Crittografia

Alessandro Pioggia, Luca Rengo, Federico Brunelli, Leon Baiocchi

1 marzo 2022

Indice

Capitolo 1

Introduzione

Impariamo due nozioni di crittografia in caso servano per le comunicazioni della terza guerra mondiale.

Capitolo 2

One-time pad

Un cifrario è perfetto se, una volta scelto il cifrario (o progettato), occorre valutare la sua sicurezza (quanto è resistente agli attacchi). Abbiamo due metodi classici che ci permettono di capire se un cifrario è robusto. Ovvero

- Protocollo a chiave segreta (metodo empirico) : si distribuisce agli scienziati il cifrario, se dopo tot tempo nessuno lo viola significa che è sicuro. Probabilmente è sicuro però, sarebbe più opportuna una dimostrazione pratica che dimostra che il protocollo non è violabile (staremmo più tranquilli).
- (metodo formale) : Dimostro che, quel protocollo crittografico non è sicuro, però per violarlo è necessario disporre di una determinata quantità di risorse (tempo di calcolo). Se le risorse sono inarrivabili, il protocollo è sicuro ed è migliore del metodo empirico esposto nel punto precedente.

Il one-time-pad è un esempio di cifrario perfetto, in cui riusciremo a dimostrare (cosa rara nei cifrari), che in determinate situazioni, il cifrario non è violabile, con qualsiasi quantità di risorse a disposizione. Problema : è troppo costoso, è quasi inutilizzabile, tranne in certi casi specifici (molto ristretto). Introduce però alcuni concetti che saranno utili nel proseguo del corso.

2.1 Esistono cifrari inviolabili?

Inviolabile = qualcuno intercettando il crittogramma che passa sul canale di comunicazione, riesce a risalire al messaggio? Anche conoscendo le funzioni di cifratura/decifratura? Considerando che l'unica cosa che non conosciamo è la chiave segreta.
La risposta è : esistono cifrari inviolabili, come ad esempio il one-time-pad ma l'impiego è quasi impossibile per via del loro altissimo costo. Dunque fra i cifrari che si utilizzano in pratica non esistono quelli inviolabili, si utilizzano invece cifrari sicuri ed economici e che vengono utilizzati ogni giorno per comunicazione di massa(acquistiamo efficienza e perdiamo sicurezza).

2.2 Processo stocastico (Modello matematico)

Abbiamo una comunicazione tra mittente (Mitt) e destinatario (Dest), questa comunicazione è modellata come un processo stocastico (ovvero probabilistico), in cui il comportamento del mittente è descritto da una variabile aleatoria M (assume certi valori con una determinata probabilità). Valori assunti da M = possibili messaggi. I possibili messaggi vengono presi dallo spazio dei messaggi (msg). Abbiamo un canale, dove viaggia il crittogramma corrispondente al

messaggio dopo averlo codificato e sul canale abbiamo una variabile aleatoria C, che assume i valori dei crittogrammi generati a partire dai messaggi. In sintesi :

- M : variabile aleatoria;
- m : valore assunto dalla variabile aleatoria in un tempo t;
- C : processo che trasforma il messaggio;
- c : crittogramma generato a partire dal messaggio, C(m) = c. il crittogramma è un valore possibile del canale. Quando m viene codificato con C, sul canale leggo c.

La distribuzione di probabilità della M dipende dalla sorgente, chi è la sorgente? la sorgente è colui che genera il messaggio (tira fuori i bit, che vengono organizzati in blocchi e poi spenditi sul canale).

Nota : Non è possibile che due messaggi uguali generino due c diverse, allo stesso modo non è possibile che due messaggi diversi abbiano uno stesso c, dal momento che in quel caso il destinatario non sarebbe in grado di ottenere con precisione il messaggio in seguito alla codifica.

Esempio applicativo : Il mittente è il signor dado, che vuole spedire un messaggio al destinatario. I messaggi che spedisce il signor dado sono $M = \{1, 2, 3, 4, 5, 6\}$. Il modello matematico dice che io spedisco $1conp = 0.5; (2, 4, 5, 6)conp = 0.1$ (questa è la distribuzione di probabilità del nostro mittente dado). Quando dado vuole spedire 4, prende la sua codifica e la cifra, $C(4)$, che genererà un crittogramma c che poi verrà spedito sul canale. Chi guarda il canale (ad esempio l'intruso), prima che passi c, sa già che il messaggio 4 esce con probabilità 0.1, stessa cosa per ogni valore. Ora supponiamo di aver letto c, noi vogliamo che conoscendo c la probabilità che il messaggio fosse 4 rimanga sempre 0.1, questo vuol dire che l'informazione rivelata dall'evento "ho letto c" è nulla, non serve a nulla. Un cifrario è perfetto se la probabilità iniziale rimane inalterata anche se viene visto ciò che è passato sul canale di comunicazione.

2.2.1 Notazione

$P(M = m) =$ { con quale probabilità il mittente vuole spedire il messaggio m } .
$P(M = m|C = c) =$ { con che probabilità il mittente ha spedito il messaggio m, sapendo che sul canale abbiamo letto il crittogramma c. }

Deduciamo che Se le due probabilità sono uguali, abbiamo che la lettura del crittogramma c non mi ha dato nessuna informazione riguardo il messaggio che è stato spedito (concetto di indipendenza : $P(A|B) = P(A)$).

2.3 Scenario ipotetico

Il crittoanalista conosce la distribuzione di probabilità con cui il mittente genera il messaggio, il cifrario e lo spazio k delle chiavi, l'unica cosa che non conosce è la chiave che è stata utilizzata (la chiave segreta).
Secondo Shannon il cifrario è perfetto quando, per ogni $m \in Msg$ e $c \in Crittogrammi$, vale la relazione : $P(M = m|C = c) = P(M = m)$. Supponendo di avere messaggi composti dalla coppia nome+cognome, non mi basta che l'intruso non sappia nè nome nè cognome, non voglio nemmeno che l'intruso scopra se l'utente sia maschio o femmina, non voglio che vengano capite nemmeno caratteristiche legate al messaggio.

2.3.1 Descrizione scenario

Supponiamo di avere dei messaggi contenenti la coppia <nome, cognome> e che tutte le coppie siano equiprobabili. Supponiamo inoltre che sulla terra ci siano 1000 coppie <nome, cognome>. Supponiamo inoltre che sul pianeta terra ci siano 300 maschi e 700 femmine. Senza leggere nulla sul canale quale è la probabilità che sia spedito un certo <nome, cognome>? 1/1000. Se io invece leggo il crittogramma c e riesco a capire che si tratti di una femmina, la probabilità varia! Perchè la probabilità che il nome contenuto nel messaggio sia <Luigi, Rossi> diventa pari a 0, perchè Luigi è un nome maschile. In un contesto perfetto, questo non deve accadere.

2.4 Numero di chiavi in un cifrario perfetto

La prima dimostrazione formale la si può fare sulla base che: In un cifrario perfetto, il numero delle chiavi possibili deve essere maggiore o uguale al numero dei messaggi possibili.

Vuol dire che se io voglio spedire messaggi lunghi 100 bit, noi sappiamo che i messaggi lunghi 100 bit sono 2^{100}. Ora dimostriamo che, se il cifrario è perfetto, allora io ho bisogno di almeno 2^{100} chiavi. In sostanza le chiavi devono cifrare i messaggi, nascondendo tutte le possibili informazioni, di conseguenza ne servono tante.

2.4.1 Dimostrazione formale

Sia N_m il numero di messaggi possibili, cioè tali che $P(M = m) > 0$ e sia N_k il numero di chiavi possibili. Poniamo per assurdo che $N_m > N_k$, se questo è vero, esiste un crittogramma c con $P(C = c) > 0$ (ovvero con probabilità che passi sul canale) a cui corrispondono $s \leq N_k$ messaggi (non necessariamente distinti), ottenuti decrittando c con tutte le possibili chiavi. Poichè $N_m \geq N_k \geq s$, esiste almeno un messaggio m con $P(M = m|C = c) = 0 \neq P(M = m)$, ovvero per $N_m > N_k$ il cifrario **non** è perfetto.

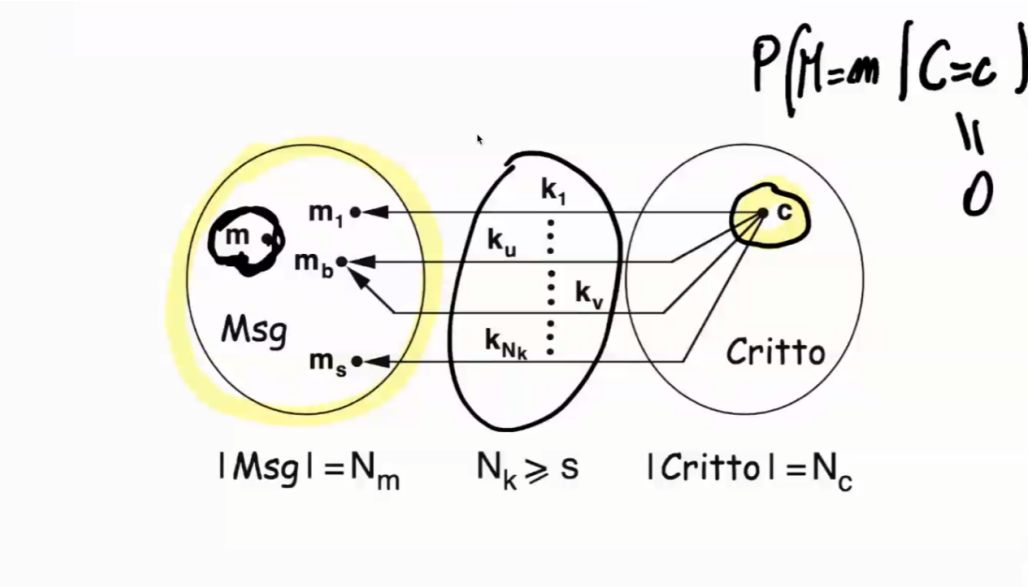


Figura 2.1: Nell'immagine viene esplicitato il ragionamento, si nota che m rimane da solo, in assenza di chiavi con cui crittografare il messaggio. Questo comporta che la $P(M = m)$ non è indipendente rispetto $P(M = m|C = c)$, dunque non si tratta di un cifrario perfetto.

2.5 Deduzione logica

Conseguentemente alla dimostrazione, otteniamo che, in un cifrario perfetto, il numero delle chiavi deve essere maggiore o uguale al numero dei messaggi possibili. Questo ci dice che i cifrari perfetti sono inutilizzabili, perchè in questo caso, se devo spedire 1 gb di messaggio, ho necessità di 1 gb di chiave ed è molto oneroso. Ci dice inoltre che se le chiavi si consumano, un messaggio di 56 bit, una volta spedito, consuma i 56 bit di chiave, dunque non può essere riutilizzata. Dal momento che le chiavi più sicure sono quelle random, inizia a maneggiare in noi il fatto che la randomness sia una risorsa, dunque avere bit a disposizione rappresenta una risorsa (e se usiamo, come in questo caso, chiavi usa e getta, la perdiamo, il cifrario diventa poco robusto). .

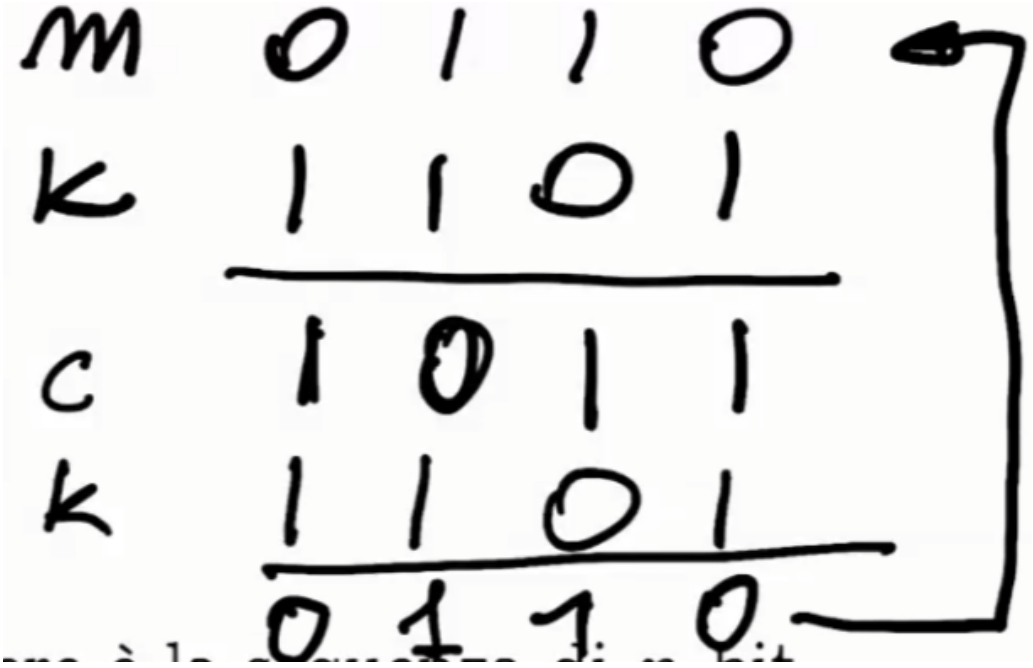
2.6 One-Time Pad

L'one-time-pad veniva usato durante la guerra fredda (Washington e Mosca), perchè la comunicazione non era così frequente (e le chiavi non si consumavano così in fretta), più si riciclano le chiavi, più il protocollo è imperfetto.

2.6.1 Funzionamento

Si costruisce una chiave segreta: Si costruisce una sequenza di bit $k = k_1, k_2, ..., k_n$, nota al mittente e al destinatario. Supponiamo di voler inviare un messaggio più corto di questa chiave e che la chiave sia scelta a caso ((p bit a 0)/(p bit a 1) = 0.5).

Cifratura: Il messaggio da trasmettere $m = m_1, m_2, ..., m_n$, il crittogramma $c = c_1, c_2, ..., c_n$ lo si ottiene eseguendo uno xor fra m e k, per ogni i.
Decifratura : Riapplicando lo xor al c risultante riottengo il messaggio originale.



Dove sta il trucco?? Sta nel fatto che i bit di k, essendo generati a caso, cancellano l'informazione contenuta nel messaggio (dunque un intruso cosa può dire del messaggio? nulla, la chiave lo ha obliterato).

2.6.2 Dimostrazione perfezione One time pad

Per semplicità assumiamo che:

- tutti i messaggi abbiano la stessa lunghezza n;
- tutte le sequenze di n bit sono messaggi possibili (messaggi possibili = tutte le possibili permutazioni della sequenza di bit).

Enunciato

Utilizzando una chiave scelta totalmente a caso per ogni messaggio, il cifrario One-Time pad è perfetto ed impiega un numero minimo di chiavi.

Dimostrazione

Per dimostrare l'enunciato è sufficiente dimostrare che $P(M = m|C = c) = P(M = m)$. E' possibile farlo applicando il teorema di bayes, per il quale :

$$P(M = m|C = c) = \frac{P(M = m, C = c)}{P(C = c)} = \frac{P(C = c|M = m)P(M = m)}{P(C = c)}$$

Osserviamo ora che l'evento $\{M = m; C = c\}$ descrive la situazione in cui Mitt ha generato il messaggio m e l'ha cifrato come crittogramma c. Per la definizione di XOR, fissato il messaggio, chiavi diverse danno origine a crittogrammi diversi, e ogni chiave può essere generata, in maniera randomica, con probabilità $(\frac{1}{2})^n$ (questo perchè o il bit è 0 o 1, per una sequenza di n elementi ed è random). Dunque, fissato m, risulta $P(C = c) = (\frac{1}{2})^n$ per ogni c, ovvero una costante, questo comporta che i due eventi siano indipendenti. Di conseguenza si giunge alla conclusione che (seguendo tutti i passaggi):

$$P(M = m|C = c) = \frac{P(C = c|M = m)P(M = m)}{P(C = c)} = \frac{P(C = c)P(M = m)}{P(C = c)} = P(M = m).$$

Inoltre è implicitamente dimostrato che Il numero di chiavi non può essere inferiore al numero di messaggi. Nel One-Time Pad il numero di chiavi è 2n che è uguale al numero di messaggi (non ne possiamo avere di meno).

Capitolo 3

Il caso

La randomness è la regina della crittografia, è vista come una risorsa, costa e richiede energie per produrla : più la randomness è randomica, più i protocolli sono sicuri.

3.1 Bit casuale

Un bit casuale è un qualcosa che può assumere valore 0 o 1 con probabilità $\frac{1}{2}$, questa definizione ci porta però a scontrarci con questa situazione:

- sequenza 1 : 11111111
- sequenza 2 : 10100100

Notiamo che la prima sequenza non è casuale mentre la seconda sì (per la legge dei grandi numeri la prima è molto improbabile che sia casuale, specie se estesa ad un numero di cifre alto). Ma secondo la prima definizione che abbiamo dato considera che $P(11111111) = P(10100100) = \frac{1}{2^{10}}$, il che non è in linea con il nostro ragionamento.

3.2 Casualità in crittografia

In crittografia casuale = "non facilmente prevedibile", un ipotetico avversario (o intruso) non deve essere in grado di prevedere ciò che noi generiamo a caso.

3.3 Casualità secondo Laplace

Laplace osserva che se tu lanci 100 volte una moneta, le combinazioni di risultati (T o M), formano sequenze regolari, facili da comprendere oppure sequenze irregolari che sono incomparabilmente più numerose (ci sono più combinazioni irregolari rispetto a quelle regolari).

3.4 Casualità secondo Kolmogorov

Kolmogorov dice che una sequenza binaria h è casuale se non ammette alcun algoritmo di generazione A la cui rappresentazione binaria sia più corta di h. Se noi prendessimo le cifre dopo la virgola di π notiamo che queste sono abbastanza irregolari. Noi possiamo però creare un programma che, a seconda della complessità, mi può restituire le cifre di π sempre più grandi (il programma però ha una lunghezza finita!). Secondo Kolmogorov quindi, la generazione di π non

è casuale, le cifre dopo la virgola è possibile determinarle in maniera assoluta da un programma di lunghezza costante. Kolmogorov introduce il concetto di complessità computazionale, se ho un algoritmo definito che genera una stringa, in maniera deterministica, non si può parlare di casualità.

3.4.1 Riflessioni

Supponiamo di avere una procedura Random(n) che produce una sequenza di n bit. Quando n cresce le sequenze prodotte da Random non saranno casuali avendo una lunghezza maggiore della rappresentazione binaria della procedura Random.

Quindi secondo Kolmogorov non esistono generatori di numeri casuali!!!

3.5 Pseudo-casualità

Noi abbiamo bisogno di casualità, è la nostra risorsa, quindi sarà necessario considerare il concetto di **pseudo-casuale** (quasi casuale). I generatori pseudo-casuali sono algoritmi che producono sequenze di bit che supereranno i test di casualità. I test di casualità sono proprietà che le stringhe prodotte dai generatori devono soddisfare. I generatori di bit pseudo-casuali sono deterministici, significa che : se prendo un algoritmo, voglio che ogni volta che gli do lo stesso input lui mi dia lo stesso output (non può inventarsi scelte casuali). Per non essere deterministici avrebbero bisogno a loro volta di bit casuali (eh però è un loop infinito, vogliamo un generatore di bit casuali, che per funzionare ha bisogno di bit casuali... determinati con? generatore di bit casuali?). Quindi producono la stessa sequenza di bit ogni volta che li invochiamo a meno che non gli forniamo un "seme" in input. Stesso seme, stessa sequenza! Stiamo dicendo che per generare casualità al mio algoritmo devo passargli casualità, perchè per conto suo l'algoritmo non è in grado di generarla, essendo l'algoritmo un processo deterministico. Seme di casualità : seme perchè cresce, germoglia e diventa più grande, dando un seme di randomness all'algoritmo, la randomness crescerà. Cosa significa? : io darò un numero all'algoritmo e lui con questo numero tirerà fuori una sequenza amplificata. Piuttosto che parlare di generatori, parliamo di amplificatori di casualità, prendo un seme piccolo (es : un numero, tipo il ciclo di clock, la data, ecc.) e lo amplifico.

3.5.1 Definizione formale : generatore di numeri pseudo-casuali

Un generatore di numeri pseudo-casuali è un algoritmo che parte da un piccolo valore iniziale detto seme, solitamente fornito come dato di ingresso, e genera una sequenza arbitrariamente lunga di numeri. Questa a sua volta contiene una sottosequenza detta periodo che si ripete indefinitamente (andando molto avanti ad un certo punto le cifre si ripeteranno). In linea di principio un generatore è tanto migliore quanto più lungo è il periodo (è migliore se riesce ad ingannarci per più tempo possibile, fino a quando crediamo che sia random siamo dentro al periodo, una volta scavallato il periodo riusciamo a capire che l'algoritmo ha finito la sua amplificazione di randomness). Questi generatori possono però essere considerati amplificatori di casualità perché se innescati da un seme casuale di lunghezza m, fornito dall'utente, generano una sequenza "apparentemente" casuale di lunghezza n » m. Una inerente limitazione è che il numero di sequenze diverse che possono essere così generate è al massimo pari al numero di semi possibili, cioè 2m nel caso binario, enormemente minore del numero complessivo 2n delle sequenze lunghe n.

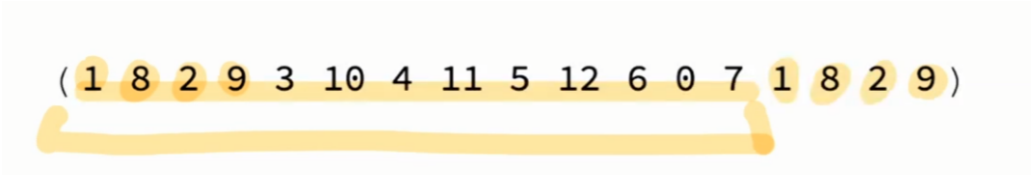


Figura 3.1: Esempio di periodo

3.6 Test statistici di casualità

Si prende una stringa, gli si applicano dei test che ci dicono se, almeno statisticamente, le parti dentro alla stringa sembrano casuali. I test di casualità sono:

- Test di frequenza : (es : la stringa 00000 non passerà il test di frequenza, perchè gli zeri dovrebbero essere circa metà della stringa, deve essere uniformemente generata. La stringa 0000011111 ironicamente passerebbe il test, anche se sembra tutto tranne che casuale)
- Poker test : Si prendono le sottosequenze di una stringa e si osserva come sono distribuite. Con la stringa 0000011111, abbiamo che la sottosequenza lunga 3 appare 3 volte, non passa il test, perchè ci sono sequenze che hanno una frequenza innaturale per essere random, non posso avere troppe stringhe con 3 zeri consecutivi o 3 uni consecutivi.
- Test di autocorrelazione : si verifica che il numero di elementi ripetuti a distanza prefissati abbiano certe caratteristiche. Nella stringa 0000011111 ci accorgiamo che, prendendo i bit, a distanza 5, risulta sempre la coppia [0, 1], non è random.
- Run test : Prendo la sottosequenza massimale tutta uguale e guardo ogni quanto deve comparire una stringa lunga n (nell'esempio è 5) di tutti zeri o uni. Nell'esempio questa ripetizione deve apparire ogni $2^{-stringLength}$ stringhe.

3.7 Generatore lineare

Generatore che supera i 4 test appena enunciati: Un generatore pseudo-casuale molto semplice che supera con successo i quattro test citati è il generatore lineare, che produce una sequenza di interi positivi $x_1, x_2, ..., x_n$ a partire da un seme casuale $x0$ secondo la relazione:
$x_i = (ax_{i-1} + b) \bmod m$, dove a, b, m sono interi positivi. Il seme x_0 è un seme di casualità, un valore di innesco che permette di far funzionare l'algoritmo (deve rispettare le proprietà definite sopra). a, b, m sono 3 valori fissati (che devono rispettare precise proprietà). Mod m significa fare a fette un numero in parti lunghe m e prendere ciò che rimane (resto della divisione per m), serve per rimpicciolire robe gigantesche.

Quindi parto da un valore di innesco x_0 (il seme), che utilizzerò per calcolare la relazione $x_i = (ax_{i-1} + b) \bmod m$, dati a, b, m fissati, per ogni valore i preso in considerazione.

3.7.1 Condizione importante (non serve memorizzarla)

Affinché il generatore abbia periodo lungo m, e quindi induca una permutazione degli interi $0, 1, ..., m-1$, i suoi parametri devono essere scelti in modo tale che $gcd(b, m) = 1$, $(a1)$ sia divisibile per ogni fattore primo di m, e $(a-1)$ sia un multiplo di 4 se anche m è un multiplo di 4 (valori consigliati sono per esempio: $a = 3141592653, b = 2718281829, m = 232$ e seme 0.
Note : gcd è il massimo comun divisore.

Conclusione Nei generatori ci sono proprietà auspicabili per a, b, m, in modo che il generatore lineare amplifichi molto la randomness del seme. I generatori soddisfano i test statistici, però non vanno comunque bene in crittografia, perchè ci sono algoritmi che riescono, dato ad un pezzo di sequenza a risalire ad a, b, m.

3.8 Generatore polinomiale

Il generatore polinomiale è una generalizzazione del generatore lineare, che segue le formulazione:
$x_i = (a_1 x_{i-1}^t + a_2 x_{i-1}^{t-1} + ... + a_t x_{i-1} + a_{t+1})$

3.9 Generatore binario

Noi non vogliamo numeri interi grandi come output, vogliamo dei bit, quindi estraiamo dei bit nel modo più casuale possibile rispetto a quelli generati dal generatore. Si calcola $r = x_i/m$: se la prima cifra decimale di r è dispari pongo il bit a 1, altrimenti a 0.

3.9.1 Difetti

I generatori lineari e polinomiali sono particolarmente efficienti (sono veloci a calcolare i bit casuali) ma non impediscono di fare previsioni sugli elementi generati, neanche quando il seme impiegato è strettamente casuale. Esistono infatti algoritmi che permettono di scoprire in tempo polinomiale i parametri del generatore partendo dall'osservazione di alcune sequenze prodotte, e questo ne svela completamente il funzionamento.

3.10 Generatori basati su funzioni one-way

One-way : senso unico, in una direzione è semplice procedere (il calcolo), nell'altra è difficile (invertire). Le funzioni one-way sono computazionalmente facili da calcolare e difficili da invertire: cioè si conosce un algoritmo polinomiale per il calcolo di $y = f(x)$, ma si conoscono solo algoritmi esponenziali per il calcolo di $x = f^{-1}(y)$. Notiamo che si opera su numeri, quindi il costo degli algoritmi deve essere riferito alla lunghezza della rappresentazione di x : un algoritmo che richiede un numero di operazioni proporzionale al valore di x sarà dunque esponenziale. Il costo computazionale è riferito alla lunghezza del numero (input), se io ho 13215798 la lunghezza del numero è 8 (ovvero il numero di cifre), è l'n in base al quale calcoliamo il costo. La lunghezza dei numeri è correlata al numero stesso tramite un logaritmo (e non con il valore stesso del numero, consideriamo il numero di cifre).

3.11 Generatori basati su funzioni one-way - Pratica

f è la nostra funzione one-way e definiamo una sequenza applicandola ad un valore.
$S = xf(x)f(f(x))f(f(f(x)))...$
Se io uso questo sistema, quando conosco $f(x)$ posso ricavare $f(f(x))$, quindi i numeri sono facilmente prevedibili. Ma cosa succede se io all'esterno fornisco questi numeri in numero inverso? ovvero f(f(x)), piuttosto che f(x)? Che per ricavare f(f(x)) è necessario invertire la funzione, che abbiamo visto, nelle funzioni one-way, essere difficile. Formalmente:
Se dunque si calcola la S per un certo numero di passi senza svelare il risultato, e si comunicano poi gli elementi uno dopo l'altro in ordine inverso, ciascun elemento non è prevedibile in tempo polinomiale pur conoscendo quelli comunicati prima di esso.

3.12 Test di prossimo bit

Test che ci dice se ciò che stiamo vedendo è abbastanza random per i nostri scopi crittografici. Dice che:
Se io ho 10001101 e voglio scoprire cosa viene dopo l'ultimo 1 mi butto ad indovinare, avendo in ogni caso probabilità $\frac{1}{2}$ di indovinare (questo vale per un qualcuno che non ha informazioni legate alla generazione di bit). Se un qualcuno non ha $P(\text{"}indovinare\text{"}) \geq \frac{1}{2} + \delta$ ho vinto, ho passato il test, se l'algoritmo passa il test viene detto crittograficamente sicuro ed è dimostrabile che passano anche i 4 test standard definiti in precedenza. Essi vengono costruiti impiegando particolari predicati delle funzioni one-way, cioè proprietà che possono essere vere o false per ogni valore di x.

3.13 Predicati hard-core

Un predicato, in logica, è una funzione che si applica ad un qualcosa e restituisce vero o falso. Formalmente:
Un predicato $b(x)$ è detto hard-core per una funzione one-way $f(x)$ se $b(x)$ è facile da calcolare conoscendo il valore di x , ma è difficile da calcolare (o anche solo da prevedere con probabilità di successo maggiore di $1/2$) conoscendo solo il valore di $f(x)$. In sostanza la proprietà hard-core, letteralmente "nucleo duro", concentra in un bit la difficoltà computazionale della f.

Abbiamo che se f è una funzione difficilmente invertibile è difficile, a partire da x_1 definire x_0 (la f è one-way). In questo caso, il predicato hardcore mi dice : è difficile calcolare tutto x_0, o c'è solo un bit difficile da calcolare? b deve essere una spremuta di x_0, di un solo bit, che comunque deve essere difficile da calcolare. In conlusione, il predicato hardcore riassume su di sè la difficoltà dell'inversione di f.

3.13.1 Esempio

Un esempio di funzione one-way è la $f(x) = x^2 \bmod n$ se n non è primo. Esempio: $n = 77$ e $x = 10$, è (polinomialmente) facile calcolare il valore $y = 102 \bmod 77 = 23$ ma è (esponenzialmente) difficile risalire al valore di $x = 10$ dato $y = 23$. Ora il predicato: $b(x) =$ "x è dispari" è hard-core per la funzione suddetta. Infatti il valore di x permette di calcolare immediatamente $b(x) = 1$ se x è dispari, $b(x) = 0$ se x è pari, ma il problema di calcolare $b(x)$ conoscendo solo il valore $y = x^2 \bmod n$ è esponenzialmente difficile.

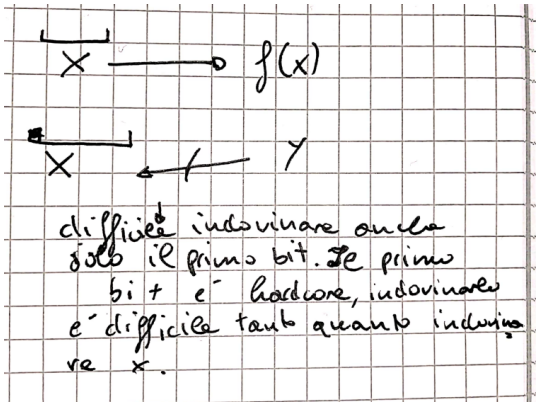


Figura 3.2: Quello che in breve, dobbiamo sapere su questo generatore

3.14 Generatore Blum, Blum, Shub

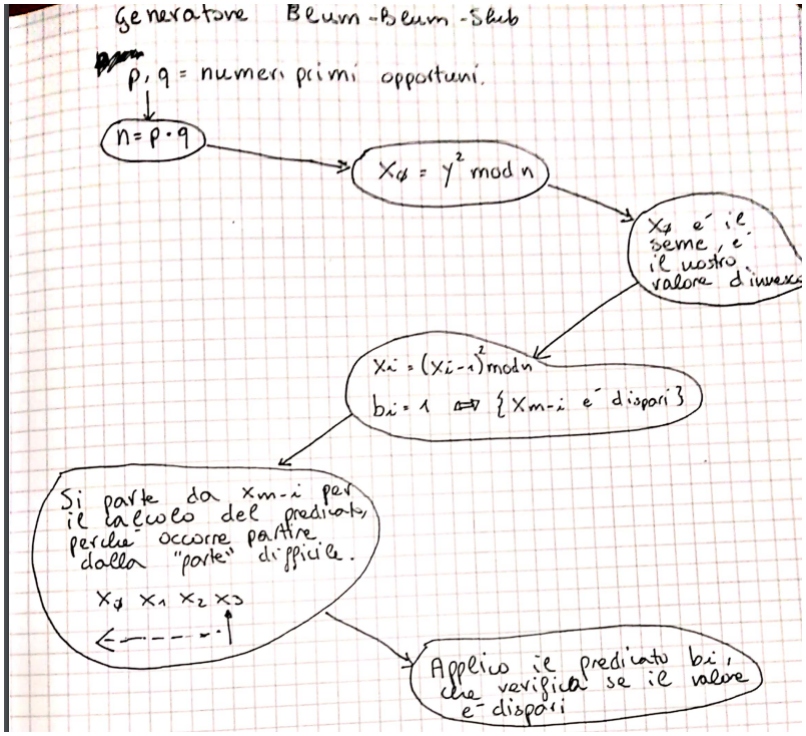


Figura 3.3: Quello che in breve, dobbiamo sapere su questo generatore

$\{x_0, x_1, ..., x_m - b_m\}$ il primo bit restituito è $x_m - b_m$. Si fa così perchè b è un predicato hardcore per questa funzione f.

3.14.1 Dimostrazione formale test di prossimo bit

Dobbiamo dimostrare che il generatore BBS supera il test di prossimo bit, ma questo è in effetti un caso particolare di un risultato generale. Poniamo che $f(x)$ sia una arbitraria funzione one-way e $b(x)$ sia un suo predicato hard-core. Indichiamo con $f^i(x)$ l'applicazione iterata della funzione per i volte consecutive ($f(f(f(f(x))))$, ecc.), e consideriamo la sequenza binaria che si ottiene partendo da un valore arbitrario di x , calcolando $b(x)$ e $f(x)$, iterando il calcolo della f un numero arbitrario di volte, e ordinando in senso inverso i valori del predicato così ottenuti.
Sequenza : $b(f^i(x)), b(f^{i-1}(x))...b(f^2(x)), b(f(x)), b(x)$
Per quanto osservato in precedenza sulle funzioni one-way, e ricordando che il predicato scelto è hard-core, segue che ciascun elemento della sequenza supera il test di prossimo bit. $b(x)$ lo tiriamo fuori in fondo perchè se lo tirassimo fuori prima daremmo informazioni sui bit successivi.
Nonostante alcuni accorgimenti di calcolo per incrementarne l'efficienza, il generatore BBS è piuttosto lento poiché il calcolo di ogni bit richiede l'esecuzione di un elevamento al quadrato di un numero di grandi dimensioni nell'algebra modulare.

3.15 Generatori basati su crittografia simmetrica

Premessa : questi cifrari sono molto efficienti, questo perchè è possibile calcolare la C agilmente, sono inoltre inviolati. Supponiamo di avere un cifrario $C(m, k)$, con m = messaggio e k = chiave, che genera un crittogramma. Come possiamo sfruttare questo cifrario per generare pseudo-casualità? utilizzando la chiave segreta k del cifrario e sostituendo il messaggio m con un opportuno valore relativo al generatore. Quindi, il messaggio viene riempito dal seme iniziale del generatore e poi dopo si applica iterativamente il cifrario per ottenere la sequenza di valori che richiediamo. In tal modo l'imprevedibilità dei risultati è garantita dalla struttura stessa dei cifrari. Un cifrario non fa altro che prendere un messaggio, prendere un po' di randomness, applicarla al messaggio e infine non fa altro che impastare queste due cose per generare il crittogramma. L'intruso fatica a risalire al messaggio perchè esso è stato impastato con della casualità. Troppa casualità → il messaggio scompare dentro il crittogramma. Un po' di casualità → l'informazione contenuta nel messaggio viene parzialmente eliminata. Applicando C tante volte a partire da un messaggio iniziale, questo è un buon metodo per generare bit pseudo casuali. Per questi esistono realizzazioni estremamente efficienti. Poiché i cifrari generano parole (i crittogrammi) composte da molti bit, ogni parola prodotta potrà essere interpretata come numero pseudo-casuale o come sequenza di bit pseudo-casuali.

3.15.1 Esempio di generatore con crittografia simmetrica

Vediamo un generatore di questo tipo, approvato come Federal Information Processing Standard (FIPS) negli Stati Uniti: il cifrario impiegato al suo interno è in genere una versione del DES di amplissima diffusione. Il DES è un macchinario che codifica informazione a blocchi di 64 bit (chunk). Detto r il numero di bit delle parole prodotte (nel DES si ha r = 64), s un seme casuale di r bit, m il numero di parole che si desidera produrre , e ricordando che k è la chiave del cifrario (stiamo usando DES per produrre dati random, non codificare), abbiamo:

```
def generatore(s, m):
    def xor(x, y):
        return x ^ y  #in python lo xor si fa cosi'
    x_list = []
    d = r # r bit presi da data e ora attuale
    y = C(d, k)
    z = s
    for i in range(1, m + 1):
        x_i = C(xor(y, z), k)
        z = C(xor(y, x_i), k)
        x_list.append(z)
    return x_list
```

- In prima istanza il generatore prende un numero r di bit pseudo-random, attraverso delle chiamate a sistema, salvandolo poi in una variabile d;
- dopodichè verrà calcolato il cifrario, prendendo in input d e k, quindi invece di codificare un messaggio, codifichiamo questo insieme di bit generati al passo precedente, lo poniamo uguale a y;
- poniamo $z = s$, ricordando che s è il seme;
- iteriamo m volte, calcolando altrettante volte il crittogramma, una volta finito il calcolo, il risultato viene aggiunto all'insieme delle soluzioni, che verrà, alla fine, comunicato all'esterno.

3.15.2 Conclusione

Cosa stiamo facendo? Stiamo cercando di sopperire alla scarsità di bit random, se noi avessimo tanti bit random, le cose sarebbero molto facili, ad esempio potremmo usare il one-time-pad... ma siccome il one-time-pad consuma la chiave, non è possibile sfruttarlo. Allora sfruttiamo dei cifrari che prendono dentro una chiave piccola, o dei generatori pseudo-casuali che però per forza di cose prendono dentro un seme di casualità molto piccolo (perchè se noi avessimo una risorsa di randomness arbitrariamente lunga, non avremmo bisogno dei generatori pseudo-casuali). Per questo motivo, se avessimo tanta randomness non sarebbe necessario studiare i generatori simmetrici, basterebbe one-time-pad. tutte le problematiche citate in questa lezione sono generate dal fatto che non abbiamo randomness in natura.

In conclusione noi cerchiamo dunque, tramite piccoli semi di randomness, di amplificare il seme della randomness nei generatori per ottenere sequenze di valori pseudo-casuali (è come se noi, in modo omeopatico, avessimo un piccolo seme random che dovesse generare tantissimi valori, (s = "seme piccolo", t = "tanti valori") $\rightarrow$ pseudo-random)

Capitolo 4

DES

4.1 Introduzione Data Encryption Standard (DES)

Primo protocollo a chiave segreta o simmetrico, utilizzato per molto tempo. Ci spiega abbastanza chiaramente quali sono le caratteristiche di un cifrario a chiave segreta (funzioni matematiche che prendono un messaggio, ci mischiano una chiave segreta condivisa col destinatario e tirano fuori il crittogramma. Il destinatario riceve il crittogramma e con la stessa chiave segreta esegue l'operazione inversa). Ci sono due criteri fondamentali alla base di questo tipo di cifrari:

- diffusione : alterazione del testo in chiaro, significa "spargere" i caratteri (del messaggio e della chiave) su tutto il testo cifrato;
- confusione : combinazione in modo complesso del messaggio e della chiave, per non permettere al crittoanalista di separare queste due sequenze mediante un'analisi del crittogramma (impastiamo uniformemente messaggio e chiave, possibile se ho tanta randomness).

Bisogna considerare che ci stiamo muovendo in un campo empirico, non ci sono teoremi, tutte le cose che facciamo durante questa parte del corso sono basate sulla pratica (es: un sistema è sicuro se non è ancora stato violato). L'esempio classico di cifrario simmetrico con diffusione e confusione è il Data Encryption Standard, brevemente DES, introdotto dalla IBM nel 1977 e divenuto per ben oltre vent'anni lo standard per le comunicazioni commerciali riservate ma "non classificate". I criteri di diffusione e confusione vengono soddisfatti attraverso una serie di messaggi sulla chiave, si tratta di permutazione, espansione (di bit) e compressione. Analizzeremo tutto il flusso di informazioni elaborate dal DES facendo riferimento all' S-box (unica parte non lineare del procedimento), da cui dipende fortemente la sicurezza del cifrario (una piccola modifica all'S-box può portare danni enormi). L'implementazione del DES è pubblica, da 50 anni, si parte dal concetto che la robustezza del cifrario non dipende dal fatto che viene tenuto nascosto, bensì dal fatto che non c'è una chiave condivisa. Il nuovo cifrario, che lo sta sostituendo è l'AES (filosoficamente simile al DES).

4.2 Struttura

Abbiamo un messaggio di n bit, divisi in blocchi di 64 bit, ogni blocco è cifrato in maniera indipendente dagli altri (questo ci permette di sfruttare il parallelismo, codifichiamo in parallelo, basta avere più processori). Il DES procede per fasi, ci sono 16 round in cui si ripetono le stesse operazioni, l'output generato da un round x_{i-1} farà da input al round x_i.

4.2.1 La chiave segreta

La chiave k è composta da 64 bit (8 byte), però ne vengono usati sono 56 come chiave, perchè 8 di questi 64 vengono utilizzati come bit di parità (rudimentale metodo di correzione d'errore, funziona se ho un numero di 1 pari). Dalla chiave k vengono create r sottochiavi ($r = 16$): $k[0], k[1], ..., k[r-1]$ impiegate una per fase. Il messaggio viene diviso in due metà S e D (sinistra e destra). In ciascuna fase si eseguono le due operazioni: $D \to S$ e $f(k[i-1], D, S) \to D$, dove f è una opportuna funzione non lineare e $i = 1, ..., r$. Alla fine delle r fasi le due metà vengono nuovamente scambiate e poi concatenate per produrre il crittogramma finale.

4.2.2 Primo colpo di genio (il genio non esiste... e a volte è un idiota)

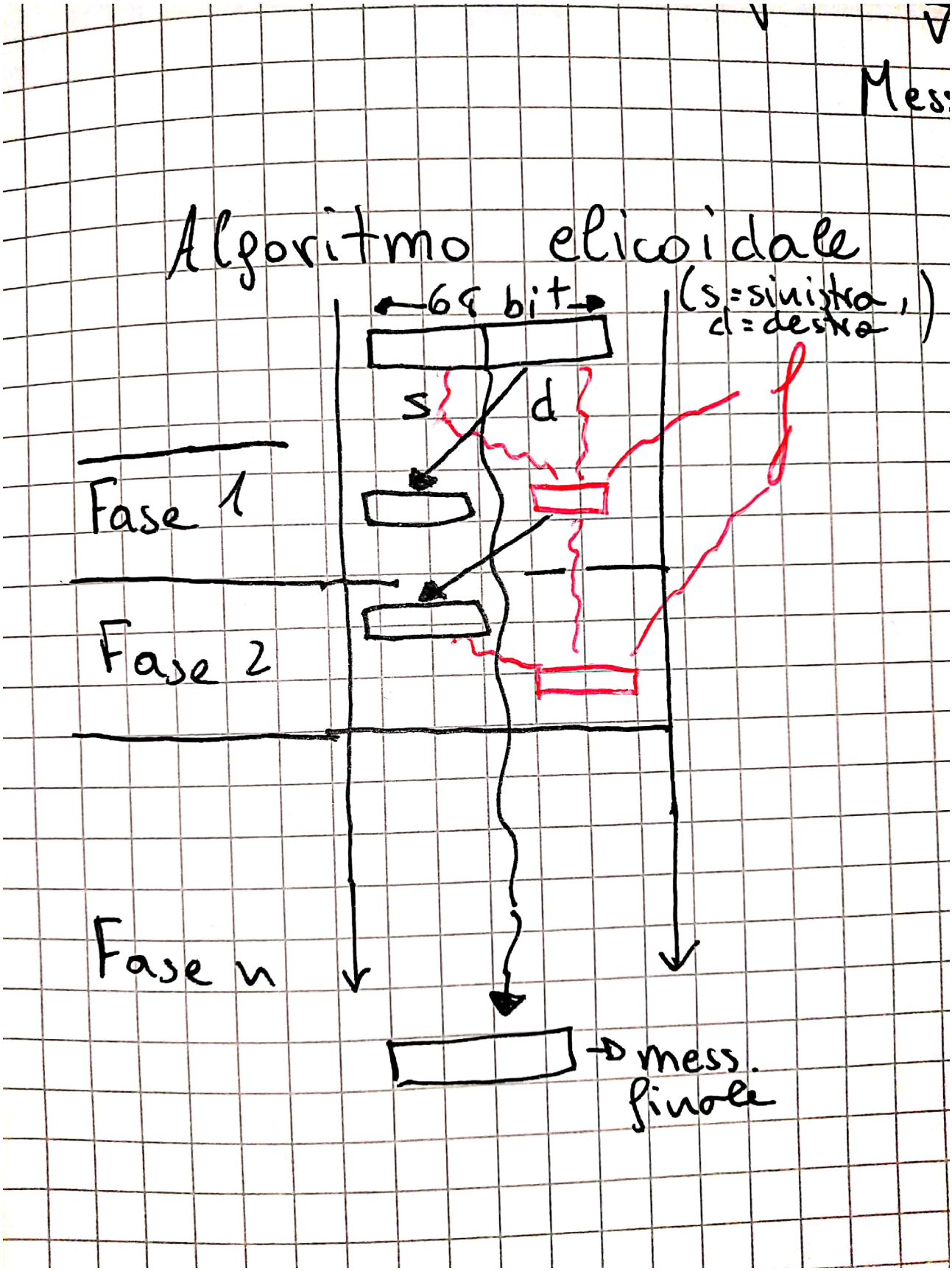
La cosa molto sorprendente è che la decifrazione (decodifica del crittogramma) consiste nel prendere il crittogramma, riapplicargli l'algoritmo con le chiavi invertite (la sequenza delle r sottochiavi viene ribaltata). Questo non è assolutamente banale.

4.3 Funzionamento

Lo schema appena descritto si realizza usando:

- permutazioni di bit (disposizioni con $n = k$);
- espansioni (ho una sequenza di bit lunga n e la voglio trasformare in una nuova sequenza lunga $n + i$, sto di fatto espandendo la sequenza iniziale);
- compressioni di sequenze binarie (operazione inversa rispetto all'espansione, ho un tot di bit e ne voglio di meno, ci saranno bit inutilizzati);
- alcune semplici funzioni combinatorie tra messaggio e chiave.

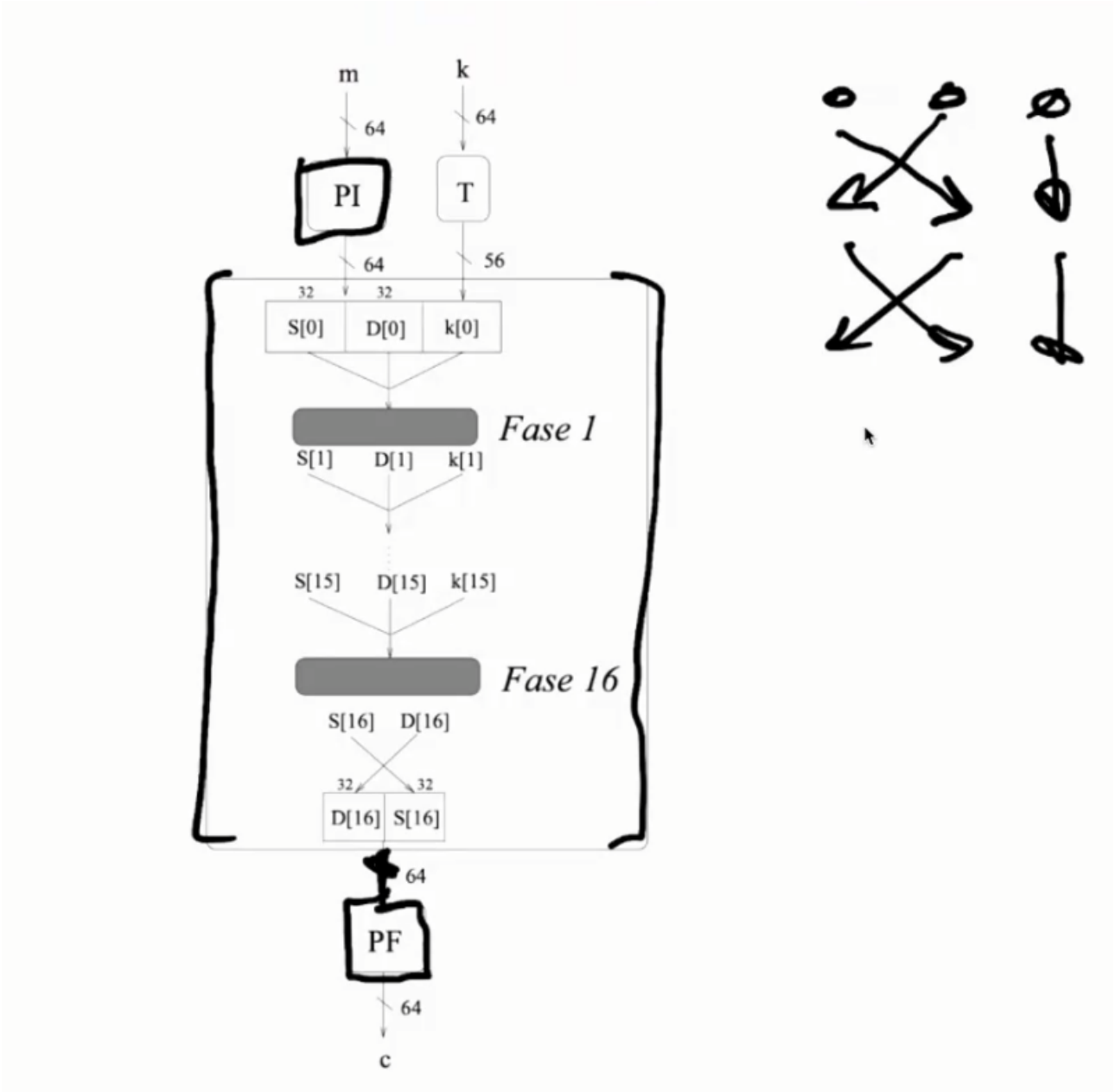
L'insieme di queste operazioni garantisce che alla fine del processo ogni bit del crittogramma dipenda da tutti i bit della chiave e da tutti i bit del messaggio in chiaro, rispondendo così ai criteri di confusione e diffusione proposti da Shannon. L'estensione della chiave con i bit di parità garantisce la corretta acquisizione e memorizzazione di tutti i bit della chiave stessa, condizioni cruciali per la decifrazione dell'intero crittogramma. La stessa protezione non è richiesta per il messaggio poiché un errore locale in esso si propaga nel crittogramma, ma è poi riprodotto identico all'originale nel messaggio decifrato con danno in genere irrilevante per la trasmissione.L'estensione della chiave con i bit di parità garantisce la corretta acquisizione e memorizzazione di tutti i bit della chiave stessa, condizioni cruciali per la decifrazione dell'intero crittogramma. La stessa protezione non è richiesta per il messaggio poiché un errore locale in esso si propaga nel crittogramma, ma è poi riprodotto identico all'originale nel messaggio decifrato con danno in genere irrilevante per la trasmissione. L'estensione ci permette di aggiungere un grado di certezza riguardo il fatto che stiamo utilizzando la chiave corretta, perchè se ci sono errori singoli sulla chiave viene identificato l'errore ed è molto utile, perchè anche un bit sbagliato nella chiave può portare ad uno stravolgimento del crittogramma (ma è anche vero che, se il ricevitore si accorge dell'errore può richiedere la ritrasmissione dell'informazione, questo perchè il canale su cui si spedisce è protetto dalla codifica; mentre attenzione, il canale in cui viene spedita la chiave è costoso, dunque sulla chiave errori non ne vogliamo fare).
Nota importante : fra hardware è molto raro che ci sia un errore, per questo si sfrutta il bit di parità, che lavora bene solo su un numero dispari di errori.



4.3.1 Rappresentazione grafica

Guida:

- la chiave k parte a 64 bit ma attraverso il processo T (trasposizione), il quale elimina i bit di parità, diventa 56 bit;
- il messaggio passa attraverso PI = "permutazione iniziale", in poche parole mischia le carte (scambiano di posto i bit);
- All'uscita viene effettuata una ulteriore permutazione, PF è il contrario della permutazione iniziale (lo schema in alto a destra ne spiega il funzionamento);



Analisi processi

Nella trasposizione è interessante notare che i bit 8, 16, ..., 56 sono stati rimossi, perchè saranno i bit di parità, questo trasformerà la dimensione della tabella, in 8x7.

58	50	42	34	26	18	10	2
60	52	44	36	28	20	12	4
62	54	46	38	30	22	14	6
64	56	48	40	32	24	16	8
57	49	41	33	25	17	9	1
59	51	43	35	27	19	11	3
61	53	45	37	29	21	13	5
63	55	47	39	31	23	15	7

Permutazione PI

il bit 58 dell'input va in posizione 1 dell'output
il bit 50 dell'input va in posizione 2 dell'output

Figura 4.1: Permutazione PI

40	8	48	16	56	24	64	32
39	7	47	15	55	23	63	31
38	6	46	14	54	22	62	30
37	5	45	13	53	21	61	29
36	4	44	12	52	20	60	28
35	3	43	11	51	19	59	27
34	2	42	10	50	18	58	26
33	1	41	9	49	17	57	25

Permutazione PF

Figura 4.2: Permutazione PF

57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
63	55	47	39	31	23	15
7	52	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

Trasposizione T

T provvede anche a scartare dalla chiave $k = k_1 k_2 \dots k_{64}$ i bit per il controllo di parità $k_8, k_{16}, \dots, k_{64}$, generando una sequenza di 56 bit che costituisce la prima sottochiave $k[0]$ (si noti che questa tabella ha dimensioni 8×7)

Figura 4.3: Trasposizione T

4.4 DES : fase i-esima

La fase i-esima prende in input 32 bit, che provengono dalla fase precedente e la chiave i-esima della fase precedente (i 56 bit provenienti dalla trasposizione). Distinguo due colonne:

- colonna di sinistra
  - viene svolto l'incrocio, mostrato nella immagine "algoritmo elicoidale";
  - i 32 bit della fase i-1 di destra formano i 32 bit della fase i della parte sinistra;
  - in parallelo il messaggio di destra della fase i-1 entra in EP, che fa una espansione, portando il messaggio a 48 bit. Il messaggio di 48 bit verrà poi messo in xor con CT;
  - terminato lo xor il messaggio entra in S, ritornando ad essere di 32 bit;
  - infine il messaggio passa in P (permutazione) e si mette in xor con il messaggio di sinistra della fase i - 1, formando così il messaggio di destra della fase i.
- colonna di destra
  - c'è una fase di calcolo e poi il risultato viene passato alla colonna di sinistra;
  - viene inoltre generata la chiave della fase successiva;
  - i 28 bit di SC entrano dentro CT, inoltre vengono ricongiunti per formare la chiave del passo i-esimo.

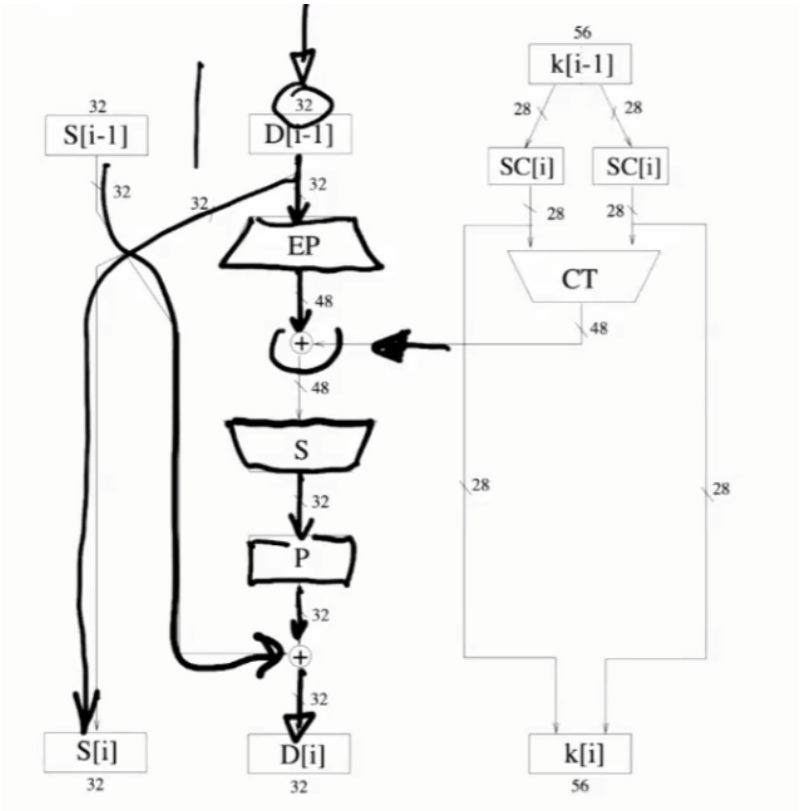


Figura 4.4: Qui viene mostrato un arbitrario passaggio intermedio, successivamente nel capitolo verranno mostrati analiticamente tutti i passaggi

4.5 SC - shift ciclici

Si tratta dello shift bit a bit, può avvenire sia a destra che a sinistra, è diverso da quello tradizionale perchè shifta sfruttando un criterio diverso. La sottochiave $k[i-1]$ di 56 bit, ricevuta dalla fase precedente, viene suddivisa in due metà di 28 bit ciascuna. Su ognuna di queste la funzione $SC[i]$ esegue uno shift ciclico verso sinistra di un numero di posizioni definito come segue: $SC[i] = 1$ per $i = 1, 2, 9, 16$, $SC[i] = 2$ altrimenti (per i fini del corso non è così importante capire il perchè). Le due parti così traslate vengono concatenate in un unico blocco di 56 bit che costituisce la sottochiave $k[i]$ per la fase successiva, ma... servono anche come chiavi dopo l'esecuzione di CT.

Figura 4.5: Esempio di shift ciclico a destra

4.6 La permutazione con selezione di CT

CT prende in input due segmenti da 28 bit. La combinazione delle funzioni SC[i] e CT garantisce che in ogni fase venga estratto dalla sottochiave un diverso sottoinsieme di bit per la cifratura. Si calcola che nella cifratura ogni bit della chiave originale k partecipi in media a 14 fasi (vuol dire che se ho 56 bit, ognuno di essi fruisce nella cifratura, questo implementa il concetto di diffusione, tutto ciò che ho in input viene diffuso nell'algoritmo di cifrazione). Selezione perchè alcuni bit vengono buttati via e gli altri rimanenti vengono permutati, infatti si passa da 56 a 48 elementi.

4.7 EP - estensione e permutazione

Entrano 32 bit e ne devono uscire 48. Abbiamo che 16 bit di ingresso vengono duplicati, per esempio il bit 32 è copiato nelle posizioni 1 e 17 dell'uscita.

4.8 S

Secondo molti è la parte cruciale di questo algoritmo, in quanto garantisce la sicurezza che lo contraddistingue. Da 48 bit si torna a 32, i 48 bit si dividono in gruppi da 6, vengono dunque formati 8 gruppi totali. Ogni gruppo fa uscire 4 elementi, da questo si arriva a 32 bit. I gruppi sono le uniche parti non lineari di DES. Come si fanno a descrivere questi gruppi? Le riscriviamo con la tabella sottostante, abbiamo 16 colonne, ciò significa che per indicizzare la colonna da 0 a 15 occorrono 4 bit, per indicizzare le righe da 0 a 4 ne occorrono 2. 2 + 4 = 6 bit, che sono esattamente il numero di ingressi dei gruppi. La funzione S-box costituisce la parte magica di DES, su cui si basa la sicurezza del cifrario. Questa funzione, progettata con molta cura dalla IBM e modificata con altrettanta cura dalla NSA, costituisce la parte cruciale e "magica" su cui si basa la sicurezza del cifrario. Essa consta di otto sottofunzioni combinatorie $S_1, S_2, ..., S_8$. L'ingresso di 48 bit viene decomposto in otto blocchi $B_1, B_2, ..., B_8$ di 6 bit ciascuno che costituiscono l'ingresso alle sottofunzioni di pari indice. Sia $B_j = b_1b_2b_3b_4b_5b_6$. Questi bit vengono divisi in due gruppi b_1b_6 e $b_2b_3b_4b_5$ che definiscono due numeri x, y, con

$0 \leq x \leq 3$ e $0 \leq y \leq 15$, utilizzati per accedere alla cella di riga x e colonna y in una tabella che definisce la sottofunzione S_j. Il numero ivi contenuto è compreso tra 0 e 15, ed è quindi rappresentato con 4 bit che costituiscono l'uscita di S_j realizzando una compressione da 6 a 4 bit. Complessivamente gli otto blocchi generano una sequenza di 32 bit. I blocchi EP e S sono studiati in modo che tutti i bit di $D[i-1]$ influenzino l'uscita di S (significa che se io cambio un bit da 0 ad 1 o viceversa in $D[i]$, questo influisce sul risultato della combinazione di EP con S), senza di che non sarebbe poi possibile decifrare il messaggio.

4.9 P

P è una permutazione di 32 bit che genera il blocco finale $D[i]$.

4.10 Linearità e non linearità

Tutte le funzioni applicate dal cifrario, a eccezione della S-box, sono lineari se riferite alla operazione di XOR, cioè vale $f(x)XORf(y) = f(xXORy)$ ove f è una funzione permutazione, espansione o compressione di un vettore binario e x, y sono vettori binari arbirari. Ciò invece non si verifica se f è una delle otto funzioni della S-box che sono dunque non lineari. Si stima che questo contribuisca in modo determinante alla sicurezza del cifrario. E' più facile predire un comportamento lineare rispetto ad un comportamento non lineare. Una funzione è lineare se è chiusa rispetto a somma e prodotto.

4.11 Attacchi

Purtroppo le chiavi possibili per il DES, ai giorni nostri, essendo formate da 56 bit sono fattibili da violare, i computer che abbiamo ad essere non è un problema elaborare 2^{56} combinazioni. Ci sono inoltre proprietà che ci permettono di dimezzare il numero di chiavi. A fronte di questa particolarità il DES è diventato sempre meno sicuro, dunque si è deciso di aumentare il numero di bit legati alle chiavi. Il punto debole del DES è costituito dal fatto che la chiave sia composta da 56 bit, questo significa che il protocollo è piccolo. Attenzione però, se arrivo a dover scandire tutte le 2^{56} chiavi significa che il protocollo non è stato violato, per essere violato occorre decifrare il crittogramma senza utilizzare tutte le chiavi.

Capitolo 5

RSA

5.1 Introduzione

Rivoluzione nel campo della crittografia, stiamo parlando di tecniche fortemente utilizzate al giorno d'oggi, ovvero di protocolli a chiave pubblica. Permette di condividere un segreto fra mittente e destinatario, nonostante non esista una chiave posso comunicare in completa segretezza. Questa rivoluzione è avvenuta negli anni 70.

5.2 Descrizione

Nei cifrari simmetrici visti sinora la chiave di cifratura è uguale a quella di decifrazione (o comunque ciascuna può essere facilmente calcolata dall'altra), ed è nota solo ai due partner che la scelgono di comune accordo e la mantengono segreta. Nei cifrari a chiave pubblica, o asimmetrici, le chiavi di cifratura e di decifrazione sono completamente diverse tra loro. Esse sono scelte dal destinatario che rende pubblica la chiave di cifratura k[pub], che è quindi nota a tutti, e mantiene segreta la chiave di decifrazione k[prv] che è quindi nota soltanto a chi la genera. Chiunque voglia spedirmi qualcosa deve conoscere la mia chiave pubblica. Esiste una coppia k[pub], k[prv] per ogni utente del sistema, scelta da questi nella sua veste di possibile destinatario Dest. La cifratura di un messaggio m da inviare a Dest è eseguita da qualunque mittente come c = C(m; k[pub]), ove sia la chiave k[pub] che la funzione di cifratura C sono note a tutti. La decifrazione è eseguita da Dest come m = D(c; k[prv]), ove D è la funzione di decifrazione anch'essa nota a tutti, ma k[prv] non è disponibile agli altri che non possono quindi ricostruire m. Il concetto base era già noto nel 76, non era invece chiara quale C e quale D avessero queste proprietà.

5.3 Asimmetria

L'appellativo di asimmetrici assegnato a questi cifrari sottolinea i ruoli completamente diversi svolti da Mitt e Dest, in contrapposizione ai ruoli intercambiabili che essi hanno nei cifrari simmetrici ove condividono la stessa informazione (cioè la chiave) segreta.

5.4 Proprietà

- Per ogni possibile messaggio m si ha:
  $D(C(m, k[pub]), k[prv]) = m$
  Dunque stiamo dicendo che $D(C(m)) = m$, ossia Dest deve avere la possibilità di interpretare qualunque messaggio che gli altri utenti decidano di spedirgli. $C(D(m)) = m$ invece non vale sempre, chi soddisfa questa proprietà commutativa è molto apprezzato, perchè permette di implementare la firma digitale.;
- La sicurezza e l'efficienza del sistema dipendono dalle funzioni C e D, e dalla relazione che esiste tra le chiavi k[prv] e k[pub] di ogni coppia;
- La coppia k[prv] e k[pub] è facile da generare, e deve risultare praticamente impossibile che due utenti scelgano la stessa chiave deve essere inoltre praticamente impossibile (computazionalmente difficile) risalire alla chiave privata dalla pubblica.
- Dati m e k[pub], è facile per il mittente calcolare il crittogramma c = C(m, k[pub]). (deve essere facile la codifica).
- Dati c e k[prv], è facile per il destinatario calcolare il messaggio originale m = D(c, k[prv]) (deve essere facile la decodifica)
- Pur conoscendo il crittogramma c, la chiave pubblica k[pub], e le funzioni C e D, è difficile per un crittoanalista risalire al messaggio m, ancora peggio alla chiave privata!. Risalire al messaggio smaschera una comunicazione (violazione puntuale), risalire alla chiave privata comporta invece una violazione totale del protocollo.

5.5 Sicurezza

I protocolli a chiave privata vengono definiti sicuri in maniera empirica, ergo se non è stato violato in n anni di utilizzo è sicuro. Nei protocolli a chiave pubblica invece la sicurezza è determinata dal fatto che per violarli dovremmo riuscire a risolvere in tempo polinomiale dei problemi che fino ad esso non siamo riusciti a risolvere (non è stato possibile risolvere questo problema in anni,secoli e millenni, per questo è un certificato di garanzia di gran lunga migliore rispetto a quello dei protocolli a chiave privata).

5.6 Funzioni one-way

C deve essere una funzione one-way, cioè facile da calcolare e difficile da invertire, ma deve contenere un meccanismo segreto detto trap-door che ne consenta la facile invertibilità solo a chi conosca tale meccanismo (questo è il succo, la sintesi dei protocolli a chiave pubblica, lo svela-segreto è la chiave privata). La conoscenza di k[pub] non fornisce alcuna indicazione sul meccanismo segreto, che è svelato da k[prv] quando questa chiave è inserita nella funzione D (D = decodifica).

5.7 Richiami di algebra modulare

$\mathbb{Z}_n$ è l'insieme di numeri modulo m. $\mathbb{Z}_n^*$ è un sottoinsieme di $\mathbb{Z}_n$ e rappresenta l'insieme di elementi di $\mathbb{Z}_n$ relativamente primi con n, 0 ed 1 esclusi. La notazione a mod n significa fare a fette a in chunk lunghi n. Se prendo due numeri e li divido per lo stesso n e ottengo lo stesso resto sono nella stessa classe di equivalenza.

- $\mathbb{Z}_n = \{0, 1, ..., n\}$;
- $\mathbb{Z}_p^* = \{1, ..., p-1\}$;
- a mod n è il resto della divisione intera fra a e n;
- $a \cong b$ mod n se e solo se a mod $n = b$ mod n;
- $a \cong b$ mod n se e solo se $a = b + kn$;
- a primo con b se e solo se il loro massimo comun divisore è uguale a 1.

5.8 Funzione di Eulero

Concetto importantissimo, non fa altro che dirci quanto è grande $\mathbb{Z}_n^*$, ovvero la sua cardinalità.Ovvero:

$\phi(n) = |\mathbb{Z}_n^*|$

5.8.1 Proprietà

- $\phi(p) = p - 1$
- $\phi(pq) = (p-1)(q-1)$
- a primo con b allora $\phi(ab) = \phi(a)\phi(b)$
- $\phi(n)$ difficile da calcolare (non riusciamo a calcolarla efficientemente, se sapessimo fattorizzare sarebbe facile anche calcolare ϕ. Anche se fosse difficile fattorizzare noi potremmo tentare di trovare ϕ con metodi alternativi).

5.9 Teorema di Eulero

Sia $n > 1$ e a primo con n, allora:

$a^{\phi(n)} \cong 1$ mod n

Capitolo 6

DH

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```

Capitolo 7

Curve ellittiche

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```

Capitolo 8

Firma digitale

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```

Capitolo 9

SSL

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```

Capitolo 10

Protocolli Zero Knowledge

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```

Capitolo 11

Bitcoin

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```

Capitolo 12

Teoria dell'informazione

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```

Capitolo 13

RFID e protocolli crittografici

```
//Hello.java
import javax.swing.JApplet;
import java.awt.Graphics;

public class Hello extends JApplet {
   public void paintComponent(Graphics g) {
      g.drawString("Hello, world!", 65, 95);
   }
}
```