Crittografia

Anno 2022-2023

23 gennaio 2023

I procioni sono meglio dei panda rossi.

Indice

Introduzione 1

Crittografia Nasconde il contenuto del messaggio, ma non l'esistenza dello stesso.

Stenografia Nasconde l'esistenza del messaggio (es tecniche: inchiostro invisibile, codifico il messaggio nei bit meno significativi dell'immagine che non sono visibili a occhio nudo). Se qualcuno scopre la tecnica che nasconde il messaggio, allora tutti i messaggi che usano quella tecnica diventano visibili.

Sistema sicuro Non esiste un concetto assoluto di sicurezza, ma si può dire che un sistema è sicuro quando il costo necessario a romperlo è superiore al vantaggio economico che si otterrebbe/al danno che si produrrebbe.

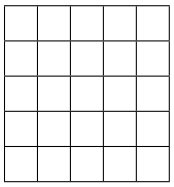
1.1 Breve storia della crittografia

Cifrario di Cesare Uno dei primi metodi crittografici della storia è quello di **Cesare**. Si basa sullo shift dell'alfabeto di n posizioni. La chiave corrisponde alla prima coppia di lettere dell'alfabeto originale e traslato (es: la chiave è AD, quindi per decifrare il messaggio devo prendere la lettera che corrisponde a 3 posizioni indietro rispetto a quella scritta nel messaggio cifrato -la lettera D cifrata corrisponderà alla lettera A-). Si tratta di una cifratura facilmente attaccabile in quanto le **possibili chiavi** sono solo 26 (**tutti gli shift dell'alfabeto**). Un attacco di questo genere (testo tutti i possibili shift) si chiama attacco di forza bruta: testo tutte le chiavi possibili finché trovo quella corretta.

Permutazione arbitraria dell'alfabeto Un'evoluzione di questo metodo di cifratura è la **permutazione arbitraria**. Questo metodo si basa sulla permutazione totale dell'alfabeto, piuttosto che sul suo semplice shift. In questo caso la **chiave** è **alfabeto permutato**. Le possibili chiavi sono [26!] (50 anni fa non era facilmente attaccabile da attacchi di forza bruta a causa dei pc poco potenti). Questa cifratura è attaccabile con l'**analisi delle frequenze**[1]. Nel caso si utilizzi un attacco a forza bruta (automatizzato dal pc), è possibile capire che se ho raggiunto il risultato corretto sempre tramite l'analisi delle frequenze.

Counter all'analisi delle frequenze Si propose di lavorare non più con una singola lettera, ma con coppie di lettere. In questo modo ad ogni lettera cifrata potevano corrispondere 2 lettere in chiaro. Uno degli approcci utilizzati è il seguente:

1: L'analisi delle frequenze è lo studio della frequenza di utilizzo delle lettere o gruppi di lettere in un testo cifrato. Questo metodo si basa sul fatto che in ogni lingua la frequenza di uso di ogni lettera è piuttosto determinata; questo è vero in modo rigoroso solo per testi lunghi, ma spesso testi anche corti hanno frequenze non molto diverse da quelle previste. Tramite questa analisi è possibile inoltre determinare anche la lingua del messaggio, in quanto ogni lingua ha delle frequenze specifiche (tralasciando le lingue orientali).



Supponiamo di avere una tabella 5×5 che andrà a contenere una password che sarà la chiave. Usiamo la parola *MONARCHY*. Nel caso la password contenga lettere ripetute, le ripetizioni vengono saltate, in modo che ogni lettera compaia una sola volta.

M	O	N	A	R
C	H	Y		

A questo punto procediamo inserendo le lettere dell'alfabeto mancanti, saltando ovviamente quelle già inserite. Consideriamo la lettera I uguale alla lettera J (ho 25 posti e le lettere sono 26).

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

Possiamo procedere ora con la codifica, lavorando sulle coppie di lettere. Le regole da seguire sono:

- Se la coppia di lettere da cifrare appartiene alla stessa riga nella tabella, si prenderà la lettera successiva nella riga, procedendo in maniera circolare.
  Esempio: Supponiamo di dover cifrare la coppia AR. Nella tabella AR appartiene alla stessa riga, quindi uso la regola sopra. La lettera A corrisponderà alla lettera R (mi sposto di una posizione), mentre la lettera R corrisponderà alla lettera M (essendo la fin della riga, la posizione successiva è l'inizio della riga stessa).
  Quindi:

  $$\text{AR} \longrightarrow \text{RM}$$
  $$\text{OA} \longrightarrow \text{RN}$$

- Se la coppia appartiene alla stessa colonna, lavoro come nel primo caso, solo che qui mi sposto verso il basso sulla colonna.
  Esempio:

  $$\text{MU} \longrightarrow \text{CM}$$

- Se la coppia appartiene a righe e colonne diverse, la prima lettera si sposterà nella colonna della seconda e viceversa.
  Esempio:

PB ⟶ SH
EA ⟶ IM oppure JM

Anche questo metodo è comunque attaccabile con l'analisi delle frequenze, in quanto, sebbene la codifica di una lettera dipende dall'altra della coppia (e quindi non valore più il caso di *a lettera uguale corrisponde lettera uguale*), vale il caso in cui "a coppia uguale corrisponde coppia uguale" e anche le coppie di lettere di una lingua hanno le loro frequenze.

Cifrario di Vigenère Il cifrario di Vigenère riprende l'idea del cifrario di Cesare: presa una chiave (es: *key*), si ripete la chiave tante volte quanto è lungo il testo (eventualmente troncando l'ultima ripetizione), e si codifica la lettera con il corrispondete cifrario di Cesare.

K E Y K E Y K E Y K E Y
P R O V A D I T E S T O

La prima lettera del cipher text sarà la lettera ottenuta dal cifrario di Cesare di chiave AP, la seconda con la chiave AR e così via. Anche questo cifrario è semplice da attaccare, si parte dalla divisione del ciphertext in gruppi di lunghezza pari a quella della chiave, e si esegue l'analisi delle frequenze su ogni gruppo. Nel caso in cui non si conosca la lunghezza della chiave, si può semplicemente provare tutte le possibili lunghezze della chiave. Ovviamente sarà necessario un testo di dimensioni maggiori ai precedente per poter effettuare l'analisi. Se la password fosse lunga quanto il testo, l'analisi delle sequenze non sarebbe possibile.

One-time pad Vediamo ora lo schema di Vigenère applicato ad un alfabeto binario. In questo caso il mio alfabeto si compone di soli due simboli: 0 e 1. Posso avere solo 4 casi di chiave-testo:

$$\begin{aligned} Key &\rightarrow 1010 \\ Text &\rightarrow 0101 \\ Cyph &\rightarrow 0110 \end{aligned}$$

Praticamente applicare questo schema ad un alfabeto binario equivale ad applicare lo XOR tra testo in chiaro e chiave. Per riottenere il plaintext a partire dal cyphertext, mi basta riapplicare lo XOR tra chiave e cyphertext. Riassumendo:

$$\begin{aligned} \text{Encryption } E(P,K) &= P \oplus K \\ \text{Decryption } D(C,K) &= C \oplus K \end{aligned}$$

Se ho una chiave lunga quanto il testo e la chiave è scelta in modo casuale (ogni bit del testo è indipendente dalla chiave (viene scelto non conoscendo la chiave)):

- La probabilità che il primo bit b del cyphertext c sia 0 è $\frac{1}{2}$ (lo stesso che sia uno);

- La probabilità che il primo bit del cyphertext sia 0 è

$$\begin{aligned} Pr[c = 0] &= Pr[b = 0 \wedge k = 0] + Pr[b = 1 \wedge k = 1] \\ &= P \cdot \frac{1}{2} + (1 - P) \cdot \frac{1}{2} \\ &= \frac{1}{2}(P + 1 - P) \\ &= \frac{1}{2} \end{aligned}$$

  Ovvero la probabilità che il cyphertext c sia 0 è data dalla probabilità P che il bit b sia stato scelto con valore 0, moltiplicata la probabilità che il bit k della chaive sia stato scelto a 0, sommata alla probabilità che il bit del plaintext b sia stato scelto a 1, quindi P - *1*, moltiplicata la probabilità che k sia 1.

Se scelgo la chiave in maniera casuale, a prescindere da come è stato scelto il plaintext, la distribuzione probabilistica sul cyphertext è quella uniforme. Quindi osservando il cyphertext vedo solo una sequenza casuale di bit, che non mi può dire nulla sul plaintext (ogni plaintext è equamente probabile dato il cyphertext). Questo sistema, detto **one-time pad**, è assolutamente sicuro.

Teorema di Shannon Per avere un sistema crittografico sicuro, tale per cui a partire dal cyphertext non si può ricavare nulla del plaintext, allora la chiave deve essere tanto lunga quanto il plaintext.

Secondo questo teorema, i sistemi usati oggi non sarebbero sicuri, in quanto usano chiavi con dimensione inferiore al messaggio scambiato. Per questo l'obiettivo è costruire sistemi dove, sebbene sia possibile ottenere informazioni dal cyphertext, questa operazione diventi troppo complessa (es: richiede molto tempo).

1.2 Problemi di sicurezza non legati alla crittografia

La sicurezza del dato non dipende solo dall'affidabilità del protocollo crittografico utilizzato. Rischi di sicurezza possono derivare, ad esempio, anche dalle implementazioni.

Vediamo il caso dell'IBM 360. Il sistema utilizzava, per il controllo della password, il semplice controllo della stringa (si confronta la prima coppia di lettere e se uguali si passa alla seconda e così via) e memorizza la password in chiaro. Si riuscì a fare in modo che una parte della password inserita per il test finisse in una pagina di memoria, mentre il resto in una seconda pagina che finisce fuori dalla chache. Si faceva quindi partire il confronto e se il sistema ritornava un page fault[2] allora la parte di password scritta nella prima pagina era corretta (se fosse stata errata il controllo si sarebbe fermato prima di raggiungere la seconda pagina, che essendo fuori dalla chache, dà page fault).

Un altro problema legato al controllo tramite confronto delle stringhe è che più lunga è la stringa, maggiore è il tempo necessario per terminare

2: Il page fault è un'eccezione generata quando un processo cerca di accedere a una pagina che è presente nel suo spazio di indirizzamento virtuale, ma che non è presente nella memoria fisica poiché mai stata caricata o perché precedentemente spostata su disco di archiviazione. Tipicamente, il sistema operativo tenta di risolvere il page fault caricando la pagina richiesta nella memoria virtuale oppure terminando il processo in caso di accesso illegale. Il componente hardware che rileva i page fault è il memory management unit, mentre quello software di gestione delle eccezioni è generalmente parte del sistema operativo (kernel).

l'operazione. Si potrebbe quindi determinare la lunghezza della stringa contando i microsecondi necessari al completamento del controllo.

Crittografia in cascata 2

L'idea è di provare a mettere in cascata una serie di sistemi crittografici deboli (es: usando chiavi piccole) per vedere se si riescono a ottenere sistemi più forti.

2.1 Permutazioni in cascata

Ho un rullo a due facce. Su ciascuna faccia sono presenti 26 contatti, che rappresentano le 26 lettere dell'alfabeto. Un qualsiasi collegamento interno al cilindro tra le due facce mi rappresenta una possibile permutazione (es: sul lato del plaintext premo il contatto che rappresenta la A e ottengo sull'altro lato, che mi rappresenta il cyphertext la lettera P). Suppongo di scegliere per il rullo la permutazione Π_1. Collego ora alla faccia del cyphertext del rullo un secondo rullo (che funziona esattamente come il primo) e per lui scelgo la permutazione Π_2. Aggiungo un terzo rullo collegato alla faccia cyphertext del secondo e scelgo la permutazione Π_3. Il mio sistema è diventato più resistente? No. La combinazione delle 3 (o anche n) permutazioni $\Pi_1 \circ \Pi_2 \circ \Pi_3$ è, alla fine, una permutazione, che sarà identica per ogni lettera del testo che voglio cifrare. Provo quindi a creare una soluzione più robusta. Decido ora di ruotare il terzo cilindro di una posizione dopo aver codificato ogni lettera del plaintext. Con questa soluzione non ho più una permutazione unica associata all'intero plaintext, ma una per ogni sua lettera, fino ad un massimo di 26 shift, in quanto poi ritorno alla posizione iniziale. Ovviamente, se decido di ruotare il secondo rullo per ogni giro completo, incremento il numero di permutazioni uniche che posso usare (26^2).

2.2 Enigma

Enigma usava un metodo di cifratura molto simile a quello descritto sopra. La sua vulnerabilità non stava tanto nel sistema, ma nel fatto che ogni messaggio scambiato iniziava sempre con lo stesso testo, dal quale gli attaccanti sono riusciti a romperla. Enigma era quindi vulnerabile al **known plaintext attack**.

2.3 Tipi di attacchi

Vediamo gli attacchi che un sistema crittografico può affrontare per essere considerato affidabile:

- Known cyphertext attack: l'attaccante conosce solo il testo cifrato del messaggio;
- Known plaintext attack: l'attaccante conosce il messaggio in chiaro e il relativo cyphertext;

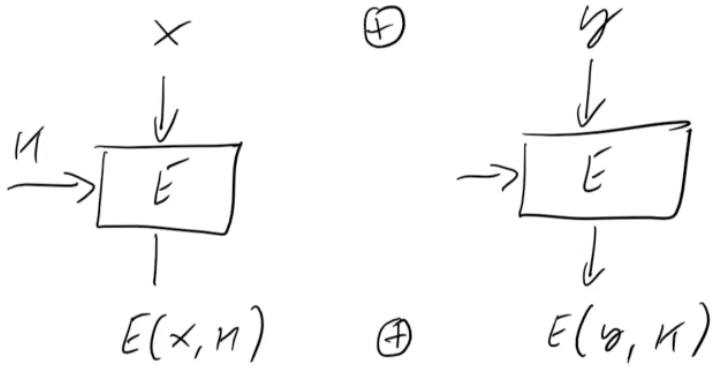
- Chosen plaintext attack: l'attaccante sceglie il messaggio da cifrare e analizza il relativo cyphertext;
- Adaptive chosen plaintext attack: l'attaccante sceglie il testo di un primo messaggio, analizza il cyphertext ottenuto e in base all'analisi sceglie un nuovo messaggio da cifrare e così via.

Se, ad esempio, il sistema è vulnerabile ad un *adaptive chosen plaintext attack*, sarà più affidabile di un sistema che è debole ad un *known cyphertext attack*.

2.4 Nascita di DES

Col tempo diventa necessario avere un sistema crittografico standard internazionale. Tra le proposte alla fine ne venne scelta una e il protocollo venne chiamata Data Encryption Standard (DES). DES venne costruito con l'idea di durare una ventina d'anni, come è effettivamente stato. DES venne bucato con la **crittoanalisi differenziale**.

A partire da due input x e y e i rispettivi cyphertext $E(x, k)$ e $E(y, k)$ cifrato con chiave k, questa analisi confrontava lo XOR degli input con lo XOR dei cyphertext. Praticamente analizzava come le differenze tra i bit di ingresso si propagassero nei bit di uscita.



L'analisi ritornava un'equazione lineare sui bit della chiave k, permettendo di costruire alla fine un sistema di equazioni indipendenti a 56 variabili, che rappresentano i 56 bit della chiave usata da DES. Si è scoperto che fissando 12 variabili, era possibile risolvere il sistema, e quindi ottenere la chiave. Di conseguenza, un attaccante non doveva più ricavare 56 bit, ma solo 12 (in quanto gli altri 44 sono ricavabili da questi), riducendo il numero di tentativi necessari per un attacco a forza bruta a solo $2^{12} = 4096$ tentativi.

Per fare questa analisi è necessario utilizzare un *adaptive chosen plaintext attack*.

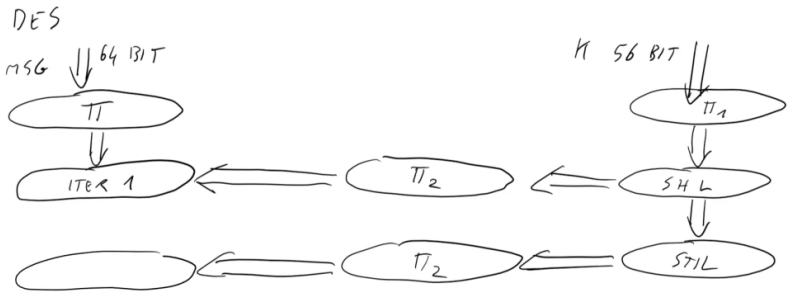
2.5 Nascita di AES

Scoperta la debolezza di DES, si cercarono nuovi protocolli per sostituirlo, che fossero resistenti alle sue vulnerabilità. Venne scelto l'algoritmo che poi venne chiamato Advanced Encryption Standard (AES), che è quello usato oggi. Sostanzialmente è una sequenza di quattro fasi, ognuna delle quali va a contrastare uno degli attacchi noti a DES. Sia per DES che per AES, non esiste una vera e propria dimostrazione che il sistema sia robusto, ma c'è solamente una analisi che dice "ci abbiamo provato in tanti e abbiamo una ragionevole convinzione che saranno robusti almeno per un po' di tempo".

Fattorizzazione di un numero In generale, non esiste un modo per costruire un protocollo dove si sa che sia sicuro alla base, ovvero che sappiamo che invulnerabile ad attacchi presenti e futuri, eccetto per one-time pad, dove abbiamo visto che esiste una dimostrazione che funzione ed è sicuro.

Gli algoritmi attualmente esistenti di fattorizzazione di un numero provano tutti i possibili divisori del numero, ed hanno una complessità esponenziale al numero di bit del numero. Molte persone, appartenenti ad abiti diversi, hanno provato a trovare un algoritmo più ottimizzato, ma senza successo. Verosimilmente un algoritmo efficiente per fattorizzare un numero non verrà trovato. Quindi, se si riuscisse a costruire un crittosistema per il quale è possibile dire che se si trova un algoritmo per attaccarlo allora si è trovato un algoritmo per fattorizzare un numero, allora in questo caso si può dire che l'attaccante non ci possa riuscire.

2.6 Overview di DES

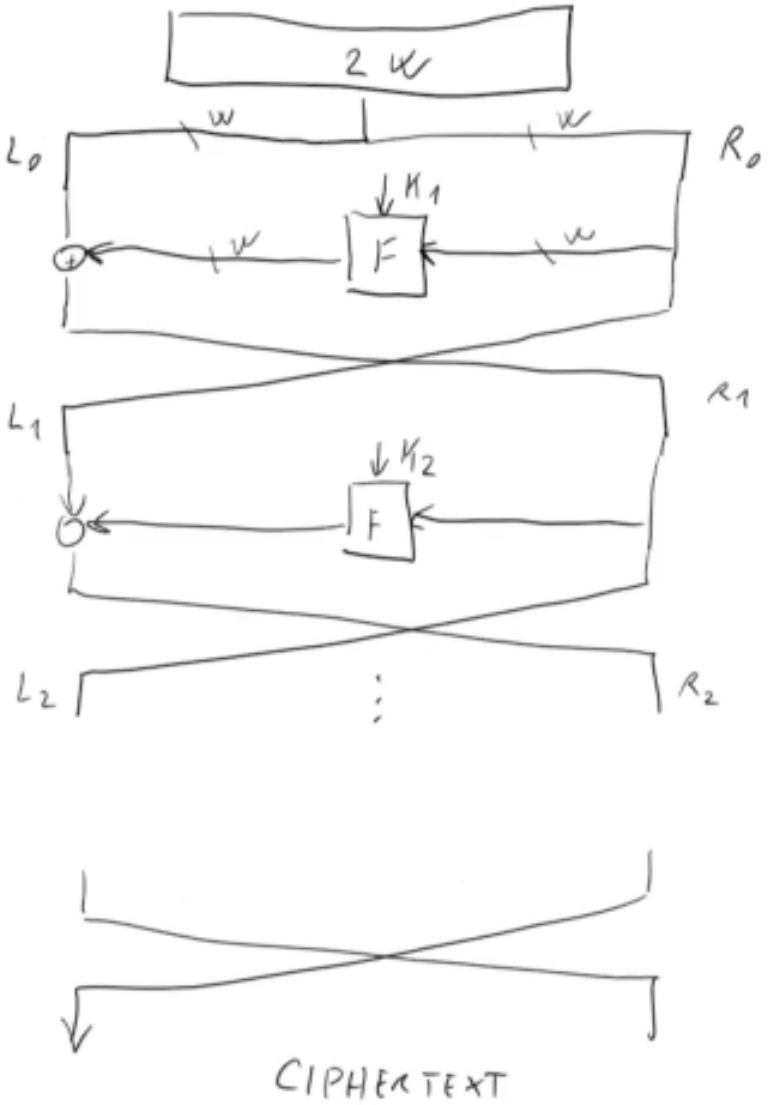


Dato un messaggio di 64 bit e una chiave di 56 bit, DES prevede i seguenti passi:

1. Permuto con permutazione Π il messaggio;
2. Permuto la chiave k con permutazione Π_1;
3. Shifto con rotazione la chiave permutata a sinistra di una posizione;
4. Applico la permutazione Π_2 alla chiave shiftata;
5. Applico la chiave ottenuta al messaggio permutato, utilizzando il meccanismo di **Feistel**;
6. Ripeto i passi 3, 4 e 5 per 16 volte;
7. Applico infine al risultato Π^{-1}, ovvero calcolo ,a permutazione inversa a quella iniziale.

Le permutazioni usate non aggiungono nulla dal punto di vista della sicurezza del sistema, riguardano infatti l'aspetto fisico del sistema: poiché il sistema è implementato in un chip non è detto, per ragioni di ottimizzazione dello spazio, che contatti di entrata e di uscita si trovino nello stesso ordine.

Il meccanismo di Feistel è fatto in modo che l'algoritmo di codifica e decodifica siano uguali. Questa soluzione mi costa meno a livello di hardware, in quanto mi basta un solo chip. Le chiavi però devono essere usate, nella decodifica, in ordine inverso rispetto alla codifica. Lo scopo di Feistel era quello di creare uno schema dove algoritmo di codifica e decodifica fossero gli stessi e che fosse applicabile a crittosistemi diversi.

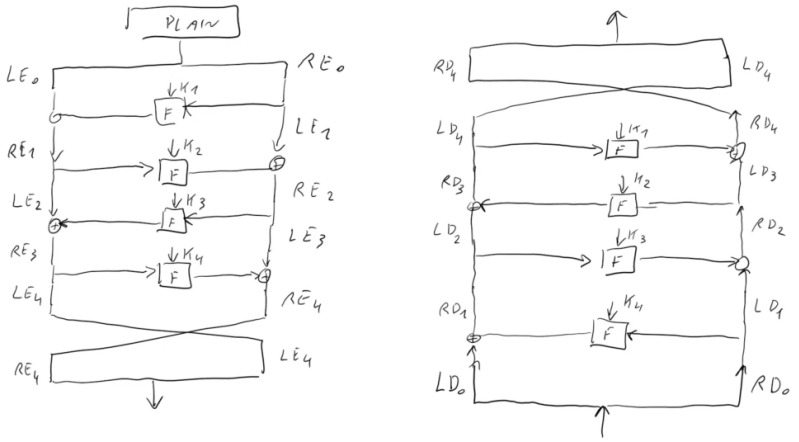


Lo schema di Feistel prende in input un flusso di dati in ingresso di dimensione $2W$ (con W numero arbitrario) ed esegue le seguenti operazioni:

1. Divide il messaggio iniziale in due messaggi L_0 e R_0 entrambi di dimensione W;
2. R_0 viene applicato ad una funzione qualsiasi F, con chiave k_1, e produce un nuove messaggio di dimensione sempre W;
3. Il messaggio risultate al punto 2 viene messo in XOR col messaggio L_0;
4. Il messaggio R_0 e il messaggio risultatane dal punto 3 vengono scambiati di posizione, diventando il messaggio L_1 e R_1;
5. I punti 2, 3 e 4 vengono ripetuti per n volte, utilizzando sempre una chiave diversa (k_2, k_3, ...);
6. Eseguo un ultimo scambio tra il messaggio di destra e quello di sinistra, ottenendo il cyphertext.

La caratteristica interessante di questo schema è che indipendentemente dalla funzione F l'applicazione di questo protocollo a partire dal cyphertext con le chiavi in ordine inverso produce il plaintext. Quindi F può essere qualsiasi funzione.

Supponiamo ora di avere il seguente schema di Feistel dicodifica e decodifica, con quattro passi, dove non scambio i due lati, ma ci cambio solo il nome, eccetto per il passo finale.

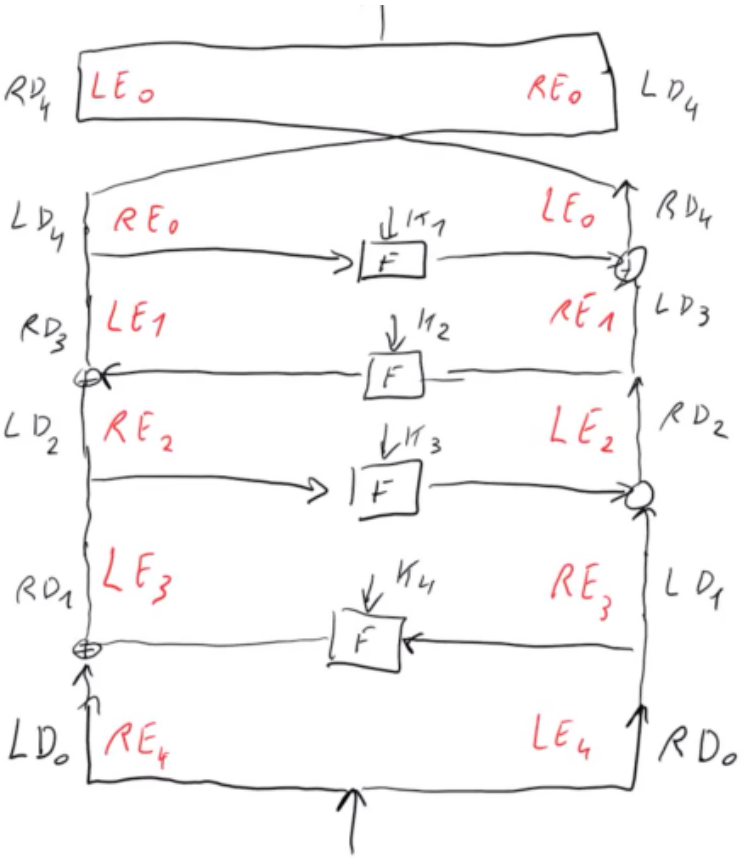


Dimostriamo ora che i due schemi sono identici:

$$
\begin{aligned}
&LD_1 = RD_0 = LE_4 = RE_3 \\
&RD_1 = LD_0 \oplus F(K_4, RD_0) = RE_4 \oplus F(K_4, RE_3) \\
&\qquad\qquad = LE_3 \oplus F(K_4, RE_3) \oplus F(K_4, RE_3) \\
&\qquad\qquad = LE_3
\end{aligned}
$$

Quindi la *Left Decryption 1* equivale alla *Right Encryption 3* e la *Right Decryption 1* equivale alla *Left Encryption 3*. Questa dimostrazione può essere estesa anche ai restanti tre passi, andando a confermare che lo schema di Feistel per codifica e decodifica è lo stesso.

Come si può vedere, la funzione F va solo ad aggiungere confusione; il prodotto del plaintext per la funzione viene automaticamente rimosso durante la decodifica, a prescindere da cosa sia. La F ha il compito di rendere difficile la decodifica non conoscendo quale sia la chiave.


RD4
LE0
RE0
LD4
LD4
RE0
K1
F
LE0
RD4
RD3
LE1
K2
F
RE1
LD3
LD2
RE2
K3
F
LE2
RD2
RD1
LE3
K4
F
RE3
LD1
LD0
RE4
LE4
RD0

Cenni alla teoria dei numeri 3

Definition 3.0.1 *Sia $m \geq 1$ un intero. Gli interi a e b sono congruenti modulo m se la loro differenza $a - b$ è divisibile per m. La formula*

$$a \equiv b \pmod{m}$$

indica che a e b sono congruenti in modulo m.

Un esempio può essere l'aritmetica delle "ore". Usando il modulo $m = 12$:

$$6 + 9 = 15 \equiv 3 \pmod{12} \qquad 2 - 3 = -1 \equiv 11 \pmod{12}$$

I numeri che soddisfano

$$a \equiv 0 \pmod{m}$$

sono tutti i numeri divisibili per m, ovvero tutti i suoi multipli km.

Dati $m \geq 1$ e a, b interi, allora

$$a \cdot b = 1 \pmod{m}$$

se e solo se $GDC(a, m) = 1$.

Dato $GDC(a, m) = 1$ allora esiste un inverso a^{-1} di $a \pmod{m}$ tale per cui $a^{-1}a \equiv 1 \pmod{m}$.

Esempio: dati $m = 5$ e $a = 2$, $GDC(5, 2) = 1$ e quindi esiste un inverso di $\overline{2} \pmod{5}$. L'inverso è 3, in quanto $2 \cdot 3 \equiv 1 \pmod{5}$.

Se a diviso m ha quoziente k e resto r, allora questo può essere riscritto come:

$$a = m \cdot k + r \qquad \text{con } 0 \leq r < m$$

3.1 Algebra modulo n

L'insieme Z_n viene definito come l'insieme delle classi di equivalenza $[0], [1], \ldots, [n-1]$ di una relazione che dice che a è congruente a b se e solo se a modulo n è uguale a b modulo n, dove il modulo è il resto della divisione per n:

$$Z_n = \{[0], [1], [2], ..., [n-1]\}$$
$$a \equiv_n b \iff a \pmod{n} = b \pmod{n}$$

Quando ho una relazione di equivalenza posso costruire gli insiemi degli insiemi che sono equivalenti tra di loro. L'insieme degli insiemi che sono equivalenti tra di loro forma una partizione dell'insieme di partenza; gli elementi della partizione vengono chiamati classi di equivalenza e si denotano mettendo tra parentesi quadre *[ ]* un elemento della classe di equivalenza.

Quindi quando parlo di un'algebra modulo n parlo di un insieme Z_n che è l'insieme delle classi di equivalenza della divisione per n; ogni singola classe di equivalenza può essere rappresentata da un qualsiasi elemento che vi appartiene. Posso infatti anche scrivere:

$$Z_n = \{0, 1, 2, ..., n-1\}$$

ovvero i primi n numeri naturali a partire dallo 0. La rappresentazione canonica in un'algebra a modulo n è un numero compreso tra 0 e $n-1$.

Il cifrario di Cesare lavora shiftando ogni lettera dell'alfabeto di un numero fissato di posizioni. Lo shift può essere descritto con un'aritmetica modulo 26:

$$(Cyphertext) \equiv (Plaintext\_Letter) + (Secret\_Key) \pmod{26}$$
$$(Plaintext\_Letter) \equiv (Cyphertext) - (Secret\_Key) \pmod{26}$$

Numeri razionali Questo tipo di notazione è già stata usata per definire i numeri razionali, che sono classi di equivalenza su coppie di numeri naturali dove due coppie sono equivalenti se hanno gli stessi prodotti incrociati (prodotto dei medi uguale prodotto degli estremi).

Ad esempio la coppia $\frac{a}{b}$ è un rappresentate delle coppie $\frac{2a}{2b}$, $\frac{3a}{3b}$ e così via. Tra tutti i possibili rappresentati di una classe ce n'è uno canonico, dove in questo caso è l'oggetto che si ottiene dalla riduzione ai minimi termini.

3.1.1 Somma nell'insieme Z_n

Definisco ora l'operazione di somma $(Z_n, +)$:

$$[a] + [b] = [a + b]$$

Questa definizione dice che la somma delle classi di equivalenza di $[a]$ e $[b]$ è uguale alla classe di equivalenza della somma di a e b $[a + b]$. Si considera buona solo se il risultato è indipendente dagli elementi che prendo come rappresentanti delle due classi da sommare. Questa operazione di somma gode di una serie di proprietà:

- La somma è commutativa: $\forall a, b : a + b = b + a$;
- La somma è associativa: $\forall a, b, c : (a + b) + c = a + (b + c)$;
- La somma ammette un elemento neutro e: $\exists\, e \forall a : a + e = e + a = a$ con $e = 0 \vee n$. Nell'algebra modulo n vale anche n (e anche $2n$, $3n$, ...) come elemento neutro in quanto è un elemento appartenente alla classe di equivalenza di 0. Similmente, nei razionali l'elemento neutro è si 1, ma anche $\frac{2}{2}$, $\frac{3}{3}$ e così via;
- La somma ammette l'elemento inverso: $\forall a\ \exists a^{-1} : a + a^{-1} = a^{-1} + a = e$. L'inverso di un numero è quel numero che messo nell'operazione di interesse con il numero originale mi dà l'elemento neutro. Ad esempio, il numero inverso nella somma è il negativo di quel numero (che nei naturali non esisterebbe, ed è per quello che si estende con gli interi); nei razionali l'inverso di $\frac{a}{b}$ nella moltiplicazione è $\frac{b}{a}$ (negli interi l'inverso della moltiplicazione non c'è, e quindi si estende con i razionali).
  In questo caso, posso scrivere l'elemento in più modi: $a^{-1} = -a = n - a$.

Una "cosa" che ha le proprietà di associatività, elemento neutro e elemento inverso è detta **gruppo**, mentre una che gode di tutte e quattro le proprietà è detta **gruppo abeliano**. Quindi l'insieme Z_n con l'operazione di somma è un gruppo.

Gli interi con l'operazione di moltiplicazione non formano un gruppo, in quanto non esiste l'elemento inverso. I razionali sono un gruppo con la somma, ma non con il prodotto, in quanto non esiste l'inverso dello 0. Se ai razionali però rimuovo l'elemento neutro della somma, diventano un gruppo anche con il prodotto.

I razionali sono un campo. Un insieme è un **campo** se è un gruppo con la somma e col prodotto a cui è rimosso l'elemento neutro della somma.

3.1.2 Prodotto in ($Z_n \backslash \{0\}, *$)

La moltiplicazione è definita coma segue:

$$[a] * [b] = [a * b]$$

Il prodotto, in Z_n senza lo 0, è commutativo e ha l'elemento neutro, ma non ha (sempre) l'elemento inverso:

$Z_{15} \backslash \{0\} = \{1, 2, 3, \ldots, 14\}$
L'inverso moltiplicativo di 2 è 8: $2 * 8 = 16 \equiv 1 \pmod{15}$
L'inverso moltiplicativo di 3 non esiste

In generale lavorando con l'operazione di moltiplicazione nei gruppo in modulo n si perde l'elemento inverso. Una possibilità è prendere un sottoinsieme di Z_n che garantisce l'elemento inverso.

3.2 Insieme Z_n^*

Sottoinsieme di Z_n tale per cui:

$$Z_n^* = \{a \in (Z_n \backslash \{0\}) \mid MCD(a, n) = 1\}$$

dove MCD indica il massimo comun divisore. Questo insieme contiene elementi a di Z_n tali per cui il massimo comun divisore tra a e n è 1. Contiene gli elementi che sono relativamente primi con n.

Se moltiplico tra loro due elementi di Z_n^* ho la garanzia di ottenere ancora un elemento di Z_n^* (se i due numeri non hanno fattori in comuni con n, allora anche il loro prodotto non avrà fattori in comune con n). La moltiplicazione quindi qui è **chiusa**. L'elemento neutro è 1.

Theorem 3.2.1 $\forall a, b \; \exists x, y : ax + by = MCD(a, b)$.

Se $a \in Z_n^*$, allora:

$$\begin{aligned} \exists x, y : ax + ny &= 1 \\ ax &= 1 - ny \\ ax &\equiv 1 \pmod{n} \end{aligned}$$

Quindi la classe a cui appartiene x è l'inverso di a. Ho confermato che Z_n^* con il prodotto è un gruppo.

Funzione di Eulero φ Dato l'insieme Z_n^* allora:

$$\varphi(n) = \mid Z_n^* \mid$$

$$\begin{aligned} \varphi(p) &= \mid p - 1 \mid \\ \varphi(pq) &= \mid (p-1) \cdot (q-1) \mid \end{aligned}$$

Sia $n = pq$ e $x < n$ un numero a caso. La probabilità che $x \in Z_n^*$ è:

$$Pr[x \in Z_n^*] = \frac{(p-1)(q-1)}{pq} = \frac{p-1}{p} \cdot \frac{q-1}{q} > \frac{1}{4}$$

3.3 Generatore di un gruppo

Sia $(G, *)$ un gruppo finito, e sia g un elemento di G, allora se prendo la serie di elementi

$$g = g^1, g * g = g^2, g * g * g = g^3, \ldots, g^{|G|}, g^{|G|+1}, \ldots$$

per il pumping lemma vi è almeno un elemento ripetuto e la sequenza diventa quindi ciclica (dall'elemento ripetuto in poi continua a ripetere la sotto-sequenza).

Theorem 3.3.1 $\forall$ *Gruppo G finito* $\forall a \in G : a^{|G|} = 1$.

Per questo teorema, se prendo un elemento qualsiasi di un gruppo e lo elevo alla cardinalità del gruppo, otteno 1. Quindi in questa catena l'elemento neutro c'è, che sarebbe $g^0 = 1$.

Se prendo tutte le potenze di g ottengo un **sottogruppo** di G, che è un sottoinsieme dell'insieme G che resta un gruppo. Se il sottogruppo che ottengo è l'intero gruppo G, allora G è **ciclico** e g è un **generatore** di G.

Theorem 3.3.2 (Teorema di Lagrange) *Se G' è un sottogruppo di G, allora $|G'| \Big| |G|$, ovvero il numero di elementi di G' è divisore del numero di elementi di G.*

Esempio

Provo ora a costruire tutti i sottogruppi di $Z_{15}^* = \{1, 2, 4, 7, 8, 11, 13, 14\}$:

$\underline{1}$

$\underline{2} * 2 = 4, \underline{4} * 2 = 8, \underline{8} * 2 = 16, 16 \pmod{15} = \underline{1}$

$\underline{4} * 4 = 16, 16 \pmod{15} = \underline{1}$

$\underline{7} * 7 = 49, 49 \pmod{15} = \underline{4}, 4 * 7 = 28, 28 \pmod{15} = \underline{13}, 13 * 7 = 91 \pmod{15} = \underline{1}$

$8, 4, 2, 1$

$11, 1$

$13, 4, 7, 1$

$14, 1$

Ho confermato che la cardinalità di tutti i generatori divide la cardinalità dell'insieme originale, ma nessuno di questi generatori è ciclico. Esistono generatori ciclici? Si, almeno quando n è primo. In generale dato p numero primo, il rispettivo insieme sarà $Z_p^* = \{1, ..., p-1\}$.

Theorem 3.3.3 *I gruppi Z_p^*, con p numero primo, sono gruppo ciclici.*

Esempio

Provo ora a cercare un generatore di $Z_7^* = \{1, 2, 3, 4, 5, 6\}$:

$1 \longrightarrow 1$ non è un generatore

$2, 4, 1 \longrightarrow 2$ non è un generatore

$3, 2, 6, 4, 5, 1 \longrightarrow 3$ è un generatore

Theorem 3.3.4 (Fermat's Little Theorem) *Sia p un numero primo e sia a*

un intero (con ka un possibile multiplo di a). Allora

$$a^{p-1} \equiv \begin{cases} 1 \pmod{p} & \text{se } p \neq ka \\ 0 \pmod{p} & \text{se } p \mid ka \end{cases}$$

Esempio

Vediamo la liste delle potenze dei numeri $1, 2, \ldots, 6$ in modulo con il numero primo 7:

$1^1 \equiv 1$	$1^2 \equiv 1$	$1^3 \equiv 1$	$1^4 \equiv 1$	$1^5 \equiv 1$	$1^6 \equiv 1$
$2^1 \equiv 2$	$2^2 \equiv 4$	$2^3 \equiv 1$	$2^4 \equiv 2$	$2^5 \equiv 4$	$2^6 \equiv 1$
$3^1 \equiv 3$	$3^2 \equiv 2$	$3^3 \equiv 6$	$3^4 \equiv 4$	$3^5 \equiv 5$	$3^6 \equiv 1$
$4^1 \equiv 4$	$4^2 \equiv 2$	$4^3 \equiv 1$	$4^4 \equiv 4$	$4^5 \equiv 2$	$4^6 \equiv 1$
$5^1 \equiv 5$	$5^2 \equiv 4$	$5^3 \equiv 6$	$5^4 \equiv 2$	$5^5 \equiv 3$	$5^6 \equiv 1$
$6^1 \equiv 6$	$6^2 \equiv 1$	$6^3 \equiv 6$	$6^4 \equiv 1$	$6^5 \equiv 6$	$6^6 \equiv 1$

Si può vedere che la colonna di destra, quella degli a^6, produce tutti 1.

3.4 Numero quadrato

Definition 3.4.1 *Sia p un numero primo dispari e sia $a \in Z_p^*$. Si dice che a è un residuo quadratico di modulo p se a è un quadrato modulo p, ovvero esiste un numero c tale che*

$$c^2 \equiv a \pmod{p}$$

In caso contrario a è detto non residuo quadratico modulo p.

Dato $(G, *)$, un gruppo G con l'operazione $*$, il numero di quadrati che contiene è esattamente uguale a $\frac{p-1}{2}$, ovvero a tutti e soli gli elementi g^{2i} (elementi con esponente pari). Ogni elemento g^{2i} ha due radici:

$$g^{2i} = \begin{cases} g^i \\ g^{i+\frac{p-1}{2}} \end{cases} \qquad \rightarrow (g^{i+\frac{p-1}{2}})^2 = g^{2i} \cdot g^{p-1} = g^{2i} \cdot 1$$

Poiché le due radici di un numero sono opposte, allora $g^{\frac{p-1}{2}} \equiv -1$. Quindi elevare un generatore alla cardinalità del gruppo dà l'unità, mentre elevarlo alla metà della cardinalità dà l'opposto dell'unità.

Quadrati in Z_{pq}^* Dato Z_{pq}^*, con p, q numeri primi, il numero di quadrati contenuto nell'insieme è $\frac{\varphi}{4}$, ovvero il numero di quadrati comuni a Z_p^* e Z_q^*.

Dato p numero primo, come è possibile capire se un dato numero a sia uguale a un quadrato modulo p? Escludendo il risolvere il problema del

logaritmo discreto per a e verificare se il risultato è pari o dispari, un modo per farlo è verificare se $x_i^2 = a$, per ogni $x_i \in 0, 1, 2, 3, 4, ..., p-1$. Esistono modi più efficienti?

Definition 3.4.2 (Simbolo di Legendre) *Il simbolo di Legendre è definito come segue:*

$$\left(\frac{a}{p}\right) = a^{\frac{p-1}{2}} \pmod{p}$$

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

Infatti:

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

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

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

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

Il simbolo di Legendre ritorna 1 se a è un quadrato modulo p o -1 se non lo è.

Si può calcolare questo algoritmo in tempo polinomiale, con numeri da migliaia di bit? Il simbolo di Legendre consiste nel calcolare una potenza, che equivale nella pratica a calcolare una serie di moltiplicazioni. Quindi la domanda diventa: esiste un algoritmo per eseguire moltiplicazioni in tempo polinomiale? Si, il **Fast Power Algorithm**.

Praticamente il simbolo di Legendre permette di ottenere il bit meno significativo del numero, che permette di capire immediatamente se il numero è pari (0) o dispari (1).

Dato $n = pq$, come è possibile capire se un numero a è un quadrato in Z_n^*? Il modo più semplice sarebbe fattorizzare n in p e q e calcolare il simbolo si Legendre di a rispetto ai due fattori. Poichè fattorizzare è difficile (non esistono algoritmi efficienti), allora anche determinare se $a \in Z_n^*$ dovrebbe essere difficile. In realtà non è proprio così.

Il simbolo di Jacobi, una generalizzazione del simbolo di Legendre, permette di determinare $\left(\frac{a}{n}\right)$ senza dover fattorizzare.

Definition 3.4.3 (Simbolo di Jacobi) *Siano a e n due interi, con n dispari e positivo. Supponendo che la fattorizzazione di n in numeri primi sia*

$$n = p_1^{e_1} \cdot p_2^{e_2} \cdot ... \cdot p_t^{e_t}$$

allora il simbolo di Jacobi $\left(\frac{a}{n}\right)$ *è definito dalla formula*

$$\left(\frac{a}{n}\right) = \left(\frac{a}{p_1}\right)^{e_1} \left(\frac{a}{p_2}\right)^{e_2} \dots \left(\frac{a}{p_t}\right)^{e_t}$$

Il simbolo di Jacobi $\left(\frac{a}{n}\right)$ gode delle seguenti proprietà:

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

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

Se n è un numero primo, allora il simbolo di Jacobi equivale al simbolo di Legendre. Se scompongo n in numeri primi, allora ogni simbolo di Jacobi $\left(\frac{a}{p_i}\right)$ mi ritorna 1 o -1:

$$\left(\frac{a}{n}\right) = \left(\frac{a}{pq}\right) = \left(\frac{a}{p}\right)\left(\frac{a}{q}\right)$$

Il simbolo di Jacobi, come detto all'inizio, è risolvibile con un algoritmo polinomiale che non richiede la scomposizione in fattori primi.

Tornando al caso iniziale $n = pq$, nel calcolo del simbolo di Jacobi $\left(\frac{a}{pq}\right)$ si possono incontrare tre casi:

1. a è un quadrato sia in Z_p^* che in Z_q^* (probabilità $\frac{1}{4}$):

$$\left(\frac{a}{pq}\right) = \left(\frac{a}{p}\right)\left(\frac{a}{q}\right) = 1 \cdot 1 = 1$$

2. a è un quadrato sia in Z_p^* ma non Z_q^* (probabilità $\frac{1}{4}$) o viceversa (probabilità $\frac{1}{4}$):

$$\left(\frac{a}{pq}\right) = \left(\frac{a}{p}\right)\left(\frac{a}{q}\right) = 1 \cdot (-1) = -1$$

$$\left(\frac{a}{pq}\right) = \left(\frac{a}{p}\right)\left(\frac{a}{q}\right) = (-1) \cdot 1 = -1$$

3. a non è un quadrato sia in Z_p^* ne in Z_q^* (probabilità $\frac{1}{4}$):

$$\left(\frac{a}{pq}\right) = \left(\frac{a}{p}\right)\left(\frac{a}{q}\right) = (-1) \cdot (-1) = 1$$

Dai tre casi sopra, si può capire che il simbolo di Jacobi è limitato rispetto al simbolo di Legendre, in quanto è in grado solamente di decidere se un numero è un non quadrato nel caso particolare in cui è un quadrato in Z_p^* e non quadrato in Z_q^* (o viceversa). Il simbolo di Jacobi si può quindi riassumere così:

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

Di conseguenza, determinare se dato un numero con simbolo di Jacobi 1 è quadrato o non quadrato è un problema difficile.

3.5 Fast Powering Algorithm

In molti crittosistemi come RSA e DH, i due attori necessiteranno di calcolare potenze di grandi dimensioni. Un modo per calcolare ad esempio $a^b \pmod n$ è per moltiplicazioni ripetute di a. Quindi

$$a_1 \equiv a \pmod n \qquad a_2 \equiv a_1 \cdot a \pmod n \qquad \dots \qquad a_b \equiv a_{b-1} \cdot a \pmod n$$

Se b è un numero elevato, l'algoritmo non è pratico. Un modo migliore per calcolare la potenza è usare l'espansione binaria di b per convertire il calcolo in una successione di quadrati e moltiplicazioni.

Ad esempio, dato $b = 75 = 1001011$, le rispettive potenze saranno:

$$2^6 + 2^3 + 2^1 + 2^0$$

Quindi:

$$a^b = a^{2^6+2^3+2^1+2^0} = a^{2^6} * a^{2^3} * a^{2^1} * a^{2^0}$$

Questa sequenza di valori è semplice da calcolare in quanto ogni numero nella sequenza è il quadrato del precedente:

$$a^{2^0} = a$$
$$a^{2^1} = a * a$$
$$a^{2^2} = a^{2^{1^2}}$$

Questo metodo è chiamato **iterative squaring** e permette di calcolare le potenze in un numero di moltiplicazioni proporzionale ai bit di b. Calcola via via le potenze di a, elevate a potenze di 2, e verifica a vedere se il corrispondente bit dell'espressione binaria di b vale 0 o 1: se vale 0 non fa nulla, se vale 1 moltiplica il risultato parziale per la potenze che ha calcolato.

Algorithm 1 Iterative squaring EXP(a, b)

```
y ← 1
t ← a
while b ≠ 0 do
    if b%2 == 1 then
        y ← y × t
    end if
    b ← b/2
    t ← t × t
end while
```

Complessità dell'algoritmo, in funzione dei bit di a e b:

Algorithm 2 Iterative squaring EXP(a, b) - complessità

```
y ← 1                                          ▷ lineare nel numero di bit di b
t ← a                                          ▷ lineare, devo copiare tutti i bit di a
while b ≠ 0 do                                 ▷ lineare
    if b%2 == 1 then                           ▷ costante, guardo il bit meno significativo
        y ← y × t                              ▷ quadratico nella lunghezza del numero
    end if
    b ← b/2                                    ▷ lineare o costante in base a come salvo il numero
    t ← t × t                                  ▷ quadratico
end while
```

L'algoritmo è quindi di complessità cubica, con il corpo del while è quadratico ripetuto per ogni bit di b, quindi l'algoritmo è cubico. Questo però non è corretto. Se ad esempio moltiplichiamo $100 * 100$, il risultato avrà quattro 0. Se moltiplichiamo il risultato ancora per 100, il numero di 0 finali diventeranno sei. Quindi in realtà la complessità sarebbe esponenziale, ovvero 2^k. Come possiamo riportare l'algoritmo a complessità cubica? Basta aggiungere dei modulo n nei calcoli intermedi:

Algorithm 3 Iterative squaring EXP(a, b) ottimizzato

```
y ← 1
t ← a
while b ≠ 0 do
    if b%2 == 1 then
        y ← (y × t)%n
    end if
    b ← b/2
    t ← (t × t)%n
end while
```

3.6 Introdurre casualità

Se moltiplico un quadrato per un non quadrato con $J = 1$ ottengo un non quadrato con $J = 1$; se moltiplico due non quadrati con $J = 1$ ottengo un quadrato. In generale:

- Se ho un oggetto e lo moltiplico per un quadrato mantengo la quadraticità;
- Se ho un oggetto e lo moltiplico per un non quadrato ne inverto la quadraticità.

Dato $a \in Z_n^*$ con $\left(\frac{a}{n}\right) = 1$, vogliamo ottenere un oggetto con la stessa quadraticità di a, ma distribuito uniformemente tra gli oggetti con la stessa quadraticità. Si può fare così:

Sia $x \in_R Z_n^*$ ($\in_R$== l'elemento viene preso uniformemente dall'insieme)

Ret $x^2 \cdot a$

3.7 Problemi difficili

Un problema si definisce difficile se non esiste un algoritmo efficiente (che funziona in quello che chiamiamo tempo polinomiale) che è in grado di risolverlo. Spesso non si sa se esiste un metodo per risolvere questi problemi difficili in modo efficiente, ma solo che attualmente non si conosce un modo per farlo. Tuttavia, la maggior parte dei problemi difficili utilizzati nella crittografia moderna sono stati ampiamente studiati e quindi si ha fiducia che l'ipotesi di difficoltà sia valida.

3.7.1 Problema del logaritmo discreto

Sia p un numero primo di grandi dimensioni. Allora esiste un generatore g tale per cui ogni elemento diverso da zero appartenente a Z_p^* è uguale a qualche potenza di g. Per il *Fermat's little theorem* $g^{p-1} = 1$ e non esiste alcuna potenza positiva di g più piccola che è uguale a 1. In modo equivalente, la lista di elementi

$$1,\ g,\ g^2,\ g^3,\ \dots,\ g^{p-1}$$

è una lista completa degli elementi di Z_p^*.

Definition 3.7.1 *Sia g un generatore di Z_p^* e sia h un elemento non zero appartenente all'insieme. Il problema del logaritmo discreto DLG è il problema di trovare un esponente x tale per cui*

$$g^x \equiv h \pmod{p}$$

Il numero x è chiamato logaritmo discreto di h per la base g.

Tuttavia, se c'è una soluzione, allora ce ne sono infinite, perché il piccolo teorema di Fermat dice che $g^{(p-1)} \equiv 1 \pmod{p}$. Quindi se x è una soluzione di $g^x = h$, allora anche $x + k(p-1)$ è una soluzione per ogni valore di k, perché

$$g^{x+k(p-1)} = g^x \cdot (g^{p-1})^k \equiv h \cdot 1^k \equiv h \pmod{p}$$

A volte, per concretezza, ci riferiamo al "logaritmo discreto" come all'intero x compreso tra 0 e $p-2$ che soddisfa la congruenza $g^x \equiv h \pmod{p}$.

3.7.2 Problema delle radici quadrate modulo pq / residuo quadratico

Sappiamo che la fattorizzazione di un numero e il logaritmo discreto di un numero sono due problemi difficili. Vogliamo ora dimostrare che il calcolare delle radici quadrate di un numero in modulo pq non è più semplice di fattorizzare.

Lemma 3.7.1 *Siano x, y due radici del quadrato $a \in Z_n^*$ con $x \neq \pm y \pmod{n}$. Allora $MCD(x+y, n)$ è p o q.*

Dimostrazione.

$$x^2 \equiv y^2$$
$$x^2 - y^2 \equiv 0$$
$$(x+y)(x-y) \equiv 0$$
$$(x+y)(x-y) = kn$$

il resto della divisione per n è 0, quindi il prodotto è un multiplo di n

Si presentano tre casi:

- Supponendo che $p \mid (x+y)$, è possibile che $q \mid (x+y)$? Se è vero allora $n = pq$ divide $(x+y)$. Allora $(x+y)$ deve essere un multiplo di n. $(x+y) \equiv 0 \pmod{n}$ e quindi $x = -y \pmod{n}$ (per definizione di congruenza modulo n). IMPOSSIBILE.
- Supponendo $q \mid (x+y)$, è possibile che $p \mid (x+y)$? IMPOSSIBILE.
- Infine, è possibile $p \nmid (x+y)$ e $q \nmid (x+y)$? p e q sono fattori del prodotto $(x+y)(x-y)$ e quindi devono essere fattori di almeno uno dei due oggetti. Se nessuno dei due è divisore di $(x+y)$, allora devono essere divisori di $(x-y)$, ma allora $x \equiv y$. IMPOSSIBILE.

□

Supponiamo che esista un algoritmo $A \in PPT$ (Probabilistic Polinomial Time) in grado di calcolare la radice quadrata di un numero:

Algorithm 4 Algoritmo di fattorizzazione FATTORIZZA(n)

```
x ∈_R Z_n^*
y ← A(x^2)
z ← MCD(x + y, n)
if z ≠ n then
    return z
else
    return FATTORIZZA(n)
end if
```

FATTORIZZA prende un quadrato a caso x^2, di cui conosce una radice x (presa uniformamene tra le sue quattro radici). Il quadrato viene quindi passato all'algoritmo A, che restituisce un'altra radice quadrata, scelta in qualche modo diverso da come vene scelto x. Qual è la probabilità che A ritorni la radice iniziale presa o il suo opposto? 1/2. Quindi sempre con probabilità 1/2 l'algoritmo A ritornerà le radici $\pm y$.

La probabilità con cui l'algoritmo ritorna la radice giusta è 1/2. Come si può portare la probabilità a 1? Basta ripetere il test un numero di volte. Ad esempio, riprovando una seconda volta la probabilità di fallire diventa $\frac{1}{4}$, alla terza $\frac{1}{8}$ e così via (distribuzione geometrica).

Supponendo che un attaccante disponga di un algoritmo A in grado di calcolare le radici di un quadrato, allora sarà in grado anche di fattorizzare un numero. Di conseguenza il problema delle radici in modulo pq è

dimostrabile sicuro. Infatti risolvere questo problema equivale a risolvere il problema di fattorizzare.

3.8 Altro

Problema del logaritmo discreto $DL(a, g)$ Dato il gruppo Z_p^*, un suo generatore g e un elemento $a \in Z_p^*$, allora esisterà una potenza i del generatore tale per cui:

$$g^i \equiv g^{i+k(p-1)} = a$$

visto che le varie potenze del generatore enumerano l'intero gruppo. Dato g e g^i, trovare i è molto difficile. Questo problema è detto problema del logaritmo discreto (discreto in quanto lavoro su un insieme finito) e ad oggi non esistono algoritmi efficienti per risolverlo.

One way function Una funzione facile da calcolare ma difficile da invertire è detta one-way function. Un esempio di funzione di questo tipo è quella del logaritmo discreto: è semplice calcolare g^i, ma è difficile calcolare i da g^i.

Come scegliere casualmente un numero primo di k bit L'uso della casualità permette di risolvere molti problemi che normalmente non saremmo in grado di risolvere. Per scegliere casualmente un numero primo, sparo un sequenza di k bit a caso. Esiste un teorema sulla densità dei numeri primi nei naturali tale per cui questa è proporzionale al numero di bit usati per rappresentare il numero. Dati k bit, per questo teorema $1\backslash k$ volte ottengo un numero primo. Quindi mi servono in media k tentativi casuali per generarlo.

Test di primalità di un numero Esistono più algoritmi per verificare se un numero è primo. Ad oggi il migliore funziona come segue: esiste un test che è possibile fare usando un numero e un numero più piccolo. Questo test disce si o no. Se dice si, il numero è composto; se dice no, non sa se il numero è composto (per quel numero è primo). Metà dei numeri più piccoli di n sono tesimoni del fatto che un numero non sia primo. Quindi prendo un numero più piccolo del numero che ho generato e faccio il test. Se il test mi dice si, il numero non è primo; se mi dice no riprovo con un numero più piccolo. Eseguo il test per x volte. Se tutte le volte mi dice che è primo allora lo accetto come primo. La probabilità di accettare un numero che non è primo è $\frac{1}{2^x}$.

Crittografia a chiave pubblica | 4

La crittografia a chiave pubblica prevede l'uso di due chiavi: P_k (Public key) e S_k (Secret Key). La chiave privata è posseduto da un solo agente; la chiave pubblica è posseduta da tutti. La P_k è usata per codificare il messaggio; la S_k è usata per decodificarlo. Senza S_k non deve essere possibile ricavare il plaintext dal cyphertext. Con questo sistema chiunque può codificare messaggi, ma solo che possiede la secret key può leggerli.

Componenti del sistema:

$$G : 1^K \to (P_k, S_k)$$
$$E : (m, P_k) \to E(m, P_k)$$
$$D : (c, S_k) \to D(c, S_k)$$

In questo sistema vale sempre:

$$\forall m \; D(E(m, P_k), S_k) = m$$

L'input 1^k del generatore G è il security parameter, che in questo caso rappresenta la lunghezza, in bit, della chiave. Non sempre rappresenta la lunghezza della chiave; in generale il security parameter è quel numero/parametro tale per cui al suo crescere il protocollo diventa sempre più sicuro.

4.1 Scambio di chiavi Diffie-Hellman

L'algoritmo di scambio di chiavi Diffie-Hellman nasce per risolvere il seguente problema: Alice e Bob vogliono condividere una chiave segreta da utilizzare in una cifratura simmetrico, ma il loro unico mezzo di comunicazione è insicuro.

Per prima cosa Alice e Bob concordano su un numero primo p e un generatore g di Z_p^*, che rendono pubblici. A questo punto Alice sceglie un intero x che terrà segreto e Bob un intero y che terrà anch'egli segreto. Bob e Alice usano i loro numeri interi segreti per calcolare

$$\underbrace{g^x \pmod p}_{Alice} \qquad \text{e} \qquad \underbrace{g^y \pmod p}_{Bob}$$

Il passo successivo consiste nello scambiare, sul canale non sicuro, i due risultati: Alice invia g^x a Bob e Bob invia g^y a Alice. Infine Alice e bob utilizzano i rispettivi interi segreti per calcolare

$$\underbrace{(g^y)^x \pmod p = g^{yx} \pmod p}_{Alice} \quad \text{e} \quad \underbrace{(g^x)^y \pmod p = g^{xy} \pmod p}_{Bob}$$

I due valori calcolati sono gli stessi.

Supponendo che un attaccante ascolti l'intera conversazione, egli potrà ricostruire la chiave segreta condivisa g^{xy} solo se riuscirà a trovare x tale per cui $g^x \pmod p$ equivale al valore inviato da Alice sul canale. Stesso discorso per y. Le linee guida attuali suggeriscono che Alice e Bob scelgano un p primo avente circa 1000 bit.

L'attaccante conosce $g^a \pmod p$ e $g^b \pmod p$, ma anche g e p. Quindi se riuscisse a estrarre uno solo degli esponenti a e b da $g^a \pmod p$ e $g^b \pmod p$, allora sarebbe in grado di calcolare la chiave di sessione g^{ab}. Potrebbe sembrare che Alice e Bob siano al sicuro a condizione che l'attaccante non sia in grado di risolvere il Problema del Logaritmo Discreto (DLP), ma questo non è del tutto corretto. È vero che un metodo per trovare il valore condiviso di Alice e Bob è risolvere il DLP, ma non è questo il problema preciso che l'attaccante deve risolvere. La sicurezza della chiave condivisa di Alice e Bob si basa sulla difficoltà del seguente problema, potenzialmente più semplice.

Definition 4.1.1 *Sia p un numero primo e g un generatore di Z_p^*. Il problema di Diffie-Hellman (DHP) è il problema di calcolare il valore $g^{ab} \pmod p$ a partire dai valori conosciuto $g^a \pmod p$ e $g^b \pmod p$.*

È chiaro che il DHP non è più difficile del DLP. Se l'attaccante riesce a risolvere il DLP, allora può calcolare gli esponenti segreti di Alice e Bob. Ma il viceversa è meno chiaro. Supponiamo che l'attaccante abbia un algoritmo che risolva efficientemente il DHP. Può usarlo anche per risolvere in modo efficiente il DLP? La risposta ad oggi non è nota, e quindi DF non è dimostrabilmente sicuro (non si è in grado di dimostrare che rompere DH è tanto difficile quando risolvere il DLP).

4.1.1 Ipotesi di Diffie-Hellman

DH non è dimostrabilmente sicuro secondo i problemi difficili per ora trovati. D'altro canto però, qualcuno potrebbe costruire degli algoritmi con alla base di DH. In questi casi tali algoritmi si dimostrano sicuri sotto l'ipotesi che DH sia sicuro. Per fare ciò nasce l'ipotesi di DH.

1: Praticamente se do la soluzione all'attaccante lui non riesce a capire che è la soluzione.

Supponiamo che sia difficile distinguere una tripla (g^x, g^y, g^{xy}), con $x, y \in \{1, ..., p-1\}$, da una tripla (g^x, g^y, g^z), con $x, y \in \{1, ..., p-1\}$[1]. Se l'attaccante identifica x, allora riesce a riconoscere la tripla corretta. In caso contrario, ha una possibilità su due di indovinare la tripla corretta. Infatti, essendo la probabilità di indovinare x $Pr = \frac{1}{2^k}$, con k i bit della chiave, allora probabilità totale di indovinare la tripla è:

$$Pr = \frac{1}{2^k} + \frac{1}{2}(1 - \frac{1}{2^k}) = \frac{1}{2} + \frac{1}{2^{k+1}}$$

L'idea è che anche se un attaccante dispone di un algoritmo per indovinare la tripla giusta, questo gli porta un vantaggio molto minimale rispetto allo sparare a caso.

4.2 Crittosistema di Rivest, Shamir e Adleman (RSA)

RSA si basa sulla seguente dicotomia:

- **Setup:** Siano p, q due numeri primi di $k/2$ bit ognuno (comunque di grandi dimensioni) e siano e, c due interi;
- **Problema:** Risolvere la congruenza $x^e \equiv c \pmod{pq}$ per la variabile x;
- **Facile:** Bob, che conosce i valori di p e q, può trovare facilmente x (teoremi sopra);
- **Difficile:** Un attaccante, che non conosce i valori dei due primi, non può calcolare facilmente x;
- **Dicotomia:** Risolvere $x^e \equiv c \pmod{pq}$ è facile per chi conosce specifiche informazioni aggiuntive (ovvero i valori di p e q), ma è difficile per chiunque altro.

La chiave segreta di Bob è una coppia di grandi primi p, q. La sua chiave pubblica è la coppia (n, e) costituita dal prodotto $n = pq$ e da un esponente e che è relativamente primo a $(p-1)(q-1)$. Alice prende il suo testo in chiaro e lo converte in un intero m compreso tra 1 e n. Crittografa m calcolando

$$c \equiv m^e \pmod{n}$$

L'intero c è il suo testo cifrato, che invia a Bob. È quindi semplice per Bob risolvere la congruenza $x^e \equiv c \pmod{n}$ per recuperare il messaggio di Alice m, perché Bob conosce la fattorizzazione $n = pq$. Un attaccante, d'altra parte, può intercettare il testo cifrato c, ma a meno che non sappia come fattorizzare n, presumibilmente avrà difficoltà a risolvere $x^e \equiv c \pmod{n}$.

4.2.1 Generazione delle chiavi, encryption e decryption

Generazione chiavi Output: chiave pubblica (n, e) e chiave privata (n, d).

1. Scegli la grandezza del security parameter k.
2. Scegli due numeri p, q di $k/2$ bit ciascuno.
3. Calcola $n = pq$.
4. Calcola $\varphi(n) = (p-1)(q-1)$.
5. Calcola l'esponente $e \in_R Z^*_{\varphi(n)}$ tale per cui

$$MCD(e, \varphi) = 1$$

6. Calcola la chiave privata d tale per cui

$$de \equiv 1 \pmod{\varphi(n)}$$

Encryption Data la chiave pubblica (n, e) e il messaggio m, la funzione di encryption è

$$c = m^e \pmod{n}$$

Decryption Data la chiave privata (n, d) e il cyphertext c, la funzione di decryption è

$$m = c^d \pmod{n}$$

La condizione $MCD(e, \varphi(n)) = 1$ assicura che assicura che l'inverso $e \pmod{\varphi(n)}$ esista, in modo che ci sia sempre una chiave privata d.

L'essenza di RSA può essere rappresentata con la seguente equazione:

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

4.2.2 Problema delle radici quadrate (versione matematica)

Se rimpiazziamo, nel *Fermat's little theorem*, il numero primo p con un numero non primo n vale ancora $a^{p-1} \equiv 1 \pmod{p}$? In generale no. Esiste però un caso particolare, quando $n = pq$ (con p, q primi), che è il caso più importante per la crittografia.

Theorem 4.2.1 (Formula di Eulero per pq) *Siano p, q due numeri primi distinti e sia $g = MCD(p-1, q-1)$. Allora*

$a^{(p-1)(q-1)/g} \equiv 1 \pmod{pq}$ *per tutti gli a che soddisfano MCD(a, pq)=1*

In particolare, se p e q sono due numeri primi dispari, allora

$a^{(p-1)(q-1)/2} \equiv 1 \pmod{pq}$ *per tutti gli a che soddisfano MCD(a, pq)=1*

Lo scambio di chiavi di DH si affida alla difficoltà di risolvere equazioni nella forma

$$a^x \equiv b \pmod{p}$$

dove a, b, p sono quantità note e x è la variabile da trovare. RSA, invece, si affida alla difficoltà di risolvere equazioni nella forma

$$x^e \equiv c \pmod{n}$$

dove le quantità note sono e, c, n e la x è sconosciuta. In altre parole, RSA si affida all'assunzione che è difficile prendere radici e-esime modulo n.

Se n è primo, risulta che è relativamente facile calcolare la radice esima in modulo n, come mostrato di seguito.

Theorem 4.2.2 *Sia p un numero primo e $e \geq 1$ un intero che soddisfa $MCD(e, p-1) = 1$. Allora e ha un inverso modulo p, ovvero*

$$de \equiv 1 \pmod{p-1}$$

Quindi la congruenza

$$x^e \equiv c \pmod{p}$$

ha l'unica soluzione $x \equiv c^d \pmod{p}$.

Il teorema sopra mostra che è facile prendere una radice e-esima se il modulo è un numero primo p. La situazione per un modulo complesso n è simile, ma è semplice da calcolare solo se i fattori di n sono conosciuti.

Vediamo ora il caso in cui $n = pq$ è il prodotto di due numeri primi.

Theorem 4.2.3 *Siano* p, q *due primi distinti e sia* $e \geq 1$ *tale per cui* $MCD(e, (p-1)(q-1)) = 1$. *Allora* e *ha un inverso modulo* $(p-1)(q-1)$, *ovvero*

$$de \equiv 1 \pmod{(p-1)(q-1)}$$

Quindi la congruenza

$$x^e \equiv c \pmod{pq}$$

ha l'unica soluzione $x \equiv c^d \pmod{pq}$.

4.2.3 Debolezze di RSA

RSA ha diverse debolezze:

- In passato è stato attaccato a causa di una sua errata implementazione. Il protocollo vuole che il valore e venga scelto **casualmente** tra tutti i numeri relativamente primi in φ_n. Gli implementatori usavano però funzioni random a 64 bit per generare e o d, andando a settare i bit "mancanti" a zero. Supponendo che il valore debba essere a 564 bit, il problema è che la probabilità di generare casualmente un numero con i primi 500 bit è estremamente bassa $(\frac{1}{2^{500}})$;
- Supponendo di lavorare in Z_n^*, se il messaggio cifrato m^e è più piccolo di n, calcolare la radice e-esima (come visto sopra) è facile. Quindi scenari con messaggi piccoli sono problematici per RSA;
- Supponiamo che l'insieme dei possibili messaggi che Alce possa spedire a Bob sia piccolo. Allora un attaccante potrebbe semplicemente cifrare tutte le possibili combinazioni e confrontarle con cyphertext inviato da Alice per capire qual è il messaggio originario. Quindi messaggi sparsi sono problematici in RSA;
- Siamo sicuri che tutti i bit dell'inverso del cyphertext siano difficili da calcolare? Ad esempio, col simbolo di Legendre, è possibile ottenere il bit meno significativo del logaritmo discreto. Per poter dire che un sistema è sicuro non basta che dal cyphertext non si possa ottenere il plaintext, ma non deve essere possibile estrarre **alcuna informazione parziale**. Per RSA non è dimostrato;
- Dati due messaggi ripetuti inviati nella stessa "comunicazione", RSA genererà lo stesso cyphertext.

RSA è sicuro perché ad oggi non si conosce un algoritmo probabilistico polinomiale per il calcolo della radice e-esima di un numero. Ma cosa

vuol dire questo? Che non esiste esiste un algoritmo che ritorna la radice e-esima? Ma se allora esistesse un algoritmo che ritorna $\sqrt[e]{a}$ con probabilità 1/100? In questo caso, cosa vuol dire con probabilità 1/100? Che l'algoritmo, se reiterato sullo stesso numero, in media con 100 tentativi ritorna il risultato corretto[2], oppure che funziona correttamente solo su 1/100 degli a possibili?

2: Applichiamo l'algoritmo una prima volta, se fallisce lo riapplichiamo una seconda e così via finché non ritorna il risultato corretto.

Quindi il fatto che l'algoritmo calcoli la radice e-esima di a con probabilità 1/100 può voler dire:

1. Dato un a a caso, la probabilità che l'algoritmo arrivi a calcolare la radice e-esima è 1/100;
2. Fissato a, la probabilità che l'algoritmo ritorni il risultato è 1/100.

Il "modo" di trasformare la probabilità da 1/100 a 1 descritta appena sopra funziona solo nel secondo caso. Quindi se fossimo nel primo scenario saremmo al sicuro? In realtà no, anche nel primo caso la probabilità può essere trasformata in 1:

Definition 4.2.1 *Dato $a \in Z_n^*$ fissato e scelto $R \in_R Z_n^*$, allora aR^e è un numero distribuito uniformemente tra gli elementi di Z_n^*.*

Dimostrazione. Essendo l'elevamento R^e invertibile, in quanto e è scelto in modo che la radice e-esima dia esattamente R, allora la funzione che mappa R in R^e è una funzione invertibile tra due insiemi che hanno la stessa cardinalità, e quindi biettiva. Un elemento scelto uniformemente in Z_n^* viene mappato in un elemento scelto uniformemente in Z_n^*. Se prendiamo un elemento distribuito uniformemente in Z_n^* e lo moltiplichiamo per a, otteniamo ancora un elemento distribuito uniformemente in Z_n^*, essendo anche la moltiplicazione una funzione invertibile. Quindi aR^e è un elemento casuale di Z_n^*. □

Siamo partiti da un elemento a fissato e siamo riusciti a costruire un elemento casuale in Z_n^*. Se a questo elemento cerchiamo di applicare l'algoritmo, 1/100 volte abbiamo la risposta. Quindi abbiamo resto l'algoritmo del primo caso indipendente dal numero fissato a. Dato il nuovo input, l'algoritmo calcolerà con $Pr = 1/100$ la radice $\sqrt[e]{aR^e}$. Da questo risultato riusciamo a ottenere la radice e-esima di a? Si, in quanto

$$\sqrt[e]{aR^e} = R\sqrt[e]{a} = \frac{R\sqrt[e]{a}}{R} = \sqrt[e]{a}$$

Siamo riusciti quindi a costruire un algoritmo che, a partire da una black-box con a casuale calcola correttamente $\sqrt[e]{a}$ con probabilità 1/100, riesce a calcolare correttamente $\sqrt[e]{a}$ con probabilità 1/100 con a fissato.

Come verifichiamo che la radice ritornata sia quella corretta? Basta semplicemente prendere il risultato e elevarlo a e.

In generale, diciamo che un sistema è attaccabile quando il tempo necessario ad attaccarlo è polinomiale. Questo vuol dire che se esistesse un qualsiasi algoritmo in grado di calcolare $\sqrt[e]{a}$ con probabilità polinomiale in k, noi saremmo in grado di trovare un algoritmo che calcola la stessa cosa con un tempo medio polinomiale in k. Poiché partiamo dall'idea

che non esiste un algoritmo PPT in grado ci calcolare $\sqrt[e]{a}$, non esiste nemmeno un algoritmo che riesce a calcolarla con una probabilità che sia polinomialmente piccola. La probabilità di successo di un eventuale algoritmo è più piccola di qualsiasi polinomio, dove per polinomio si intende

$$\forall c : Pr[correttezza] < \frac{1}{k^c}$$

Questa formula non è del tutto corretta, in quanto, solitamente, con chiavi piccole i protocolli sono attaccabili. La forma corretta è la seguente

$$\exists c \; \forall \bar{k} \; \exists k > \bar{k} \; \left| Pr[attacco\_ha\_successo] \right| < k^{-c}$$

Fissato un livello di difficoltà c, vogliamo che la probabilità di successo di un attaccante sia più piccola del polinomio costruito col quel livello di difficoltà k^{-c}. Non abbiamo però garanzia che questo sia sempre vero, ma se fissiamo una chiave sufficientemente lunga $\bar{k}$, allora possiamo rendere la probabilità di attacco inferiore al polinomio k^{-c}. Secondo questa definizione, idealmente aumentando la dimensione della chiave l'attacco diventa più difficile.

Dato n, è vero che non esiste un algoritmo PPT in grado di calcolare $\sqrt[e]{a}$? Esiste, ed è quello che conosce la fattorizzazione di n. Se fissiamo n, l'algoritmo che calcola $\sqrt[e]{a}$ esiste ed è quello che conosce l'informazione **trapdoor**, ovvero la fattorizzazione di n.

Quando diciamo che una funzione è difficile da calcolare dobbiamo essere precisi, dobbiamo dire input e output della funzione. Per RSA si pensa che non esista un algoritmo PPT che dati n, e, a calcola $\sqrt[e]{a}$ in Z_n^*.

4.2.4 Complessità di RSA

Da un punto di vista computazionale, quanto è complesso RSA? Quanti passi di computazione dobbiamo fare con una chiave di k bit? La complessità di m^e è polinomiale: sommare due numeri a k bit costa k, moltiplicarli costa k^2, elevarli tra loro costa k^3 (grazie all'iterative squaring). Quindi codifica e decodifica di RSA costano k^3. Se lavoriamo con 1'000 bit, allora queste operazioni ci costano un miliardo di operazioni.

Diversamente AES ha un costo di codifica sul singolo blocco che è costante, in quanto alla fine lavora su tabelle e registri. Per questo motivo la crittografia a chiave pubblica non è generalmente usata per cifrare i messaggi, ma per cifrare e scambiare una chiave di sessione usata poi in codifiche simmetriche, molto meno costose.

Diversamente da RSA, che permette di inviare una chiavi qualsiasi, Diffie-Hellman permette di scambiare un'unica chiave, sempre che non si crei una nuova chiave di DH ogni volta.

Crittografia a blocchi 5

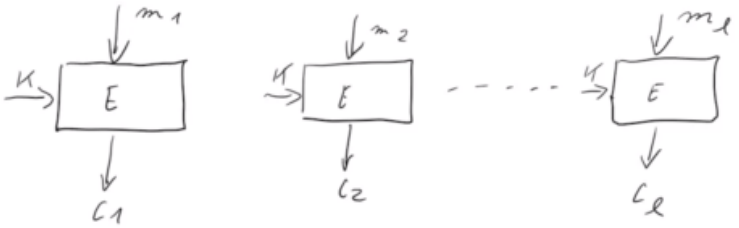
I protocolli visti fin'ora permettono di codificare messaggi di dimensioni precise (per RSA dipende dalla dimensione della chiave). Se però dovessimo inviare un messaggio che supera queste dimensioni? Dato un messaggio di n bit, con $n > k$, possiamo suddividere il messaggio in una serie di blocchi $m = m_1\ m_2\ ...\ m_l$, di k bit ciascuno, cifrare ciascun blocco e inviare la sequenza di questi blocchi cifrati.

La crittografia a blocchi (block cypher) differisce da quella a flusso (stream cypher), che cifra il messaggio bit a bit man mano che viene reso disponibile.

Esistono più tecniche per cifrare i blocchi.

5.1 Electronic Code Book (ECB)

ECB è il metodo più semplice per cifrare un messaggio diviso in blocchi: codifichiamo separatamente ciascun blocco con la chiave k e il crittosistema E e poi ricostruiamo la sequenza in ordine.



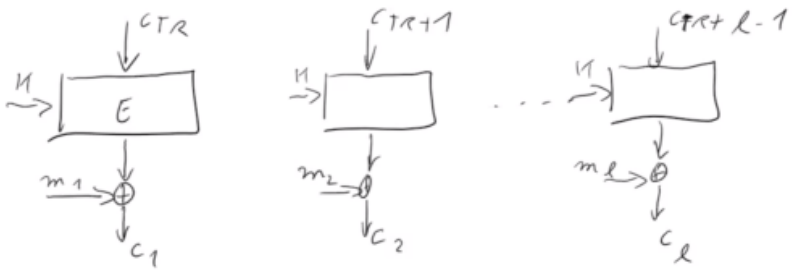
Questo sistema ha molte debolezze:

- Blocchi che si ripetono producono lo stesso cyphertext, quindi sono riconoscibili;
- L'ordine dei blocchi, se scambiato, può generare messaggi ancora sensati. Noi possiamo ad esempio invertire i blocchi 2 e 7 e ottenere una manipolazione nota del plaintext pur non conoscendo il plaintext lavorando direttamente sul cyphertext (**malleabilità**);
- È possibile fare un collage di blocchi provenienti da messaggi diversi ("Paga 100 euro a" -> "paga 10000 euro a").

L'unico vantaggio che ha è la possibilità di parallelizzare il lavoro.

5.2 Counter (CTR)

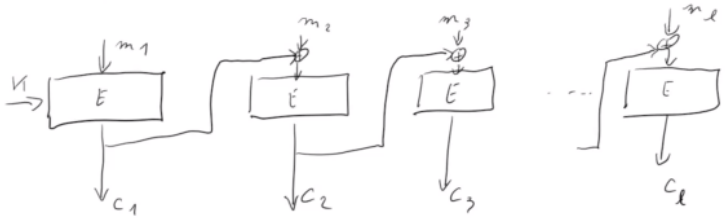
CTR prevede di prendere di codificare un counter ctr con il crittosistema E e la chiave k, e mettere in XOR il risultato col blocco del messaggio da cifrare. Ad ogni cifratura, il counter viene incrementato di 1. Il cyphertext finale è dato dalla ricomposizione dei risultati per ciascun blocco.



Con questo schema blocchi ripetuto generano cyphertext differenti e non è possibile fare collage di blocchi provenienti da messaggi differenti, a patto che il contatore sia differente. La debolezza di questo schema è il contatore: mittente e destinatario devono condividere il contatore e tenerlo sincronizzato.

5.3 Cypher Block Chaining (CBC)

In CBC codifichiamo i blocchi del messaggio con il crittosistema E e la chiave k e, in ogni blocco successivo al primo, mettiamo in XOR il blocco con il cyphertext del blocco che lo precede.



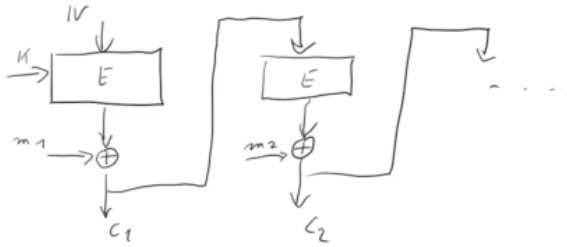
Il cyphertext del blocco precedente viene usato per modificare la sequenza di bit del blocco successivo. Con questo schema il collage non funziona e non esiste alcun contatore da mantenere condiviso. Il problema sta nel fatto che, se il cyphertext di un blocco contiene errori, questi vengono poi propagati in tutti i blocchi che lo seguono (es: in seguito ad errori di trasmissione).

Soffre inoltre del problema dei messaggi ripetuti. Questo problema può essere risolto aggiungendo un blocco casuale all'inizio del messaggio.

Vogliamo quindi trovare un sistema per evitare che l'errore di trasmissione si propaghi agli altri blocchi. Infatti qui, ad esempio, il blocco

5.4 Cypher Feedback

Variazione del CBC che permette di trasmettere messaggi codificati bit per bit, piuttosto che blocco per blocco. Infatti in CBC il blocco c_2, ad esempio, non può essere inviato finché non abbiamo ottenuto completamente il blocco m_2.

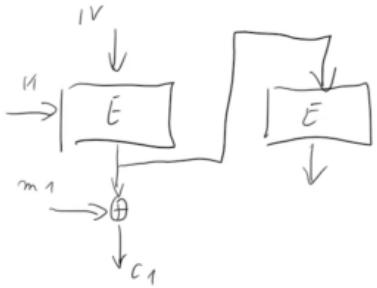


Questo sistema utilizza un Initialization Vector IV (da concordare con la controparte) che viene codificato con il crittosistema E e la chiave k. Il risultato viene messo in XOR con il blocco da cifrare. Il risultato di questo viene usato sia come cyphertext del blocco corrente sia come nuovo IV per la codifica del blocco successivo.

Il vantaggio di questo schema è che possiamo inviare il risultato dello XOR col blocco alla codifica del successivo bit a bit mentre viene generato. Lo schema che abbiamo costruito è a blocchi ma permette di fare la trasmissione a stream. Non è resistente ad errori di trasmissione.

5.5 Output Feedback

In questo caso, diversamente da Cypher Feedback, inviamo alla codifica del blocco successivo l'output del crittosistema E, e non il risultato dello XOR.



Con questo scema un errore di trasmissione non ha influenza nel blocco successivo. Infatti, conoscendo l'IV, siamo in grado di calcolare tutte le uscite di ogni blocco senza dover leggere i cyphertext dei blocchi precedenti. Praticamente in ogni blocco usiamo una codifica della codifica di IV.

Codifica di un singolo bit e di sequenze di bit

6

Supponiamo che Alice voglia usare un crittosistema a chiave pubblica per inviare a Bob 1 bit di informazione. A prima vista questa disposizione può sembrare intrinsecamente non sicura, in quanto tutto quello che deve fare un attaccante è cifrare i due possibili plaintext $m = 1$ e $m = 0$ e confrontarli con il cyphertext inviato da Alice. In generale, in ogni crittosistema in cui il pool di possibili messaggi è piccolo, tutto quello che deve fare un attaccante è cifrare tutte le possibilità e confrontarle con cyphertext inviato.

6.1 Crittosistema di Goldwasser-Micali

La **cifratura probabilistica** è stata inventata da Goldwasser e Micali per aggirare questo problema. L'idea è che Alice scelga un plaintext m e una stringa casuale di dati r che andrà a cifrare con la chiave pubblica di Bob. Idealmente, poiché r potrebbe variare su tutti i suoi possibili valori, il cyphertext (m, r) varierà casualmente su tutti i possibili cyphertext generabili. Più precisamente, per ogni m_1, m_2 fissati e per un r variabile, i cyphertext $E(m_1, r)$ e $E(m2, r)$ dovrebbero essere indistinguibili. Ovviamente per Bob non è necessario recuperare l'intero messaggio (m, r) ma solamente m, quando decodifica.

A partire da questa idea, Goldwasser e Micali hanno creato un schema che, sebbene impratico in quando codifica un bit per volta, è semplice da descrivere e analizzare. Lo scema si basa sulla difficoltà del seguente problema:

Definition 6.1.1 (Problema del residuo quadratico) *Siano p, q due numeri primi segreti di $k/2$ bit ciascuno e sia $n = pq$ pubblico. Per un dato numero a, determinare se a è un quadrato modulo n, ovvero determinare se esiste un intero x tale per cui $x^2 \equiv a \pmod{n}$.*

Bob, che conosce la fattorizzazione di n, può risolvere il problema molto facilmente in quanto

$$a \text{ è un quadrato modulo } n \iff \left(\frac{a}{p}\right) = 1 \wedge \left(\frac{a}{q}\right) = 1$$

Un attaccante invece ha molta più difficoltà in quanto conosce solo il valore n. Il massimo che può fare è calcolare $\left(\frac{a}{n}\right)$, ma questo non dice se a è un quadrato modulo n.

6.1.1 Generazione delle chiavi, encryption e decryption

Generazione chiavi Bob sceglie:

- p, q primi segreti di $k/2$ bit.

- $y \in_R Z_n^*$ non quadrato con $(\frac{y}{p}) = (\frac{y}{q}) = -1$.
- Pubblica $n = pq$ e y.

Encryption Alice:

- Sceglie $m \in \{0, 1\}$.
- Sceglie $r \in_R Z_n^*$.
- Usa la chiave pubblica (n, y) per calcolare:

$$c = \begin{cases} r^2 \pmod{n} & \text{se } m = 0 \\ yr^2 \pmod{n} & \text{se } m = 1 \end{cases}$$

- Invia c a Bob.

Decryption Bob decifra il messaggio:

$$m = \begin{cases} 0 \pmod{n} & \text{se } \left(\frac{c}{p}\right) = \left(\frac{c}{q}\right) = 1 \\ 1 \pmod{n} & \text{se } \left(\frac{c}{p}\right) = \left(\frac{c}{q}\right) = -1 \end{cases}$$

Il crittosistema di Goldwasser-Micali funziona come previsto in quanto:

$$\left(\frac{c}{p}\right) = \begin{cases} \left(\frac{r^2}{p}\right) = \left(\frac{r^2}{p}\right)^2 = 1 & \text{se } m = 0 \\ \left(\frac{yr^2}{p}\right) = \left(\frac{y}{p}\right)\left(\frac{r}{p}\right)^2 = \left(\frac{y}{p}\right) = -1 & \text{se } m = 1 \end{cases}$$

Inoltre, poiché Alice sceglie r random, l'insieme dei possibili valori che un attaccante vede quando Alice cifra $m = 0$ sono tutti i possibili quadrati modulo n, mentre quando cifra $m = 1$ sono tutti i possibili numeri c che soddisfano $\left(\frac{c}{n}\right)$ che non sono quadrati modulo n.

Questo crittosistema non è pratico, in quanto ogni bit del plaintext viene cifrato con un numero modulo n. Affinché sia sicuro, è necessario che un attaccante non sia in grado di fattorizzare $n = pq$ (o risolvere il problema del residuo quadratico), quindi nella pratica n deve essere di almeno 1000 bit. Quindi se Alice vuole inviare un messaggio di k bit, il cyphertext sarà di $1000k$ bit.

6.1.2 Dimostrazione di correttezza del crittosistema

Supponiamo che esista un algoritmo A che dati in input il cyphertext c e la chiave pubblica (n, a) ritorna il bit b in tempo polinomiale, quando possiamo affermare che questo algoritmo è riuscito ad attaccare il sistema? L'algoritmo attacca il sistema quando indovina, quindi potrebbero esserci casi in cui indovini e non indovini. Quale deve essere allora la probabilità $Pr[A\_indovina]$ affinché attacchi?

L'attacco funziona ovviamente se $Pr[A\_indovina] > \frac{1}{2}$, ovvero ha più probabilità di aver successo rispetto allo sparare a caso. Ma allo stesso modo ha successo anche quando $Pr[A_indovina] < \frac{1}{2}$. Perché? Prendiamo il caso estremo in cui $Pr[A\_indovina] = 0$. L'attaccante, sapendo che l'algoritmo sbaglia sempre, deve solamente scegliere il valore opposto a quello che A ritorna per portare il successo dell'attacco a 1.

Nel nostro caso, dunque, diciamo che l'attacco ha successo se la sua probabilità (di successo) è **diversa** da $\frac{1}{2}$. Quanto però deve allontanarsi da $\frac{1}{2}$? La differenza deve essere tale da permettere di accorgersi che con A si indovina di più rispetto allo sparare a caso. Nella nostra vita siamo in grado, al massimo, di effettuare un numero di esperimenti polinomiale, quindi la quantità deve essere polinomiale. La formula di correttezza/sicurezza del crittosistema può essere così riscritta:

$$\exists c \; \forall \bar{k} \; \exists k > \bar{k} \; \left| Pr[attacco\_ha\_successo] - \frac{1}{2} \right| < k^{-c}$$

Passiamo ora alla dimostrazione di correttezza.

Dimostrazione. Supponiamo per assurdo che il crittosistema non sia sicuro. Questo significa che la formula di sicurezza non vale e quindi:

$$\exists c \; \forall \bar{k} \; \exists k > \bar{k} \; \left| Pr[attacco\_ha\_successo] - \frac{1}{2} \right| \geq k^{-c}$$

La probabilità che l'attaccante ha di avere successo, ovvero che la sua decodifica gli permetta di ottenere il messaggio originale, è

$$Pr[attacco\_ha\_successo] > \frac{1}{2} + k^{-c}$$

Supponiamo di disporre di un algoritmo A che ha come probabilità di successo 3/4. Se facessimo 100 esperimenti indipendenti tra loro per stimare il plaintext, ci spetteremmo che 75 di questi diano il bit corretto. Quindi il valore corretto sarebbe quello che compare più volte, con alta probabilità beccheremmo il bit corretto.

Tornando alla nostra probabilità effettiva, ci potremmo aspettare che $\frac{1}{2} + k^{-c}$ degli esperimenti ritorni la risposta corretta. Più k^{-c} è piccolo, più esperimenti è necessario fare per avere una buona probabilità. Quindi dato k^{-c}, dobbiamo capire qual è il numero di esperimenti che bisogna fare per poter avere una buona garanzia per dire che il valore visto più volte è la risposta corretta.

I problemi da risolvere sono due:

1. Capire come, dato il cyphertext, costruire una serie di esperimenti indipendenti sulla macchina A, ognuno dei quali ha una sua probabilità di dare la risposta corretta;
2. Dato k^{-c}, capire quanti esperimenti fare.

Partiamo dal secondo punto. In modo molto semplicistico, nella statistica si fanno degli esperimenti e sulla base di questi si cerca di dedurre qualcosa circa il mondo che si sta osservando. Poiché è impossibile osservare l'intero mondo, se ne osservano solo dei pezzi. Da questi pezzi si cerca di stimare come si comporta effettivamente il mondo. Quando facciamo queste stime, sappiamo i dati li stiamo campionando da alcune variabile casuali (quando peschiamo una persona, il risultato dell'esperimento con quella persona è una variabile casuale). Di queste variabili casuali possiamo osservare più cose, e possiamo anche cercare

di calcolare qual è la probabilità che il risultato che osserviamo da un insieme di variabili casuali sia effettivamente la rappresentazione corretta della realtà, da cui queste variabili casuali sono state campionate. Si cerca quindi di trovare delle disequazioni sulle provabilità di ottenere qualcosa. Quello che interessa a noi in questo caso è il limite di Chernoff.

Definition 6.1.2 (Limite di Chernoff per valori binari) *Siano $x_1, ..., x_n$ variabili casuali binarie indipendenti con probabilità di successo $P > \frac{1}{2}$. La probabilità che più della metà di queste variabili valga 1 è*

$$P = \sum_{i=\frac{n}{2}+1}^{n} \left(\binom{n}{i} p^i (1-p)^{n-i} \right)$$

Affinché più della metà degli elementi abbiano successo (1), il numero di esperimenti i che hanno avuto successo possono essere $\frac{n}{2} + 1$ oppure $\frac{n}{2} + 2$ oppure $\frac{n}{2} + 3$ oppure eccetera. Li sommiamo tutti. Quindi quello che abbiamo è la probabilità che esattamente i elementi hanno successo. Per sapere la probabilità che esattamente i elementi hanno successo, dobbiamo scegliere un sottoinsieme di i variabili e vedere qual è la probabilità che esattamente quel sottoinsieme di variabili ha successo (e le restanti no); è un'unione di eventi disgiunti, che sono tanti quanti i sottoinsiemi di i variabili, quindi $\binom{n}{i}$. Deciso quali sono le variabili che hanno successo e quali no, la probabilità che le i variabili di successo abbiano successo è p^i, mentre la probabilità che le altre non siano di successo è $(1-p)^{n-i}$ ($(1-p)$ alla numero delle altre variabili).

Chernoff ha fatto uno studio dal quale si ottiene che

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

Questa formula dice che la probabilità che più della metà degli elementi dia 1 è esponenzialmente vicina a 1, dove per esponenzialmente si intende esponenzialmente in n, dove

$$\left(P - \frac{1}{2}\right)^2 = k^{-2c}$$

e quindi probabilità di errore è

$$e^{-2nk^{-2c}}$$

Vogliamo rendere questa probabilità molto piccola, ma, come si può osservare, questa si abbassa esponenzialmente al crescere di n. Supponiamo di voler rendere

$$1 - e^{-2nk^{-2c}} \geq 1 - \frac{1}{e^k}$$

Quindi

$$\begin{aligned}\frac{1}{e^k} &\geq \frac{1}{e^{2nk^{-2c}}}\\ e^k &\leq e^{2nk^{-2c}}\\ e^k &\leq e^{\frac{2n}{k^{2c}}}\\ 2n &\geq k^{1+2c}\\ n &\geq \frac{k^{1+2c}}{2}\end{aligned}$$

Il numero di esperimenti da fare è polinomiale in k. Questo ci dice che se il vantaggio che abbiamo è polinomiale in qualche security parameter, riusciamo a ottenere una probabilità di errore esponenzialmente piccola nel security parameter scegliendo una quantità di esperimenti polinomiale nel security parameter.

Noi partiamo dall'idea che tutto ciò che è polinomiale è facile, per cui diciamo che un sistema è attaccato nel momento in cui c'è un algoritmo polinomiale in grado di attaccarlo. Con una quantità polinomiale di esperimenti riusciamo effettivamente ad attaccare il sistema. Visto che la nostra ipotesi dice che non esistono algoritmi probabilistici polinomiali in grado di risolvere il problema, e visto che qui abbiamo trovato un algoritmo probabilistico polinomiale in grado di risolvere il problema diciamo che il protocollo dovrebbe essere sicuro, in quanto siamo arrivati ad un ASSURDO.

Passiamo ora la primo punto. Supponiamo che la probabilità di successo sia polinomiale. Come creiamo gli esperimenti indipendenti? Prendiamo come input un input z tale per cui

$$\left(\frac{z}{n}\right) = 1$$

Vogliamo capire se z è quadrato o no. Scegliamo

$$R_1, \ldots, R_n \in_R Z_n^*$$

Calcoliamo

$$w_i = zR_i^2$$

Sia

$$b_i = A(w_i)$$

Cosa stiamo facendo? Prendiamo il cyphertext z, che non sappiamo se è una codifica di uno 0 o di un 1, e lo moltiplichiamo per un quadrato a caso R_i^2. A questo punto eseguiamo l'algoritmo A sui vari esperimenti e scegliamo come valore corretto quello che si ripete di più tra i vari b_i risultanti.

Partiamo da uno z, che non sappiamo se è un quadrato o un non quadrato, quindi non sappiamo se è una codifica di uno 0 o di un 1. Il nostro obiettivo è creare n esperimenti indipendenti da dare in pasto ad A per farci dire se z è un quadrato oppure no. Sappiamo che A ha successo con probabilità

$1/2 + k^{-c}$, quindi il modo in cui creiamo gli esperimenti indipendenti è trasformando z, che è un quadrato o un non quadrato scelto a caso, in un oggetto che è sempre scelto a caso con una misura di probabilità diversa; è come se fosse stato codificato n volte in maniera indipendente ognuna delle n volte. Moltiplicando z per un quadrato a caso sto costruendo un nuovo quadrato o non quadrato scelto a caso. Quindi gli n esperimenti che facciamo sono effettivamente indipendenti, è come se lo stesso bit fosse stato codificato n volte indipendentemente. Su ognuna di queste codifiche lanciamo l'algoritmo di attacco A, che cerca di dirci se il plaintext è 0 o 1 con una probabilità di successo $1/2 + k^{-c}$. Abbiamo quindi creato lo scenario che volevamo, dove facciamo n esperimenti indipendenti. La risposta che prenderemo sarà quella che occorre il più delle volte.

Questo ragionamento è corretto per creare n esperimenti indipendenti da dare in pasto ad A per farci dire il valore di z? Supponiamo che A abbia una probabilità di successo $Pr[A\_successo] = 51/100$, dove:

- Il successo sui quadrati è 40/100;
- Il successo sui non quadrati è 62/100.

La probabilità totale è quindi:

$$Pr[A\_successo] = \frac{1}{2} \cdot \frac{40}{100} + \frac{1}{2} \cdot \frac{62}{100} = \frac{51}{100}$$

Se diamo in input un cyphertext scelto **a caso**, la probabilità di indovinare quel cyphertext è 51/100. Ma noi stiamo dando un cyphertext a caso alla macchina A? No, in quanto il cyphertext lo stiamo scegliendo a caso tra le codifiche di 0 **oppure** tra le codifiche di 1, non lo stiamo prendendo a caso da entrambe. La distribuzione che stiamo usando in input su A non è quella che abbiamo usato per definire che l'algoritmo ha successo con probabilità 51/100. L'input deve essere scelto a caso tra tutte le codifiche di numeri con simbolo di Jacobi 1. Infatti in questo caso:

- Se passiamo in input una codifica di uno 0, gli esperimenti stanno passando ad A sempre un quadrato a caso. L'algoritmo ritornerà solo 40/100 volte che l'input è un quadrato, quindi sbaglierà dicendo che l'input è un non quadrato;
- Se passiamo in input una codifica di un 1, gli esperimenti stanno passando ad A sempre un non quadrato a caso. L'algoritmo ritornerà 62/100 volte che l'input è un non quadrato, quindi ritornerà il valore corretto dicendo che l'input è un non quadrato.

L'algoritmo che abbiamo costruito ritornerà sempre che l'input passato è un non quadrato, con questa divisione della probabilità. Noi non sappiamo che algoritmo A abbiamo a disposizione quando ci viene detto che ha probabilità 51/100.

L'approccio che abbiamo usato per la creazione degli esperimenti indipendenti non è corretto, perché non fornisce ad A gli input secondo la misura di probabilità che A si aspetta quando diciamo che ha una certa probabilità di successo (come dimostrato dal controesempio creato).

Quando replichiamo un esperimento, questo consiste nel fornire un input ad una macchina e osservarne l'output. La macchina ha un determinato comportamento, solitamente definito nel seguente modo: se l'input ha una

determinata distribuzione di probabilità, allora l'output ne ha un'altra. Nel costruire l'esperimento dobbiamo assicurarci che l'input abbia la distribuzione di probabilità giusta, quella che si aspetta l'algoritmo.

Vediamo ora come costruire gli esperimenti in modo corretto. A si aspetta un input che sia distribuito uniformemente tra gli elementi con simbolo di Jacobi 1, ma la nostra z è un quadrato o un non quadrato. Dobbiamo far si che i W_i sia distribuiti uniformemente solo tra i quadrato o solo tra i non quadrati. In particolare, a prescindere da cos'è z, i w_i devono essere distribuiti uniformemente con probabilità $1/2$ tra i quadrati e $1/2$ tra i non quadrati. Per generare un oggetto distribuito uniformemente tra gli elementi con simbolo di Jacobi 1, potremmo prima lanciare per decidere se distribuirlo tra i quadrati o i non quadrati e poi distribuirlo tra i non quadrati e i non quadrati. Questo si può fare similmente lanciando una moneta per decidere se distribuirlo tra gli oggetti che hanno la stessa quadraticità di z o tra quelli con la quadraticità opposta. Tra tutti gli elementi che abbiamo a disposizione, ne abbiamo uno che permetta di cambiare la quadraticità di z? Lo schema di Micali, nella chiave pubblica, contiene un non quadrato. Possiamo quindi, in base al risultato del lancio, moltiplicare o meno moltiplicare W_i per y.

Il nuovo algoritmo diventa così:

- Siano $R_1, ..., R_n \in_R Z_N^*$ e siano $s_1, ..., s_n \in \{0, 1\}$;
- Per ogni i allora:

$$w_i = \begin{cases} zR_i^2 & \text{se } s_i = 0 \\ yzR_i^2 & \text{se } s_i = 1 \end{cases}$$

- Sia $b_i = A(W_i)$. Non posso prendere come risultato **finale** i b_i maggiori, in quanto b_i è il risultato al problema trasformato w_i, ma devo ripristinare la quadraticità iniziale di z:

$$\forall i \; b' = \begin{cases} b_i & \text{se } s_i = 0 \\ \bar{b_i} & \text{se } s_i = 1 \text{ (prendo l'opposto di } b_i) \end{cases}$$

Abbiamo dimostrato che un attacco all'algoritmo di codifica di Goldwasser-Micali si traduce in un algoritmo in grado di calcolare la quadraticità di un numero, quindi che calcola il residuo quadratico. In realtà non lo abbiamo dimostrato ancora completamente, manca un piccolo dettaglio. L'algoritmo che calcola il residuo quadratico prende in input z e ritorna 0 o 1. Il nostra algoritmo A prende in input z e y e ritorna 0 o 1. Per funzionare A necessita di un non quadrato con simbolo di Jacobi 1, quindi non abbiamo costruito un algoritmo che prende solo z.

Per poter usare A per risolvere il problema del residuo quadratico dovremmo fare in modo che A si calcoli y internamente. Conosciamo algoritmi in grado di calcolare non quadrati con simbolo di Jacobi 1 senza conoscere la fattorizzazione di n? Il non quadrato pubblicato nella chiave pubblica è stato costruito prendendo un oggetto a caso, verificandone l'appartenenza a Z_n^* e calcolandone il simbolo di Legendre rispetto a p e q, i due fattori di n. Con la fattorizzazione di n siamo capaci di costruire y, ma senza?

E se noi prendessimo un y a caso di $Z_n?*$ e verificassimo se la macchina A funziona? Ma come facciamo a sapere se A sta effettivamente dando i risultati corretti? Potremmo fare un esperimento di questo genere:

- Prendiamo $x \in_R Z_n^*$ e $s \in_R \{0,1\}$
- Calcoliamo w tale che

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

- Sia b la risposta di $A(w)$, allora la macchina funziona con y se $b = s$.

In questo modo passiamo alla macchina un quadrato a caso se $s = 0$ o un non quadrato se $s = 1$. Se noi passiamo un quadrato alla macchina e questa dice quadrato, allora ha indovinato; se noi passiamo un non quadrato e la macchina ritorna non quadrato allora ha indovinato. Se la macchina indovina con una probabilità che si discosta significativamente da $1/2$ allora basta fare un numero di esperimenti dato dal limite di Chernoff. Se indovina la maggior parte delle volte allora funziona. Questo vale solo se y è un quadrato.

Se y è un quadrato però stiamo testando quadrato in entrambi i casi ($y_{quadrato} * y^2 = quadrato$), la macchina si comporta quindi sempre allo stesso modo. Quando la macchina funziona, avvero le stiamo passando effettivamente un non quadrato? Quando il comportamento statistico della macchina sugli x^2 è diverso da quello sugli yx^2. Se le passiamo un non quadrato, stiamo testando la macchina effettivamente su quadrati e non quadrati, se le passiamo un quadrato la stiamo testando solo su quadrati, Quindi ce ne accorgiamo se stiamo testando la macchina con y quadrato, in quanto sappiamo che A, di fronte ad un y non quadrato, indovina quadrati da non quadrati con vantaggio polinomiale. □

7 Codifica di sequenze di bit

Supponiamo di voler codificare una sequenza di l bit $b_1 ... b_l$, ha senso codificare i singoli bit e poi ricomporre la sequenza come $E(b_1)...E(b_l)$? Se teniamo presente la malleabilità, questa soluzione non è possibile, in quanto una codifica di questo tipo è malleabile.

Non consideriamo per ora la malleabilità, ma concentriamoci solo nel fare in modo che dal cyphertext non possa essere estratta **alcuna informazione binaria** del plaintext. Allora il sistema appena descritto funziona. Ma cosa vuol dire che non è possibile estrarre alcuna informazione dal cyphertext relativa al plaintext?

7.1 Distinguisher e capacità di distinguere

Definition 7.1.1 *Un distinguisher $D \in PPT$ è un algoritmo che dato un input restituisce 0 (con probabilità x) o 1 (con probabilità $100 - x$).*

In generale, se la differenza tra le probabilità che D ha di dire 0 e 1 è molto piccola (più piccola di ogni polinomio), per avere un vantaggio il numero di esperimenti da fare è molto alto (più grande di ogni polinomio).

Definition 7.1.2 *Sia $P_k^{D,m}$ la probabilità che D restituisca 1 su una codifica di m quando il security parameter vale k:*

1. *Un crittosistema E nasconde m_1 e m_2 a D quando*

$$\forall c \; \exists \bar{n} \; \forall n > \bar{n} \; \left| P_k^{D,m_1} - P_k^{D,m_2} \right| < k^{-c}$$

   Un'altra notazione per scrivere questa formula è

$$\left| P_k^{D,m_1} - P_k^{D,m_2} \right| < k^{-\omega(1)}$$

   dove $\omega(1)$ è l'insieme di tutti i naturali.
2. *E nasconde a D se*

$$\forall m_1, m_2 \; E \text{ nasconde } m_1, m_2 \text{ a } D$$

3. *E nasconde se $\forall D \in PPT$ E nasconde a D, ovvero*

$$\forall D \in PPT \; \forall m_1, m_2 \; \left| P_k^{D,m_1} - P_k^{D,m_2} \right| < k^{-c}$$

   ovvero E impedisce a chiunque di distinguere nessuna coppia di messaggi.

Esiste comunque una qualche proprietà binaria dei messaggi che D è in grado di distinguere? Se esiste allora dati un messaggio che la soddisfa e un messaggio che non la soddisfa, D dovrebbe essere in grado di vedere la

differenza tra i due messaggio, ma per la definizione sopra il distinguisher non la può vedere. Questa è veramente una buona definizione? Se un crittosistema la soddisfa allora non è attaccabile? Assolutamente no. Ad oggi, sulla base dell'esperienza che abbiamo, questa appare essere una buona definizione di correttezza, ma non è stata dimostrata.

Quando definiamo un protocollo e i suoi parametri, siamo sicuri che il crittosistema esista? Potrebbe capitare che la definizione sia così pensante da non poter essere realizzata nella realtà. In questo caso diventa necessario indebolire la definizione per implementarla.

L'algoritmo che abbiamo descritto sopra, che concatena le codifiche dei singoli bit, è un algoritmo che soddisfa le proprietà del distinguisher. Il fatto che un crittosistema sia sicuro rispetto alla definizione del distinguisher non implica che sia sicuro rispetto alla malleabilità.

Dimostriamo ora che il crittosistema costruito nasconde ad ogni distinguisher. La tecnica usata sarà utile in vari contesti.

Dimostrazione. Supponiamo per assurdo che il crittosistema non sia sicuro e quindi non soddisfi la definizione. Allora:

$$\exists D \in PPT\ \exists m_1, m_2\ \exists c\ \forall \bar{k}\ \exists k > \bar{k} \left| P_k^{D,m_1} - P_k^{D,m_2} \right| > k^{-c}$$

Siano $m_1 = \alpha_1\ \alpha_2\ ...\ \alpha_l$ e $m_2 = \beta_1\ \beta_2\ ...\ \beta_l$ con:

- $E(m_1) = E(\beta_1)\ E(\beta_2)\ ...\ E(\beta_l)$;
- $E(m_2) = E(\alpha_1)\ E(\alpha_2)\ ...\ E(\alpha_l)$.

Per dimostrare che possiamo attaccare questa codifica, andiamo a vedere se a partire dall'esistenza dei due messaggio m_1 e m_2 riusciamo a dimostrare l'esistenza di altri due messaggi distinti per un solo bit.

Definiamo per ogni $i \in \{0, ..., l\}$

$$m(i) = \beta_1 ... \beta_i\ \alpha_{i+1} ... \alpha_l$$

Ad esempio:

$$m(0) = \alpha_1 ... \alpha_l \equiv m_1$$
$$m(l) = \beta_1 ... \beta_l \equiv m_2$$

Con questa costruzione dei messaggi, i messaggi $m(i)$ non son altro che i messaggi intermedi tra m_1 e m_2 che permettono di passare m_1 a m_2 un bit alla volta. In totale sono $l + 1$ messaggi intermedi. Tra il messaggi i e $i + 1$ cambia un solo bit, che è quello $i + 1$.

Il nostro obiettivo è trovare due messaggi tra questa serie di messaggi intermedi che siano distinguibili che differiscono solo per un bit. Per fare ciò definiamo

$$P(i) = P_k^{D,m_i}$$

la proprietà che il distinguisher ritorni 1 se gli passiamo in input una codifica del messaggio $m(i)$. La nostra ipotesi diventa quindi

$$\begin{aligned} k^{-c} &< \left| P_k^{D,m_1} - P_k^{D,m_2} \right| \\ &= |P(0) - P(l)| \\ &= \underbrace{\left| \sum_{i=1}^{l} (P(i-1) - P(i)) \right|}_{P(0)-P(1)+P(1)-P(2)+\ldots+P(l-1)-P(l)} \end{aligned}$$

Abbiamo riscritto la probabilità con cui D ritorna 1 su m_1 e 1 su m_2 come una somma di differenze di probabilità con cui D ritorna 1 su messaggi che sono sequenziali. Ogni coppia di messaggi sequenziale differisce per un solo bit.

In questo modo abbiamo espresso il risultato del nostro esperimento come somma dei risultati di esperimenti fatti su coppie di messaggi che differiscono. Possiamo ora la disuguaglianza di Weierstrass, che dice che il modulo della somma di qualcosa è minore uguale della somma dei moduli:

$$\begin{aligned} &= \left| \sum_{i=1}^{l} (P(i-1) - P(i)) \right| \\ &\leq \sum_{i=1}^{l} |P(i-1) - P(i)| \end{aligned}$$

Arrivati qui vediamo che abbiamo una somma di moduli che eccede k^{-c}. Se noi abbiamo una somma di numeri che eccede un determinato valore, allora sappiamo che esiste almeno un elemento che è maggiore uguale della media[1].

1: In ogni sommatoria è sempre presente un elemento che è maggiore o uguale alla media.

Definition 7.1.3 *Supponendo di avere una generica sommatoria*

$$\sum_{i=0}^{n} x_i = L$$

allora

$$\exists i \; x_i \geq \frac{L}{n}$$

Dimostrazione. Supponiamo per assurdo

$$\forall i \; x_i < \frac{L}{m}$$

allora

$$\sum x_i < \sum \underbrace{\frac{L}{n}}_{L}$$

IMPOSSIBILE □

Nella nostra sommatoria stiamo sommando tutti elementi positivi, il sui risultato è maggiore di k^{-c}. La media di questa somma è maggiore uguale di $\frac{k^{-c}}{l}$. Da questo possiamo dire che:

$$\exists i \left| P(i-1) - P(i) \right| > \frac{k^{-c}}{l} > k^{-c+1} \text{ con } k \text{ abbastanza grande}$$

Abbiamo trovato due messaggi m_i e m_{i+1} che il D è in grado di riconoscere con un vantaggio polinomiale $k^{c'} = k^{-c+1}$. Non è la stessa c di prima, ma a noi basta trovare una c.

Possiamo quindi concludere che esiste un D che riesce a distinguere i messaggi m_i e m_{i+1}. La tecnica che abbiamo usato è detta **Tecnica di Interpolazione**.

Definition 7.1.4 (Tecnica di interpolazione) *Dati due cose molto diverse che vengono distinte, la tecnica di interpolazione consiste nel cercare di trovare una regola di trasformazione che permetta di passare da una all'altra cambiando poche cose alla volta in maniera tale che alla fine si arrivi a distinguere nuovi oggetti che però sono molto più simili tra di loro e sono sufficientemente simili per usarli come base per un attacco a qualcos'altro.*

Vediamo ora se, seguendo quanto dice la tecnica di interpolazione, riusciamo a utilizzare la coppia di messaggi trovata per attaccare un crittosistema che codifica un singolo bit:

$$m(i-1) = \beta_1 \dots \beta_{i-1}\ \alpha_i\ \alpha_{i+1} \dots \alpha_l$$
$$m(i) = \beta_1 \dots \beta_{i-1}\ \beta_i\ \alpha_{i+1} \dots \alpha_l$$

I due messaggi sono identici eccetto per il bit i-esimo. Come facciamo a dire effettivamente il bit β_i e α_i sono diversi? Visto che i due messaggi sono identici eccetto per il bit i-esimo e il distinguisher li distingue, è chiaro che i due messaggi sono diversi.

Supponiamo senza perdita di generalità che $\alpha_i = 0$ e $\beta_i = 1$[2]. Sia z un elemento di Z_n^* con $\left(\frac{z}{n}\right) = 1$. Allora z potrebbe essere la codifica di 0 o la codifica di 1. Il nostro obiettivo di vedere se D riesce a vedere la differenza tra la codifica di uno 0 o la codifica di un 1. D ritorna uno 0 o un 1, ma vogliamo che la probabilità con cui ritorna 1 nel caso in cui z sia la codifica di uno 0 sia sufficientemente diversa dalla probabilità con cui D ritorna 0 nel caso in cui z sia la codifica di uno 1. Allora dato z:

- Costruisci $E(\beta_1)\ E(\beta_2) \dots E(\beta_i)\ z\ E(\alpha_{i+2}) \dots E(\alpha_l)$;
- Lancia D sul risultato.

Abbiamo costruito un qualcosa che se z è la codifica di 0 abbiamo costruito la codifica di un messaggio $m(i-1)$, mentre se z è la codifica di 1 abbiamo costruito la codifica di un messaggio $m(i)$. La probabilità con cui D restituisce 1 è

$$P(i-1) \text{ se } z \text{ codifica } 0$$
$$P(i) \text{ se } z \text{ codifica } 1$$

2: Per fare una dimostrazione completa avremmo dovuto risolvere i due casi, quello dove $\alpha_i = 0$ e $\beta_i = 1$ e quello dove $\alpha_i = 1$ e $\beta_i = 0$. Quando diciamo "supponiamo senza perdita di generalità" che vale uno dei due casi, è perché l'altro caso è sostanzialmente identico. In questo caso, se dovesse valere il contrario rispetto al caso che stiamo dimostrando, basta scambiare $m(i-1)$ e $m(i)$ e tornare al caso che stiamo dimostrando.

Allora $P(i-1)$ è la probabilità con cui D restituisce 1 se gli diamo in input una codifica del messaggio $m(i-1)$, codifica costruita secondo l'algoritmo per codificare il messaggio $m(i-1)$. Cosa vuol dire codificare il messaggio $m(i-1)$? Significa applicare $E(\beta_1)$, $E(\beta_2)$, ..., $E(\beta_i)$, $E(\alpha_{i+2})$, ... Ma per quanto riguarda z, abbiamo applicato l'algoritmo? No, perché l'algoritmo di codifica prevede di dare un quadrato a caso se vogliamo codificare uno 0 e un non quadrato a caso se vogliamo codificare un 1. Ma z non è un quadrato a caso, ma un quadrato o un non quadrato che ci viene dato, e che non sappiamo come è stato pescato. Quindi se vogliamo veramente costruire un oggetto che è una codifica di $m(i-1)$ o $m(i)$, dobbiamo prendere zx^2 dove $x \in_R Z_n^*$. Dobbiamo prendere l'oggetto che ci è stato dato e trasformarlo in una nuova codifica dello stesso plaintext ma costruito secondo la distribuzione che si aspetta l'algoritmo D. L'algoritmo quindi diventa:

- Costruisci $E(\beta_1)\ E(\beta_2)\ ...\ E(\beta_i)\ zx^2\ E(\alpha_{i+2})\ ...\ E(\alpha_l)$;
- Lancia D sul risultato.

Ora il costrutto è effettivamente un cyphertext a caso che codifica il messaggio $m(i-1)$ e $m(i)$ e quindi abbiamo $P(i-1)$ se z codifica uno 0 e $P(i)$ se z codifica un 1. A questo punto il distinguisher vede effettivamente la differenza tra uno z che codifica uno 0 e uno z che codifica un 1.

La nostra definizione di correttezza per la codifica del singolo bit non dice "non esiste un distinguisher", ma "non esiste un algoritmo A in grado di calcolare il bit con un vantaggio polinomiale". Quindi la non esistenza del distinguisher implica la non esistenza dell'algoritmo A che non riesce a calcolare? Le due definizioni sono sostanzialmente equivalenti, dire che non esiste un distinguisher e dire che non esiste un algoritmo in grado di calcolare il plaintext è la stessa cosa.

Dimostrazione. Supponiamo che:

- P_0 la probabilità che D ritorni un 1 dato in input uno 0. Quindi $P_0 = P(i-1)$;
- P_1 la probabilità che D ritorni un 1 dato in input uno 1. Quindi $P_1 = P(i)$.

Allora se riceviamo la codifica di un bit scelto a caso, con quale probabilità riusciamo a indovinarlo? Usiamo il risultato di D come tentativo. La probabilità di indovinare è

$$\begin{aligned} Pr[indovinare] = \frac{1}{2}(1-P_0) + \frac{1}{2}P_1 &= \frac{1}{2} + \frac{1}{2}P_1 - \frac{1}{2}P_0 \\ &= \frac{1}{2} + \frac{1}{2}(P_1 - P_0) \end{aligned}$$

Se il bit dato è uno z (con probabilità 1/2), z sarà una codifica di uno 0 e la probabilità con cui D ritorna 1 è P_0, ma la macchina indovina se restituisce 0, quindi la probabilità che la macchina ha di indovinare è $1 - P_0$. Similmente per il bit a 1, solo che in questo caso D ritorna 1 con probabilità P_1 e quindi la probabilità per la macchina resta P_1. Mettiamo

ora in modulo il risultato ottenuto e togliamo $\frac{1}{2}$:

$$\left|\frac{1}{2} + \frac{1}{2}(P_1 - P_0) - \frac{1}{2}\right| = \left|\frac{1}{2}(P_1 - P_0)\right| = \frac{1}{2}|(P_1 - P_0)|$$

Noi sappiamo che $|(P_1 - P_0)| > k^{-c'}$, quindi:

$$\begin{aligned}\frac{1}{2}\left|(P_1 - P_0)\right| &= \frac{1}{2}\left|(P(i) - P(i-1))\right| \\ &> \frac{1}{2}k^{-c'}\end{aligned}$$

Abbiamo scoperto che il nostro algoritmo, che usa il risultato di D come strumento per indovinare il bit che è stato codificato ha un vantaggio rispetto a $\frac{1}{2}$ che è più grande del polinomio $k^{-c'}$. Effettivamente quell'algoritmo è il nostro attaccante.

50:00 - 1:00:00 dimostrazione fatta a lezione.

□

□

Siamo arrivati ad uno schema di codifica che è dimostrabile sicuro, in quanto se qualcuno riuscisse a ricavare informazione (vedere differenza tra i cyphertext che hanno l'informazione e quelli che non ce l'hanno) allora riuscirebbe a trovare un algoritmo in grado di indovinare un plaintext a partire da un cyphertext con un vantaggio polinomiale. Abbiamo già dimostrato, nel capitolo precedente, che se un tale algoritmo dovesse esistere, allora quell'algoritmo può essere usato come black-box per risolvere il problema del residuo quadratico.

Questo schema dimostra anche che se qualcuno ci fornisse uno schema di codifica di natura diversa, noi potremmo dimostrare la correttezza dello schema che codifica tanti bit riconducendoci allo schema che codifica il singolo bit.

Siamo riusciti ad ottenere la sicurezza dimostrabile. Però se dovessi codificare un singolo bit, dovrei inviare 1000 bit; se dovessimo codificare 1000 bit, dovremmo inviare 1'000'000 bit. Noi vorremmo un sistema che codifica un bit con un singolo bit. Se codificassi un bit con un solo bit non potremmo creare cyphertext enormi, ma solo cyphertext che sono 0 o 1. Ci servirebbe un meccanismo in grado di codificare una sequenza di bit arbitraria con una nuova sequenza di bit lunga quanto la sequenza di partenza, ma in maniera tale che la codifica del singolo bit sia comunque dimostrabilmente sicura.

La parte difficile è come prendere da uno schema è come prendere uno schema che codifica un bit basandosi sulla difficoltà di applicare funzioni a migliaia di bit e convertirlo in uno schema che si basi sulla difficoltà di calcolare un singolo bit, anziché la difficoltà di calcolare 1000 bit. Per fare questo studieremo un problema laterale alla crittografia, i generatori pseudocasuali, che servono per fare le scelte a caso. Una volta che avremo un generatore pseudocasuale crittograficamente sicuro, lo useremo per generare una sequenza di bit casuale da usare come chiave per one-time

pad. La vera chiave segreta sarà in qualche modo legata al seme con cui generiamo lo stream di numeri. In questo modo prenderemo una sequenza di bit casuali lunga quanto il messaggio e la metteremo in XOR con il messaggio in chiaro e otterremo un cyphertext lungo quanto il plaintext.

Fin'ora abbiamo lavorato con funzione one-way, ma ora necessitiamo anche di lavorare con l'incapacità di ricavare una qualche informazione binaria da una funzione one-way. Dovremmo cercare di usare funzioni one-way per sfruttare la difficoltà di una qualche informazione binaria su queste funzioni one-way e l'informazione binaria sarà in qualche modo quel bit che vogliamo usare per fare le nostre codifiche.

Lancio della moneta e Hard Core Predicate

8

Supponiamo di avere Alice e Bob che vogliono lanciare una moneta in rete. Vogliamo costruire un protocollo che sia simile a quello che si ha quando la moneta viene lanciato in presenza. Si potrebbe fare in modo che uno dei due lanci la moneta e poi comunichi il risultato all'altro. Questa soluzione non va bene in quanto chi lancia la moneta può sceglierne arbitrariamente il valore, senza che l'altro agente abbia modo di verificare l'autenticità del valore ricevuto. Non è possibile verificare se la moneta lanciato sia equa.

Possiamo quindi far lanciare una moneta ad entrambi e poi far loro scambiare i risultati. Supponiamo che il risultato sia 0 quando i valori delle due monete sono diversi e 1 quando sono uguali. Se almeno uno dei due agenti lancia un moneta equa, la probabilità che il risultato finale sia 0 è $1/2$. Anche questo soluzione è vulnerabile. Bob potrebbe fingere di spedire il suo risultato e attendere il messaggio di Alice, per poi forgiare di conseguenza il suo messaggio e indurre il risultato che vuole, fingendo un ritardo sulla rete.

Possiamo appoggiarci ad una terza parte, ma questa soluzione sposta solo il problema della fiducia dai due attori originali ad un altro.

Il nostro obiettivo è costruire un lancio della moneta sicuro che coinvolga solo i due attori iniziali. Supponiamo che $coin_A$ sia una variabile di Alice che dice qual è il risultato finale del lancio della moneta e che $coin_B$ sia una variabile di Bob che dice lo stesso risultato.

Vogliamo quindi un protocollo tale per cui, alla fine:

$$coin_A = coin_B \wedge Pr[coin_A = 0] = \frac{1}{2}$$

Questo può avvenire solo se A e B seguono correttamente il protocollo (sono onesti). In caso contrario si possono verificare tre casi:

- Se A è onesto e B è disonesto, allora dobbiamo far si che

$$\left| Pr[coin_A = 0] - \frac{1}{2} \right| < k^{-c}$$

- Se B è onesto e A è disonesto, allora dobbiamo far si che

$$\left| Pr[coin_B = 0] - \frac{1}{2} \right| < k^{-c}$$

- Se A e B sono disonesti, non facciamo nulla (non c'è nessuna parte onesta da tutelare).

Come facciamo ad implementare un protocollo del genere?

8.1 Lancio di moneta nel pozzo

Alice:

- Sceglie a caso due numeri primi p, q e calcola $n = pq$;
- Calcola $z \in_R Z_n^*$ con simbolo di Jacobi $\left(\frac{z}{n}\right) = 1$;
- Invia n, z a Bob.

Bob:

- Sceglie $b \in_R \{0, 1\}$;
- Invia b a Alice.

Alice

- Invia p, q a Bob.

Risultato Il risultato del lancio della moneta è dato dal seguente calcolo:

$$b \oplus is\_square(z)$$

Alice lancia la moneta scegliendo a caso un quadrato o un non quadrato in Z_n^*. Infatti scegliendo uniformemente un numero in Z_n^* con simbolo di Jacobi 1, stiamo scegliendo un quadrato con probabilità $1/2$ o un non quadrato con probabilità $1/2$. Bob sceglie anche lui un bit a caso, che assegna a b. Quando Alice invia a Bob z, Bob non ha alcun modo di vedere il valore scelto da Alice, in quanto per farlo, non conoscendo ancora p e q, dovrebbe risolvere il problema del residuo quadratico. Quindi quando Bob invia il suo lancio, il valore non è stato in alcun modo scelto in funzione del bit scelto da Alice. Tramite lo XOR tra i due lanci, scopriamo se i risultati sono uguali oppure no.

Poiché l'ordine dei messaggi e quando inviarli è deciso dal protocollo, la vulnerabilità del messaggio inviato in ritardo è risolta. Ma questo protocollo effettivamente funziona?

Questo protocollo è un'istanza del sistema dove Alice e Bob lanciano indipendentemente una moneta, si scambiano il risultato e il risultato è 0 allora le due monete sono uguali o 1 se sono diverse. Il vantaggio di questo protocollo è che se almeno uno dei due lancia la moneta in maniera equa, il risultato è equo. Vediamo se l'algoritmo funziona rispetto ai vari casi che si possono verificare:

- Se Alice e Bob sono onesti, allora z è stato campionato secondo una misura casuale e quindi è un quadrato con probabilità $1/2$ o un non quadrato con probabilità $1/2$. Il valore b vale 0 con probabilità $1/2$ o 1 con probabilità $1/2$. Tutti conoscono p, q, z, n, b e possono calcolare il risultato. Sia $coin_A$ sia $coin_B$ avranno lo stesso risultato e

$$Pr[coin_A = 0] = \frac{1}{2}$$

- Se Bob è disonesto, lo può fare in due modi:
  1. Può provare a calcolare b in modo non casuale, usando una distribuzione diversa (es: $b = 0$ con probabilità $3/4$ e $b = 1$ con probabilità $1/4$). Fintanto che Bob calcola, nel modo che vuole, b in modo indipendente dal valore di z, il valore scelto

sarà sempre uguale alla quadraticità di z con probabilità $1/2$, in quanto lo XOR tra un bit completamente casuale e un bit calcolato in modo non casuale ma indipendente dall'altro elemento dell'operazione resta comunque casuale;
2. Può calcolare b in funzione di z. Ma questo è equivalente a indovinare la quadraticità di z, e quindi ad usare un algoritmo che è in grado di calcolare la quadraticità di z. Questo algoritmo dovrebbe essere in grado di stabilire se un numero è un quadrato con un vantaggio polinomiale rispetto a $1/2$. A questo punto l'eventuale attacco di Bob diventerebbe un algoritmo per la risoluzione del problema del residuo quadratico, che consideriamo difficile.
Abbiamo quindi dimostrato che se questo protocollo fosse attaccabile da Bob, allora il problema che sta alla base sarebbe facile, cosa non possibile in quanto è un problema difficile.

- Se Alice è disonesta, lo può fare in più modi:

1. Può calcolare p, q **non** primi, ma Bob se ne accorgerebbe quando li riceve al termine del protocollo. In questo caso possiamo considerare nullo il risultato e far rilanciare una moneta a Bob;
2. Può scegliere z non uniformemente tra gli elementi con simbolo di Jacobi 1. Se Alice sceglie un elemento con simbolo di Jacobi -1, Bob se ne accorgerebbe comunque alla fine. Potrebbe quindi scegliere z con un algoritmo diversa da quello casuale. Bob, quando riceve z, non ha modo di accorgersi che Alice ha scelto z in maniera non conforme. Ma Bob sceglie comunque il suo bit in modo casuale, rendendo comunque lo XOR casuale;
3. Può inviare p, q sbagliati, ma Bob se ne accorgerebbe e quindi potrebbe lanciare una sua moneta per decidere il risultato. Ma se invece non li inviasse proprio a Bob?
Dire che Bob lancia la sua moneta crea una grandissimo svantaggio nei suoi confronti. Alice, una volta ricevuto b da Bob potrebbe rifiutarsi di inviare il messaggio finale, influenzando la probabilità con cui Bob ottiene, ad esempio 0. Come fa ad alterare la probabilità? Dopo il secondo messaggio alice conosce $coin_A$, in quanto può calcolare lo XOR, ma Bob non conosce ancora $coin_B$. Supponiamo che il risultato del lancio della moneta decida chi sale sull'elicottero funzionante e che sull'elicottero manomesso. Se $coin_A$ dice ad Alice di salire sull'elicottero che cade, e quindi le è sfavorevole, allora Alice può non spedire il terzo messaggio. Bob allora lancia la propria moneta, che con probabilità $1/2$ sarà favorevole a lui e con probabilità $1/2$ sarà favorevole ad Alice. Supponendo che $coin_B = 0$ sia il risultato favorevole ad Alice, la probabilità di ottenerlo diventa

$$Pr[coin_B = 0] = Pr[coin_A = 0] + Pr[coin_A = 1] \cdot Pr[secondo\_lancio = 0]$$
$$Pr[coin_B = 0] = \frac{1}{2} + \frac{1}{2} \cdot \frac{1}{2} = \frac{3}{4}$$

Il protocollo non soddisfa quindi le proprietà di un protocollo di lancio della moneta. Esistono però casi in questo protocollo può comunque

essere accettato:

- Se Bob conosce conosce il risultato a se favorevole, allora, nel caso in cui Alice non sia onesta, può scegliere quel risultato;
- Se ne Bob ne Alice conoscono il risultato a loro favorevole (in questo caso, anche se Alice non invia l'ultimo messaggio, la probabilità non cambia).

Possiamo riscrivere il protocollo in modo che possa funzionare?

Definition 8.1.1 (Lancio di moneta) *Un protocollo di lancio della moneta deve essere tale per cui gli agenti possano conoscere il risultato anche senza inviare/ricevere l'ultimo messaggio.*

Sia n il numero minimo di messaggi di un protocollo che funziona. Allora entrambi gli agenti devono conoscere il risultato del lancio della moneta anche senza aver inviato l'ultimo messaggio. Questo significa che l'ultimo messaggio non serve, ma allora il messaggio che permette all'altra parte è il penultimo messaggio (basandoci sul protocollo sopra, Bob conosce il risultato e Alice no). Quindi esiste ancora un messaggio fondamentale che permette all'altro agente di conoscere il risultato. Continuando con questo ragionamento si arriverebbe a dire che entrambi gli attori devono conoscere i risultato senza scambiarsi alcun messaggio.

Quindi qualunque protocollo si utilizza, esiste sempre un messaggio senza il quale l'altra parte non conosce il risultato, ed è quello dove possiamo applicare l'attacco di Alice, alterando la probabilità. In generale, dati n agenti che partecipano al lancio della moneta, è possibile garantire la sicurezza del protocollo solo se meno della metà degli agenti barano (con due agenti, ne basta uno che bari).

Dobbiamo quindi indebolire la definizione.

Il problema che stiamo affrontando con il lancio della moneta può essere rappresentato con il seguente scenario:

- Bob lancia la moneta in un pozzo, che è lontano da lui;
- Alice, che è vicina al pozzo riesce a veder il risultato;
- Bob, per veder il risultato, deve avvicinarsi al pozzo ma questo è protetto dal cane rabbioso Alice;
- Alice può quindi decidere se far avvicinare Bob o meno in base al suo interesse: se il risultate le è favorevole fa avvicinare Bob, se le è sfavorevole non lo fa avvicinare, costringendolo a lanciare una nuova moneta (ovviamente fuori dal pozzo) influenzando così la probabilità di vittoria.

8.1.1 Oblivius transfer

Supponiamo che Bob sappia per certo l'andamento della borsa nel prossimo anno. Bob decidere di vendere questa informazione ad Alice in modo particolare: Alice paga 500 euro per accedere e riceve l'informazione solo se il lancio di una moneta produce testa.

In questo contesto Bob deve trasferire l'informazione con probabilità 1/2, ma a lui non interessa se l'informazione è stata trasferita o no, a lui interessa solo che Alice otterrà l'informazione con probabilità 1/2.

Questo è il classico caso dove Alice e Bob devono lanciare una moneta, ma a Bob non interessa conoscere il risultato del lancio della moneta, gli basta sapere che Alice riceverà il risultato con probabilità 1/2. Si tratta quindi di un caso di lancio della moneta nel pozzo.

Vediamo un protocollo di questo genere:

Bob:

- Sceglie a caso p, q primi;
- Calcola $n = pq$;
- Calcola e chiave pubblica e d chiave privata per RSA;
- Codifica il segreto con RSA ottenendo il cyphertext c;
- Invia n, e, c ad Alice.

Alice:

- Sceglie $x \in_R Z_n^*$;
- Invia x^2 a Bob.

Bob:

- Calcola una radice quadrata di x^2 e la invia ad Alice.

Alice manda un quadrato a caso a Bob, di cui lui calcola una radice quadrata. Le radici quadrata sono quattro, quindi Bob ne sceglierà una con una sua misura qualsiasi, ma con probabilità 1/2 quella radice sarà diversa da $\pm x$, ovvero il numero di partenza scelto da Alice. Abbiamo già visto che quando un agente possiede due radici quadrate di uno stesso numero che non sono una l'opposta dell'altra, allora l'agente p in grado di fattorizzare n, calcolando $MCD(n, (x + y))$. Con probabilità 1/2 invierà ad Alice una radice $y \neq \pm x$, mettendola così in condizione di calcolare p e q. Così Alice potrà calcolare d e decifrare c ricavando l'informazione segreta.

Non siamo in grado di fare un lancio della moneta regolare, quello che possiamo fare è un lancio di moneta nel pozzo. Esistono casi in cui questo ci può andare bene, e sono tutti quelli dove a Bob non interessa conoscere il risultato del lancio della moneta (oblivius transfer) oppure Bob sa qual è il risultato a se favorevole e quindi può sceglierlo se Alice non segue il protocollo.

8.2 Hard Core Predicate

Proviamo ora a riscrivere il protocollo mostrano per il lancio della moneta nel pozzo usando il logaritmo discreto.

Dati p primo e g generatore di Z_p^* conosciuti da entrambi gli attori, il protocollo procede come segue:

Alice:

- Sceglie $x \in_R \{0, ..., P - 2\}$;
- Calcola $z = g^x$;
- Invia z a Bob.

Bob:

- Sceglie $b \in_R \{0, 1\}$;
- Invia b ad Alice.

Alice:

- Invia x a Bob.

Risultato:

$$b \oplus \left(x < \frac{p-1}{2} \right)$$

Questo protocollo funziona come il primo: Alice calcola qualcosa e lo invia a Bob, Bob sceglie un bit a caso e lo invia ad Alice, Alice invia a Bob le informazione necessarie a fare il calcolo finale. Bob, quando sceglie il suo bit, non è ancora in grado di calcolare il bit finale in quanto, in questo caso, dovrebbe verificare se $x < \frac{p-1}{2}$, conoscendo solo g^x. Dovrebbe quindi fare un test binario sul logaritmo discreto di $z = g^x$.

A patto che Bob non sia in grado di fare il test binario, questo protocollo è un protocollo di lancio di moneta nel pozzo. Ma siamo sicuri che Bob non sia in grado di calcolare il predicato binario $x < \frac{p-1}{2}$, a meno di un vantaggio meno che polinomiale? Noi sappiamo che il logaritmo discreto è un problema difficile, ovvero che la funzione g^x è **one-way** e l'inversa è difficile da calcolare, e fin'ora abbiamo lavorato con quella. Questo algoritmo però lavora con un predicato binario sull'inversa, non ci interessa sapere il valore di x, ma solo se il predicato vale o non vale. Il problema quindi diventa: siamo sicuri che data una funzione one-way, dove per sua natura la funzione inversa è difficile da calcolare, qualsiasi predicato binario (sull'inversa) sia difficile da calcolare? No, data una funzione one-way, possono esistere predicati binari sull'inversa che sono facili da calcolare. Per il logaritmo discreto un predicato binario che sappiamo calcolare facilmente è il bit meno significativo, tramite il simbolo di Legendre.

Il predicato $x < \frac{p-1}{2}$, che ci dice se ci troviamo nella prima metà dei $p-1$ logaritmi discreti o nella seconda metà, riteniamo sia difficile da calcolare, e quindi è un hard core predicate per il logaritmo discreto.

Definition 8.2.1 (Hard Core Predicate) *Un hard core predicate per una funzione one-way è un predicato binario sull'inversa della funzione che è difficile da calcolare (ovvero la probabilità di indovinare il suo valore si discosta di una probabilità meno che polinomiale da* $1/2$*).*

Un domanda che può sorgere ora è questa: data una funzione one-way a caso, siamo sicuri che esista, per tale funzione, un hard core predicate? La risposta è si.

8.2.1 Hard Core Predicate per il logaritmo discreto

Abbiamo detto che $x < \frac{p-1}{2}$ è un predicato binario difficile da risolvere per il logaritmo discreto, ovvero che dato x a caso, la probabilità con cui

un algoritmo riesce a valutare x conoscendo g^x si allontana da $1/2$ di una quantità più piccola di ogni polinomio.

Dimostrazione. Sia $y = g^x$, se y è un quadrato allora ammette due radici distinte z_1 e z_2. Di queste due radici una ha $LD < \frac{p-1}{2}$, mentre l'altra ha $LD \geq \frac{p-1}{2}$. Sia la radice con $LD < \frac{p-1}{2}$ la radice principale. Se qualcuno riesce a fornirci l'algoritmo per risolvere l'hard core predicate del logaritmo discreto, allora possiamo usarlo anche per individuare la radice principale. Per dimostrare questa affermazione dobbiamo dimostrare che:

1. Se abbiamo a disposizione un algoritmo che calcola PSQR, allora possiamo calcolare DL;
2. Se abbiamo un algoritmo che calcola PSQR con probabilità esponenzialmente vicina ad 1, allora abbiamo un algoritmo per il logaritmo discreto che funziona con probabilità almeno $\frac{1}{2}$;
3.

1. Vogliamo calcolare il logaritmo discreto di un numero y $DL(y)$. Come farà l'algoritmo a calcolare la radice quadrata principale di un numero avendo a disposizione il predicato binario $x < \frac{p-1}{2}$? Preso un numero y, ne calcola la radice quadrata aritmetica e il suo opposto e invoca il predicato binario su una delle due radici per vedere se il predicato vale o no. Se il predicato vale, la risposta sarà la radice scelta, se non vale sarà l'altra.

Sappiamo che un quadrato in Z_p^* ha due radici, della forma:

$$g^i \text{ e } g^{i+\frac{p-1}{2}}$$

Di queste due radici, una si trova nella prima metà dei logaritmi discreti, mentre l'altra nella seconda metà. Definiamo **principale** la radice che sta nella prima metà. Queste due radici siamo in grado di calcolarle aritmeticamente, ma non siamo in grado di dire qual è la principale. Se qualcuno ci dà l'algoritmo per calcolare se un numero x è minore di $\frac{p-1}{2}$ possiamo individuare la principale. Le radici di un numero sono un numero e il suo opposto. Quindi, dal punto di vista aritmetico, se qualcuno ci dà un quadrato, possiamo calcolare la sua radice quadrata aritmetica e prendere il suo opposto per avere le due radici quadrate del numero. Tuttavia non è detto che la radice aritmetica sia quella che ha il logaritmo discreto più piccolo.

Vogliamo dimostrare che calcolare il predicato binario $x < \frac{p-1}{2}$ sul logaritmo discreto di un numero oppure calcolare la radice quadrata principale di un numero sono due problemi equivalenti. Quindi piuttosto che dire "supponiamo per assurdo che calcolare il predicato binario $x < \frac{p-1}{2}$ sia facile", diciamo "supponiamo per assurdo che il calcolo della radice principale di un numero sia facile".

Supponiamo che esista un algoritmo che calcola il predicato binario $x < \frac{p-1}{2}$ dato in input $y = g^x$, allora esiste un algoritmo che calcola la radice quadrata principale di un numero. Quindi possiamo usare questo algoritmo per calcolare il logaritmo discreto di un numero. Vediamo l'algoritmo:

Algorithm 5 Algoritmo per il calcolo del logaritmo discreto DL(y)

```
if y == 1 then
    return 0                          ▷ Se y è l'unità del gruppo, il DL è 0
else
    b ← LSB(y)
    if b then
        y ← y × g^{-1}                ▷ Mettiamo a 0 LSB(x)
    end if
    y ← PSQR(y)                       ▷ Scorriamo a destra di 1 i bit di x
    return 2 × DL(y) + b
end if
```

Se y è l'unità del gruppo, allora il logaritmo discreto è 0, in quanto $g^0 = 1$. Altrimenti assegniamo a b il Least Significant Bit del logaritmo discreto di y, ovvero verifichiamo $\left(\frac{y}{p}\right) == 1$, se è vero prendiamo $b = 0$ altrimenti $b = 1$[1]. Una volta che abbiamo calcolato LSB, se il logaritmo discreto di y è dispari gli togliamo una unità, ovvero prendiamo l'LSB e lo facciamo diventare 0. A questo punto calcoliamo la radice principale di y, ovvero scorriamo a destra di 1 i bit di x. Fatto questo possiamo invocare ricorsivamente il calcolo del logaritmo discreto su y, che □

1: Praticamente verifichiamo se y è un quadrato in Z_p^*, e quindi il suo logaritmo discreto è pari e termina con 0, altrimenti y è un non quadrato e quindi il suo logaritmo discreto è dispari e termina con 1.

2. Supponiamo di avere un algoritmo B per PSQR che funziona con probabilità $1 - \epsilon$. Qual è la probabilità che DL dia la risposta corretta? Noi calcoliamo i bit del logaritmo discreto uno alla volta e ogni volta dobbiamo calcolare una radice quadrata principale. Quante oltre calcoliamo una radice quadrata di y prima di arrivare al caso base? Dobbiamo fare tanti shift quanti i numeri di bit che ci sono nel logaritmo discreto di y affinché spariscano tutti e resti lo 0. Dobbiamo fare lo shift a destra per k volte, dove k rappresenta il numero di bit che usiamo per rappresentare y. L'algoritmo deve essere invocato k volte e ogni volta deve fornire la risposta corretta. La probabilità che l'algoritmo invocato k volte dia sempre la risposta corretta è:

$$Pr = (1 - \epsilon)^k$$

Allora abbiamo un algoritmo per il logaritmo discreto che potrebbe non funzionare. Abbiamo modo per capire che l'algoritmo non funziona? Se il calcolo della radice principale funziona sempre correttamente, dopo k iterazioni arrivo al caso base, quindi potrei contare le iterazioni per capire se il calcolo è corretto. Potremmo terminare prima nel caso in cui il logaritmo discreto di y ha meno di k bit, ma non dovremmo mai finire in più di k passi. Tuttavia anche se algoritmo termina, potrebbe terminare terminare con un risultato R non corretto. In questo vaso basta verificare se $g^R = y$. Nel momento in cui l'algoritmo termina e ci fornisce un numero, quindi, possiamo verificare se il risultato è corretto. Se il risultato non è corretto, possiamo rilanciare l'algoritmo. Sappiamo che, in media, con un numero di tentativi che è il reciproco della probabilità otteniamo la risposta corretta, quindi con $\frac{1}{(1-\epsilon)^k}$ tentativi.

Se $\epsilon \leq \frac{1}{2^k}$ allora $(1 - \epsilon)^k > \frac{1}{2}$. Da questo sappiamo che se abbiamo un algoritmo che riesce a risolvere il problema con una probabilità esponenzialmente vicina ad 1 (l'errore è limitato da $\frac{1}{2^k}$) allora abbiamo

un algoritmo che calcola la radice principale di un numero con probabilità di almeno $\frac{1}{2}$. Questo vuol dire che se reiteriamo il nostro algoritmo per un numero atteso di 2 tentativi, otteniamo il risultato.

Abbiamo quindi dimostrato che se abbiamo un algoritmo per la radice quadrata principale con probabilità esponenzialmente vicina ad 1, allora abbiamo un algoritmo per DL che funziona almeno con probabilità almeno $\frac{1}{2}$. Di conseguenza l'algoritmo che ripete il calcolo del logaritmo discreto finché non è corretto è un algoritmo che in tempo medio 2 iterazioni, ovvero polinomiale, risolve il problema del logaritmo discreto. Questa idea funziona in quanto quando abbiamo calcolato la soluzione siamo in grado di verificare se è corretta oppure no. □

3. Abbiamo quindi un algoritmo B che risolver il problema in $1 - \frac{1}{2^k}$. Vogliamo ora dimostrare che l'algoritmo che abbiamo veramente è sempre un algoritmo B ma che risolver il problema in $\frac{1}{2} + \eta$, con $\eta = k^c$. Cioè abbiamo un algoritmo che ha un vantaggio polinomiale rispetto ad $\frac{1}{2}$, ma per completare la dimostrazione ci serve un algoritmo che funziona con probabilità esponenzialmente vicina ad 1. Riusciamo quindi a fare una riduzione $B_{1-\frac{1}{2^k}} \leq B_{\frac{1}{2}+\eta}$, cioè riusciamo a far vedere che se qualcuno ci dà un algoritmo che ha vantaggio polinomiale rispetta ad $\frac{1}{2}$ allora possiamo ricavare una algoritmo che ha un probabilità esponenzialmente vicina ad 1 per funzionare? Possiamo farlo utilizzando la stessa tecnica usata nella dimostrazione per il residuo quadratico.

Nel problema del residuo quadratico avevamo un algoritmo che aveva vantaggio polinomiale rispetto a $\frac{1}{2}$ e creando tante istanze indipendenti dello stesso problema e prendendo come risultato finale quel numero che osservavamo nella maggio parte dei casi, avevamo un algoritmo con probabilità esponenzialmente vicino ad 1. Avevamo inoltre dimostrato con il limite di Chernoff che se noi richiediamo di avere una risposta corretta con un errore massimo di $\frac{1}{2^k}$, la quantità di esperimenti che dovevamo fare era polinomiale in k^c. Di conseguenza con un numero polinomiale di esperimenti avevamo una risposta corretta con errore massimo $\frac{1}{2^k}$.

Qui abbiamo ancora una situazione dove abbiamo un vantaggio polinomiale e sappiamo che se riusciamo a fare tanti esperimenti indipendenti dai quali possiamo ricavare la risposta del problema originale, allora con un numero l polinomiale di esperimenti riusciamo ad avere la risposta corretta rispetto a B. Il problema è come riusciamo a costruire degli esperimenti indipendenti tali che una volta che conosciamo la risposta al problema che abbiamo costruito allora riusciamo a trovare la risposta al problema originale.

Come costruiamo un esperimento indipendente y' tale per cui dalla risposta a y' otteniamo la risposta a y? Sia y un quadrato e sia $r \in_R \left[0, ..., \frac{p-1}{2}\right)$ un esponente scelto a caso nella prima metà. Sia $y' = y \cdot g^{2r}$. Stiamo moltiplicando quindi y per un quadrato a caso, di cui conosciamo il logaritmo discreto, ottenendo un altro quadrato a caso distribuito uniformemente tra i quadrati di Z_p^*. Vogliamo ora calcolare la radice quadrata principale di y' e da quella ricavare la radice quadrata principale di y. Siamo in grado di farlo?

Lemma 8.2.1 *Se $x + 2r < p - 1$ allora zg^r è PSQR di y' se e solo se z è PSQR di y.*

Questo lemma dice che se x è il logaritmo discreto di y. Se moltiplichiamo y per il nuovo quadrato e non facciamo operazioni di modulo sugli esponenti allora se qualcuno ci dà una radice quadrata principale del tuo nuovo oggetto, dividiamo questa per g^r, che conosciamo, e otteniamo la radice quadrata principale dell'y originario.

Dimostrazione del lemma. Se z è la radice quadrata principale di un numero che ha logaritmo discreto x, allora il logaritmo discreto di questa radice quadrata è $\frac{x}{2}$. Se $z = g^w$, allora

$$g^w g^r \text{ è radice quadrata principale di } y' = g^x g^{2r}$$
$$\Longleftrightarrow$$
$$g^w \text{ è radice quadrata principale di } g^x$$

Sappiamo che $x = \frac{w}{2}$, e quindi è $\frac{x}{2}$ che è radice principale di g^x. Allora:

$$g^{\frac{x}{2}r} \text{ è radice quadrata principale di } y' = g^x g^{2r}$$
$$\Longleftrightarrow$$
$$g^{\frac{x}{2}} \text{ è radice quadrata principale di } g^x$$

Dobbiamo dimostrare che se vale la parte destra vale la parte sinistra e viceversa. Partiamo dall'osservazione che il logaritmo discreto della radice quadrata principale di y è $\frac{x}{2}$, in quanto:

$$\sqrt[2]{z} = \sqrt[2]{g^w} = \sqrt[2]{g^x} = g^{x \cdot \frac{1}{2}} = g^{\frac{x}{2}}$$

$\longleftarrow$. La radice quadrata principale di z è $g^{\frac{x}{2}}$. Le radici quadrate di y sono due: la radice con logaritmo discreto $\frac{x}{2}$ e quella con logaritmo discreto $\frac{x}{2} + \frac{p-1}{2}$. La principale è $\frac{x}{2}$. Supponiamo che z sia radice quadrata principale di y, quindi il logaritmo discreto di z è esattamente $\frac{x}{2}$. Ci chiediamo se zg^r è radice quadrata di y'. Per saperlo dobbiamo verificare se il logaritmo discreto di zg^r, $\frac{x}{2} + r$ è minore di $\frac{p-1}{2}$. Sappiamo dal lemma appena scritto che $x + 2r < p - 1$. Di conseguenza:

$$x + 2r < p - 1$$
$$\frac{x}{2} + r < \frac{p-1}{2}$$

Abbiamo quindi dimostrato che $\frac{x}{2} + r < \frac{p-1}{2}$. □

$\longrightarrow$. Supponiamo che g^r sia radice quadrata principale di y, allora z è una radice quadrata di y. Questo lo sappiamo per che se g^r è è la radice quadrata principale di g^{2r}, l'altro elemento di y' deve essere per forza la radice di y. Ma z è la radice $\frac{x}{2}$ o $\frac{x}{2} + \frac{p-1}{2}$? Visto che il logaritmo discreto di z può essere una di quelle due radici, andiamo a vedere quali delle due rende vero il lemma. Sappiamo che $w + r < \frac{p-1}{2}$, quindi tra le due w qual è? zg^r è radice quadrata principale di y, di conseguenza z è un $g^w g^r$; w è $\frac{x}{2}$ oppure $\frac{x}{2} + \frac{p-1}{2}$. Vogliamo sapere quale dei due è w, sapendo

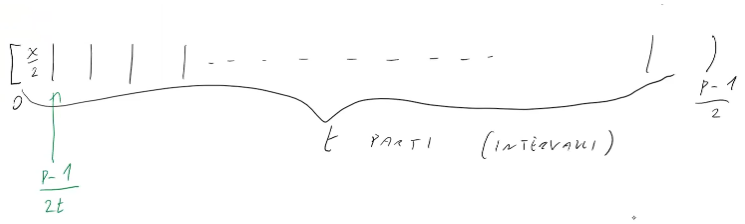
che $w + r < \frac{p-1}{2}$. Può essere $\frac{x}{2}$? Se mettiamo $\frac{x}{2}$ al posto di al posto di w, scopriamo che $w + r < \frac{p-1}{2}$, perché vale l'ipotesi sopra. Se invece $w = \frac{x}{2} + \frac{p-1}{2}$ la disequazione non vale. Quindi w può essere solo $\frac{x}{2}$. □

Se vale questa questa ipotesi, nel momento in cui ci viene data una radice principale di y', sappiamo che dividendo quella radice quadrata principale per g^r otteniamo z, la radice quadrata principale di y. Questo è il modo per ricavare la risposta al problema originale, cioè trovare la radice quadrata principale di y, avendo in mano la risposta al problema trasformato. Se il nostro r lo abbiamo scelto on modo che valga l'ipotesi sopra, allora dalla risposta al problema trasformato riusciamo a ricavare la risposta al problema originale.

Il problema è che r viene scelto a caso, quindi non abbiamo garanzia che soddisfi le condizioni del lemma.

Più x è piccolo e più è probabile trovare un r che vada bene. Se il numero y dato ha un logaritmo discreto piccolo, è più probabile che un r soddisfi la proprietà del lemma.

Prendiamo ora l'intervallo $[0, \frac{p-1}{2})$ e dividiamolo in t parti tutte uguali.



Supponiamo che $\frac{x}{2}$ sia nel primo intervallo. Qual è la probabilità di trovare un r che vada bene? Se r sta nell'ultimo intervallo, quello che termina con $\frac{p-1}{2}$, effettivamente è possibile che $\frac{x}{2} + r \not< \frac{p-1}{2}$. Ma se r si trova in uno degli altri intervalli, non sarà mai possibile andare oltre $\frac{p-1}{2}$, in quanto il valore di $\frac{x}{2}$ non eccede la dimensione di un intervallo. Quindi la probabilità di trovare un r che vada bene corrisponde alla probabilità di non finire nell'ultimo intervallo. Questa probabilità è:

$$\frac{t-1}{t}$$

ovvero la probabilità di essere in qualsiasi intervallo tranne l'ultimo.

Prendiamo l'esperimento completo:

- Prendi r a caso;
- Sia a la PSQR di yg^{2r};
- Ritorna ag^{-r}.

Scegliamo un r a caso, produciamo il nuovo problema e ne calcoliamo la radice quadrata principale, da quella ricaviamo la risposta al problema originale.

Qual è la probabilità che la risposta sia corretta? Affinché possiamo avere la garanzia che la risposta sia corretta, ci serve una risposta corretta

la problema trasformato ($Pr = \frac{1}{2}k^{-c}$) e sapere di aver scelto una r che soddisfa l'ipotesi del lemma ($Pr = \frac{t-1}{t}$). Quindi la probabilità è:

$$\left(\frac{t-1}{t}\right)\left(\frac{1}{2}k^{-c}\right)$$

Affinché il trucco di fare tanti esperimenti indipendenti e prendere il maggior numero di risposte corrette come risposta corretta funziona fintanto che la probabilità di dare la risposta corretta sia polinomialmente distante da $\frac{1}{2}$. Quindi la probabilità che avviamo travato deve essere:

$$\left(\frac{t-1}{t}\right)\left(\frac{1}{2}k^{-c}\right) \geq \frac{1}{2}k^{-2c}$$

Usiamo k^{-2c} in quanto $\frac{t-1}{t}$ rende più piccolo $\left(\frac{1}{2}k^{-c}\right)$, e quindi sarà più vicino a $\frac{1}{2}$. Praticamente la nostra probabilità deve essere maggiore di $\frac{1}{2}$ di un qualsiasi polinomio, solo più piccolo del k^{-c} che avevamo originariamente definito, in quanto sappiamo che $\frac{t-1}{t}$ riduce la nostra probabilità.

Abbiamo quindi imposto che questa grandezza sia sufficientemente distante da $\frac{1}{2}$. Se risolviamo la disequazione in t, riusciamo a trovare un limite inferiore al numero di intervalli da usare. Il limite inferiore che ci darà la disequazione sarà un limite polinomiale in k.

Quindi dividendo per una quantità di intervalli che è polinomiale in k, riusciamo ad ottenere un sistema tale per cui se x è nel primo intervallo, la probabilità di successo (di fornire una radice quadrata principale di y) è polinomialmente distante da $\frac{1}{2}$. Di conseguenza abbiamo un algoritmo polinomiale per calcolare la radice quadrata principale con probabilità di successo esponenzialmente vicina a 1, grazie al limite di Chernoff.

Abbiamo costruito un algoritmo che con probabilità $\frac{1}{2}$ calcola il logaritmo discreto di un numero a patto che questo logaritmo discreto stia nel primo intervallo. Per portare la probabilità a 1, basta ripetere l'algoritmo più volte.

Il problema ora è capire come costruire un algoritmo che funzioni su ogni intervallo a partire da uno che funziona con x nel primo intervallo.

Supponiamo di sapere che $\frac{x}{2}$ si trovi nell'intervallo i. L'intervallo i-esimo va da $i \cdot \frac{p-1}{2t}$ a $(i+1) \cdot \frac{p-1}{2t}$, supponendo di numerare gli intervalli con 0, 1, 2, ... Prendiamo il nostro y e lo moltiplichiamo per $g^{-\frac{p-1}{t}i}$. Cosa stiamo facendo? Sappiamo che $y = g^x$ e che $i \cdot \frac{p-1}{2t} \leq \frac{x}{2} < (i+1) \cdot \frac{p-1}{2t}$, quindi se facciamo

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

significa che

$$\frac{x'}{2} = \frac{x - \frac{p-1}{t}i}{2} = \frac{x}{2} - \frac{p-1}{2t}i$$

Con questa moltiplicazione (che diventa una sottrazione sugli esponenti) rimuoviamo il limite sinistro dell'intervallo, spostando il mostro proble-

ma dall'intervallo i-esimo al primo. Ci siamo quindi ricondotti al caso che il nostro algoritmo riesce a risolvere. Una volta che il nostro algoritmo avrà risolto il problema, basterà riaggiungere $\frac{p-1}{t}i$ al logaritmo discreto risultate per ottenere la risposta per l'intervallo i-esimo.

Ci resta solo un problema: come possiamo risolvere il problema se non sappiamo in che intervallo ci troviamo? Basta provarli tutti, tanto sono una quantità polinomiale. Con il primo algoritmo visto siamo in grado di calcolare il logaritmo discreto con probabilità $\frac{1}{2}$ e verificare che sia corretto. Basta ripetere il test per ogni intervallo finché non ci troveremo in quello che rende vera l'ipotesi del lemma. Quindi l'algoritmo che testa una volta ognuna delle ipotesi finché non trova una risposta corretta sappiamo che ci darà la risposta corretta con probabilità almeno $\frac{1}{2}$. Se nessuna delle ipotesi ha dato la risposta corretta, basta ripetere il giro un'altra volta. IN un numero atteso di 2 tentativi avremo la risposta corretta, il logaritmo discreto di y. □

□

□

Generatori di bit pseudocasuali 9

Esiste un teorema generico che dice che ogni funzione one-way ammette hard core predicate. Se esistono le funzioni one-way allora esistono algoritmi per il lancio della moneta nel pozzo. Abbiamo dimostrato che gli algoritmi di lancio della moneta non esistono, in quanto c'è sempre un problema di sincronizzazione dei messaggi, ovvero esiste sempre un ultimo messaggio che può essere non spedito che impedisce all'altro agente di vedere il risultato del lancio. Si algoritmi per il lancio della moneta nel pozzo abbiamo visto quello basato sul residuo quadratico e quello basato sul logaritmo discreto.

Da quello basato sul logaritmo discreto abbiamo ricavato che in generale l'esistenza di funzioni one-way è la condizione sufficiente per l'esistenza di algoritmi di lancio nel pozzo.

Abbiamo dimostrato che la funzione one-way g^x ammette hard core predicate $x < \frac{p-1}{2}$ in diversi passi:

- Calcolare l'hard core predicate della funzione one-way equivale a calcolare la radice quadrata principale di un numero in Z_p^*. Quando abbiamo un quadrato (che sappiamo esserlo tramite il simbolo di Legendre), questo ammette due radici: quella aritmetica e il suo opposto. Ognuna di queste radici quadrata ha il suo logaritmo discreto, solo che non sappiamo quale delle due ha il logaritmo discreto più basso. Quindi il test del predicato binario ci permette capire quale dei due logaritmi discreti è nella prima metà e di conseguenza quale delle due radici è la principale. Se abbiamo un sistema che calcola la radice quadrata principale di un numero, dato x calcoliamo x^2 e lo passiamo in input al sistema: se ci ritorna il numero iniziale, allora x è la radice quadrata principale, altrimenti basta prendere il suo opposto.
  Abbiamo usato il calcolo della radice quadrata principale come base per mostrare che se esiste un algoritmo per l'hard core predicate allora esiste un algoritmo per il calcolo del logaritmo discreto (totale). Se noi partiamo col dire che calcolare il logaritmo discreto è difficile, allora deve essere difficile anche calcolare la radice quadrata principale e di conseguenza il calcolo di questo predicato binario.
  Abbiamo scritto un algoritmo che calcola il logaritmo discreto di un numero partendo dal bit meno significativo e riconducendosi a calcolare il logaritmo discreto di un numero il cui logaritmo discreto è lo shift a destra dei bit del logaritmo discreto del numero originale. Il logaritmo discreto di 1 è banalmente 0; il bit meno significativo del logaritmo discreto di y è facile da calcolare con il simbolo di di Legendre. Se il bit è 1 lo possiamo rendere 0 dividendo per g e ottenendo un nuovo numero il cui logaritmo discreto ha il bit meno significativo a 0, cioè è un quadrato. Calcolando la radice quadrata principale del nuovo numero calcoliamo la radice discreta che è la divisione per 2 del logaritmo discreto per y. Divisione per 2 coincide con lo spostamento a destra di tutti i bit logaritmo discreto

di y. A questo punto abbiamo il logaritmo discreto del numero ottenuto shiftando a destra tutti i bit del logaritmo discreto. Lo rishftiamo a sinistra moltiplicando per 2 e aggiungiamo il bit meno significativo ottenendo il logaritmo discreto finale del numero y.

- Abbiamo detto che siamo in grado di calcolare il logaritmo discreto tramite il logaritmo che calcola la radice quadrata principale, ma noi non abbiamo un algoritmo per il calcolo della radice quadrata principale, ne abbiamo uno che la calcola a meno di un certo errore ϵ. Se l'errore con cui calcoliamo la radice quadrata principale è ϵ, qual è la probabilità di ottenere la risposta corretta con l'algoritmo descritto sopra? $(1-\epsilon)^k$, perché calcolare k radici quadrate principali e ogni calcolo deve dare la risposta corretta.
  Vogliamo che l'errore finale sia non troppo alto, quindi ϵ deve essere molto piccolo. Per il teorema di analisi, se ϵ è esponenzialmente piccolo nel security parameter, allora la probabilità di successo è almeno $\frac{1}{2}$. Con questa probabilità abbiamo il vantaggio che reiterando più volte, in media dopo 2 tentativi abbiamo la risposta corretta.
- In realtà quando ci dicono che qualcuno è riuscito a rompere il protocollo di lancio della moneta, questo qualcuno ci dice che abbiamo in mano un algoritmo in grado di indovinare il predicato binario di qualche cosa con un vantaggio polinomiale rispetto ad $\frac{1}{2}$. Il nostro obiettivo è costruire un qualcosa che abbia un errore esponenzialmente piccolo. Per farlo basta costruire tanti esperimenti indipendenti per l'algoritmo B. Con esperimenti indipendenti intendiamo un problema nuovo che è distribuito esattamente secondo la distribuzione di probabilità che si aspetta l'algoritmo B per dire che indovina con probabilità $\frac{1}{2}+\eta$. Questi esperimenti indipendenti devono essere tali che dalla risposta al nuovo esperimento sia possibile risalire alla risposta del problema originale.
  Dobbiamo costruire un nuovo input y' per la macchina B tale per cui dalla risposta che diamo ad y' siamo in grado di dare la risposta per y. Per costruire l'esperimento indipendente scegliamo un quadrato a caso e lo moltiplichiamo per y, ottenendo un quadrato a caso. L'algoritmo B sarà in grado di calcolare il logaritmo discreto di y' facilmente. Il quadrato a caso lo abbiamo costruito in modo da conoscere noi stessi il suo logaritmo discreto. Prima abbiamo calcolato il suo logaritmo discreto nella prima metà e poi abbiamo costruito il nuovo oggetto. L'oggetto lo vogliamo costruito in maniera tale che dalla risposta, ovvero dalla radice quadrata di questo oggetto, siamo in grado di trovare la radice quadrata principale di y.
  Il lemma che abbiamo dimostrato dice che questa cosa la sappiamo fare solo quando $x + 2r < p - 1$, ovvero quando nel calcolare i quadrati non andiamo oltre $p - 1$ e non ci serve usare l'operazione di modulo (a livello di esponenti stiamo solo facendo un'operazione aritmetica).
  Se $x+2r$ è sufficientemente piccolo riusciamo ad ottenere la risposta corretta. Ma a noi interessa ottenere sempre la risposta corretta. Più x è piccolo, più è probabile scegliere un r che funzioni.
  Abbiamo definito cosa significa per noi "essere piccolo": abbiamo diviso l'intervallo dei possibili logaritmi discreti in un numero di parti che non conosciamo t. Supponiamo che x stia nella prima

delle t parti e ci chiediamo quale sia la probabilità di trovare un r che vada bene per il lemma. Questa probabilità è equivalente alla probabilità di avere un r che stia in tutti gli intervalli eccetto l'ultimo. Solo quando r sta nell'ultimo intervallo, rischiamo di uscire dal limite se aggiungiamo la nostra x. La probabilità di andare in tutti gli intervalli eccetto l'ultimo è $\frac{t-1}{t}$. A questo punto abbiamo potuto calcolare qual è la probabilità di ottenere la risposta corretta all'esperimento completo: trasformare il problema, prendere la risposta al problema trasformato, dividerla per g^r per ottenere la risposta al problema originale.
La probabilità che abbiamo trovato è un limite inferiore in quanto $\frac{t-1}{t}$ non comprende anche quegli r appartenenti all'ultimo intervallo che andrebbero comunque bene. Al fine di usare il teorema di Chernoff e poter fare un numero polinomiale di esperimenti per avere la probabilità esponenzialmente vicina ad 1 di avere la risposta corretta, la nuova probabilità deve essere polinomialmente distante da $\frac{1}{2}$.
Scegliamo quindi un polinomio che ci vada bene, tipo k^{-2c}, e verifichiamo se riusciamo a trovare una soluzione alla disequazione che si genera con la probabilità di successo dell'esperimento. La soluzione ci determina il numero (polinomiale in k) minimo di intervalli in cui dobbiamo dividere il numero di logaritmi discreti. Se non ci riusciamo, non siamo in grado di dimostrare che abbiamo un vantaggio polinomiale.
In questo modo, con una quantità polinomiale in k di intervalli, abbiamo la garanzia di poter essere polinomialmente distante da $\frac{1}{2}$ con il nuovo esperimento. Grazie a questo possiamo avere la risposta corretta con probabilità esponenzialmente vicina ad 1 e quindi possiamo arrivare a calcolare il logaritmo discreto con probabilità almeno $\frac{1}{2}$.
Ma se x non si trova nel primo intervallo e qualcuno ci dice dove sta? Possiamo moltiplicare y per una quantità opportuna tale per cui il nuovo logaritmo discreto sta nel primo intervallo.
Ma se non sappiamo in che intervallo si trova x? Grazie al fatto che il numero di intervalli è polinomiale in k, possiamo provare la nostra ipotesi in ciascun intervallo. Prima o poi la proveremo in quello corretto e con probabilità $\frac{1}{2}$ avremo in mano il nostro logaritmo discreto. Se una volta provati tutti li intervallo non avremo trovato il logaritmo discreto, ritentiamo un'altra volta (con probabilità $\frac{1}{2}$ potremo non trovarlo). In media con 2 tentativi ce la facciamo.

9.1 Bit pseudocasuali

Vogliamo costruire una sequenza di bit che sia equivalente all'aver lanciato una moneta ogni volta. All'interno di un computer non vi è nulla di casuale, è tutto deterministico. Se conosciamo lo stato del pc, possiamo prevedere tutto quello che farà. Vogliamo fare in modo che il pc possa generare una serie di bit in modo deterministico che appaiano casuali a chi li osserva (per questo sono chiamato pseudocasuali).

La nostra generazione di bit pseudocasuali non può essere un processo semplice che costruisce qualcosa distribuito uniformemente, ma deve

essere un qualcosa che produce una sequenza di bit tale che nessuno con potenza di calcolo probabilistica polinomiale sia in grado di capirla.

Vogliamo un sistema PRSG (Pseudo Random Sequence Generator) che dato in input un seme di dimensione k produce in output una sequenza di bit di lunghezza l, con $l > k$, tale che nessuno che osserva la sequenza riesca a dire che non è propriamente casuale. Il seme è scelto in maniera realmente casuale. Questi generatori da un seme iniziale producono una sequenza di bit maggiore la cui difficoltà di identificazione cresce al crescere di k. Più bit usiamo per il seme iniziale, più difficile è capire che la sequenza generata non è casuale.

I generatori che andremo a costruire li possiamo vedere come moltiplicatori di casualità: a partire da una casualità vera, il seme, andremo a generare una casualità più grande rispetto a quella originale. I bit che genereremo saranno tali per cui nessun algoritmo PPT riesca a trovare regolarità al suo interno. Della regolarità esiste, noi saremmo in grado di rappresentare la sequenza in maniera più compatta, ma nessun algoritmo PPT deve essere in grado di trovare questa regola.

Cosa vuol dire generare bit pseudocasuali? Se riceviamo $b_0, b_1, ..., b_l$ facciamo fatica ad indovinare b_{l+1}, ovvero lo indoviniamo con vantaggio più piccolo di ogni polinomio. Quindi la probabilità $P[\Sigma]$ di:

- Prendiamo il seme $S \in \{0,1\}^k$ e la sequenza $b_1, ..., b_l, b_{l+1}$ generata dal generatore $G(S)$;
- Prendiamo b ottenuto dall'algoritmo $A(b_1, ..., b_l)$, che tenta di predire il bit successivo a partire dalla sequenza $(b_1, ..., b_l$;
- Verifichiamo se $b == b_{l+1}$.

deve essere:

$$\left| P[\Sigma] - \frac{1}{2} \right| < k^{-\omega(1)} \tag{9.1}$$

Questa formula determina quando il generatore pseudocasuale è corretto, ovvero quando un attaccante non è capace di indovinare il bit successivo conoscendo i bit precedenti.

9.2 Generazione di bit pseudocasuali basata sulla difficoltà del logaritmo discreto

Vediamo ora un algoritmo per la generazione di bit pseudocasuali basato sulla difficoltà del logaritmo discreto:

1. Fissiamo p, g;
2. Prendiamo il seme $x \in_R \{0, p-1\}$;
3. Definiamo:

a_0	a_1	a_2	a_3	...	a_l
x	g^x	g^{g^x}	$g^{g^{g^x}}$	...	$g^{g^{\cdot^{\cdot^{g^x}}}}$

In maniera compatta questo può essere riscritto come:

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

Stiamo generando una sequenza di elementi $a_0, a_1, \ldots$ di Z_p^*, dove il primo elemento è il nostro seme, e ogni elemento successivo non è altro che il generatore elevato all'elemento precedente.

4. Definiamo ora i nostri bit:

$$\begin{array}{cccccc} a_0 & a_1 & a_2 & a_3 & \ldots & a_l \\ x & g^x & g^{g^x} & g^{g^{g^x}} & \ldots & g^{g^{\cdot^{\cdot^{g^x}}}} \end{array}$$

$$\begin{array}{cccccc} b_0 & b_1 & b_2 & b_3 & \ldots & b_l \end{array}$$

dove:

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

5. Ritorniamo la sequenza

$$b_l \, b_{l-1} \, b_{l-2} \ldots b_1 \, b_0$$

Dimostrazione di correttezza. Supponiamo per assurdo che esiste un algoritmo in grado di predire il successivo. Questo algoritmo prende in input una quantità di bit e indovina il bit successivo con un vantaggio che è più grande di qualche polinomio.

Supponiamo che dati in input i bit $b_l \ldots b_3$ ($l-3$ bit) l'algoritmo riesca ad indovinare il bit b_2.

Se osserviamo la sequenza notiamo che:

- b_2 è l'hard core predicate di a_3:

$$\begin{aligned} a_3 &= g^{a_2} \\ b2 &= hcp(a_2) \end{aligned}$$

- a_2 è il logaritmo discreto di a_3:

$$\begin{aligned} a_3 &= g^{a_2} \\ DL(a_3) &= DL(g^{a_2}) = a_2 \end{aligned}$$

Se noi abbiamo a_3, calcolare b_2 vuol dire calcolare l'hard core predicate di a_3. Dimostriamo ora che se abbiamo un algoritmo che indovina b_2 a partire dalla sequenza $b_l \ldots b_3$ con vantaggio polinomiale, allora abbiamo un algoritmo che calcola l'hard core predicate con vantaggio polinomiale.

Preso in input il numero, lo vediamo come se fosse a_3 e generiamo la sequenza fino ad a_l. Generati questi oggetti, possiamo costruire la sequenza $b_l \ldots b_3$. Applichiamo l'algoritmo che indovina b_2. L'algoritmo

indovina con vantaggio polinomiale rispetto a $\frac{1}{2}$, cioè abbiamo calcolato l'hard core predicate di a_3 con vantaggio polinomiale rispetto a $\frac{1}{2}$.

L'eventuale previsione del bit successivo diventa il calcolo dell'hard core predicate (fino all'ottenimento del seme). L'algoritmo che abbiamo costruito restituisce i bit alla rovescia in quanto altrimenti il bit successivo della sequenza sarebbe stato il risultato di una funzione facile da calcolare. Noi abbiamo usato una funzione difficile da calcolare sui semi per dire che non riusciamo a indovinare il bit successivo. Restituirli alla diritta equivale a calcolare una funzione one-way, che sappiamo calcolare, per ottenere il bit successivo.

Restituendo i bit alla diritta questa dimostrazione non avrebbe funzionato. Ma questo significa che non è possibile restituire i bit alla diritta? No, se noi non siamo capaci di trovare una dimostrazione per qualcosa, non significa che quel qualcosa sia falso. La nostra dimostrazione del protocollo alla rovescia si basa sul fatto che conoscendo i semi non siamo in grado di calcolare b_2. Se non siamo in grado di calcolarlo conoscendo i semi, a maggior ragione non siamo in grado di calcolarlo conoscendo i bit. Se conosciamo solo bit, siamo molto più in difficoltà, ma questa dimostrazione non ci permette di catturare questo concetto.

Definizione con distinguisher:

$$\forall D \in PPT\ \forall l$$
$$P_K^{D,G} = Pr[S \in_R \{0,1\}^k; b_0 \dots b_l \leftarrow G(S); D(b_1 \dots b_l) == 1]$$
$$P_K^{D,U} = Pr[S \in_R \{0,1\}^{l+1}; D(b_0 \dots b_l) == 1]$$
$$\left| P_K^{D,G} - P_K^{D,U} \right| < k^{-\omega(1)}$$

Se $b_l \dots b_0$ soddisfa la definizione di indovinare allora soddisfa anche la definizione di distinguere, quindi la sequenza $b_l \dots b_0$ è indistinguibile da una sequenza casuale.

Possiamo di conseguenza che $b_0 \dots b_l$ sia indistinguibile? Sia D un distinguisher per $b_l \dots b_0$ (che distingue tra una sequenza veramente casuale e non) e sia D' una macchina tale per cui:

$$D' : b_l \dots b_0 \rightarrow D(b_0 \dots b_l)$$
$$b_0 \dots b_l \rightarrow D(b_l \dots b_0)$$

Se D' riesce a distinguere i bit diritti, vuol dire che D sta distinguendo o bit rovesci; se D' riesce a distinguere i bit rovesci, vuol dire che D sta distinguendo o bit diritti. Quindi distinguere i bit rovesci o dritti è la stessa cosa: se abbiamo distinguisher per i bit rovesci basta invertire i bit dati in input e costruiamo un distinguisher per i bit diritti.

Con la definizione di indovinare siamo riusciti a dimostrare che i bit alla rovescia funzionano, ma non per i bit diritti. Grazie all'equivalenza tra indovinare e distinguere siamo riusciti a dimostrare che anche i bit diritti funzionano.

Il vantaggio di restituire i bit alla rovescia sta nel fatto che dopo aver ritornato b_0 non siamo in grado di andare avanti con la sequenza: per ottenere il bit successivo dovremmo calcolare il logaritmo discreto del

seme (non siamo in grado di calcolare il logaritmo discreto). Il vantaggio dei bit diritti sta nel fatto che possiamo generare bit ogni volta che sono richiesti (dimensione arbitraria). □

9.2.1 Indovinare ↔ Distinguere

indovinare → distinguere. Su input $b_0 \dots b_{l-1}$ indoviniamo b_l con vantaggio polinomiale:

$$\left| Pr[A(b_0 \dots b_{l-1}) = b_l] - \frac{1}{2} \right| > k^{-\omega(1)}$$

Quindi

$$Pr[A(b_0 \dots b_{l-1}) = b_l] > \frac{1}{2} + k^{-\omega(1)}$$

Per dimostrare che violando la definizione di indovinare violiamo quella di distinguere possiamo costruire un distinguisher che:

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

A partire dall'algoritmo A che attacca il sistema di generazione dei bit pseudocasuali secondo la definizione di indovinare, cerchiamo di costruire un distinguisher che distingue sequenza casuali da pseudocasuali.

Vediamo qual è la probabilità che l'algoritmo A indovini un bit che viene scelto in modo veramente casuale e indipendente da $b_0 \dots b_{l-1}$. Noi sappiamo che se c'è un qualsiasi algoritmo che genera un bit e lo mettiamo in XOR con un bit veramente casuale, la probabilità di uguaglianza è esattamente $\frac{1}{2}$. Quindi la probabilità che $A(b_0 \dots b_{l-1}) = b_l$ è esattamente $\frac{1}{2}$:

$$P_k^{D,U} = \frac{1}{2} \tag{9.2}$$

Sappiamo che la probabilità che A indovini b_l è:

$$P_k^{D,G} = Pr[A(b_0 \dots b_{l-1}) = b_l] > \frac{1}{2} + k^{-c} \tag{9.3}$$

Di conseguenza

$$P_k^{D,G} - P_k^{D,G}\frac{1}{2} + k^{-c} - \frac{1}{2} = k^{-c} \tag{9.4}$$

La differenza tra le due probabilità è maggiore del polinomio k^{-c}. Di conseguenza. Quindi se abbiamo l'attaccante per la definizione di indovinare abbiamo anche il distinguisher per la definizione di distinguere. □

indovinare → distinguere. Date la sequenza $b_1 \dots b_l$ pseudocasuale e la sequenza $r_1 \dots r_l$ casuale, sia D distinguisher che vede al differenza tra le

due. Con queste due sequenza non possiamo fare molto per proseguire con la dimostrazione.

Definiamo la sequenza $S_i = b_1 \dots b_i\ R_{i+1} \dots r_l$. Allora S_0 è la sequenza pseudocasuale e S_l è la sequenza casuale.

Definiamo P_i come la probabilità che D restituisca 1 con input S_i; allora $P_0 = P_k^{D,R}$ e $P_l = P_k^{D,G}$. Sapendo che $|P_l - P_0| > k^{-c}$, facendo la costruzione per interpolazione arriviamo a dire che

$$\exists i\ |P_i - P_{i+1}| > k^{-c}$$

Cerchiamo ora di definire A che predice il bit $i+1$ conoscendo i bit precedenti:

$$A(b_1 \dots b_l) = \begin{cases} R_{i+1} & \text{se } D(b_1 \dots b_i\ R_{i+1} \dots r_l) = 1 \\ \overline{R_{i+1}} & \text{se } D(b_1 \dots b_i\ R_{i+1} \dots r_l) = 0 \end{cases}$$

Vediamo ora qual è la probabilità che $Pr[A(b_1 \dots b_l) = b_{l+1}]$:

$$Pr[A(b_1 \dots b_l) = b_{l+1}] = Pr[R_{i+1} = b_{b+1}] \cdot P_{i+1} + Pr[R_{i+1} = \overline{b_{b+1}}] \cdot X$$

Quando A indovina? Distinguiamo sul valore di R_{i+1}: la probabilità di indovinare è data dalla probabilità di indovinare quando il bit successivo è generato (b_{i+1}) sommata alla probabilità di indovinare quando il bit è casuale. Se il bit successivo è quello pseudogenerato, la probabilità di indovinare è P_{i+1}, ovvero la probabilità che il distinguisher dia 1 quando sappiamo che i primi $i+1$ bit sono generati (sequenza $b_1 \dots b_i\ b_{i+1}\ r_{i+2} \dots r_l$); se il bit successivo è casuale, la probabilità di indovinare è X, che è la probabilità che restituisca 0 sulla sequenza $b_1 \dots b_i\ \overline{b_{i+1}}\ r_{i+2} \dots r_l$.

Procediamo col calcolo della probabilità. Poiché il bit b_{i+1} è scelto in maniera casuale, la probabilità che sia r_{i+1} è esattamente $\frac{1}{2}$:

$$Pr[A(b_1 \dots b_l) = b_{l+1}] = \frac{1}{2}P_{i+1} + \frac{1}{2}X$$

Proviamo ora a capire che relazione esiste tra X, P_{i+1} e P_i. Vediamo quanto vale P_i, ovvero la probabilità che D restituisca 1 con input la sequenza con bit i-esimo casuale, e i successivi pseudocasuali:

$$\begin{aligned} P_i =& Pr[\textbf{bit } i+1 = b_{i+1}]Pr[D \text{ dia 1 su } b_1 \dots b_i\ b_{i+1} \dots b_l] + \\ & Pr[\textbf{bit } i+1 = \overline{b_{i+1}}]Pr[D \text{ dia 1 su } b_1 \dots b_i\ \overline{b_{i+1}} \dots b_l] \end{aligned}$$

Risolvendo:

$$P_i = \frac{1}{2}P_{i+1} + \frac{1}{2}(1 - X)$$

Da questa formula ricaviamo che

$$X = 1 - (2P_i - P_{i+1})$$

Risolvo X nell'espressione sopra:

$$\begin{aligned} Pr[A(b_1 \dots b_l) = b_{l+1}] &= \frac{1}{2}P_{i+1} + \frac{1}{2}X \\ &= \frac{1}{2}P_{i+1} + \frac{1}{2}(1 - (2P_i - P_{i+1})) \\ &= \frac{1}{2}P_{i+1} + \frac{1}{2} - P_i + \frac{1}{2}P_{i+1} \\ &= \frac{1}{2} + P_{i+1} - P_i \end{aligned}$$

$$Pr[A(b_1 \dots b_l) = b_{l+1}] - \frac{1}{2} = P_{i+1} - P_i$$

Di conseguenza

$$\left| \underbrace{Pr[A(b_1 \dots b_l) = b_{l+1}]}_{Pr[A \text{ corretto}]} - \frac{1}{2} \right| = |P_{i+1} - P_i| \geq k^{-c}$$

□

9.3 Algoritmo di Blum Blum Shoup

Algoritmo per la generazione di bit pseudocasuali basato sulla difficoltà del calcolo di radici quadrate in Z_n^*. La funzione elevamento al quadrato in Z_{pq}^* è una funzione one-way trapdoor. Sappiamo che l'inversione equivale a fattorizzare (dimostrazione $MCD(x + y, n)$). Se conosciamo p, q allora sappiamo calcolare le radici.

Theorem 9.3.1 (Primi di Blum) *Se*

$$p, q \equiv 3 \pmod 4$$

allora per ogni quadrato x esiste un'unica radice che un quadrato

L'hard core predicate che usiamo per la radice quadrata è il $LSB(\sqrt{x})$, quando vale il teorema sopra.

Vediamo l'algoritmo:

- Sia $n = pq$ con p, q primi di Blum scelti a caso;
- Sia $x \in_R Z_n^*$;
- Definiamo:

a_1	a_2	a_3	...	a_l
x^2	$(x^2)^2$	$((x^2)^2)^2$	...	x^{2^l}

dove:

$$\begin{cases} a_1 & = x^2 \\ a_{i+1} & = a_i^2 \end{cases}$$

Stiamo applicando successivamente la nostra funzione one-way trapdoor. Definiamo $b_i = LSB(a_i)$:

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

$$\begin{array}{ccccc} b_1 & b_2 & b_3 & \dots & b_l \end{array}$$

- Restituiamo i bit $b_l \dots b_1$.

9.4 Sistema a chiave pubblica dimostrabilmente sicuro - Algoritmo di Blum Goldwasser

Generazione chiavi

- $p, q \in_R$ primi di Blum con k bit;
- $n = pq$;
- $P_k = n, S_n = (p, q)$.

Encryption

- Vogliamo cifrare il messaggio $m : \ m_1 \dots m_l$;
- $x \in Z_n^*$;
- Usiamo Blum Blum Shoup (BBS) per generare l bit pseudocasuali:

$$\begin{array}{cccccc} & x^2 & (x^2)^2 & (x^2)^3 & \dots & x^{2^l} \\ & b_1 & b_2 & b_3 & \dots & b_l \\ \oplus & m_1 & m_2 & m_3 & \dots & m_l \\ \hline & c_1 & c_2 & c_3 & \dots & c_l \quad x^{2^{(l+1)}} \end{array}$$

Abbiamo usato one-time pad con una chiave che è pseudogenerata a partire da un seme scelto a caso. Questo protocollo è praticamente one-time pad. Noi sappiamo che se la sequenza $b_1 \dots b_l$ è veramente casuale, allora il cyphertext non contiene alcuna informazione circa il plaintext (quindi il cyphertext costruito è completamente sicuro). Supponiamo che qualcuno riesca a ottenere qualcosa dal cyphertext, che riesca a distinguere qualsiasi coppia di messaggi cifrata con questo sistema. Quel qualcuno che riesce a vedere se due cyphertext sono dello stesso messaggio o di messaggi diversi sappiamo che non lo riesce a fare in presenza di una chiave veramente casuale. Quel distinguisher per due messaggi del nostro crittosistema diventerebbe un distinguisher per un generatore di bit pseudocasuali: se sono veramente pseudocasuali D non distingue, se i bit sono pseudocasuali si comporta in maniera diversa. Questa macchina diventa un distinguisher per Blum Blum Shoup, che soddisfa le proprietà di un generatore di bit pseudocasuali.

L'ultimo elemento del cyphertext $x^{2^{(l+1)}}$, agli occhi di chi non ha la chiave pubblica è un elemento casuale di Z_n^*.

Decryption

- Per ricavare x^2, conoscendo la fattorizzazione di n, calcoliamo le radici quadrate di ogni elemento (per i primi di Blum ne esiste solo una che è un quadrato). Esiste un algoritmo che calcola direttamente x^2 da $x^{2^{(l+1)}}$ senza calcolare le radici intermedie;
- Generiamo $b_1 \dots b_l$ a partire da x^2;
- Eseguiamo

$$\begin{array}{cccccc} & b_1 & b_2 & b_3 & \dots & b_l \\ \oplus & c_1 & c_2 & c_3 & \dots & c_l \end{array}$$

  Per ottenere $m_1 \dots m_l$.

Questo sistema è dimostrabilmente sicuro in quanto riuscire a distinguere messaggi dal loro cyphertext equivale a distinguere la sequenza $b_1 \dots b_l$ da una sequenza puramente casuale equivale ad avere un distinguisher per il sistema Blum Blum Shoup.

Il crittosistema è complesso quanto RSA: l'algoritmo calcola quadrati l volte, dove il calcolo di ogni quadrato costa k^2. Quindi la complessità della codifica è

$$\Theta(lk^2)$$

RSA codifica a blocchi; codificando anche qui a blocchi di k bit la complessità diventa

$$\Theta(lk^2) = \Theta(k \cdot k^2) = \Theta(k^3)$$

Funzioni pseudocasuali | 10

Vogliamo ora trovare un modo per evitare che nessuno possa modificare il messaggio. Possiamo usare sistemi crittografici non malleabili oppure possiamo aggiungere della ridondanza (codice ciclico, bit di parità).

Noi vogliamo però costruire dei sistemi che sono resistenti anche da chi tenta di modificare il messaggio in maniera controllata per far quadrare anche il codice di ridondanza. I codici ciclici di ridondanza vengono solitamente usati per proteggerci dall'errore di trasmissione dei dati, che è un errore casuale.

Vogliamo quindi accodare al messaggio un qualcosa che nessuno riesca a capire se non il reale destinatario finale. Se per esempio scegliessimo al messaggio una funzione casuale (da k bit a k bit) condiviso con l'altra parte, potremmo accodare al messaggio $f(m)$ come **autenticazione**. In questo modo chi non conosce la funzione casule, che vede solo bit casuali appesi al messaggio, non può modificare in l'autenticazione $f(m)$ del messaggio in modo corretto (lo indovina con probabilità $\frac{1}{2^k}$, con k lunghezza dell'autenticazione).

10.1 Generazione di funzioni pseudocasuali

Sia U_k l'insieme delle funzioni

$$U_k : \{0,1\}^k \to \{0,1\}^k$$

Vogliamo scegliere $f \in_R U_k$. Come facciamo? Generiamo una sequenza di bit pseudocasuali per generare l'indice di f in una enumerazione di $\{0,1\}^k \to \{0,1\}^k$. L'indice che generiamo è indistinguibile da un indice casuale, e quindi la funzione è come se fosse stata scelta casualmente.

Questa definizione ha però un problema: la cardinalità di un insieme di funzioni da A a B é $|B|^{|A|}$, quindi

$$|U_k| = (2^k)^{(2^k)}$$

Se la cardinalità di questo insieme è $(2^k)^{(2^k)}$, quanti bit ci servono per denotare l'indice di una funzione? I bit che ci servono sono

$$\log_2 |U_k| = k \cdot 2^k$$

Ci servirebbe una quantità di bit esponenziale, e quindi un tempo esponenziale solo per generare gli indici. Noi vogliamo denotare una funzione usando k bit. Dobbiamo lavorare con un insieme $F_k \subseteq U_k$. Le funzioni che andremo ad usare saranno quindi un sottoinsieme delle possibili funzioni. Vogliamo comunque sceglierle in modo tale che nessuno sia in grado di accorgersi che la funzione sia stata scelta da un insieme più piccolo.

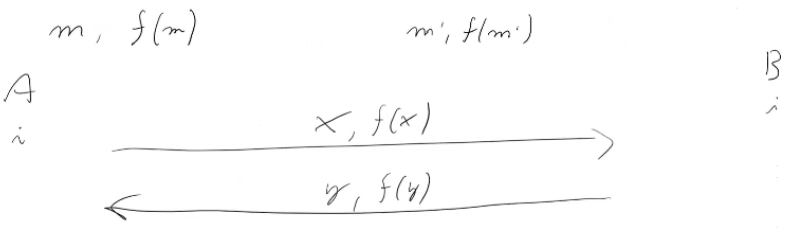


Figura 10.1: Alice invia un messaggio, Bob lo verifica, Bob risponde, Alice verifica la risposta.

Definition 10.1.1 (Funzione pseudocasuale) *Sia $F_k \subseteq U_k$ tale per cui:*

1. $|F_k| = 2^k$;
2. $\exists A \in PPT$ *che calcola la seguente funzione:*

$$i, x \rightarrow f_i(x)$$

Ogni volta che vogliamo applicare la funzione i-esima all'input x, applichiamo un algoritmo polinomiale in grado di farlo;

3. *Sia $D \in PPT$ una macchina che interroga F e ritorna $\{0,1\}$. Siano*
   - $P_k^{D,U}$ *la probabilità che D restituisca 1 quando $f \in_R U_k$ (campionata casualmente dall'intero insieme U_k);*
   - $P_k^{D,F}$ *la probabilità che D restituisca 1 quando $f \in_R F_k$.*

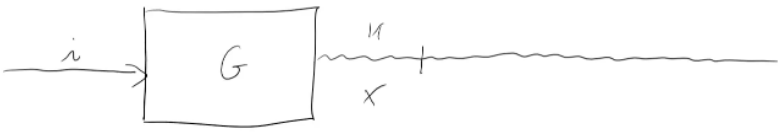
   Allora

$$\left| P_k^{D,U} - P_K^{D,F} \right| < k^{-\omega(1)}$$

Nessuno di coloro che conosce l'indice della funzione deve accorgersi che la funzione è generata (ovvero presa dall'insieme piccolissimo F_k).

10.1.1 Funzione 1

Siano G un PRSG (Pseudo Random Sequence Generator) e i il seme. Allora il seguente sistema:



su input x genera k bit, li memorizza e li restituisce. Su un successivo input y, se $y = x$ restituisce il risultato precedenti, altrimenti genera altri k bit, li memorizza e li restituisce. In generale, dato un x già usato, G ritorna i k bit associati.

Questo è un generatore di bit pesudocasuali? Supponiamo che Alice e Bob vogliano scambiarsi dei messaggi autenticati $(m, f(m))$ con questo sistema. Entrambi conoscono il seme i da passare al generatore di funzioni pseudocasuali. Se la comunicazione è sequenziale, come in figura 10.1, il sistema funziona correttamente in quanto l'ordine in cui vengono generate le sequenze di k bit è la stessa per Alice e Bob.

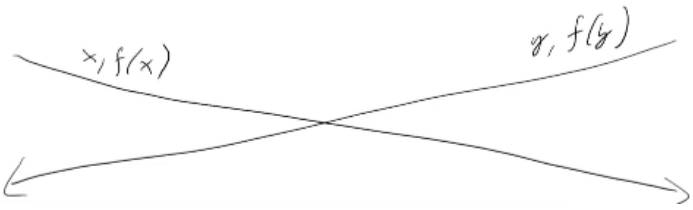


Figura 10.2: Alice invia un messaggio, Bob invia un messaggio, Bob verifica, Alice verifica.

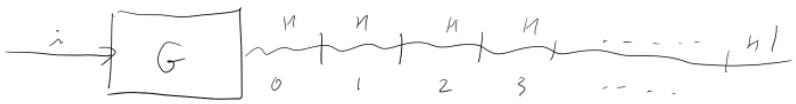
Se invece Alice e Bob inviano ciascuno un messaggio all'altro contemporaneamente, i messaggi non vengono autenticati 10.2. Il sistema descritto associa la prima sequenza di bit al primo input passato, la seconda al secondo (se diverso da quello già memorizzato) e così via. Quindi il sistema di Bob assocerà la prima sequenza a y, mentre quello di Alice a x, rendendo così errata l'autenticazione.

Questo sistema potrebbe essere usato quando esiste un repository centralizzato. Inoltre, necessita di un database che mantenga tutte le richieste già effettuate; maggiore sono le entry, più tempo è necessario a ottenere una risposta dalle interrogazioni.

Questo sistema non soddisfa la nostra definizione di funzioni pseudocasuali in quanto F_k non è un insieme ben definito.

10.1.2 Funzione 2

Siano G un PRSG (Pseudo Random Sequence Generator) e i il seme. Dato il seguente sistema



dividiamo l'output di G in tanti gruppi di k bit ciascuno. Associo il primo gruppo a f_0, il secondo a f_1, il terzo a f_3 e così via. Allora

$$f_i(x) = \text{ x-esimo gruppo di } k \text{ bit restituito da } G$$

Con questo sistema l'insieme F_k è ben definito e la sua cardinalità è $|2^k|$. La seconda proprietà è verificata? No, non esiste alcun algoritmo $A \in PPT$ che soddisfi al proprietà.

Dimostrazione. Sia x una sequenza di k bit a 1, ovvero:

$$x = 1 \ldots . 1 = 2^k - 1$$

Per calcolare $f_i(x)$, dobbiamo prendere il gruppo in posizione $2^k - 1$. Per come è definito un generatore id bit pseudocasuali, per calcolare questo gruppo dobbiamo generare tutti i bit che vengono prima, che sono $k \cdot 2^k$ bit. Ci serve un tempo esponenziale. □

10.1.3 Funzione 3

Siano G un PRSG (Pseudo Random Sequence Generator) e $i \oplus x$ il seme. Dato il seguente sistema:

prendiamo solo i primi k bit come risultato. Allora:

$$f_i(x) = \text{ primi } k \text{ bit di } G \text{ con seme } i \oplus x$$

Verifichiamo se rispetta le tre proprietà:

1. L'insieme ha cardinalità $|2^k|$ e l'indice i definisce in maniera univoca la funzione ($f_i(x)$ è ben definita);
2. La funzione $f_i(x)$ è calcolabile polinomialmente;
3. Supponiamo per assurdo che esiste D tale per cui:

$$\left| P_k^{D,U} - P_K^{D,F} \right| > k^{-\omega(1)}$$

   Abbiamo quindi una macchina PPT che vede la differenza. Vorremmo trasformare D in macchina che attacca il generatore G, ovvero che ci permette di violare la sicurezza di G (che permette di distinguere sequenze di bit pseudocasuali da realmente casuali oppure permette di predire il bit successivo). Per farlo dobbiamo prendere il distinguisher D e vedere gli esperimenti che questo fa sulla funzione f come esperimenti che noi facciamo sul generatore G. Se gli esperimenti che facciamo sono compatibili, ovvero soddisfano tutti i vincoli che abbiamo dato nel definire la correttezza di un generatore di bit pseudocasuali, possiamo vedere D come un distinguisher per il generatore G.
   Affinché D sia veramente un distinguisher che ci permetta di vedere qualcosa di interessante sul generatore G, è importante che gli esperimenti che D sta facendo implicitamente su G passando per f siano compatibili con gli esperimenti che un distinguisher sta facendo su G quando diciamo che G non è distinguibile.
   Se riprendiamo la definizione di generatore di bit pseudocasuali, l'esperimento che si fa è il seguente:

   - Genera un seme a caso, usa il seme per generare una sequenza e fa osservare a D la sequenza generata. D deve cercare di dire qualcosa;
   - Genera un altro seme a caso, usa il seme per generare una sequenza e fa osservare a D la nuova sequenza generata. D deve cercare di dire di nuovo qualcosa;
   - E così via.

   A D possiamo far vedere un numero qualsiasi di sequenze, ma gli esperimenti che D può fare devono essere esperimenti indipendenti.

Nel caso specifici della funzione 3, il distinguisher può scegliere valori diversi di x (può interrogare f su $x_1, x_2, ...$), ma il generatore G conosce $x_1, x_2, ...$. Se prendiamo lo XOR tra 2 degli input usati da G, otteniamo lo XOR del seme che è stato usato da G per generare $f_i(x)$. Il nostro distinguisher può fare diversi esperimenti dove va a controllare $k$bit generati da G, ma a differenza della nostra definizione di bit pseudocasuali, il nostro distinguisher è in grado di osservare l'output di G su 2 semi di cui è nota la differenza.
Se ad esempio interroghiamo $f(0) = a$ e $f(1^k) = b$, sappiamo che il seme usato per generare a è il complemento del seme usato per generare b. Il seme usato per $f(0)$ è esattamente i, mentre il seme usato per $f(1^k)$ è il complemento di i. In generale se interroghiamo $f(x) = a'$ e $f(y) = a''$, lo XOR dei semi usati per generare a' e a'' è $x \oplus y$.
Gli esperimenti che il distinguisher D riesce a fare implicitamente su G sono esperimenti che il distinguisher che abbiamo usato per definire la correttezza di G non è in grado di fare. Quindi il distinguisher che sta lavorando sulle funzioni pseudocasuali è più potente del distinguisher che abbiamo ammesso nella definizione di correttezza di un generatore di bit pseudocasuali. I casi sono due:

- Abbiamo definito un generatore di bit pseudocasuali troppo debole e quindi ci serve una definizione più forte che ammetta anche esperimenti di questi tipo, oppure possiamo dimostrare che il generatore che abbiamo definito resiste anche agli attacchi che questo D è in grado di fare;
- Non vediamo alcun vantaggio che l'attaccante possa trarre da questo tipo di esperimento più potente e quindi lo teniamo "buono".

In crittografia bisogna sempre essere scettici: se non vediamo un modo in cui si riesca a rompere qualche cosa non significa che qualcun altro possa trovarlo.
In questo caso è possibile costruire un generatore G, che è un generatore di bit pseudocasuali, che costruito in questo contesto permette effettivamente di costruire un distinguisher. Nel momento in cui siamo in grado di farlo, questa dimostrazione non potrà mai funzionare, ovvero non si potrà mai partire di un distinguisher D per il generatore di funzioni pseudocasuali e costruire un distinguisher per il generatore di bit pseudocasuali.
Costruiamo ora un generatore di bit pseudocasuali che in questo contesto è distinguibile.
Supponiamo di avere un generatore G che prende in input $k-1$ bit e produce la sequenza pseudocasuale. Il generatore è contenuto in una macchina G' che prende un seme i di k bit. La nostra richiesta è di produrre un generatore che prende in input k bit, ma noi ne abbiamo a disposizione solo uno a $k-1$ bit. Allora procediamo così:

- Dato il seme $i = i_0\ i_1\ ...\ i_{k-1}$ di k bit, usiamo il bit i_0 some selettore;
- Se $i_0 = 0$, allora ritorniamo:

$$G(i_1\ ...\ i_{k-1})$$

- Se $i_0 = 1$, allora ritorniamo:

$$G(\overline{i_1 \dots i_{k-1}})$$

Poiché il complemento di una sequenza casuale resta una sequenza casuale, stiamo producendo sequenza pseudocasuali di bit con seme da $k-1$ bit in entrambi i casi. G prende sempre in input una sequenza di bit casuali che è generata uniformemente tra i gruppi di $k-1$ bit: con probabilità $\frac{1}{2}$ il seme è la sequenza casuale originale generata, con probabilità $\frac{1}{2}$ è il suo complemento. L'output di G' è l'output di un seme distribuito uniformemente tra tutti i semi. Quindi G' si comporta esattamente come G, anche se prende in input un bit in più. Tuttavia se questo oggetto lo usiamo nel contesto sopra scopriamo che $f(0)$ e $f(1^k)$ sono la stessa cosa:

- se con $f(0)$ generiamo il seme $i_0\ i_1 \dots i_{k-1}$, con $i_0 = 0$, allora

$$G' = G(i_1 \dots i_{k-1})$$

- Di conseguenza, $f(1^k)$ genererà il seme $\overline{i_0\ i_1 \dots i_{k-1}}$, con $\overline{i_0} = 1$, e quindi:

$$G' = G(\overline{\overline{i_1 \dots i_{k-1}}}) = G(i_1 \dots i_{k-1})$$

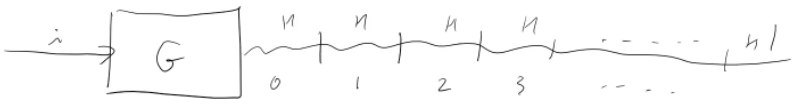
Il distinguisher diventa una macchina banale:

- Calcola $f(0)$ e $f(1^k)$;
- Se sono uguale ritorna 1;
- Se sono diversi ritorna 0.

Questo distinguisher di fronte ad una funzione pseudogenerata rispenderà sempre 1, di fronte ad una funzione casuale risponderà 1 solo quando $f(0) = f(1^k)$, ovvero con probabilità $\frac{1}{2^k}$.
Questa costruzione, nonostante sia stupida, è permessa dalla nostra definizione di generatore pseducasuale.

10.1.4 Funzione 4

Torniamo alla funzione 2:



dividiamo l'output di G in tanti gruppi di k bit ciascuno. Associo il primo gruppo a f_0, il secondo a f_1, il terzo a f_3 e così via. Allora

$$f_i(x) = \text{ x-esimo gruppo di } k \text{ bit restituito da } G$$

Per poter calcolare $f_i(x)$ in PPT usiamo un generatore BBS (Blum Blum Shoup). Quando lavoriamo col logaritmo discreto non siamo in grado di

calcolare elemente troppo distanti, ma se dobbiamo calcolare:

$$f_i(2^k - i) = \text{ bit dalla posizione } k \cdot (2^k - 1) \text{ a } k \cdot 2^k$$

Come facciamo? Dobbiamo prendere

$$i^{2^{(2^k-1)-1}}$$

Questo esponenziale sembra complicato da calcolare. Tuttavia sappiamo che gli esponenti lavorano modulo $\varphi(n)$, quindi stiamo calcolando un esponente modulo $\varphi(n)$:

$$i^{2^{(2^k-1)-1} \pmod{\varphi(n)}}$$

Questo lo sappiamo calcolare con l'iterative squaring. Una volta calcolato l'esponente possiamo calcolare l'esponenziale modulo $\varphi(n)$ in tempo polinomiale:

$$i^{2^{(2^k-1)-1} \pmod{\varphi(n)}} \pmod{\varphi(n)}$$

Di conseguenza questo oggetto è calcolabile in PPT, quindi disponiamo di un A che calcola $f_i(x)$ in tempo polinomiale. Non possiamo fare questo con ogni generatore, ma solo con BBS riusciamo ad ottimizzare le operazione e calcolare direttamente un bit senza calcolare quelli precedenti.

A questo punto disponiamo di un generatore G che rispetta i punti 1 e 2. Vediamo ora se rispetta anche il punto 3.

Supponiamo di avere un distinguisher che interroga f, facendo ovviamente implicitamente degli esperimenti su G (genera un seme, guarda alcuni gruppi a caso - $0, 1, 5, 2^k - 1$ -). Il distinguisher che stiamo usando qui ha la stessa potenza del distinguisher usato nella definizione di G? Quando parliamo di esperimenti su G, generiamo una sequenza di bit di una data lunghezza e andiamo a guardarla, quindi generiamo tutti i bit. In questo caso possiamo generare una sottosequenza di bit senza generare i bit che la precedono (possiamo generare il gruppo 4 senza generare i primi 3). Potremmo quindi anche andarci bene, in quanto vedere solo un gruppo è "peggio" che vedere un gruppo e quelli che lo precedono.

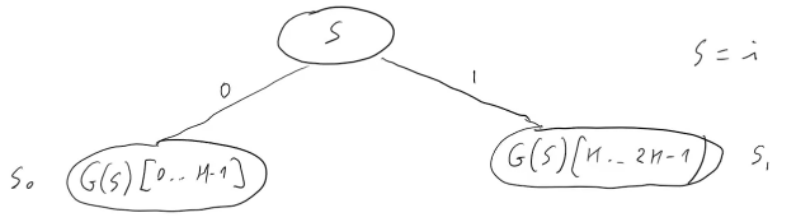
Il problema è che il nostro distinguisher può vedere anche il gruppo $2^k - 1$ come una sequenza polinomiale di bit, mentre il distinguisher per G vedrebbe $2^k - 1$, più tutta la sequenza precedente, come una sequenza esponenziale. Noi non permettiamo ad un algoritmo polinomiale di vedere una sequenza esponenziale, di conseguenza D è in grado di vedere dei bit che il distinguisher per G non potrebbe mai vedere. Se quei bit li fossero quelli utili a capire se ci troviamo di fronte a un generatore di bit pseudocasuali o no, per la definizione di G questi non sarebbero un problema perchè il distinguisher non li vedrebbe mai. Con D però siamo in grado di vedere questi bit, quindi l'esperimento che stimo facendo con D non può essere tradotto in un esperimento contro il generatore G.

Come facciamo a concludere che l'esistenza di un distinguisher per le funzioni implica l'esistenza di un distinguisher per un generatore, quando questo distinguisher sta facendo cose che non potrebbero essere mai fatte su G? Non possiamo farlo.

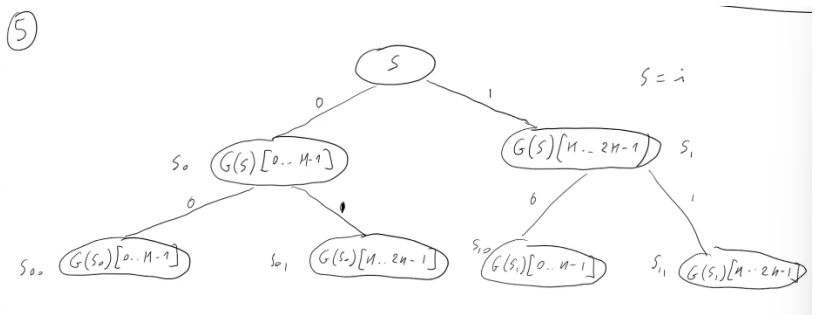
In generale i crittografi, se non trovano una dimostrazione di correttezza per un sistema, tendono a scartarlo, a meno che non ve ne siano altri usabili.

10.1.5 Funzione 5

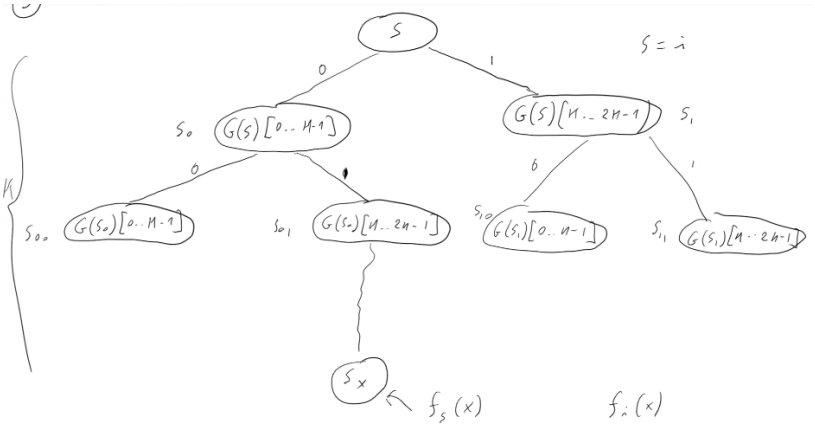
Partiamo da un seme $s = i$ e usiamo usiamo un generatore G per generare $G(s)[0 \dots k-1]$ (generatore per i bit $\dots k-1$) e usiamo lo stesso generatore per generare $G(s)[k \dots 2k-1]$. Stiamo usando il generatore G per generare $2k$ bit: i primi k bit li posizioniamo a sinistra, i restanti a destra. Chiamiamo il ramo di destra 0 e il ramo di sinistra 1. Chiamiamo i due semi che abbiamo generato s_0 e s_1.



Aggiungiamo un altro livello.



Ripetiamo il procedimento fino ad arrivare al nodo s_x.



Siamo scesi per k livelli, arrivando a nodi dove i semi sono indicizzati da k bit. Il seme indicizzato da x è $f_s(x)$, cioè $f_i(x)$, in quanto abbiamo ridenominato i in s.

Abbiamo costruito un albero di profondità k e abbiamo scoperto come costruire i nodi di questo albero. L'albero ha 2^k nodi. Le foglie rappresentano i valori della funzione sui vari argomenti, a partire dal seme s messo in radice.

Dato s la funzione è ben f_s definita (costruiamo l'albero e andiamo a prendere la foglia di indice x). Esiste un algoritmo PPT in grado di calcolare $f_s(x)$? Per calcolare $f_s(x)$ non è necessario calcolare tutto l'albero, ma basta seguire il percorso da s verso s_x. Per ogni livello generiamo massimo $2k$ bit per k bit. Il generatore BBS, per generare un bit fa un elevamento al quadrato. Se deve costruire $2k$ bit, fa $2k$ volte un elevamento al quadrato. Quindi per ognuno dei $2k$ bit deve fare un elevamento al quadrato, che corrisponde ad una moltiplicazione che costa k^2. Quindi la complessità totale è:

$$k \cdot 2k \cdot k^2 = \Theta(k^4)$$

Quindi l'algoritmo A PPT che calcola $f_s(x)$ esiste.

Vediamo ora il punto 3. Che tipo di esperimenti stiamo facendo implicitamente su G? Stiamo usando G su un insieme di semi che non sono casuali, ma nidificati. L'uso nidificato delle funzioni, tuttavia, risulta migliore rispetto all'uso sequenziale o su argomenti in relazione tra di loro.

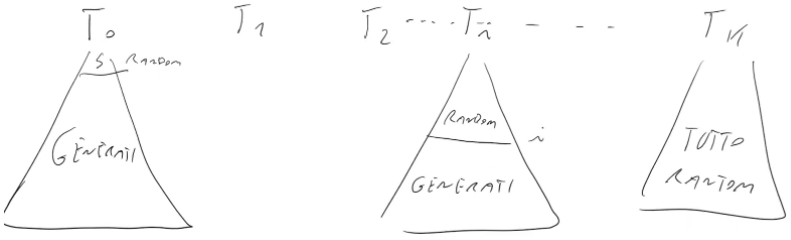
Ci troviamo ad un dato livello, e da quello stimo passando al successivo (dal livello 1 al livello 2, dal 2 al 3 e così via). Noi potremmo guardare questo albero per livelli. Un generatore pseudocasuale trasforma una certa quantità di casualità in una quantità di casualità diversa. Il nostro generatore lavora da un livello al successivo. Se riuscissimo a ricondurci ad un albero dove stiamo generando le funzioni pseudocasuali, ma in qualche modo ci concentriamo sul passaggio da un livello al successivo, allora siamo in un contesto dove stiamo lavorando con un generatore di bit pseudocasuali classico.

Per ogni livello $i = 0 \ldots k$ definiamo l'albero T_i tale che:

1. I nodi dei livelli 0 ... i sono tutti casuali;
2. I nodi dei livelli $i + 1$ ... k sono generati dai livelli precedenti.

Nel nostro albero la radice è casuale, mentre ogni altro nodo è stato generato ognuno a partire dal livello precedente.

Abbiamo T_0 dove s è random e il resto è pseudogenerato, T_k dove tutto è random, T_i dove fino al livello i è random e il resto è generato.



Definiamo H_i la funzione costruita da un albero T_i (vogliamo calcolare una funzione, per farlo costruiamo un albero T_i e abbiamo una funzione campionata da T_i). LO schema T_0 è uno schema dove la radice è casuale e tutto il resto è generato. Un albero costruito secondo questo schema è esattamente l'albero che abbiamo costruito nella funzione 5. Un albero T_k tutti i nodi sono veramente casuali, per calcolare f_x andiamo a guardare la foglia x che è scelta in maniera veramente casuale. Un albero costruito secondo lo schema T_k rappresenta una funzione campionata da U_k. L'albero T_0 rappresenta una funzione campionata da F_k.

Lo schema i-esimo è un modo per generare funzioni pseudocasulae dove, anziché usare come seme solo il livello 0, usiamo come semi tutti i livelli fino al livello i-esimo. Sono $k + 1$ modi diversi per generare una funzione pseudocasuale.

Definiamo come $P(i)$ la probabilità che D restituisca 1 quando interagisce con una funzione geenrata secondo lo schema T_i. Possiamo scegliere come generare una funzione pseudocasuale con ognuno di questi schemi. Scelto lo schema, possiamo generare una funzione che lo segue, darla in pasto a D e verificare con che probabilità ritorna 1. A questo punto è chiaro che:

$$P(0) = P_k^{D,G} \qquad\qquad P(k) = P_k^{D,U}$$

Supponiamo che il nostro distinguisher distingua, ovvero:

$$\left|P_k^{D,G} - P_k^{D,U}\right| > k^{-c} = |P(0) - P(k)| > k^{-c}$$

Possiamo lavorare ora a livello di interpolazione:

$$\exists i \ |P(i) - P(i+1)| > k^{-c'}$$

Abbiamo visto che gli schemi estremi sono quelli su cui abbiamo le ipotesi. Se abbiamo la capacità di distinguere due schemi estremi allora abbiamo anche la capacità di distinguere due schemi adiacenti. Se prendiamo gli

schemi T_i e T_{i+1}, nel primo caso il livello $i+1$ è generato dal livello i, nel secondo caso è causale. Quindi se confrontiamo i due schemi, l'unica differenza che troviamo sta nel livello $i+1$. Con il nostro G non facciamo altro che andare a fare dei test sui bi del livello $i+1$.

Quali sono i test che stiamo facendo su questo livello? Quando chiediamo $G(x)$, andiamo indietro e prendiamo un nodo specifico al livello $i+1$, non stiamo facendo altro che fare dei calcoli che ci portano a vedere x, non vediamo tutti i calcoli intermedi (il distinguisher D non vede nodi tra il fondo e x), anche se il generatore li genera. Delle cose che possiamo costruire a partire da un nodo del livello $i+1$ vediamo solo una parte.

Non è un problema vedere meno di quello che un distinguisher su un generatore G può vedere. Il punto è che se calcoliamo f_x su T_{i+1}, stiamo praticamente dicendo di generare dei bit casuali di partenza e di fare delle elaborazioni per ottenere x; se lo stiamo facendo sull'albero T_i stiamo dicendo di generare dei bit casuali, generare un seme, usare il seme per generare dei bit da usare per calcolare x. Il nostro esperimento sta lavorando, in un caso, su bit veramente casuale, nell'altro sta lavorando su un seme casuale usato per generare dei bit per arrivare a x.

E se chiediamo una y che fa riferimento allo stesso nodo? Questo si traduce nello stesso identico esperimento sul livello $i+1$, quindi non è un esperimento nuovo, abbiamo fatto due calcoli a partire dalla stessa osservazione fatta. E se lavoriamo su una z che va su un nodo diverso? In questo caso stiamo facendo un nuovo esperimento su G, dove stiamo lavorando con altri bit casuali o generato. I bit generati vengono però generati a partire da un altro seme che è indipendente dal precedente. I semi sono indipendenti; una volta che abbiamo definito gli argomenti $x_1, x_2, x_3, \ldots$, risalendo troviamo un insieme di nodi, ma tutto ciò che parte dai nodi e scende è un esperimento indipendente dagli altri. Il test che stiamo facendo su f è un test che stiamo facendo sul livello i-esimo per capire se è casuale o generato, ed è fatto nella forma corretta che era stata prevista quando abbiamo definito i generatori di bit pseudocasuali G.

Questo ci dimostra che un eventuale attaccante a un generatore di funzioni si traduce in un eventuale attaccante a un generatore di bit pseudocasuali, e di conseguenza un distinguisher non esiste se partiamo dalla definizione di G che abbiamo dato.

Se noi prendiamo un nodo del livello $i-1$ e prendiamo tutto il suo albero, tutto le interrogazioni che facciamo lì sotto, sono in realtà interrogazione su $2k$ bit generati a partire dal livello precedente. Per noi sono un esperimento unico, perché effettivamente l'esperimento che facciamo parte dal generatore e calcola tutto ciò che vuole. Noi vediamo il livello $i+1$ pseudogenerato o veramente casuale, e facciamo degli esperimenti mirati a calcolare della roba che viene sotto. Questo è ciò che fa il distinguisher, e questo distinguisher può fare l'esperimento più volte, ma sono tutti esperimenti indipendenti in quanto ciascun seme è generato in maniera indipendente (è come usare il distinguisher per il singolo bit più volte, a differenza degli altri casi dive utilizzavamo i distinguisher su semi che conoscevamo, oppure facevamo esperimenti che non si potevano fare in tempo polinomiale)

Siamo partiti dalla generazione di bit pseudocasuali. Da questa dopo aver trovato un sistema dimostrabilmente sicuro siamo passati alla generazione di funzioni pseudocasuali. Una funzione pseudocasuale è una scatola nera che contiene una funzione che possiamo interrogare con argomenti che ottenere risposte. La funzione è scelta casualmente fra tutte le funzioni tra k bit a k bit. In realtà la funzione non è scelta tra tutte le funzioni ma tra un insieme ristretto, in quanto dobbiamo denotare questa funzione con un insieme di k bit. L'indice scelto determina la funzione da usare.

Se i k bit vengono scelti a caso, la cardinalità dell'insieme in cui scegliamo le funzioni è 2^k. Questo generatore funziona bene nella misura in cui chi interagisce con la scatola nera non capisce se questa sta calcolando una funzione scelta dall'insieme grande o da quello ristretto.

Visto che la scatola nera la dobbiamo costruire, vogliamo che dato, indice e argomento, in tempo probabilistico polinomiale sia possibile calcolare $f_i(x)$. Abbiamo provato varie soluzioni, di cui abbiamo sempre trovato qualche difetto.

Autenticazione di messaggi 11

Vogliamo ora risolvere il problema dell'autenticazione. Uno dei motivi per cui abbiamo introdotto le funzioni pseudocasuali è per autenticare messaggi. Supponiamo di trovarci nel casi in cui due o più agenti debbano scambiarsi dei messaggi in rete e che non siano interessati alla segretezza dei messaggi. Vogliono però essere sicuri che i messaggi siano creati solo da loro e non da agenti esterni al gruppo.

Una delle possibili soluzioni consiste nello scegliere una funzione pseudocasuale f da condividere tra gli agenti (condividendo il seme tramite un canale sicuro). Tutti gli agenti condividono la funzione e sono in grado di calcolarla. Ad ogni messaggio viene quindi accodato il valore $f(m)$. Questa è l'autenticazione del messaggio. La funzione è pseudocasuale e quindi chiunque non conosca il seme vede questa sequenza come una sequenza casuale.

In quale senso è sicuro?

Definition 11.0.1 (Falsificatore (forger)) *Sia F un algoritmo PPT che interroga f su una sequenza di messaggi $m_1, m_2, \ldots, m_l$ anche in modo adattivo. Alla fine produce una coppia (m, σ). Diciamo che F ha successo se:*

$$\sigma = f(m) \ \wedge \ \forall i \ m \neq m_i \tag{11.1}$$

Il forger ha quindi successo se riesce a costruire l'autenticazione corretta di un nuovo messaggio diverso da tutti quelli che ha usato per interrogare f.

Vogliamo ovviamente che la probabilità del forger di avere successo sia più piccola di ogni polinomio:

$$\forall F \in PPT \ Pr[F\_ha\_successo] < k^{-w(1)}$$

Sappiamo che questa probabilità è più piccola di ogni polinomio quando la funzione è effettivamente casuale, in quanto l'autenticazione di un messaggio diverso dagli altri osservati è scelta casualmente tra le sequenze di k bit. Di conseguenza indovinerà con probabilità $\frac{1}{2^k}$. Se la sequenza è pseudocasuale e indovinasse con una probabilità significativamente diversa, se noi costruiamo il distinguisher che applica F e ritorna 1 se ha successo, allora:

- La probabilità che ritorni 1 davanti ad una sequenza veramente casuale è $\frac{1}{2^k}$;
- La probabilità che ritorni 1 davanti ad una sequenza pseudocasuale è ampiamente alta.

Questo è un distinguisher. Di conseguenza sappiamo che se la funzione è pseudocasuale il forger non esiste, in quanto potrebbe essere usato per creare un distinguisher. Sappiamo che usare una funzione pseudocasuale per creare un'autenticazione dei messaggi funziona.

Abbiamo trovato un modo per autenticare i messaggi, ma i messaggi che stiamo autenticando sono di k bit. Se qualcuno dovesse autenticare messaggi più lunghi come dovrebbe fare?

11.1 Autenticazione di messaggi di lunghezza arbitraria

Potremmo prendere il messaggio m e scomporlo in blocchi m_i di k bit. A questo punto possiamo cercare di costruire un'autenticazione di m applicando f ad ogni m_i e combinando i risultati. Definiamo quindi $f(m)$ come:

$$f(m) = f(m_1) \oplus ... \oplus f(m_l)$$

Questa soluzione funziona? Lo XOR è un'operazione commutativa, di conseguenza un attaccante potrebbe invertire l'ordine dei blocchi mantenendo il nuovo messaggi comunque autenticato. Riusciamo a dimostrare allora che questa soluzione permette di costruire un forger? Il forger è semplice da costruire: prende un messaggio composto da almeno 2 blocchi, lo autentica, prende una permutazione del messaggio diversa dall'identità e usa la stessa autenticazione. Riesce quindi a costruire l'autenticazione corretta per un messaggio diverso da quelli osservati.

Esempio: Se noi chiediamo $f(m_1, m_2)$, la coppia $(m_2\ m_1, f(m_1, m_2))$ è autentica.

Siamo anche in grado, in questi schema, di creare un messaggio autenticato senza neanche interrogare f: se il numero dei blocchi è pari e sono tutti uguali, il loro XOR è 0[1].

1: Infatti $f(m_1\ m_1) = 0$

Vogliamo vietare lo scambio dei blocchi, facendo in modo che la posizione dei blocchi impatti su f. Proviamo a definire ore $f(m)$ come:

$$f(m) = f(< 1 > m_1) \oplus ... \oplus f(< l > m_l)$$

Dove il numero tra parentesi angolari indica la sequenza di bit necessari a codificare la posizione del blocco. I blocchi non saranno più di k bit, ma di $k - bit\_posizione$. Se invertissimo m_1 con m_2, l'autenticazione cambierebbe perché calcoleremmo $f(< 1 > m_2)$. Non funziona più neanche la creazione del messaggio autenticato senza interrogare f, in quanto anche se i blocchi fossero uguali, i bit di posizione associati sono diversi.

Questo sistema funziona? Anche in questo sistema esiste un forger. Supponiamo di voler autenticare il messaggio $m_1\ m_2$, allora:

$$f(m_1\ m_2) = f(< 1 > m_1) \oplus f(< 2 > m_2)$$

Supponiamo ora di voler autenticare i messaggi $m_1' \; m_2$ e $m_1 \; m_2'$:

$$f(m_1' \; m_2) = f(<1> m_1') \oplus f(<2> m_2)$$
$$f(m_1 \; m_2') = f(<1> m_1) \oplus f(<2> m_2')$$

Ci siamo fatti autenticare tre messaggi di 2 blocchi, dove poi abbiamo cambiato prima il blocco di sinistra e poi quello di destra. Se calcoliamo lo XOR delle tre autenticazioni otteniamo:

$$\begin{array}{rl} & f(m_1 \; m_2) = f(<1> m_1) \oplus f(<2> m_2) \\ & f(m_1' \; m_2) = f(<1> m_1') \oplus f(<2> m_2) \\ \oplus & f(m_1 \; m_2') = f(<1> m_1) \oplus f(<2> m_2') \\ \hline & f(m_1' \; m_2') = f(<1> m_1') \oplus f(<2> m_2') \end{array}$$

Siamo riusciti a costruire l'autenticazione di un messaggio mai autenticato prima e quindi abbiamo creato un forger. Per creare un messaggio nuovo, abbiamo provato a modificare il messaggio originale un blocco alla volta.

Il problema di questa soluzione sta nello XOR usato in maniera piatta. Quando usiamo lo XOR allo stesso livello più volte, è un'operazione associativa, commutativa e che lo XOR tra due oggetti uguali è 0.

Se noi però applicassimo lo XOR in maniera nidificata, allora la storia sarebbe diversa. Definiamo quindi:

$$f(m_1 \ldots m_l) = f(m_l \oplus \ldots \oplus f(m_3 \oplus f(m_2 \oplus f(m_1))))$$

Questo è un sistema di autenticazione (è ben definito, dato un messaggio, come si fa il calcolo). Ma è sicuro? Si, è dimostrabilmente sicuro.

Alla fine abbiamo la nostra funzione pseudocasuale f da cui costruiamo una funzione f' che prende una qualunque sequenza di bit e produce una funzione:

$$f' : \{0,1\}^{kl} \to \{0,1\}^k$$

Quindi moltiplichiamo la dimensione dell'input. Non è più una lunghezza di k bit, ma una lunghezza arbitraria purché multipla di k. Potremmo vedere questo oggetto come una estensione della nostra idea di funzione pseudocasuale. Supponiamo di avere una funzione scelta casualmente tra le funzioni da $\{0,1\}^{kl}$ a $\{0,1\}^k$, oppure una funzione da kl bit a k bit scelta fra un insieme più ristretto (perché dipende dallo stesso seme usato per f).

Vogliamo dimostrare che questo sistema è sicure nel senso che non esiste nessun algoritmo PPT che è in grado di distinguere una funzione casuale da una pseudocasuale. Poiché il forger può essere trasformato in distinguisher, dire che questo è un sistema di autenticazione o una funzione pseudocasuale da kl bit a k bit è la stessa cosa. L'esistenza di un forger implica l'esistenza di un distinguisher.

Vogliamo mostrare che la nuova funzione che abbiamo ottenuto è una funzione pseudocasuale secondo la definizione che abbiamo dato. Costruiamo una F_i tale che:

$$F_i(m_1 \dots m_l) = R(m_l \oplus \dots \oplus R(m_{i+1} \oplus f(m_i \oplus \dots \oplus f(m_2 \oplus f(m_1)))))$$

Per i primi i livelli applichiamo la f vera e propria, mentre per i livelli successivi applichiamo una funzione casuale R. A questo punto sappiamo che che $F(0)$ è il caso in cui usiamo solo la funzione R, mentre $F(l)$ è il caso in cui usiamo solo al funzione pseudocasuale f. Possiamo quindi costruire tutti i livelli intermedi e sappiamo che esiste un i tale per cui:

$$|P(i) - P(i-1)| > k^{-c}$$

Riusciamo quindi a creare un distinguisher che, sul livello i-esimo, è in grado di riconoscere l'uso di una funziona casuale da una pseudocasuale. Di conseguenza abbiamo un distinguisher per f.

Nidifichiamo f, ovvero la applichiamo di volta in volta, e contiamo quante volte la applichiamo. Arrivati ad applicarla i volte (o $i-1$ o $i+1$), il distinguisher se ne accorge, ovvero quello è il punto in cui si vede la differenza tra l'uso della funzione casuale e pseudocasuale. Come facciamo a fare questo esperimento? La f pseudocasuale la conosciamo quando applichiamo gli esperimenti, la applichiamo per i primi i livelli e per il livello $i+1$ o diamo una sequenza di bit pseudocasuali o continuiamo ad usare f. A questo punto abbiamo generato un esperimento compatibile con quello del forger.

11.2 Firma digitale

Abbiamo trovato un modo per autenticare messaggi arbitrari. Se Alice e Bob si scambino messaggi e se li autenticano tra di loro, allora hanno la garanzia che nessuno di esterno possa interferire su quello che si stanno dicendo. Ma se Alice e Bob non si fidano gli uno degli altri? Se Alice fa delle promesse a Bob e poi non le mantiene, come può Bob dimostrare che effettivamente Alice ha fatto queste promesse? Supponendo che i messaggi siano autenticati, sorgono due problemi:

- Come può la terza parte sapere che i messaggi sono veramente autenticati? Per verificarlo è necessario sapere il seme di f. Conoscendo il seme è però possibile sempre generare nuovi messaggi autentici, quindi che è messo nelle condizioni di verificare l'autenticità e di conseguenza messo nella condizione di poter creare nuovi messaggi autentici;
- Anche conoscendo il seme, per verificare l'autenticità del messaggio, non è possibile provare chi ha originato il messaggio (il messaggio "io Alice pagherò Bob" può essere scritto e autenticato da Bob).

Questo è il problema della **non repudiation**: noi vogliamo un meccanismo che non permetta di ripudiare un messaggio. La semplice autenticazione di messaggi che abbiamo visto fino ad ora non è sufficiente. Questo problema è legato al concetto di firma digitale. L'idea è di autenticare

il messaggio garantendo la non ripudiabilità. Innanzitutto non deve essere possibile per chi è in grado di verificare la firmo di poterla rifare in qualche modo. Solamente l'agente che firma deve essere in grado di firmare.

Definition 11.2.1 (Concetto di firma digitale) *Solo uno (o pochi) agente/i può firmare; tutti possono verificare la firma. Di conseguenza chi può verificare non può firmare.*

Devono esistere due chiavi (semi): una privata per firmare e una pubblica per verificare. Dalla chiave pubblica non deve essere possibile ricavare la chiave privata. Un algoritmo per fare questo è RSA.

Con RSA per firmare si usa la chiave privata (n, d) per firmare e la chiave pubblica (n, e) per verificare. La firma di un messaggio m diventa (m, m^d). La verifica di un messaggio firmato (m, σ) è $\sigma^e == m$.

Per definire la correttezza per un protocollo di firma ci appoggiamo alla definizione di forger:

Definition 11.2.2 (Sistema di firma digitale) *Un sistema di firma digitale è una tripla di algoritmo PPT:*

1. *Generazione chiavi* $G : 1^k \rightarrow (P_k, S_k)$*;*
2. *Sign* $S : S_k, m \rightarrow (\sigma)$*;*
3. *Verify* $V : P_k, m, \sigma \rightarrow \{0, 1\}$ *(accetta, non accetta);*

Questi algoritmo devono soddisfare una serie di proprietà:

- Se la firma è costruita correttamente secondo l'algoritmo di firma, allora il sistema deve dire 1 con probabilità 1[2]:

$$\forall m \; Pr[V(P_k, m, S(S_k, m)) = 1] = 1$$

- Sia il forger $F \in PPT$ un algoritmo che chiede la verifica di messaggi $m_1, m_2, \ldots, m_l$, anche in modo adattivo, e che produce una coppia (m, σ). Allora F ha successo se l'algoritmo di verifica dive "accetta" e il messaggio firmato è differente da tutti i messaggi precedenti:

$$V(P_k, m, \sigma) = 1 \wedge \forall i \; m \neq m_i$$

  Allora la probabilità che F abbia successo è minore di ogni polinomio:

$$\forall F \in PPT \; Pr[F\_ha\_successo] < k^{-c}$$

2: Diciamo "deve dire 1 con probabilità 1", e non "deve dire sempre si", in quanto l'algoritmo di verifica potrebbe essere un algoritmo probabilistico. Di conseguenza potrebbe sbagliare, ma vogliamo che sbagli con probabilità 0, quindi mai.

La definizione soddisfa le proprietà che abbiamo dato? Ad F diamo in pasto solo la chiave pubblica. Di conseguenza se non è in grado di generare una firma falsa è evidente che la chiave pubblica non può produrre firme. Di conseguenza un sistema che soddisfa questa definizione è un sistema che soddisfa anche la proprietà di non ripudiabilità.

RSA soddisfa questa definizione? No, non è vero che ogni forger ha una probabilità trascurabile. Supponiamo di avere due messaggi e la loro firma (m_1, m_1^d) e (m_2, m_2^d). Se prendiamo il messaggio $m_1 \cdot m_2$, la sua firma sarà $(m_1 \cdot m_2, m_1^d \cdot m_2^d)$. Chiunque è in grado di creare quest'ultima

firma conoscendo la firma di m_1 e m_2. Se il messaggio è m^e, la sua firma sarà (m^e, m). Abbiamo ottenuto due messaggi firmati mai visti prima. Nel secondo caso, soprattutto, è possibile ottenere un messaggio cifrato senza interrogare l'agente che conosce la chiave privata.

Si potrebbe obiettare che m^e e il prodotto di due messaggi sono messaggi senza senso; è vero, ma la definizione di forger accetta la generazione di un qualsiasi messaggio, non per forza sensato (la difficoltà resta nel firmare un messaggio dato).

RSA non è corretto secondo la definizione data sopra.

11.2.1 Firmare messaggi lunghi

Abbiamo visto che RSA non è resistente ad attacchi di forgery. Ora vogliamo vedere come cifrare messaggi lunghi (es: DVD). Prendiamo l'intero messaggio e lo codifichiamo con Cypher Block Chaining, usando il risultato come firma? Così finiremmo per ottenere una firma lunga quanto il messaggio. Non è ragionevole come approccio (es: mi servirebbe un secondo DVD per contenere la firma).

Per fare questo usiamo le funzione hash crittograficamente sicure. Infatti noi andiamo a firmare con RSA l'hash del messaggio, quindi:

$$\sigma = H(m)^d$$

Firmando l'hash del messaggio otteniamo una firma lunga k e non più lunga quanto il messaggio (oltre al fatto che firmare l'hash è computazionalmente meno costoso).

Cosa succede se $H(m_1) = H(m_2)$, ovvero abbiamo collisione? Sarebbe possibile costruire un forger, in quanto chiedendo la firma σ di m_1, possiamo usarla per produrre (m_2, σ), e quindi firmare un messaggio mai visto. Se ad esempio l'hash di un messaggio sono i suoi primi/ultimi k bit, è semplice trovare altri messaggi che producono lo stesso hash. Inoltre, in questo caso, ottenuta la firma del primo messaggio, abbiamo ottenuto la firma anche per tutti gli altri messaggi che iniziano/finiscono con gli stessi bit.

Le funzioni hash, per natura intrinseca, hanno collisione. Quindi coppie di messaggi con lo stesso hash possono essere usate da un forger per attaccare. Per questo è necessario usare funzioni hash collision resistant, ovvero dove sia computazionalmente difficile trovare due messaggi con lo stesso hash.

Definition 11.2.3 (Funzione collision resistant) *Funzione tale per cui per ogni algoritmo PPT la probabilità che questo riesca a restituire una collisione sia più piccola di ogni polinomio.*

Alcune proposte di funzioni collision resistant sono MD4 (Message Digest -> prendo un messaggio lungo e lo digerisco per produrre un messaggio più corto), MD5, MD6, SHA1 (Secure Hash), SHA2.

SHA2 è ad oggi la funzione che sembra più inattaccabile. Come facciamo a far vedere che uno di questi modi non è più sicuro? Basta mostrare una

collisione. Se partiamo dall'idea che trovare collisione sia difficile, vuol dire che nessuno è in grado di trovarla.

Se all'interno di RSA usiamo funzioni collision resistant allora la firma dell'hash di un messaggio diventa sicura. Ad esempio non siamo più in grado di produrre la firma di $m_1 \cdot m_2$ a partire dalle firme di m_1 e m_2. L'hash di $m_1 \cdot m_2$ non è più il prodotto dell'hash di m_1 e dell'hash di m_2, è qualcosa di diverso, e quindi non siamo più in grado di produrre le firme. Non è neanche più valida la firma (m^e, m), in quanto adesso diventa $(m^e, H(m^e)^d)$.

L'uso delle funzioni hash collision resistand risolve sia il problema della firma dei messaggi lunghi sia il problema del forgery attack su RSA puro.

Ad oggi l'unica funzione che viene considerata sicura è SHA2.

Definition 11.2.4 (Funzioni hash one way) *Una funzione hash è one way se dato x è difficile trovare un qualsiasi m tale che $H(m) = x$.*

Se H è collision resistant allora H è one way.

Dimostrazione. Supponiamo che H non sia one way. Dimostriamo che H non è collision resistant[3].

Prendiamo un messaggio m casuale. Calcoliamo $x = H(m)$ e diamo in pasto x alla macchina H^{-1}, che produrrà un messaggio m', tale per cui $H(m') = x$. Qual è la probabilità che $m = m'$?

Ci sono infiniti messaggi che vengono mappati su x, e noi ne abbiamo scelto uno a caso uniformemente tra tutti. H^{-1} ne sceglie un altro. Qual è la probabilità che quello scelto sia quello che abbiamo scelto noi? La probabilità è:

$$Pr = \frac{1}{cardinalita'\_dei\_messaggi}$$

Sostanzialmente poiché il messaggio m lo abbiamo scelto a caso, la probabilità che H^{-1} ritorni il messaggio che abbiamo scelto è praticamente 0. Quindi con probabilità sostanzialmente 1 abbiamo trovato una collisione. Se H non è collision resistant abbiamo un algoritmo molto semplice che produce una collisione.

Quindi avere una funzione collision resistant è più forte di avere una funzione hash one way. □

3: Poichè $(A \rightarrow B) \leftrightarrow (\neg B \rightarrow \neg A)$. Quindi dimostriamo $\neg B \rightarrow \neg A$ per dimostrare la prima parte.

11.3 Blind signature

Come possiamo firmare un messaggio di cui non conosciamo il contenuto? Supponiamo che qualcuno debba mandarci il pin dello nostra carta, e che questo ci venga spedito in una lettera firmata dal direttore della banca. Nella lettera il pin viene coperto da un adesivo, in modo che chi firma non possa vedere il pin. Questo è un esempio di un messaggio

che si vuole firmare senza far conoscere il contenuto. Come si può fare questo in formato elettronico?

In un messaggio finora abbiamo preso H del messaggio e lo abbiamo firmato. Ma come facciamo a calcolare l'hash se non possiamo conoscere il contenuto del messaggio?

Abbiamo un messaggio che non conosciamo e che non possiamo conoscere e lo dobbiamo firmare. Partiamo dall'idea che ci fidiamo della persona che di passa il messaggio. Per fare questo introduciamo il concetto di **blind signature**.

Definition 11.3.1 (Blind signature) *Esiste un messaggio m che l'agente B deve firmare. B non può conoscere m.*

L'agente A che ha bisogno della firma procede così:

- Sceglie $b \in_R Z_n^*$ (**blinding factor**);
- Calcola mb^e e ne chiede la firma.

L'agente B:

- Calcola la firma $\sigma = (mb^e)^d$ (usiamo uno schema di firma senza funzioni hash; useremo la struttura dei messaggi o la ridondanza dei messaggi per evitare gli attacchi visti nella firma con RSA senza hash - è fondamentale che i messaggi abbiano una struttura);
- Comunica σ ad A.

L'agente A:

- Calcola $\sigma' = \frac{\sigma}{b}$, ovvero σb^{-1};
- Produce (m, σ').

Verifichiamo se (m, σ') è una firma valida.

Dimostrazione.

$$\sigma' = \sigma \cdot b^{-1} = (mb^e)^d b^{-1} = m^d b^{ed} b^{-1} = m^d \cdot b \cdot b^{-1} = m^d$$

□

Quindi la firma prodotta da A (m, σ') a partire dalla firma di b è una firma valida per m. Durante il processo di firma, B non ha modo di conoscere m, in quanto moltiplicare il messaggio per un oggetto casuale di Z_n^* trasforma il messaggio originale in un elemento casuale di Z_n^*.

Vogliamo ora provare a migliorare questo sistema affinché B, anche se continua a non avere conoscenza del messaggio, abbia la garanzia che il messaggio sia costruito in maniera adeguata a quello che doveva comunicare (es: se lo scopo del messaggio è comunicare un pin, deve effettivamente contenere quel pin).

Vogliamo quindi che B abbia la garanzia che il messaggio sia ben formato. Ci sono più modi per farlo:

1. La chiave pubblica usata da B può essere usata solo per spedire quel tipo di messaggio (es: pin);
2. Complichiamo leggermente il protocollo.

Vediamo come funziona la soluzione 2.

A:

- Genera $m_1, ..., m_l$ differenti messaggi di comunicazione pin. I messaggi devono essere tutti identici, tranne per il pin;
- Genera b_1 ... b_l blinding factors;
- Spedisce $m_1 b_1^e, ..., m_l b_l^e$.

B:

- Sceglie $i \in_R \{1, ..., l\}$ (sceglie una lettera);
- Comunica i da A.

A:

- Comunica $m_1, b_1, ..., m_{i-1}, b_{i-1}, , m_{i+1}, b_{i+1}, m_l, b_l$. Comunica tutto tranne m_i, b_i.

B:

- Verifica la correttezza dei dati ricevuti (verifica che $m_1 \cdot b_1^e$ è il primo oggetto che ha ricevuto e così via);
- Se la verifica ha successo per tutti i messaggi, allora produce $(m_i b_i^e)^d$, ovvero la firma.

La probabilità che B firmi il messaggio anomalo, supponendo che ve ne sia uno nella lista di messaggi spediti da A, è $\frac{1}{l}$. Se A invia più di un messaggio anomalo, B se ne accorge subito.

Protocolli vari 12

12.1 Bit commitment

Supponiamo di trovarci nel caso un cui un agente conosca l'andamento della borsa nel futuro e inserisca questa informazione in una busta chiusa, che verrà aperta tra un anno. Vogliamo tentare di costruire il concetto di busta chiusa con i bit. Supponiamo di lavorare con singoli bit in quanto, se siamo in grado di "imbustare" un bit, possiamo farlo anche con sequenze di bit (es: concateniamo buste chiuse nell'ordine corretto).

Chiunque abbia davanti a se un bit in busta chiusa deve poter indovinare quel bit con probabilità $\frac{1}{2}$. Definiamo ora i requisiti che il protocollo deve soddisfare.

Creazione della busta: A invia un messaggio m a B tale per cui:

- m deve contenere un bit b fissato;
- A non deve essere in grado di cambiare b;
- B non deve essere in grado di indovinare b[1].

1: Se noi costruiamo un algoritmo dove: Alice sceglie il bit b, genera il messaggio m, lo invia a Bob e Bob applica un qualche algoritmo per produrre un nuovo bit, la probabilità che questo sia uguale a b differisce da $\frac{1}{2}$ di una quantità più piccola di ogni polinomio

Apertura della busta: A invia un messaggio m' a B tale per cui:

- B riesce a ricavare il bit b da m e m';
- A non deve riuscire a creare due coppie (m, m') (m, m'') che portano B ad ottenere due risultati diversi.

12.1.1 Protocollo 1

Vediamo ora una prima implementazione del protocollo.

1. B invia ad A un numero R random (**nonce**[2]);
2. A invia a B $E_k(Rb)$, ovvero la codifica con chiave k della sequenza Rb, dove k è una chiave casuale e E è un algoritmo di crittografia simmetrica;
3. B conosce ora il cyphertext, eccetto per un bit. Tuttavia DES e AES non sono vulnerabili al known plaintext attack. In realtà non abbiamo usa dimostrazione che effettivamente non siano vulnerabili, ma tutte le analisi fatte fino ad ora ci portano a dire che non è possibile ricavare la chiave anche se conosciamo il plaintext di un dato cyphertext. Di conseguenza B non ha modo di ricavare b, tranne provare tutte le possibili chiavi (attacco a forza bruta);
4. A invia k a B;
5. B decodifica il messaggio e verifica che il contenuto del messaggio eccetto l'ultimo bit sia identico a R.

2: Usato per autenticare l'agente. Il nonce è un number once, ovvero un numero mai usato in precedenza. Il modo più semplice per generarlo è generare una sequenza di bit a caso: se questa è sufficientemente lunga, la probabilità che esistano due agenti che abbiano usato lo stesso numero è più piccola di ogni polinomio.

In questa implementazione A non può imbrogliare in quanto dovrebbe trovare due chiavi k_0, k_1 tali per cui:

$$E_{k0}(R, 0) = E_{k1}(R, 1)$$

cioè dovrebbe trovare due chiavi tali per cui una permetta di aprire la busto rivelando il bit 0, mentre la seconda permetta di aprire la busta rivelando il bit 1 (su AES non siamo in grado di farlo).

Questa soluzione presenta però due problemi:

- Abbiamo detto che "non conosciamo altro metodo per" ricavare la chiave senza provarle tutte e per generare due chiave tali per cui una rivela 0 e l'altra 1. Abbiamo sempre detto che non è buona cosa usare un protocollo solo perché non siamo in grado di attaccarlo, quindi sarebbe meglio trovare qualcosa di dimostrabilmente sicuro;
- A per creare la busta chiusa necessita del supporto di B, che deve generare R[3]. Se B non invia R, A non può inviare il bit.

3: senza R il messaggio diventerebbe arbitrario - contiene solo il bit scelto da A - e quindi creare buste con valori diversi sarebbe banale, in quanto metà delle chiavi decodificherebbero un cyphertext come 0 e l'altra 1

12.1.2 Protocollo 2

Sia H una funzione hash one way. Allora il protocollo è implementato come segue:

- A invia a B $H(R_1\ R_2\ b), R_1$. $R_1 0$ e R_2 sono sequenze di bit casuali;
- A invia a B R_2, b per aprire la busta.

Con il primo messaggio conosce l'hash del messaggio più parte del suo contenuto. L'unico modo che ha ora per ricavare il messaggio è invertire la funzione H. Se non fosse presente R_2, a B basterebbe calcolare $H(R_1\ 0)$ e $H(R_1\ 1)$ per scoprire il valore del bit b.

Quando riceve il secondo messaggio, B può calcolarne l'hash e verificare che A non stia mentendo.

A non è in grado di creare una busta che riveli due bit in quanto dovrebbe creare una situazione in cui:

$$H(R_1\ R_2\ 0) = H(R_1\ R_2'\ 1)$$

Questo equivale a trovare conflitti nella funzione H, cosa che non siamo in grado di fare.

Siamo riusciti a creare un protocollo dove non è necessario che B partecipi alla creazione della busta. Sia in questo protocollo che nel precedente possiamo sostituire il bit con una sequenza di bit. Anche questo protocollo non è dimostrabilmente sicuro.

12.1.3 Protocollo 3

Vediamo ora un protocollo dimostrabilmente sicuri che però richiede la collaborazione di B. Sarà molto complesso (a livello di complessità) e richiederà al sua applicazione ad ogni bit che si vuole "imbustare". Vediamo come funziona:

- B invia ad A R, dove $R = r_1 \dots r_l$;
- A genera un seme s e una pseudostringa di bit $X = x_1 \dots x_l$. Genera poi:

$$\forall i \; y_i = \begin{cases} x_i & \text{se } r_i = 0 \\ x_i \oplus b & \text{se } r_i = 1 \end{cases}$$

  Invia a B $y_1 \dots y_l$. La sequenza Y è una sequenza di bit generati o in modo pseudocasuale o messi in XOR col bit b. Nel momento in cui una quantità casuale viene messa in XOR con qualcosa, il risultato è casuale. Quindi se gli x_i sono veramente casuali, allora la sequenza degli y_i è anch'essa casuale. Ciò significherebbe che non contiene alcuna informazione circa il bit b.
  La sequenza X è pseudocasuale. Se B fosse in grado di ricavare dalla sequenza Y informazioni circa il bit b, allora l'algoritmo per ricavare b sarebbe un distinguisher per il generatore di bit pseudocasuali.
  Il messaggio che A invia non rivela quindi il bit b;
- A invia a B il seme s. B diventa in grado di costruire ora la sequenza X e verificare che tutti gli y_i siano stati costruiti correttamente.

Se tutti gli r_i dovessero valere 0, allora in nessuno degli y_i sarebbe presenta l'informazione riguardo il bit b. Ma la probabilità che questo avvenga è $\frac{1}{2^l}$, trascurabile.

A non può creare due buste in dovrebbe trovare due semi s_0 e s_1, tali per cui il primo porta alla stringa $x_1 \dots x_l$ e il secondo alla stringa $x'_1 \dots x'_l$ e :

$$\forall i \text{ se } r_i = 1 \text{ allora } x_i = \overline{x'_i}$$

ovvero:

$$0 \oplus x_i = 1 \oplus x'_i$$

Questa cosa è dimostrabile, anche se non lo facciamo.

12.2 Schemi a barriera

Supponiamo di conoscere un segreto (es: chiave che sblocca gli armamenti nucleari) s, che è una sequenza di bit. Lo distribuiamo a n agenti creando delle parti (**share**) $s_1 \dots s_n$. Nessuno degli agenti possiede il segreto completo, ma solo un pezzo, in modo che mettendo assieme tutte le parti si ricava S. Mettendo assieme meno di n parti, non si conosce niente di S (non se ne conosce un singolo bit).

Per farlo usiamo la funzione XOR. Siano s_1 ... s_{n-1} sequenze casuali e sia:

$$s_n = s \oplus s_1 ... \oplus s_{n-1}$$

Se s_1 ... s_{n-1} sono sequenze casuali allora anche il loro XOR è casuale. Lo XOR tra s e $s_1 \oplus ... \oplus s_{n-1}$ è anch'esso casuale. Per come abbiamo costruito s_n, vale che:

$$s_1 ... \oplus s_n = s$$

Abbiamo trovato un modo per per ripartire un segreto in n parti tale per cui se mettiamo assieme le n parti otteniamo il segreto, mentre se ne mettiamo assieme meno di n parti non otteniamo il segreto.

Questo schema ha un difetto: se vogliamo impedire di usare l'arsenale nucleare, è sufficiente che ci facciamo inserire tra gli n agenti che condividono parte del segreto. Oppure uno degli agenti è stato eliminato.

12.2.1 Schema di Shamir

Vogliamo costruire uno schema a barriera (n, k) che permetta di costruire il segreto s con k parti, mentre con meno di k parti non permetta di conoscere nulla di s. Per farlo usiamo lo schema di Shamir, che si basa sul seguente teorema.

Theorem 12.2.1 *Un polinomio di grado $k-1$ è individuato univocamente da k punti (per k punti passa esattamente un polinomio di grado $k-1$).*

Costruiamo un polinomio P casuale di grado k tale che $P[0] = s$:

$$P[x] = a_0 + a_1 x + a_2 x^2 + ... + a_{k-1} x^{k-1} \qquad \text{(polinomio generico)}$$
$$P[0] = a_0$$

Per costruire un polinomio casuale di grado k basta scegliere a_1 ... a_{k-1} a caso (ponendo $a_0 = s$). Ad ogni agente i comunico $P[i]$. Se k agenti mettono insieme il loro segreto, conosciamo il valore del polinomio in k punti.

12.3 Problema dei crittografi mangiatori

Ci sono tre crittografi che hanno mangiato. Forse uno di loro ha pagato. Tutti vogliono sapere se a pagare sia stato uno di loro. Chi ha pagato, se esiste, non vuole rivelare la propria identità.

I tre crittografi devono interagire tra loro con un qualche protocollo al termine del quale calcolano una funzione. Questa ad esempio può essere:

- Ogni crittografo dispone di un bit segreto;
- Tutti i crittografi tranne al più uno hanno il bit a 0;

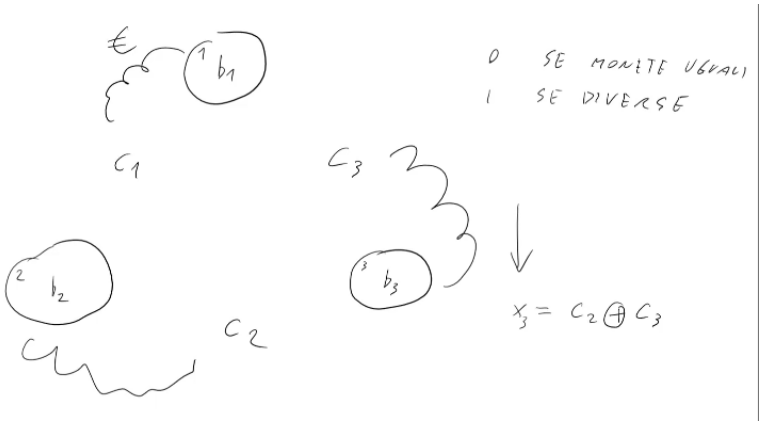


Figura 12.1: Schema grafico del protocollo

- Un crittografo potrebbe avere il bit a 1;
- I crittografi calcolano l'OR sui dati segreti.

I crittografi stanno calcolando sui loro dati segreti, ma la vogliono calcolare in modo tale che alla fine ognuno conosca il valore della funzione, ognuno conosca l'OR dei bit, ma nessuno sappia quanto valgono i bit di qualunque degli altri crittografi, se non ciò che si può ricavare dal risultato finale.

Vediamo un protocollo che i crittografi possono seguire (12.1).

1. Abbiamo tre crittografi 1, 2, 3. Di conseguenza abbiamo b_1, b_2, b_3;
2. Ogni crittografo lancia una moneta (c_1, c_2, c_2) alla propria destra. Ogni moneta è visibile solo dal crittografo che l'ha lanciata e da quello alla sua destra;
3. Ogni crittografo vede due monete: quella che ha lanciato e quella alla sua sinistra;
4. Ogni crittografo calcola x dato dallo XOR delle monete che vede (es: $x_3 = c_2 \oplus c_3$);
5. Ogni crittografo ottiene 0 se le monete sono uguali o 1 se sono diverse;
6. Il crittografo che ha pagato complementa il proprio x;
7. Ogni crittografo comunica il proprio risultato: solo il crittografo che ha pagato comunicherà l'opposto del risultato che ha ottenuto;
8. Contando gli 0 e gli 1 i crittografi sono in grado di dire se uno di loro ha pagato o no.

Come fanno i crittografi a dire se uno di loro ha pagato contando gli 0 e gli 1? Sappiamo che in condizioni normali (nessuno ha pagato) vale che

$$x_1 \oplus x_2 \oplus x_3 = 0$$

Se qualcuno paga allora il calcolo diventa:

$$x_1 \oplus \bar{x_2} \oplus x_3 \equiv x_1 \oplus x_2 \oplus x_3 \oplus 1$$

In quanto il complemento di un bit equivale a quel bit XOR 1. Di conseguenza lo XOR dei tre bit annunciati dai crittografi ritorna 1 al posto di 0.

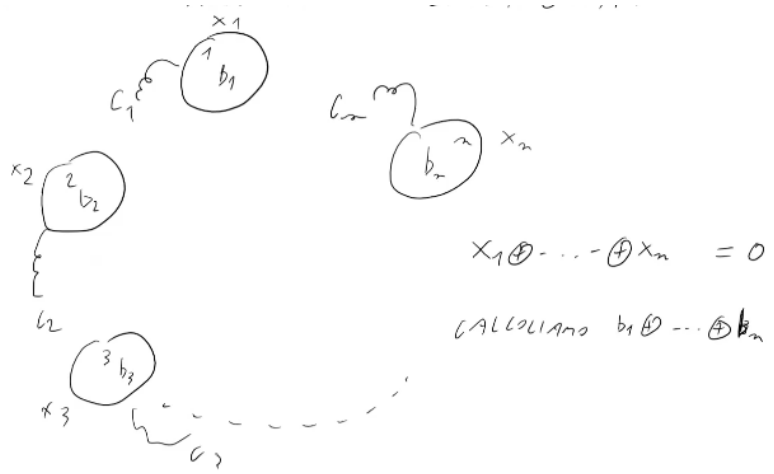


Figura 12.2: Schema grafico del protocollo

Possiamo vederla in un altro modo. Contiamo quanto sono i bit a 1 annunciati dai crittografi. Questi possono essere 0 o 2: sono 0 quando tutti i bit hanno lo stesso valore ($c_1 = c_2 = c_3$), sono 2 quando due valori sono uguali e uno è diverso (se $c_1 = c_3$, b_1 dirà 0, b_2 e b_3 diranno 1). Se un crittografo complementa il proprio bit, allora questi diventano o 1 o 3 (equivale a dire "lo XOR dà 1").

Con questa funzioni tutti i crittografi sono in grado di calcolare la funzione. Se i crittografi che mentono sono due, allora questa funzione non va bene. Lavora bene solo se a pagare è solo un crittografo.

Supponiamo che il risultato che otteniamo è 0. Allora ogni crittografo sa che nessuno dei tre ha pagato. Ogni crittografo sa tutto degli altri crittografi. Non lo sa perché il protocollo gli ha rivelato informazioni aggiuntive, ma sa esattamente quello che può sapere conoscendo il proprio bit e il risultato della funzione. In questo caso, essendo il risultato 0, conosce di conseguenza il valore del bit degli altri crittografi.

Se il risultato della funzione è 1, il crittografo che ha pagato sa tutto anche degli altri crittografi, mentre gli altri sanno solo che qualcuno tra loro ha pagato, ma non chi.

Il protocollo è generalizzabile a n crittografi (12.2). Come sopra $x_1 \oplus \ldots \oplus x_n = 0$, poiché ogni moneta viene conteggiata due volte. Tuttavia se un crittografo complementa il proprio bit, che equivale ad aggiungere un $\oplus 1$, il risultato diventa 1. Questo sistema funziona solo se il numero di bit a 1 è dispari (se è pari il loro XOR dà 0).

A cosa ci può servire un protocollo del genere? Incarichiamo un solo crittografo di spedire un bit agli altri agenti. Il crittografo si comporta come il crittografo che ha pagato nel caso in cui il bit da spedire sia 1. Gli altri crittografi osservano il bit trasmesso, ma non sanno il suo valore.

Abbiamo creato una situazione in cui un crittografo può inviare un bit senza rivelare la propria identità.

Controllo del canale di comunicazione Cambiamo contesto. Ora non c'è più un agente esterno che dice al crittografo di inviare il bit, ma è il crittografo che vuole inviare il bit in broadcast a tutti. Come fa a trasmettere? Supponiamo di avere un qualcosa che scaglioni il tempo. Quando scatta il tic i crittografi lanciano al moneta e la comunicano a lato. Al tic successivo i crittografi comunicano il valore di x. Un crittografo che vuole spedire un bit a 0 non fa nulla (si comporta come un crittografo che non ha mangiato), mentre uno che vuole spedire 1 si comporta come un crittografo che ha mangiato. Ad ogni coppia di tic, il crittografo che vuole trasmettere codifica il bit che vuole spedire.

Supponiamo che il crittografo che trasmette sia uno solo. In questo caso riesce a trasmettere con successo e tutti vedono il risultato della trasmissione. Il crittografo può anche inviare una sequenza di bit.

Potrebbero essere però presenti dei conflitti, ad esempio quando due crittografi decidono di spedire contemporaneamente. I due agenti hanno modo di accorgersi del conflitto? Ogni crittografo, quando spedisce il bit, riesce anche ad osservare il bit che è stato trasmesso sulla rete (lui, come tutti gli altri, fa lo XOR degli x_i). Se trasmette un bit e vede che nella rete c'è esattamente il bit che ha trasmesso, potrebbero anche esserci dei conflitti (se tre crittografi inviano 1, il risultato è comunque 1), ma non è un problema perché sta circolando il messaggio che volevamo.

Nel momento in cui c'è in circolo un altro bit diverso da nostro, uno dei due osserverà che il bit che sta circolando è diverso da quello che aveva inviato. Tutti i crittografi che si accorgono di aver creato un conflitto si fermano e lasciano continuare gli altri (simile al protocollo ethernet).

Quindi se un crittografo vuole trasmettere, procede come segue:

- Verifica che nessuno stia trasmettendo;
- Se il canale è vuoto inizia a trasmettere, continuando a controllare eventuali conflitti.

Invio del messaggio Supponiamo ora che un crittografo i voglia inviare un messaggio a un crittografo j. Poiché può trasmettere solo in broadcast, il messaggio conterrà l'identità del destinatario. Come possiamo nascondere il destinatario di un messaggio? Potremmo semplicemente prender il messaggio e cifrarlo con la chiave pubblica del destinatario.

Abbiamo realizzato un sistema di comunicazione completamente non tracciabile: non si sa chi ha inviato il messaggio, non si sa chi ha ricevuto il messaggio e non si sa qual è il contenuto del messaggio.

Si tratta comunque di un protocollo dispendioso, in quanto gli agente dovrebbe restare attivi durante tutte le fasi. Presenta inoltre un difetto: se un agente vuole oscurare il canale, lo può fare facilmente inviando bit casuali, senza che nessun altro agente sappia che è lui (è debole al denial of service -> tinen il canale occupato e causa conflitti nel caso un altro agente invii un messaggio).

Zero knowledge 13

Abbiamo ora tutti gli strumento per studiare la zero knowledge, ovvero l'idea di trasmettere un segreto senza conoscere il segreto. Abbiamo due agenti Peggy e Victor, che sono rispettivamente **Proover** e **Verifier**. La situazione che vogliamo creare è la seguente: il proover conosce un segreto e vuole convincere il verifier che conosce il segreto, senza però comunicarlo al verifier; inoltre non vuole nemmeno che il verifier possa a dire ad altri che egli conosce il segreto.

Vogliamo quindi che il verifier apprenda solo la conoscenza che il proover conosce il segreto, ma senza che questa conoscenza sia sufficiente a convincere altri che il proover conosce il segreto.

Esempio

Supponiamo che P e V si trovino in una caverna come in figura 13.1, dove le sezioni C e D sono separate da una porta che si apre solo con una parola chiave. P vuole dimostrare a V di conoscere la parola chiave. Vediamo come potrebbe fare:

- V segue P mentre parte dal punto B e ritorna al punto B. IN questo modo però V entra a conoscenza della parola segreta che apre al porta;
- V aspetta P al punto B e lo vede andare verso il lato C e tornare dal lato D. V però potrebbe attaccare una videocamera a P e mostrare il filmato ad altri per dimostrare che P conosce la parola chiave.

Nessuna di queste due soluzione va bene. Proviamo a vederne una che applica il seguente protocollo:

1. P, V si trovano all'entrata della caverna (punto A);
2. P raggiunge B e lancia una moneta per decidere C o D;
3. P raggiunge C o D;

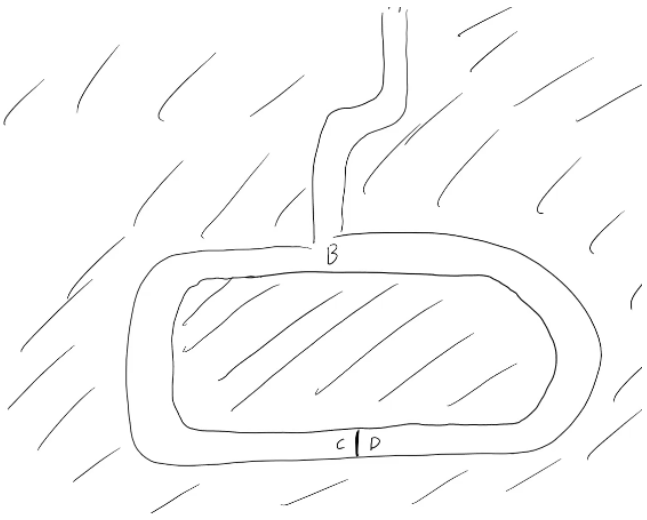


Figura 13.1: Caverna

4. V raggiunge B e lancia una moneta per decidere C o D;
5. V chiede a P di uscire dal lato risultante;
6. P obbedisce.

Con questo protocollo se P conosce il segreto non ha problemi, mentre se non lo conosce ha una probabilità di $\frac{1}{2}$ di essere scoperto da V, in quanto non sarebbe in grado di uscire dal lato richiesto. Di conseguenza V ha una probabilità di $\frac{1}{2}$ di verificare correttamente se P conosce o meno il segreto.

La probabilità che V ha di sbagliare è troppo altra, dobbiamo fare in modo che si abbassi. Come possiamo fare? Possiamo ripetere il protocollo k volte, rendendo la probabilità di sbagliare a $\frac{1}{2^k}$.

Abbiamo quindi creato un protocollo dove:

- Se P conosce il segreto, allora V lo accetterà sempre;
- Se P non conosce il segreto, possiamo rendere la probabilità che V accetti esponenzialmente piccola (sostanzialmente V rifiuta).

Questo protocollo risolve anche il problema del "filmato": poiché non viene filmata l'apertura della porta, potrebbe essere che P e V si fossero messi d'accordo prima di girare il filmato per convincere gli altri che P conosca il segreto.

Abbiamo raggiunto la zero knowledge totale.

13.1 Interactive Proof System per un linguaggio L

Un IPS per un linguaggio L si compone di una coppia di algoritmi/macchine di Turing) (P, V) tali che:

- Il proover ha capacità di calcolo illimitata;
- Il verifier ha capacità di calcolo $\in PPT$.

Poiché dimostrare è più difficile dimostrare che verificare, al proover diamo una potenza di calcolo illimitata. P e V interagiscono scambiandosi messaggi. Alla fine V accetta o rifiuta.

Un linguaggio L ammette IPS se esistano P, V tali che:

$$\forall x \in L \; Pr[(P, V) \text{ accetta } x] > \frac{2}{3}$$
$$\forall x \notin L \; \forall P' \; Pr[(P', V) \text{ accetta } x] < \frac{1}{3}$$

Nel primo caso va bene qualsiasi numero maggiore di $\frac{2}{3}$ (alta), mentre nel secondo minore di $\frac{1}{2}$ (bassa).

Qualunque linguaggio NP accetta un IPS. La classe dei linguaggi che ammette un IPS si chiama IP.

13.1.1 Problema del Quadratic Non Residuosity

Abbiamo un numero in Z_n^* che non è un quadrato e vogliamo costruire un IPS che lo dimostri:

- Input (x, n);
- Output accetta se x non è un quadrato.

Vediamo lo scambio di messaggi:

- V sceglie $w_1 \dots w_n \in_R Z_n^*$, $b_1 \dots b_n \in_R \{0,1\}$ e invia a P:

$$y_i = \begin{cases} W_i^2 & \text{se } b_i = 0 \\ W_i x^2 & \text{se } b_i = 1 \end{cases}$$

- P invia a V la sequenza $c_1 \dots c_n$, dove:

$$\forall i \; c_i = \begin{cases} 0 & \text{se } y_i \text{ quadrato} \\ 1 & \text{altrimenti} \end{cases}$$

- V accetta se e solo se $b_1 \dots b_l = c_1 \dots c_l$.

Questo è un IPS in quanto. Se x veramente non è un quadrato allora i w_i saranno quadrati quando $b_i = 0$ e non quadrati quando $b_i = 1$. P verifica la quadraticità dei w_i ed effettivamente sarà il caso che i $b_i = c_i$.

Se invece x è un quadrato, ovvero non appartiene al nostro linguaggio, tutti i w_i sono quadrato. In questo caso, che P conosca sia in grado di provare o no, non è in grado di ricavare informazione circa i b_i. Visto che i b_i sono stati scelti a caso, per ogni i la probabilità che c_i sia uguale al b_i è $\frac{1}{2}$. Quindi la probabilità che sia uguale per tutti gli i è $\frac{1}{2^n}$.

Ripetendo l'esperimento più volte, la probabilità che V accetti nel secondo caso diventa arbitrariamente piccola.

P vuole dimostrare a V di sapere la fattorizzazione di n. Come fa a dimostrarlo? Dice "dammi un numero e io ti dico se è un quadrato o no". Se P è in grado di dire la quadraticità di un numero, vuol dire che P conosce quell'informazione trapdoor che permette di capire se è un quadrato o un non quadrato. Ad oggi l'informazione trapdoor che permette di capire questo è la fattorizzazione di n.

13.1.2 Problema dei Grafi non isomorfi

Vogliamo stabilire se dati due grafi questi sono non isomorfi:

- Input (G_1, G_2), conosciuti da P, V;
- Output accetta se x non è un quadrato.

Definition 13.1.1 *Due gradi sono isomorfi se traslando i nodi del grafo A nel grafo B, gli archi che esistono nel grafo A esistono anche nel grafo B.*

Vediamo lo scambio dei messaggi:

- V sceglie $i \in_R \{1,2\}$, calcola G' permutazione di casuale di G_i e invia G' a P;

- P risponde:

$$\begin{cases} 0 & \text{se } G' \simeq G_1 \\ 1 & \text{se } G' \simeq G_2 \end{cases}$$

- V accetta se P indovina.

Se $G_1 \not\simeq G_2$, allora G' è isomorfo solo al grafo da cui si è calcolata la permutazione. P quindi indovina (avendo potenza di calcolo illimitata) e V accetta il risultato con probabilità 1. La capacita di P di indovinare è legata puramente alla sua capacità di calcolo, in quanto non conosciamo alcuna informazione trapdoor per gli isomorfismi tra grafi.

Se $G_1 \simeq G_2$ (ovvero non appartiene al linguaggio), G' è isomorfo ad entrambi, e di conseguenza non contiene alcuna informazione circa la scelta fatta da V. P può solo sparare a caso, avendo probabilità $\frac{1}{2}$ di indovinare G_i.

Come sempre, ripetendo l'esperimento più volte, la probabilità che V accetti nel secondo caso diventa arbitrariamente piccola.

Vogliamo arrivare ad un punto dove V non impara nulla di più del semplice fatto che due grafi sono non isomorfi. In questo contesto, in realtà, V è in grado di imparare qualcosa. Non propriamente il V descritto sopra, ma un V che cerca di sfruttate P per imparare qualcosa di più.

In generale abbiamo un V che pone delle domande e un P che risponde a queste domande, ma potremmo anche avere un V non tanto interessato a capire che due grafi sono non isomorfi, ma che cerca di capire se un dato grafo G_2 e isomorfo a un altro grafo G_3. Parte da un'istanza in cui si vuole vedere se $G_1 \simeq G_2$ e mentre comunica con P gli invia, come grafo da verificare, G_3 al posto di G'. Se P risponde fa sapere a V se $G_2 \simeq G_3$.

In generale questo si verifica quando V vuole risolvere un problema ma non ha abbastanza potenza di calcolo per poterlo fare. Quindi pone il problema a P.

Vogliamo che V non possa imparare nulla.

13.2 Zero knowledge

Se prendiamo (P, V) che interagiscono, la loro interazione genera una sequenza di messaggi. La sequenza è casuale, in quanto P e V lanciano monete (sono entrambi algoritmi stocastici -fanno scelte casuali-). Per essere precisi, dalla loro interazione si produce una misura di probabilità su sequenza di messaggi $View(P, V, x, h)$. La $View$ dipende dal proover, dal verifier, dalla stringa su cui stiamo cercando di riconoscere il linguaggio e da una sequenza h (**hint**). L'**hint** serve per catturare l'idea che il verifier non riesce a ricavare alcuna informazione dall'interazione al proover nemmeno quando c'è una terza parte che fa domande al verifier (dà dei suggerimenti).[1]

La $View(P, V, x, h)$ è il risultato tra l'interazione tra proover e verifier[2]. Quando possiamo dire che il verifier non apprende nulla se non il fatto

1: Questo h cattura il caso in cui qualcuno passa al verifier un grafo G_3 chiedendogli se è isomorfo a G_1 o G_2. Un verifier che riceve una stringa x in input su cui fare la verifica e un qualche suggerimento h, interagisce con il proover e da questa interazione si costruisce una misura di probabilità su questa sequenza di messaggi, chiamata $View(p, x)$. Quando abbiamo un IPS, vogliamo che un verifier non apprenda nulla (da protocollo). Il verifier che stiamo usando ora è un verifier fasullo V', che appare a P come il vero verifier, che fa domanda atte a ottenere altri risultati (h) non sia in grado di imparare nulla.

2: Praticamente è il filmato che V ha fatto.

che $x \in L$ dall'interazione con P? L'unica cosa che V ha in mano quando ha interagito con P è un elemento campionato da $View$. Di conseguenza qualcuno che vede questa roba campionata da $View$, oppure vede più interazione tra V e P[3], non dice nulla ad una terza parte se il verifier poteva campionare messaggi dalla stessa misura di probabilità senza interagire con Proover.

3: È possibile effettuare delle indagini statistiche fino ad una quantità polinomiale di interazioni.

Se il verifier fosse in grado di costruire messaggi con la stessa misura di probabilità di $View$, potremmo dire che il fatto di vedere l'interazione tra P e V non dà info aggiuntive.

Definition 13.2.1 ((Perfect) zero knowledge) *Diciamo che P, V è **zero knowledge** se $\forall V' \in PPT$ esiste un simulatore M in PPT tale che:*

$$M(P, V', x, h) = View(P, V', x, h)$$

Quindi esiste un algoritmo $M \in PPT$ che da solo produce sequenze di messaggi che sono distribuite come $View$[4]. Poiché possiamo usare il simulatore M per generare la sequenza di messaggi, nessuno garantisce che $View(P, V, x, h)$ derivi dall'interazione tra P e V, e che non sia stato generato da V.

4: Senza interagire con Proover, siamo in grado di costruire sequenze di messaggi scambiati tra P e V' con la stessa identica provabilità di $View$.

Possiamo definire tre tipi di zero knowledge:

1. **Perfect Zero Knowledge**: la differenza tra $View(P, V', x, h)$ e $M(P, V', x, h)$ è 0:

$$View(P, V', x, h) = M(P, V', x, h)$$

2. **Statistical Zero Knowledge**: la differenza tra $View(P, V', x, h)$ e $M(P, V', x, h)$ è più piccola di ogni polinomio:

$$\sum_{\alpha} \left| View(P, V', x, h)(\alpha) - M(P, V', x, h)(\alpha) \right| < k^{-c}$$

   Non è necessario che il simulatore produce esattamente la stessa identica misura di probabilità prodotta da $View$, ma basta che le due misure sia sufficientemente vicine che nessun test statistico polinomiale possa vedere la differenza[5].
3. **Computational Zero Knowledge**: $View(P, V', x, h)$ e $M(P, V', x, h)$ sono computazionalmente (polinomialmente) indistinguibili[6]. Siano $P_k^{D,V}$ la probabilità che D restituisca 1 campionando da $View$ e $P_k^{D,M}$ la probabilità che D restituisca 1 campionando da M, allora:

$$\forall D \in PPT \left| P_k^{D,V} - P_k^{D,M} \right| < k^{\omega(1)}$$

5: Se noi campioniamo più volte $View$ ed M, il numero di campioni che dobbiamo fare per vedere una differenza deve essere almeno pari al reciproco della differenza di probabilità sulle due cose che sto campionando. Quello che ci sta dicendo la formula è che la **somma** delle differenze di probabilità su tutti gli oggetti che stanno nello spazio campione della misura $View$ e della misura M deve essere più piccola di qualsiasi polinomio. DI conseguenza, la quantità di esperimenti da fare per vedere una differenza è più grande di qualsiasi polinomio.

6: Anche se vediamo il filmato, e questo è diverso dalla vera interazione tra Proover e Verifier, non siamo capaci di vedere la differenza. Non li sappiamo distinguere.

Dal punto di vista pratico, chiedere la perfect o la computational è la stessa cosa. inoltre perfect implica statistical e statistical implica computational.

13.2.1 Problema del Ciclo hamiltoniano

Verificare se G ammette ciclo hamiltoniano.

Definition 13.2.2 (Ciclo hamiltoniano) *Un ciclo hamiltoniano se esiste un cammino che visita ogni nodo esattamente una volta che riporta al nodo di partenza.*

L'input è un grafo G di cui non conosciamo un ciclo hamiltoniano. Vediamo lo scambio dei messaggi:

- P permuta G per ottenere H, codifica tutti i bit della matrice di adiacenza di H con bit commitment. Sia H' il risultato;
- P invia H' a V;
- V lancia una moneta per decidere se chiedere a P:
  A. L'evidenza che $G \simeq H$;
  B. L'evidenza che H ammette ciclo hamiltoniano.
- P obbedisce. Per dare evidenza:
  A. Nel primo caso rivela H e l'isomorfismo tra G ed H[7];
  B. Nel secondo caso scopre solo i bit di H che costituiscono un ciclo hamiltoniano[8].

Supponiamo che G ammetta ciclo hamiltoniano. Allora quando V fa al domanda, P è in grado di rispondere ad entrambe le domande[9]. Quando G ammette ciclo hamiltoniano e P lo conosce, effettivamente il Proover è in gradi di rispondere sempre alle domande di V.

Se G non ammette ciclo hamiltoniano, o il Proover non lo conosce (in un contesto la potenza di calcolo illimitata di P deriva da una conoscenza di un'informazione trapdoor), allora quando il Proover manda un grafo H coperto dal bit commitment al Verifier, non è possibile che H sia contemporaneamente isomorfo a G e ammetta ciclo hamiltoniano:

- Se H è isomorfo a a G, allora anche H non ammette ciclo hamiltoniano. Di conseguenza il Proover sa rispondere solo alla domanda **A**;
- Se H ammette ciclo hamiltoniano, allora H non può essere isomorfo a G. DI conseguenza il Proover sa rispondere solo alla domanda **B**.

Visto che il verifier sceglie la domanda casualmente, con probabilità $\frac{1}{2}$, il Proover sa rispondere.

Questo è uno dei casi in cui P conosce un segreto e V vuole verificare quel segreto. Lo vogliamo fare in maniera tale che V non possa usare l'interazione con P per convincere altri.

Come costruiamo M? Scegliamo casualmente la domanda di V. Se:

- La domanda è la **A**, costruiamo H' secondo il protocollo (H', domanda, risposta);
- La domanda è **B**, costruiamo H' bit commitment del grafo completo. Scegliamo una permutazione casuale dei nodi come ciclo (H', domanda, risposta).

La distribuzione delle domande è quella corretta (con probabilità $\frac{1}{2}$ chiediamo la domanda **A**, con $\frac{1}{2}$ la domanda **B**); la distribuzione delle risposte è quella corretta: nel caso **A** è quella dovuta alla permutazione casuale del grafo, nel caso **B** è una permutazione casuale dei nodi di H'. Se prendiamo un grafo che ammette un ciclo hamiltoniano e ne permutiamo a caso i suoi nodi, questo diventa effettivamente la

7: Se due grafi sono isomorfi, uno ammette ciclo hamiltoniano se e solo se anche l'altro lo ammette.

8: Il Proover non va a scoprire l'intera matrice di adiacenza, ma scopre solo alcuni archi, che sono quelli che mostrano che esiste un ciclo hamiltoniano. Effettivamente in questo caso il Verifier ha l'evidenza che la matrice di adiacenza di H contiene un ciclo hamiltoniano, ma non ha altra conoscenza sulla struttura di H.

9: Il Proover ha costruito l'isomorfismo tra G e H, quindi conosce H, G, l'isomorfismo (è in grado di scoprire tutto) e conosce il ciclo hamiltoniano su G. Conoscendo un ciclo hamiltoniano su G, allora può applicare l'isomorfismo che ha costruito per costruire un ciclo hamiltoniano su H. A questo punto è un grado di scoprire i bit di H' corretti per rivelare l'esistenza del ciclo

permutazione casuale dei nodi di un gradi. Quindi ogni permutazione dei nodi ha la stessa identica probabilità di essere un ciclo hamiltoniano, nel protocollo originale. Quindi anche la misura di probabilità delle risposte è quella originale.

La misura di probabilità di H' è uguale all'originale nel caso della domanda **A**, ma non nella domanda **B**. Nel caso **B** è un bit commitment non di una permutazione del grafo originale, ma di una permutazione di un grafo completo. Il punto chiave: prendiamo un qualunque osservatore che vede i messaggi costruiti da Me un osservatore che vede i messaggi scambiati da P e V. Se questo osservatore ha potenza di calcolo polinomiale riesce a vedere la differenza? Vede che le domande sono distribuite uniformemente (scelte casualmente) e le risposte sono costruite correttamente. Però vediamo che il primo messaggio mandato, H', in alcuni casi è il bit commitment della permutazione del grafo originale, negli altri casi è il bit commitment di un grafo completo. Ma un osservatore riesce a vedere che la matrice di adiacenza di H' è uguale a quella di un grafo completo? Qualunque osservatore con potenza di calcolo polinomiale nono riesce ad avere alcuna informazione circa i bit nascosti da bit commitment. Di conseguenza non riesce a vedere la differenza.

Non è vero che quello che ha prodotto M è identico all'interazione tra P e V. Ma è vero che qualunque algoritmo con capacità di calcolo polinomiale, messo davanti all'interazione costruita da M o a quella reale, non riesce a vedere la differenza. Le due misure di probabilità sono polinomialmente indistinguibili (solito concetto di distinguisher).

Questa cosa è computational zero knowledge in quanto nel caso **B** inviamo qualcosa di diverso rispetto alla normale interazione tra P e V, solo che è indistinguibile all'esterno in quanto protetta con bit commitment.