantaggio più grande di qualche polinomio. I bit presi in input sono $b_l, ..., b_3$, quindi l'algoritmo $\mathcal{A}$ predice il bit b_2. ma a_3 è l'hardcore predicate di b_2, quindi non è possibile predire il bit b_2 con un vantaggio maggiore di qualche polinomio.

$$a^3 = g^{g^{g^x}} \quad \text{e} \quad a^2 = g^{g^x}$$

La costruzione della sequenza $a_0, a_1, a_2, a_3, \dots, a_l$ e della sequenza $b_l, b_{l-1}, b_{l-2}, b_{l-3}, \dots, b_0$ è facilmente costruibile. Se i bit fossero stati restituiti in ordine sequenziale, $b_0, b_1, b_2, b_3, \dots, b_l$, non saremmo riusciti a far funzionare l'algoritmo perché i bit successivi della sequenza sarebbero stati il risultato di una funzione facilmente calcolabile. La nostra costruzione si è basata sul fatto che la funzione da calcolare sui semi è difficile al fine di dire che non riusciamo a prevedere il bit successivo.

Il fatto che la dimostrazione non funziona non implica che non sia possibile restituire i bit in ordine di enumerazione crescente.

La costruzione di funzionamento con i bit restituiti alla rovescia si basa sul fatto che conoscendo i semi non siamo in grado di calcolare b_2, se non so calcolare b_2 conoscendo i semi, a maggior ragione non sono in grado di calcolarlo conoscendo i bit, ma tale dimostrazione non mi permette di catturare tale concetto.

Prendendo in considerazione la formula 3.9, se i bit restituiti alla rovescia sono indistinguibili da una sequenza casuale, allora anche i bit restituiti in ordine crescente sono indistinguibili da una sequenza casuale.

Se $b_l, \dots, b_0$ soddisfa la definizione 3.9, allora soddisfa la definizione 3.8. Quindi $b_l, \dots, b_0$ è indistinguibile da una sequenza casuale e possiamo concludere che $b_0, \dots, b_l$ è indistinguibile da una sequenza casuale.

Sia un distinguisher $\mathcal{D}$ per $b_l, \dots, b_0$, quindi $\mathcal{D}$ prende in input $b_l, \dots, b_0$ e restituisce 1 se la sequenza è pseudocasuale e 0 se la sequenza è casuale. Sia $\mathcal{D}'$ un distinguisher per $b_l, \dots, b_0$, quindi $\mathcal{D}'$ prende in input $b_0, \dots, b_l$ e restituisce $\mathcal{D}(b_0, \dots, b_l)$, quindi rovescia la sequenza e restituisce il risultato di $\mathcal{D}$ basata sulla sequenza rovesciata.

Se $\mathcal{D}$ distingue $b_l, \dots, b_0$ allora $\mathcal{D}'$ distingue $b_0, \dots, b_l$ e viceversa. Quindi distinguere i bit in ordine di enumerazione crescente è equivalente a distinguere i bit in ordine di enumerazione decrescente.

Restituire i bit alla rovescia è svantaggioso perché non è possibile calcolare il bit successivo, mentre restituire i bit in ordine di enumerazione crescente permette di andare avanti con la sequenza.

3.9.3 Distinguere equivale a prevedere

Teorema 3.9.1

1. Sia distinguere equivalente a:

$$\forall_{\mathcal{D} \in \text{PPT}} \quad \left| \mathcal{P}_k^{\mathcal{D},\mathcal{G}} - \mathcal{P}_k^{\mathcal{D},\mathcal{R}} \right| < k^{-\omega(1)} \tag{3.10}$$

2. Sia prevedere equivalente a:

$$\left| \mathbb{P}[s \in_R \{0,1\}^k; b_0 b_1 ... b_l \leftarrow \mathcal{G}(s); b \leftarrow \mathcal{A}(b_0 b_1 ... b_l); b = b_{l+1}] - \frac{1}{2} \right| \leq k^{-\omega(1)} \tag{3.11}$$

Allora:

- $1 \implies 2$
- $2 \implies 1$

Dimostrazione. ($1 \implies 2$) Su input $b_0, \dots, b_{l-1}$ indoviniamo b_l con vantaggio polinomiale, ovvero:

$$\mathbb{P}[\mathcal{A}(b_0, \dots, b_{l-1}) = b_l] - \frac{1}{2} \geq k^{-c}$$

$$\mathbb{P}[\mathcal{A}(b_0, \dots, b_{l-1}) = b_l] \geq \frac{1}{2} + k^{-c}$$

e possiamo affermarlo senza perdita di generalità. Se vogliamo dimostrare che la proprietà (2) vale, ovvero che violando la proprietà (1) si viola anche la proprietà (2), allora possiamo costruire un distinguisher.

$$\mathcal{D}(b_0, \dots, b_l) = \begin{cases} 1 & \text{se } \mathcal{A}(b_0, \dots, b_{l-1}) = b_l \\ 0 & \text{altrimenti} \end{cases}$$

Sappiamo che la probabilità che l'algoritmo $\mathcal{A}$ indovini un bit scelto in maniera veramente casuale e indipendente da $b_0, \dots, b_{l-1}$ è esattamente $\frac{1}{2}$ (*one-time pad*), ovvero:

$$\mathcal{P}_k^{\mathcal{D},\mathcal{U}} = \frac{1}{2}$$

Mentre, la probabilità che $\mathcal{A}$ con in input $b_0, \ldots, b_{l-1}$ indovini b_l è maggiore di $\frac{1}{2}+k^{-c}$, ovvero:

$$\mathcal{P}_k^{\mathcal{D},\mathcal{G}} \geq \frac{1}{2} + k^{-c}$$

Di conseguenza:

$$\mathcal{P}_k^{\mathcal{D},\mathcal{G}} - \mathcal{P}_k^{\mathcal{D},\mathcal{U}} \geq \frac{1}{2} + k^{-c} - \frac{1}{2} = k^{-c}$$

Se esiste l'attaccante per la proprietà (1), allora violiamo la proprietà (2).

($2 \implies 1$) Disponiamo di una sequenza di bit $b_1, \ldots, b_l$ generata da $\mathcal{G}$ e una sequenza di bit $r_1, \ldots, r_l$ generata in maniera casuale e indipendente da $\mathcal{G}$, supponiamo che $\mathcal{D}$ sia un distinguisher per $b_1, \ldots, b_l$ e $r_1, \ldots, r_l$.

Sappiamo che:

$$\begin{array}{lcllllllll} \mathcal{S}_0 & = & b^1 & \ldots & b^{i-1} & b^i & b^{i+1} & b^{i+2} & \ldots & b^l \\ \mathcal{S}_l & = & r^1 & \ldots & r^{i-1} & r^i & r^{i+1} & r^{i+2} & \ldots & r^l \end{array}$$

Con la tecnica di interpolazione, supponendo che le sequenze differiscano di un solo bit (*come dimostrato nella sezione 3.6*), siamo in grado di affermare che:

$$\left| \mathcal{P}_k^{\mathcal{D},\mathcal{G}} - \mathcal{P}_k^{\mathcal{D},\mathcal{R}} \right| \geq \frac{1}{2} - k^{-c}$$

$$\left| \sum_{i=0}^{l} (\mathcal{P}_i - \mathcal{P}_{i-1}) \right| \leq \sum_{i=0}^{l} \left| (\mathcal{P}_i - \mathcal{P}_{i-1}) \right|$$

Quindi

$$\exists i \in \{0, \ldots, l\} \quad \left| \mathcal{P}_i - \mathcal{P}_{i-1} \right| \geq \frac{k^{-c}}{l} = k^{-c'}$$

$$\begin{array}{lcllllllll} \mathcal{S}_i & = & b^1 & \ldots & b^{i-1} & b^i & r^{i+1} & r^{i+2} & \ldots & r^l \\ \mathcal{S}_{i+1} & = & b^1 & \ldots & b^{i-1} & b^i & b^{i+1} & r^{i+2} & \ldots & b^l \end{array}$$

In qualche modo il distinguisher si comporta diversamente quando i bit differiscono di un solo bit.

Vorremmo cercare di costruire un algoritmo che indovini i bit pseudocasuali. Sia

$$\mathcal{F} = \mathcal{D}(b_0, \ldots, b_i, r_{i+1}, \ldots, r_l)$$

$$\mathcal{A}(b_1, ..., b_l) = \begin{cases} r_{i+1} & \text{se } \mathcal{F} = 1 \\ \bar{r}_{i+1} & \text{se } \mathcal{F} = 0 \end{cases}$$

Utilizziamo quindi il distinguisher per indovinare il bit successivo.

Senza perdita di generalità $\mathcal{P}_{i+1} - \mathcal{P}_i \geq k^{-c'}$, supponiamo che la probabilità che $\mathcal{D}$ restituisca 0 su input $b_1, \ldots, b_i, \bar{b}_{i+1}, \ldots, r_l$ sia x, allora la probabilità che $\mathcal{A}$ indovini il bit successivo è:

$$\mathbb{P}[\mathcal{A}(b_1, ..., b_l) = b_{i+1}] = \mathbb{P}[r_{i+1} = b_{i+1}] \cdot \mathcal{P}_{i+1} + \mathbb{P}[r_{i+1} = \bar{b}_{i+1}] \cdot x$$

$$\frac{1}{2} \cdot \mathcal{P}_{i+1} + \frac{1}{2} \cdot x$$

Proviamo attualmente a cercare una relazione linearmente indipendente con $\mathcal{P}_{i+1}$. Il valore equivale alla probabilità che il distinguisher restituisca 1 su input $b_1, \dots, b_i, r_{i+1}, \dots, r_l$.

$$\begin{aligned}
\mathcal{P}_i &= \mathbb{P}[\textit{il bit i-esimo} = b_{i+1}] \cdot \mathbb{P}[\mathcal{D} \textit{ dia 1 su } b_1, \dots, b_i, b_{i+1}, \dots, r_l] + \\
&+ \mathbb{P}[\textit{il bit i-esimo} = \bar{b}_{i+1}] \cdot \mathbb{P}[\mathcal{D} \textit{ dia 0 su } b_1, \dots, b_i, \bar{b}_{i+1}, \dots, r_l] = \\
&= \frac{1}{2} \cdot \mathcal{P}_{i+1} + \frac{1}{2} \cdot (1 - x) = \frac{1}{2} \cdot \mathcal{P}_{i+1} + \frac{1}{2} - \frac{1}{2} \cdot x
\end{aligned}$$

Ricaviamo quindi che:

$$x = \mathcal{P}_{i+1} + 1 - 2 \cdot \mathcal{P}_i$$

Sostituendo x nell'equazione precedente:

$$\begin{aligned}
&= \frac{1}{2} \cdot \mathcal{P}_{i+1} + \frac{1}{2} \cdot \mathcal{P}_{i+1} + \frac{1}{2} - \mathcal{P}_i \\
&= \frac{1}{2} + \mathcal{P}_{i+1} - \mathcal{P}_i
\end{aligned}$$

Ma sappiamo che $\mathcal{P}_{i+1} - \mathcal{P}_i \geq k^{-c'}$, quindi:

$$\frac{1}{2} + \mathcal{P}_{i+1} - \mathcal{P}_i \geq \frac{1}{2} + k^{-c'}$$

□

3.10 Blum Blum Shub

Si tratta di un algoritmo di generazione di numeri pseudocasuali basato sulla difficoltà del calcolo di radici quadratiche in $\mathbb{Z}_n^*$.

La funzione di elevamento al quadrato è una funzione *one-way trapdoor* in $\mathbb{Z}_{p \cdot q}^*$, poiché la conoscenza di p e q, ovvero la fattorizzazione di n, permette di calcolare la radice quadrata in tempo polinomiale. Sappiamo inoltre che ogni funzione *one-way* ammette un *hardcore predicate*, infatti:

$$\texttt{LSB}(x) = (\sqrt{x})$$

Il problema è che x ha più di una radice quadrata, che radice bisogna selezionare?

Teorema 3.10.1 **Primi di Blum**
Se p e q sono primi congrui a $3 \mod 4$ allora per ogni quadrato x esiste una sola radice quadrata y tale che $y \equiv \sqrt{x} \mod n$.

Sia $n = p \cdot q$ con p e q primi di Blum scelti casualmente, sia $x \in_R \mathcal{Z}^*$, allora

$$\begin{array}{lllll}
a_1 & a_2 & a_3 & \dots & a_l \\
x^2 & (x^2)^2 & ((x^2)^2)^2 & \dots & x^(2^l)
\end{array}$$

ovvero

$$\begin{cases} x_i = x^2 \\ x_{i+1} = a_i^2 \end{cases}$$

Definiamo quindi $b_i = \texttt{LSB}(a_i)$, quindi:

$$\begin{array}{lllll} a_1 & a_2 & a_3 & \ldots & a_l \\ x^2 & (x^2)^2 & ((x^2)^2)^2 & \ldots & x^(2^l) \\ b_1 & b_2 & b_3 & \ldots & b_l \end{array}$$

Dove il valore di ritorno sarà la sequenza $b_l, b_{l-1}, \ldots, b_1$. Utilizzando il principio spiegato nella sezione precedente (3.9.2) possiamo restituire i bit in ordine di enumerazione crescente, ovvero $b_1, b_2, \ldots, b_l$.

La differenza è che se il numero n è conosciuto da tutti, allora è possibile restituire i bit solamente nell'ordine $b_1, b_2, \ldots, b_l$, altrimenti in qualsiasi ordine.

3.11 Crittosistema di Blum Goldwasser

L'idea è stata di utilizzare un crittosistema dato dai soliti tre algoritmi, quello di generazione della chiave, quello di codifica e quello di decodifica. La chiave è data da p, q primi di Blum con k bit, $n = p \cdot q$, dove la chiave pubblica è data da n e la chiave privata è data da p e q. Chi possiede la chiave segreta riesce a calcolare radici quadrate in modulo n. L'algoritmo di codifica è dato da un $x \in_R \mathbb{Z}_n^*$, e attraverso blum blum shub, generiamo l bit pseudocausali $b_1, \ldots, b_l$. Supponiamo di avere un messaggio m di lunghezza l, allora il messaggio cifrato sarà:

$$c_1, \ldots, c_l, x^{2^{l+1}} = m_1 \oplus b_1, \ldots, m_l \oplus b_l$$

Dove $x^{2^{l+1}}$ è il primo elemento della sequenza generata da blum blum shub.

Sostanzialmente il messaggio viene cifrato attraverso one time pad, con una chiave non realmente casuale, ma con numeri pseudocasuali mediante un seme.

Se la sequenza $b_1, \ldots, b_l$ è casuale, allora il cyphertext non contiene alcuna informazione sul plaintext, quindi il crittosistema è sicuro.

Supponendo per assurdo che dal ciphertext sia possibile ricavare informazioni sul plaintext, allora si riesce a distinguere qualunque coppia di messaggi, ma sappiamo che non lo si riesce a fare in presenza di una chiave realmente casuale. Ma nel caso specifico del nostro crittosistema, si dovrebbe riuscire a distinguere sul generato di bit pseudocausali, ma questo è assurdo poiché non siamo in grado di distinguere una sequenza pseudocasuale da una casuale.

La decodifica ricava x^2, infatti, data la conoscenza della fattorizzazione di n, è possibile calcolare $\sqrt{x^{2^{l+1}}} = x^{2^l}$, e così via. Si ricava quindi $x^{2^l}, x^{2^{l-1}}, \ldots, x^2$ e si calcolano i bit $b_1, \ldots, b_l$. Attraverso questi bit è possibile decifrare il messaggio con l'operazione inversa:

$$m_i = c_i \oplus b_i$$

L'algoritmo in questione effettua il calcolo di l quadrati, con un costo di $\Theta(n^2)$ per ciascuna operazione di elevamento al quadrato. Pertanto, il costo totale dell'algoritmo è $\Theta(l \cdot n^2)$. Questo algoritmo è uno stream cypher, a differenza di `RSA`, che è invece un block cypher. Nonostante questa differenza nel metodo di cifratura, le complessità computazionali di entrambi gli algoritmi risultano essere simili. La distinzione principale risiede nel fatto che l'algoritmo di Blum Goldwasser gode di una dimostrazione di sicurezza, dimostrabilità che manca invece in `RSA`.

Capitolo 4

Autenticazione

4.1 Introduzione

I crittosistemi costruiti fino ad ora sono malleabili, quando riceviamo un messaggio vorremmo essere sicuri che nessuno l'abbia modificato. Per farlo possiamo usare sistemi non malleabili dove le modifiche non sono prevedibili, oppure aggiungere della ridondanza, aggiungendo bit di parità o un codice ciclico di ridondanza.

Possedendo però un messaggio cifrato e la sua decifratura, o dalla chiave pubblica, è possibile modificare un messaggio cifrato attraverso il non quadrato ricavato, alterando qualsiasi bit in maniera controllata, e quindi modificare in maniera controllata anche i codici di ridondanza.

Per garantire l'integrità e l'autenticità dei messaggi trasmessi, è fondamentale adottare sistemi di autenticazione avanzati. Uno di questi approcci consiste nell'utilizzare una funzione casuale condivisa tra mittente e destinatario.

Quando un mittente invia un messaggio, genera una sequenza di bit aggiuntiva chiamata Message Authentication Code (MAC). Questo MAC viene ottenuto applicando una funzione casuale precedentemente concordata al messaggio stesso. La casualità della funzione è cruciale e deve essere condivisa solo tra le parti interessate, rimanendo sconosciuta agli altri.

Il destinatario, una volta ricevuto il messaggio, applica la stessa funzione casuale al messaggio ricevuto per calcolare un nuovo MAC. Se il MAC calcolato corrisponde a quello ricevuto con il messaggio, ciò indica che il messaggio non è stato modificato in modo controllato e l'autenticazione è riuscita.

L'uso di una funzione casuale impedisce a un potenziale aggressore di prevedere la sequenza di bit aggiuntiva al messaggio. Inoltre, permette di verificare l'autenticità del messaggio, poiché solo il mittente e il destinatario conoscono la funzione casuale utilizzata.

Un possibile attaccante che dispone di un messaggio, non sarà in grado di produrre il codice di autenticazione corretto, poiché il risultato della applicazione di una funzione causale è una sequenza di bit casuale, e quindi non prevedibile e distinguibile.

4.2 Funzioni pseudo-casuali

Avendo imparato a generare bit pseudocasuali, siamo in grado di generare funzioni pseudo-casuali, ovvero funzioni che non sono realmente casuali, ma sono generate da un seme, ma agli occhi di un osservatore esterno sono casuali.

Sia $\mathcal{U}_k$ l'insieme delle funzioni da $\{0,1\}^k$ a $\{0,1\}^k$, vogliamo scegliere f appartenente a $\mathcal{U}_k$ in maniera casuale.

Possiamo generare una sequenza di bit pseudocasuali per generare l'indice di f in una enumerazione di $\{0,1\}^k \to \{0,1\}^k$. Dalla teoria del calcolo combinatorio sappiamo che la cardinalità di delle funzioni da A a B è $|B|^{|A|}$, quindi la cardinalità di $\mathcal{U}_k$ è $\{0,1\}^{k \cdot \{0,1\}^k} = 2^{k \cdot 2^k}$. Per denotare l'indice della funzione avremmo quindi bisogno di $\log_2(2^{k \cdot 2^k}) = k \cdot 2^k$ bit, ovvero una quantità di bit esponenziale, cosa che non ci possiamo permettere.

Ci farebbe comodo denotare una funzione con k bit, il problema è che con i nostri indici dovremmo denotare un insieme di funzioni minori dell'insieme delle funzioni in $\mathcal{U}_k$, dobbiamo quindi lavorare con un insieme $\mathcal{F}_k \subseteq \mathcal{U}_k$. Vogliamo comunque far in modo che chiunque non sia in grado di accorgersi che sia stata scelta una funzione da un insieme più piccolo.

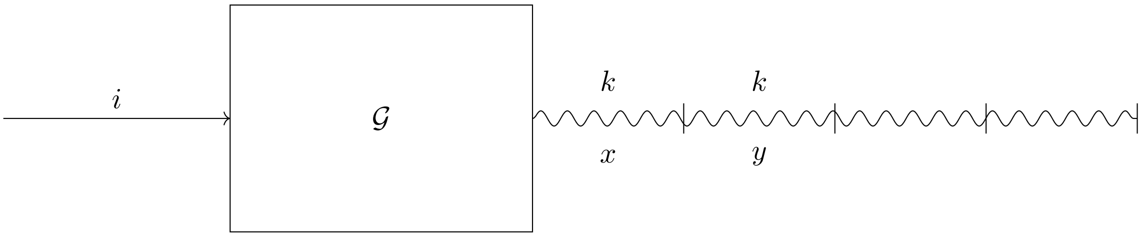
In un generatore di funzioni pseudocasuali, partiamo dall'idea che:

1. Troviamo un insieme di funzioni $\mathcal{F}_k \subseteq \mathcal{U}_k$, tale che $|\mathcal{F}_k| = 2^k$; imponendo per definizione tale dimensione, siamo in grado di generare funzioni casuali avendo a disposizione k bit;
2. Esiste un algoritmo probabilistico polinomiale che calcola la funzione $(i, x) \mapsto f_i(x)$;
3. Sia $\mathcal{D} \in \mathcal{PPT}$ un algoritmo che, su input 1^k (*il security parameter rappresentato in unario, nel valore, ovvero nel numero di bit che usiamo per rappresentare le nostre informazioni*), che interroga f aribitrariamente, anche in maniera adattiva (*scegliendo l'argomento su cui interrogare la funzione f anche sulla base dei risultati ottenuti dalle precedenti interrogazioni, ma sempre in tempo polinomiale*). Siano $\mathcal{P}_k^{\mathcal{D},\mathcal{U}}$ la probabilità che $\mathcal{D}$ restituisca 1 quando $f \in_R \mathcal{U}_k$ e $\mathcal{P}_k^{\mathcal{D},\mathcal{F}}$ la probabilità che $\mathcal{D}$ restituisca 1 quando $f \in_R \mathcal{F}_k$, allora
$$\left|\mathcal{P}_k^{\mathcal{D},\mathcal{U}} - \mathcal{P}_k^{\mathcal{D},\mathcal{F}}\right| \leq k^{-\omega(1)}$$

4.2.1 Costruzione delle funzioni pseudo-casuali

Primo metodo

Sia $\mathcal{G}$ un psudo random sequence generator (`PRSG`), ovvero un algoritmo che genera una sequenza di bit pseudocasuali.



Su input x generiamo k bit e li restituiamo. Oltre a restituirli, ci ricordiamo che x è associata ai primi k bit generati dal `PRSG`. Sul successivo input y se $y = x$ restituiamo il risultato precedente, altrimenti generiamo k bit nuovi e li restituiamo.

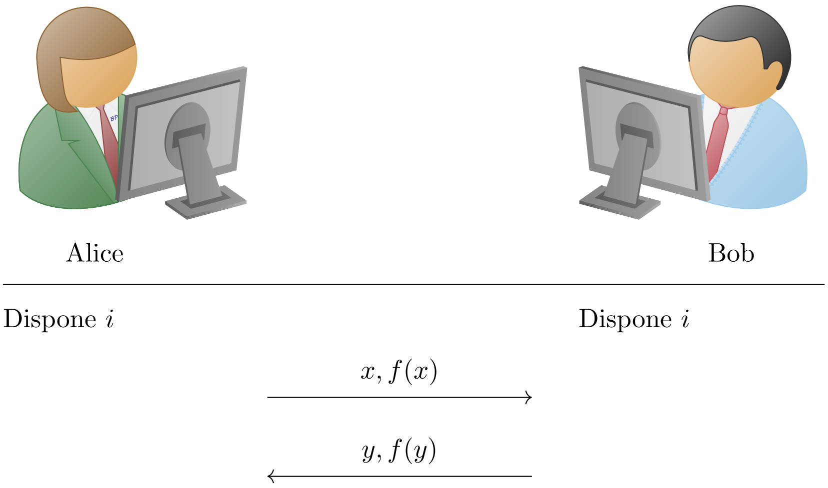
Ogni volta che abbiamo un nuovo input, generiamo k bit nuovi, altrimenti restituiamo il risultato fornito in precedenza. Il problema di fondo abbastanza grosso è che una volta che abbiamo definito il seme i, la funzione è definita in maniera univoca da i? No, perché la funzione dipende dall'ordine in cui viene interrogata. Quindi passando prima x e poi y oppure prima y e poi x potremmo ottenere risultati diversi.

Ma agli occhi di chi osserva, siamo in presenza di una funzione casuale, se scegliamo un numero nuovo vediamo un risultato scelto uniformemente a caso, se scegliamo un numero già visto, vediamo il risultato che abbiamo visto in precedenza.

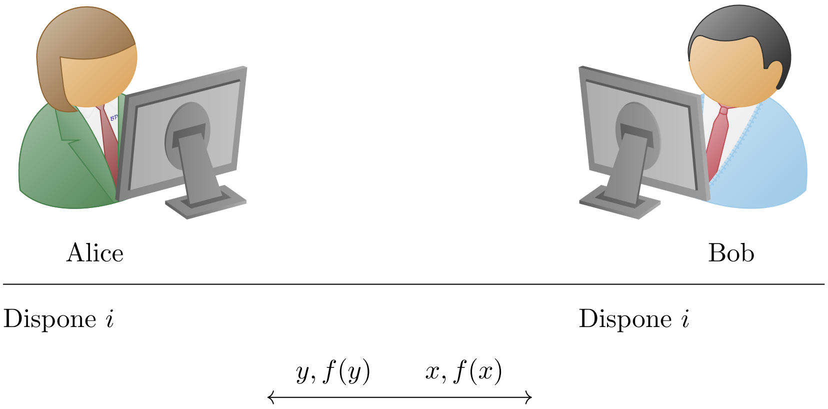
Tuttavia questa costruzione non soddisfa la definizione che abbiamo dato, quindi possiamo far in modo che la definizione ammetta questa costruzione per la generazione di funzioni pseudo-casuali. Ma per la costruzione che abbiamo in mente, ovvero per l'autenticazione di messaggi, questa costruzione non è adatta.

$$m, f(m)$$

Chiunque non conosca f, e veda in $f(m)$ una sequenza casuale, e non sia in grado di calcolare $f(m')$ per $m' \neq m$. L'autenticazione di un messaggio m' è possibile crearla solo con f.



Se le cose vanno esattamente come nel diagramma rappresentato, allora tutto va secondo i piani, poiché ogni agente riesce a verificare l'autenticità di ogni messaggio.

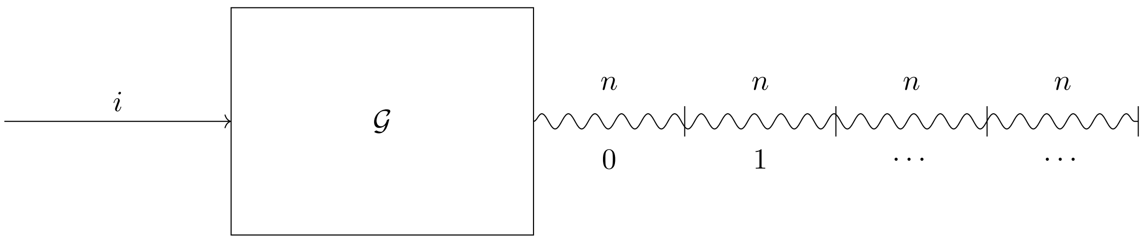


Scambiandosi i messaggi in maniera sincrona, poiché Alice utilizza il generatore e la prima volta che lo invoca, lo invoca su x, mentre Bob lo invoca su y, Alice e Bob si trovano in una situazione in cui non sono in grado di verificare poiché $f(x) = f(y)$.

Nel momento in cui la funzione f dipende dall'ordine di interrogazione, non è possibile distribuire la funzione f a più agenti, e ci vorrebbe un oggetto centralizzato che distribuisce la funzione f a tutti gli agenti.

Tale costruzione è utile in caso di generazione di funzioni pseudocasuali in cui l'utilizzo è locale, tuttavia la crescita di richieste fa si che la dimensione del database cresca e quindi allunghi i tempi di risposta e quindi ci si accorga della pseudocasualità nella generazione.

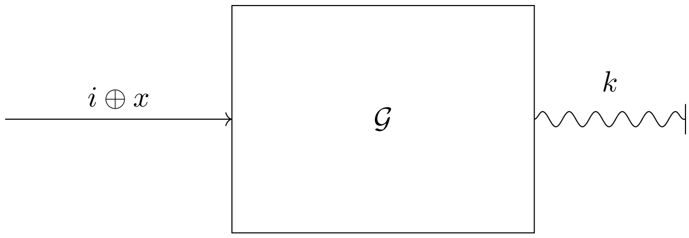
Secondo metodo



Dove f_i è definita come l'*x-esimo gruppo di n bit restituito da $\mathcal{G}$*. In questo caso l'insieme f_i è definito in maniera univoca. Per ogni i c'è una funzione f_i definita in maniera univoca, quindi l'insieme $\mathcal{F}_k$ è l'insieme delle funzioni f_i definite nella maniera appena descritta.

Il problema sorge nella condizione che indica che esiste un algoritmo probabilistico polinomiale tale che $(i, x) \mapsto f_i(x)$. Sia $x = 1...1 = 2^{k-1} - 1$, per come è definito il generatore, per farlo dobbiamo generare tutti i numeri da 0 a $2^{k-1} - 1$, quindi i bit totali da generare sono $k \cdot 2^{k-1}$, quindi il tempo di esecuzione non è polinomiale.

Terzo metodo



Come seme usiamo $i \oplus x$, e prendiamo i primi k bit del risultato. Quindi $f_i(x)$ è definita come i primi k bit di $\mathcal{G}(i \oplus x)$. Sappiamo che ogni indice i determina una funzione f_i definita in maniera univoca. Inoltre l'algoritmo che genera f_i è polinomiale, poiché per generare f_i è sufficiente eseguire lo `XOR` tra i e x e, che è un'operazione polinomiale, e poi generare con il generatore $\mathcal{G}$ i primi k bit del risultato, e sappiamo che $\mathcal{G}$ è un algoritmo polinomiale.

Dobbiamo capire se esiste distinguisher in grado di distinguere f_i da una funzione casuale. Supponiamo che esista $\mathcal{D}$ tale per cui $\left|\mathcal{P}_k^{\mathcal{D},\mathcal{U}} - \mathcal{P}_k^{\mathcal{D},\mathcal{F}}\right| > k^{-\omega(1)}$, vorremmo trasformare $\mathcal{D}$ in un algoritmo che viola la correttezza del generatore $\mathcal{G}$.

Per farlo dobbiamo assicurarci che il distinguisher $\mathcal{D}$ stia facendo implicitamente su $\mathcal{G}$, tutti esperimenti che siano compatibili con quello che abbiamo detto per definire la correttezza di $\mathcal{G}$.

Nel caso specifico il distinguisher $\mathcal{D}$ può scegliere valori distinti di x, ma il generatore $\mathcal{G}$ conosce tali valori $x_1, x_2, \ldots, x_n$, prendendo lo `XOR` di uno dei due valori in input a $\mathcal{G}$, otteniamo lo `XOR` del seme usato in $\mathcal{G}$, allora il distinguisher è in grado di osservare l'output di $\mathcal{G}$ su due semi di cui è nota la differenza.

Se interroghiamo $f(0) = a$ e $f(1^k) = b$, sappiamo che il seme usato per generare a è esattamente il complemento del seme usato per generare b. In generale se $f(x) = a'$ e $f(y) = a''$, allora sappiamo che `XOR` dei semi usati per generare a' e a'' è esattamente $x \oplus y$. Gli esperimenti che il distinguisher riesce a fare implicitamente su $\mathcal{G}$ sono esperimenti che il distinguisher che abbiamo usato per definire la correttezza su $\mathcal{G}$ non è in grado di fare. Quindi il distinguisher che sta lavorando sulle funzioni pseudocasuali è più potente del distinguisher ammesso nella definizione di correttezza di $\mathcal{G}$, quindi abbiamo una contraddizione.

In questo caso i casi possono essere due:

- Abbiamo definito un generatore troppo debole, poiché abbiamo bisogno di una definizione più forte, che ammetta esperimenti di questo tipo;
- Possiamo far vedere che se abbiamo un generatore di bit pseudocasuali, allora tale generatore resiste ad attacchi di questo tipo.

In questo caso è possibile costruire un generatore di bit pseudocasuali $\mathcal{G}$ e che se usato in questo contesto permette di costruire un distinguisher $\mathcal{D}$.

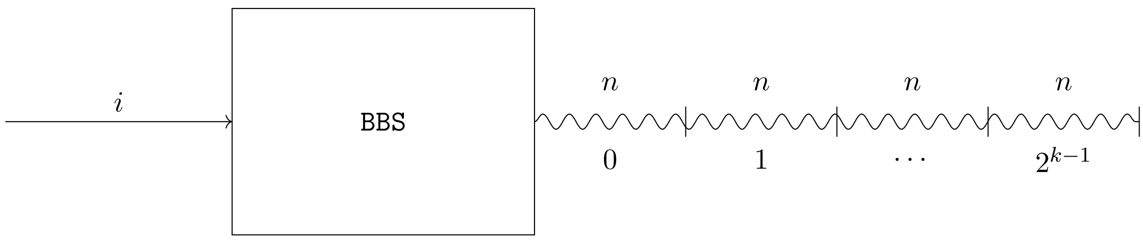
Incapsuliamo il generatore in un generatore più grande $\mathcal{G}'$, che prende in input $i_0, i_1, \ldots, i_{k-1}$. Se $i_0 = 0$ allora si inviano $i_1, i_2, \ldots, i_{k-1}$ al generatore $\mathcal{G}$, altrimenti si invia $\overline{i_1}, \overline{i_2}, \ldots, \overline{i_{k-1}}$ al generatore $\mathcal{G}$. Scegliendo un seme a caso usiamo $\mathcal{G}$ con un seme a caso, poiché il complemento di una sequenza di bit casuali è sempre una sequenza casuale, quindi stiamo generando una sequenza di bit casuali con security parameter $k-1$.

Il nuovo generatore $\mathcal{G}'$ è un generatore di bit pseudocasuali che soddisfa la definizione di correttezza. L'output di $\mathcal{G}'$ è effettivamente un seme distribuito uniformemente tra tutti i possibili semi, e quindi si comporta come $\mathcal{G}$. Tuttavia usando tale oggetto nel contesto riportato sopra, scopriamo che $f(0) = f(1^k)$. Il distinguisher $\mathcal{D}$ in questo caso riusciamo quindi a costruirlo facilmente, poiché basta che controlli se $f(0) = f(1^k)$, e se questo è vero allora $\mathcal{D}$ restituisce 1, altrimenti restituisce 0. Di fronte ad una funzione pseudocasuale risponderà sempre 1, di fronte ad una funzione casuale risponderà 1 solamente quando $f(0) = f(1^k)$, ma tale uguaglianza avviene con probabilità 2^{-k}, perché è la probabilità che due sequenze di k bit siamo uguali, la differenza essendo enorme distingue con probabilità 1.

La soluzione alla quale vogliamo ambire è una soluzione che cercherà di evitare il problema.

Quarto metodo

Torniamo al secondo metodo, dove $f_i(x)$ è l'$x-esimo$ gruppo di $\mathcal{G}$, eravamo rimasti al fatto che non riuscissimo a calcolare $f_i(x)$ in tempo polinomiale. Per poter calcolare $f_i(x)$ in tempo polinomiale, usiamo un generatore di Blum-Blum-Shub, lavorando con il logaritmo discreto non siamo in grado di calcolare elementi molto distanti, ma se dobbiamo calcolare $f_i(2^k - 1)$ servono solamente i bit dalla posizione $k \cdot (2^k - 1) - 1$ a $k \cdot k \cdot 2^k$.



Per calcolare tali bit, avendo a disposizione il seme i, per calcolare il bit $k \cdot (2^k - 1) - 1$ dobbiamo calcolare $i^{k \cdot (2^k - 1) - 1}$, che è abbastanza complicato da calcolare. Tuttavia gli esponenti lavorano in modulo $\phi(n)$ e una volta calcolato l'esponente possiamo lavorare in modulo n, quindi possiamo calcolare $i^{(k \cdot (2^k - 1) - 1) \mod \phi(n)} \mod n$, in tempo polinomiale k^3 dove k è il numero di bit. Di conseguenza l'oggetto è calcolabile in tempo polinomiale, quindi abbiamo un algoritmo `PPT` che è in grado di calcolare $f_i(x)$.

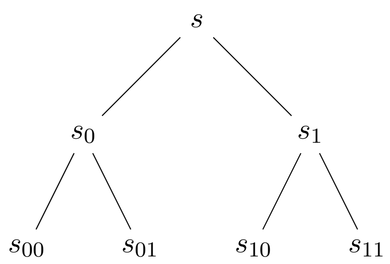
Questa operazione non è possibile con tutti i generatori, il generatore di Blum-Blum-Shub lavora in un gruppo di cardinalità finita ed ha una struttura particolare che permette di fare questo tipo di operazioni.

Dobbiamo vedere se tale oggetto soddisfa la terza proprietà di un oggetto pseudocasuale, ovvero che sia indistinguibile da una funzione realmente casuale. Il distinguisher $\mathcal{D}$ interroga la funzione f e nell'interrogare la funzione f sta implicitamente facendo degli esperimenti su $\mathcal{G}$, ma gli esperimenti possono essere fatti in tempo polinomiale, quindi un numero esponenziale di bit non può essere vista dal distinguisher. Quindi con il generatore di funzioni pseudocasuali siamo in grado di vedere dei bit che il distinguisher non è in grado di vedere.

In questa situazione però non siamo riusciti a costruire un controesempio.

Quinto metodo

Usiamo l'indice della funzione i di k bit, come seme e di base usiamo un generatore pseudocasuale $\mathcal{G}$, per generare $2k$ bit, $\mathcal{G}(s)[0,\dots,k-1]$ ovvero s_0 e lo stesso per $\mathcal{G}(s)[k,\dots,2k-1]$ ovvero s_1. e ripetiamo ricorsivamente il procedimento generando s_{00}, s_{01}, s_{10} e s_{11}, che sono rispettivamente $s_0[0,\dots,k-1], s_0[k,\dots,2k-1], s_1[0,\dots,k-1]$.



Scendendo nell'albero, arriviamo al seme indicizzato da x, ovvero s_x, che è un seme di k bit, che è il risultato della funzione $f_i(x)$. Ovviamente questo albero ha 2^k nodi e le foglie rappresentano i valori delle funzioni sui diversi argomenti a partire da s in radice, quindi la funzione $f_s(x)$ ovvero $f_i(x)$ è ben definita.

Per calcolare tale funzione in tempo polinomiale, dobbiamo calcolare calcolare solamente un cammino dell'albero, quindi il tempo di calcolo è polinomiale. Infatti per calcolare un nodo dell'albero nel caso pessimo calcoliamo $2k$ bit per k volte a cui si moltiplica il tempo di generazione di $2k$ bit dal generatore di Blum-Blum-Shub, che esegue un elevamento al quadrato per ogni bit, quindi il tempo di calcolo è $\theta(k^4)$.

Dobbiamo verificare che non esista un distinguisher $\mathcal{D}$ che sia in grado di distinguere la funzione $f_s(x)$ da una funzione realmente casuale. Creiamo un distinguisher $\mathcal{D}$ che interroga la funzione $f_s(x)$; implicitamente sta facendo degli esperimenti sul generatore $\mathcal{G}$, in maniera nidificata. Guardando l'albero a livelli, un generatore pseudocasuale trasforma una certa quantità di casualità in una quantità di casualità diversa.

Definiamo gli alberi:

$$\begin{array}{lcllllllll} \mathcal{T}_0 & = & s^1 & \dots & s^{i-1} & s^i & s^{i+1} & s^{i+2} & \dots & s^l \\ \mathcal{T}_l & = & r^1 & \dots & r^{i-1} & r^i & r^{i+1} & r^{i+2} & \dots & r^l \end{array}$$

Dove $\mathcal{T}_0$ è l'albero che rappresenta esattamente l'albero definito per la generazione di funzioni pseudocasuali, mentre $\mathcal{T}_l$ corrispondente ad una funzione casuale campionata uniformemente dell'insieme $\mathcal{U}_k$.

Il distinguisher secondo la nostra ipotesi vede la differenza tra un albero costruito secondo la prima tecnica e un albero costruito secondo la seconda.

$$\left|\mathcal{P}_k^{\mathcal{D},\mathcal{G}} - \mathcal{P}_k^{\mathcal{D},\mathcal{U}}\right| > k^{-c} \qquad \text{quindi} \qquad \left|\mathcal{P}(0) - \mathcal{P}(k)\right| > k^{-c}$$

In questo caso costruiamo tutti gli alberi intermedi dove per ogni livello $i = 0, \ldots, k$ definiamo l'albero $\mathcal{T}_i$ tale che:

- I nodi dei livelli $0, \ldots, i$ sono tutti casuali;
- I nodi dei livelli $i + 1, \ldots, k$ sono tutti pseudocasuali.

Con la solita tecnica di interpolazione, possiamo affermare che se il distinguisher è in grado di rilevare la differenza tra un albero costruito secondo lo schema $\mathcal{T}_0$ e un albero costruito secondo lo schema $\mathcal{T}_k$, allora può individuare la differenza tra livelli adiacenti mediante interpolazione. Poiché due livelli adiacenti differiscono solamente per un livello specifico, otteniamo due alberi distinti che si differenziano solo per quel livello particolare. Pertanto, calcolando $f_s(x)$ risulterebbe equivalente a retrocedere al nodo i-esimo dell'albero $\mathcal{T}$ e osservarlo casualmente in un caso e pseudocasualmente nell'altro, con un seme generato al livello precedente in maniera veramente casuale. L'esperimento di calcolare $f_s(x)$ è quindi il classico test su un generatore di bit pseudocasuali.

$$\exists i \in \{0, \ldots, k-1\} \quad \left|\mathcal{P}(i) - \mathcal{P}(i+1)\right| > k^{-c'}$$

Effettuando un esperimento su $f_s(y)$ che proviene dall'istanza dello stesso nodo i-esimo dell'albero $\mathcal{T}$, possiamo considerarlo come un'istanza dello stesso esperimento, ossia un esperimento sui bit di livello i casuali o pseudocasuali. Se l'esperimento è condotto su $f_s(z)$ proveniente da una nuova istanza di un nodo i-esimo dell'albero $\mathcal{T}$, allora può essere considerato un nuovo esperimento indipendente, consentendo la realizzazione di più esperimenti indipendenti.

Se la probabilità di successo con un singolo esperimento è inferiore a qualsiasi polinomio, la probabilità di successo con un numero polinomiale di esperimenti indipendenti è comunque inferiore a qualsiasi polinomio.

Di conseguenza un esistenza di un attaccante al generatore di funzioni si traduce in un attaccante al generatore di bit pseudocasuali $\mathcal{G}$, quindi il distinguisher non esiste se assumiamo che il generatore di partenza soddisfi la definizione.

4.3 Autenticazione

Ci sono due o più agenti che si scambiano messaggi, non necessariamente orientati alla segretezza dei messaggi, ma vogliono essere sicuri che i messaggi siano creati solo da loro e non da altri.

Una possibilità è quella di scegliere una funzione pseudocasuale f da condividere tra gli agenti, la funzione può essere generata con un qualsiasi schema, per esempio lo schema precedente che sfrutta l'utilizzo di un seme, e questo seme viene scambiato tra gli agenti attraverso un canale sicuro. Ad ogni messaggio m si aggiunge il valore $f(m)$, spedendo quindi la coppia $(m, f(m))$, in altri termini $f(m)$ è l'autenticazione di m, non mandando quindi solo il messaggio m, ma anche la sequenza di bit che è l'autenticazione di m.

La funzione è pseudocasuale, di conseguenza chiunque non conosca il seme di partenza vede di fianco ad ogni messaggio una sequenza di k bit scelta uniformemente a tra le sequenze di k bit. Qualunque agente che conosce il seme può calcolare autonomamente $f(m)$ e confrontarlo con il secondo elemento della coppia, se sono uguali allora il messaggio è autentico, altrimenti non è autentico.

4.3.1 Forger

Sia $\mathcal{F}$ un **forger**, ovvero un algoritmo probabilistico polinomiale che interroga f su una sequenza di messaggi $m_1, m_2, \ldots, m_l$, anche in modo adattivo e alla fine restituisce una coppia (m, σ), diciamo che $\mathcal{F}$ ha successo se $f(m) = \sigma$ e $\forall i\, m \neq m_i$, quindi restituisce una coppia (m, σ) dove m non è mai stato interrogato e σ è l'autenticazione di m.

$$\forall \mathcal{F} \in \texttt{PPT} \quad \mathcal{P}\Big[\text{Succ}(\mathcal{F}) = 1\Big] \leq k^{-\omega(1)}$$

4.3.2 Autenticare messaggi lunghi

L'idea è quella di dividere il messaggio in blocchi di lunghezza k, quindi $m = m_1, \ldots, m_l$, possiamo cercare di creare l'autenticazione applicando la funzione f a ogni blocco, quindi $f(m_1), \ldots, f(m_l)$ e definire il messaggio m come:

$$m = m_1 \oplus f(m_1) \oplus f(m_2) \oplus \cdots \oplus f(m_l)$$

Il problema è che il forger in questo caso può combinare i blocchi in modo diverso, quindi può creare un messaggio m' che è diverso da m, ma che ha la stessa autenticazione. Infatti chiedendo la coppia $(m_1m_2, f(m_1m_2))$, la coppia $(m_2m_1, f(m_2m_1))$ è una coppia autentica.

Costruire un messaggio autenticato senza interrogare la funzione f in questo caso è molto semplice, infatti costruendo la coppia $(m_1m_1, f(m_1m_1))$, lo `XOR` di due messaggi uguali è sempre 0, e dal punto di vista della definizione abbiamo trovato il forger, quindi abbiamo la necessità di utilizzare un meccanismo più complesso per autenticare messaggi lunghi vietando inoltre lo scambio dei blocchi. Per farlo la posizione di ogni singolo blocco deve avere un impatto sulla funzione f.

$$f(m) = f(\langle 1\rangle m_1) \oplus f(\langle 2\rangle m_2) \oplus \cdots \oplus f(\langle l\rangle m_l)$$

Dove $\langle i\rangle$ è una stringa di pochi bit che rappresenta la posizione del blocco m_i. Purtroppo però anche con questo sistema il forger può creare un messaggio autentico in modo semplice:

$$\begin{array}{rcl}
f(m_1m_2) & = & f(\langle 1\rangle m_1) \oplus f(\langle 2\rangle m_2) \\
f(m_1'm_2) & = & f(\langle 1\rangle m_1') \oplus f(\langle 2\rangle m_2) \\
f(m_1m_2') & = & f(\langle 1\rangle m_1) \oplus f(\langle 2\rangle m_2') \\
\hline
f(m_1'm_2') & = & f(\langle 1\rangle m_1') \oplus f(\langle 2\rangle m_2')
\end{array}$$

L'idea per risolvere il problema potrebbe essere l'applicazione nidificata della funzione f in stile cypher block chaining, ovvero:

$$f'(m_1 \dots m_l) = f(m_l \oplus \dots \oplus f(m_3 \oplus f(m_2 \oplus f(m_1))))$$

Questo sistema è dimostrabilmente sicuro; difatti, ciò che viene realizzato è la costruzione di una nuova funzione $f' : \{0,1\}^{kl} \to \{0,1\}^k$, partendo da una funzione pseudocasuale di k bit in k bit, che incrementa la dimensione dell'input. Possiamo concepire questo concetto come un'estensione dell'idea di generazione di funzioni pseudocasuali. Supponiamo di avere una funzione scelta in modo uniforme tra le funzioni da $\{0,1\}^k$ a $\{0,1\}^k$, oppure una funzione da kl bit a k bit, selezionata da un insieme di dimensione più ridotta, poiché dipende dallo stesso seme utilizzato per f. Il nostro obiettivo è affermare che questo sistema è sicuro nel senso che non esiste alcun algoritmo probabilistico polinomiale in grado di distinguere una funzione pseudocasuale da una funzione casuale.

L'eventuale forger può essere convertito in un distinguisher; pertanto, affermare che si tratta di un sistema di autenticazione o di una funzione pseudocasuale da kl bit a k bit è equivalente, quindi l'esistenza di un forger implica l'esistenza di un distinguisher, e viceversa.

Dimostriamo quindi che la costruzione della funzione f' è una costruzione di una funzione pseudocasuale da kl bit a k bit, partendo da una funzione pseudocasuale da k bit a k bit, e dimostriamo che è una funzione pseudocasuale nel senso delle definizioni date sulle funzioni pseudocasuali, ovvero non esiste alcun algoritmo probabilistico polinomiale in grado di distinguere una funzione pseudocasuale da una funzione casuale.

Per farlo costruiamo $\mathcal{F}_i(m_1, \dots, m_i)$ prende i primi i blocchi del messaggio e applica la funzione f pseudocasuale a questi blocchi e per i successivi $l-i$ blocchi applica la funzione r casuale, quindi:

$$\mathcal{F}(m_1 \dots m_l) = r(m_l \oplus \dots \oplus r(m_{i+1} \oplus f(m_i \oplus \dots \oplus f(m_3 \oplus f(m_2 \oplus f(m_1))))))$$

A questo punto, sapendo che f_0 è sempre casuale in tutti i punti e f_1 è pseudocasuale in tutti i punti, sapendo distinguere i due estremi possiamo, per interpolazione, distinguere anche i punti intermedi. Sappiamo che $\mathcal{P}(i)$ è la probabilità che il distinguisher restituisca 1 se la funzione di autenticazione è la funzione $\mathcal{F}_i$, allora:

$$\left|\mathcal{P}(i) - \mathcal{P}(i+1)\right| > k^{-c}$$

Quindi distinguere sul livello i-esimo e il livello $i+1$-esimo si traduce in una distinzione nell'utilizzo di una funzione pseudocasuale e una funzione casuale, quindi:

$$\exists i \in \{0, \dots, k-1\} \quad \left|\mathcal{P}(i) - \mathcal{P}(i+1)\right| > k^{-c'}$$

Diventando quindi un distinguisher per la funzione pseudocasuale f.

4.3.3 Problema della non ripudiabilità

Vorremmo avere a disposizione un sistema che permetta di autenticare un messaggio, in modo tale che il mittente non possa negare di averlo inviato, poiché con le tecniche utilizzate fino ad ora i messaggi potrebbero essere stati creati da chiunque. Tale problema è legato al concetto di firma digitale, chi firma il messaggio non può negare di averlo firmato, in quanto la firma è univoca e non può essere riprodotta da nessun altro.

In sostanza, solo un agente può firmare un messaggio, e tutti possono verificare la firma. Chi può verificare non può firmare, nel momento in cui la firma è stata apposta non può essere contestata. Devono quindi esistere due chiavi, una chiave privata per firmare e una chiave pubblica per verificare, e dalla chiave pubblica non deve essere possibile risalire alla chiave privata.

`RSA` nella firma digitale

L'algoritmo `RSA` può essere utilizzato per la firma digitale, in quanto è un algoritmo a chiave pubblica e privata, e la chiave pubblica non può essere utilizzata per risalire alla chiave privata.

In `RSA` classico, il messaggio da criptare si eleva alla potenza e e si prende il modulo n, e per decriptare si eleva alla potenza d e si prende il modulo n. Per il teorema di Eulero, n elevato alla cardinalità del gruppo fornisce l'unità, quindi $m^{\phi(n)} \equiv 1 \mod n$, quindi $m^{k\phi(n)+1} \equiv m \mod n$.

Per firmare un messaggio m si calcola $m^d \mod n$, e per verificare la firma si calcola $(m^d)^e \mod n$, che è uguale a m. Solamente chi possiede la chiave privata può firmare, mentre la verifica può essere effettuata da chiunque.

$$(m^d)^e \equiv m^{de} \equiv m^{k\phi(n)+1} \equiv m \mod n$$

4.3.4 Definizione di sistema di firma digitale

Un sistema di firma digitale è una tripla di algoritmi $(\mathsf{Gen}, \mathsf{Sign}, \mathsf{Ver})$ probabilistici polinomiali.

- $\mathcal{G}$ è un algoritmo probabilistico polinomiale che prende in input 1^n (*poiché l'algoritmo deve essere polinomiale nel valore del security parameter, ovvero il numero di bit che usiamo per rappresentare le nostre chiavi, visto che sulle macchine di Turing la dimensione del problema è pari al numero di celle che usiamo nel nastro di input*) che produce una coppia di chiavi (p_k, s_k);
- $\mathcal{S}$ è un algoritmo probabilistico polinomiale che prende in input una chiave privata s_k e il messaggio m e produce una firma σ;
- $\mathcal{V}$ è un algoritmo probabilistico polinomiale che prende in input una chiave pubblica p_k, e la firma σ e restituisce 1 se la firma è valida, 0 altrimenti.

La probabilità di verificare una firma costruito secondo lo schema di firma digitale è 1.

$$\forall \mathcal{P}\left[\mathcal{V}(p_k, \mathcal{S}(s_k, m)) = 1\right] = 1$$

Si parla di probabilità perché l'algoritmo di verifica potrebbe essere un algoritmo probabilistico polinomiale, e di conseguenza sbagliare, ma la probabilità che questo accada è 0.

Sia $\mathcal{F} \in \texttt{PPT}$ un algoritmo che chiede la firma di messaggi $m_0, m_1, \ldots, m_l$, anche in modo adattivo, e che produce una coppia (m, σ), dove m è il messaggio che viene firmato e σ ed $\mathcal{F}$ ha successo se $\mathcal{V}(p_k, \sigma) = 1$ e $m \notin \{m_0, m_1, \ldots, m_l\}$.

$$\forall \mathcal{F} \in \texttt{PPT} \qquad \mathcal{P}\left[\mathcal{F} \text{ ha successo}\right] \leq k^{-\omega(1)}$$

Se $\mathcal{F}$ non riesce a produrre una firma falsa avendo a disposizione la chiave pubblica, è evidente che chiunque abbia la chiave pubblica non può produrre firme; non è possibile ricavare la chiave privata, perché l'unico modo che conosciamo per produrre una firma è mediante l'algoritmo $\mathcal{S}$, che prende in input la chiave privata e di conseguenza un crittosistema di firma digitale che soddisfa queste definizioni è un sistema che soddisfa la proprietà di non ripudiabilità.

`RSA` però non è un buon sistema di firma digitale, nel senso che non soddisfa la definizione data, infatti non è vero che ogni forger ha una probabilità di successo trascurabile, infatti è possibile forgiare messaggi, dati (m_1, m_1^d) e (m_2, m_2^d), creando un nuovo messaggio $(m_1 m_2, (m_1 m_2)^d)$. Chiaro che inserendo sufficiente ridondanza ai messaggi in modo tale che molto difficilmente $m_1 m_2$ sia un messaggio significativo questi attacchi non producono nulla.

4.3.5 Firma di messaggi lunghi