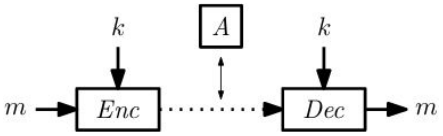
Crittografia Core Concepts

La **crittografia** è un insieme di tecniche di natura algoritmica che permette a due o più entità (agenti) di comunicare tra loro in modo sicuro.

Introduzione alla Crittografia



Vi è una chiave unica, preventivamente scambiata tra le parti, che viene usata prima per cifrare il messaggio in chiaro, poi per decifrare il messaggio cifrato.

Uno **schema di codifica** (o cifrario) è dato da tre spazi *K*, *M* e *C* ovvero lo spazio delle chiavi, dei messaggi e dei crittogrammi, solitamente sono tutti insiemi di stringhe (es. stringhe binarie, da alfabeto naturale, da alfabeto non già esistente, ecc.). Si ha inoltre una tripla di algoritmi (Gen, Enc, Dec) dove:

Gen: $1 \to K$ Enc: $M \times K \to C$ Dec: $C \times K \to M$

Enc prende in input un messaggio e una chiave e produce in output un crittogramma, il contrario di *Dec*. *Gen* invece è un algoritmo che produce in output una chiave a partire da un insieme 1, ovvero un qualunque insieme che contiene un solo elemento. Questo input 1 "fittizio" è necessario per evitare che si intenda Gen come un elemento che produce sempre lo stesso output. Gen non può essere vista come una funzione, in quanto lo stesso input NON produce lo stesso output, è considerato quindi un algoritmo probabilistico.

Uno schema è considerato *corretto* quando Dec(Enc(x, k), k) = x. La correttezza è considerata un requisito minimo.

Dato un avversario A che può interferire nella comunicazione tra Alice e Bob, il **principio di Kerchoff** assume che A conosca il funzionamento interno di Gen, Enc, Dec, quindi l'unico elemento sconosciuto è la chiave k.

Possibili attacchi:
- *Ciphertext-Only Attack*: è conosciuto solo il crittogramma, l'attaccante non fa altro che analizzare i soli elementi crittografati (attacco passivo).
- *Known-Plaintext Attack*: l'avversario conosce i crittogrammi di alcuni messaggi in chiaro (attacco passivo).
- *Chosen-Plaintext Attack*: l'avversario sceglie attivamente testi in chiaro da cifrare per scoprirne il crittogramma (attacco attivo). Quindi $Enc_k(m) = Enc(m, k)$, è possibile invocare la funzione di cifratura anche senza conoscerne la chiave.
- *Chosen-Ciphertext Attack*: l'avversario partecipa attivamente alla comunicazione avendo accesso al sistema di decifrazione in modalità di oracolo $Dec_k(\cdot)$ (attacco attivo).

Cifrario di Cesare
I messaggi in M sono testi in una qualunque lingua, vi è una sola chiave in K che non cambia mai, ad esempio K = {4}. *Enc* non fa altro che traslare di k posizioni sull'alfabeto ogni carattere del messaggio. È molto debole in quanto lo spazio delle chiavi contiene un solo elemento.

Cifrario a Rotazione
È una generalizzazione del cifrario di Cesare dove lo spazio delle chiavi diventa $K = \{1 \dots |\sum| - 1\}$, quindi contiene tutte le posizioni dei possibili caratteri dell'alfabeto utilizzato. Si applica sempre uno shift di k simboli ad ogni carattere del messaggio, con k un numero tra quelli disponibili all'interno di *K*. Aumenta lo spazio delle chiavi ma rimangono troppo poche (da 2 in caso di alfabeto binario, a poche decine in caso di alfabeti standard)..

Cifrario a Sostituzione Monoalfabetica
È un'ulteriore generalizzazione del cifrario a Rotazione, dove lo spazio dei messaggi rimane lo stesso, mentre lo spazio delle chiavi diventa $K = \{\sigma \mid \sigma : \sum \to \sum$ è una permutazione$\}$. Quindi ogni chiave rappresenta una permutazione dell'alfabeto originario, utilizzare la chiave $\sigma(C)$ significa sostituire la lettera C con la lettera in terza posizione all'interno della permutazione scelta σ. In questo modo lo spazio delle chiave è rappresentato da tutte le possibili permutazioni dell'alfabeto Σ, ovvero $|\Sigma|!$ che risulta essere impraticabile anche da un attacco brute-force. Tuttavia simboli uguali vengono cifrati con valori uguali e per questo esistono attacchi statistici.

Gli attacchi statistici analizzano le frequenze di ciascun simbolo nel crittogramma, confrontandola con quella degli stessi simboli nei messaggi in *M*. Ogni linguaggio ha lettere che appaiono più volte di altre, così anche i crittogrammi che ne risultano avranno lettere (seppur prive di significato nel contesto) che appaiono più frequentemente.

Cifrario di Vigenère
In questo cifrario lo spazio delle chiavi è l'insieme delle stringhe di lunghezza finita in Σ, ossia $K = \Sigma^*$. Ogni carattere è ruotato di un certo numero di elementi (ovvero ha un alfabeto diverso, definito dalla chiave in quel carattere) che differisce dal carattere successivo, e così via. La lunghezza della chiave $|k|$ indica ogni quanti caratteri la rotazione si ripete. Anche qui gli attacchi brute force non sono applicabili, però se il periodo è noto la chiave può essere facilmente rotta applicando tecniche statistiche a tutti i caratteri che condividono la stessa chiave all'interno del testo cifrato.

Sicurezza Perfetta

Nella sicurezza perfetta, anche *Enc*, così come *Gen*, può procedere in modo probabilistico: a stesso messaggio e stessa chiave possono corrispondere testi cifrati diversi. *Dec* rimane chiaramente deterministico. La scelta del messaggio e della chiave, la codifica e la decodifica sono viste come processo probabilistico.

Possiamo definire tre variabili casuali:
K: che corrisponde alla chiave utilizzata e la cui distribuzione di probabilità dipende dall'algoritmo *Gen* (l'ottimo si ha se tutte le chiavi hanno uguale probabilità, ma non è sempre facile).
M: che cattura il messaggio prodotto dal mittente (Alice con una certa probabilità produrrà un qualche messaggio, in una conversazione ad esempio si suppone siano più probabili messaggi di senso compiuto che incompiuto).
C: corrisponde al crittogramma e che dipenderà da **K**, **M** ed *Enc*.
Possiamo calcolare quantità come $\Pr(\mathbf{K} = k)$ dove $k \in K$ è una chiave, oppure la probabilità condizionata $\Pr(\mathbf{K} = k \mid \mathbf{M} = m)$, dove $m \in M$.
Nota: **K** è una variabile aleatoria, *K* è l'insieme di chiavi e k è una singola chiave.

La scelta della chiave e del messaggio sono sempre da considerarsi *indipendenti*, vorremmo inoltre catturare il fatto che conoscere il valore di **C** non modifichi la conoscenza di **M**.

Definizione (Sicurezza Perfetta)
Uno schema di codifica (Gen, Enc, Dec) si dice **perfettamente sicuro** sse per ogni messaggio m ∈ *M* e per ogni crittogramma c ∈ *C* tale che Pr(**C** = c) > 0 vale che

$$Pr(\mathbf{M} = m \mid \mathbf{C} = c) = Pr(\mathbf{M} = m).$$

Quindi la sicurezza perfetta si ha quando la probabilità di scoprire il messaggio osservando il crittogramma equivale a conoscere il messaggio in chiaro, ovvero con il crittogramma non si riesce a risalire al relativo messaggio in chiaro con probabilità maggiore che cercare di indovinare a caso il messaggio in chiaro.

Lemma
Uno schema di codifica (Gen, Enc, Dec) si dice **perfettamente sicuro** sse per ogni messaggio m ∈ *M* e per ogni crittogramma c ∈ *C* vale che

$$Pr(\mathbf{C} = c \mid \mathbf{M} = m) = Pr(\mathbf{C} = c).$$

Deve essere quindi vero anche il contrario, conoscendo il solo messaggio non si riesce a risalire al suo crittogramma con probabilità maggiore che cercare indovinare a caso il crittogramma.

Dimostrazione

$$\Rightarrow Pr(\mathbf{M} = m) = Pr(\mathbf{M} = m \mid \mathbf{C} = c) = \frac{Pr(\mathbf{M} = m)Pr(\mathbf{C} = c \mid \mathbf{M} = m)}{Pr(\mathbf{C} = c)} = Pr(\mathbf{C} = c) = Pr(\mathbf{C} = c \mid \mathbf{M} = m)$$

$$\Leftarrow \textit{La dimostrazione e' sostanzialmente identica a quella sopra}$$

Nella seconda uguaglianza viene applicato il teorema di Bayes:

$$P(A \mid B) = (P(B \mid A) * P(A)) / P(B)$$

Nella terza uguaglianza viene diviso per Pr(**M** = m) entrambi i membri semplificandolo, e portando in alto Pr(**C** = c).

Lemma
Uno schema di codifica (Gen, Enc, Dec) si dice **perfettamente sicuro** sse per ogni coppia di messaggi $m_0, m_1 \in M$ e per ogni crittogramma c ∈ *C* vale che

$$Pr(\mathbf{C} = c \mid \mathbf{M} = m_0) = Pr(\mathbf{C} = c \mid \mathbf{M} = m_1).$$

Quindi tutti i messaggi sono equivalenti, non si è avvantaggiati nel scoprire il crittogramma conoscendo un messaggio o un altro.

Dimostrazione

$$\Rightarrow Pr(\mathbf{C} = c \mid \mathbf{M} = m_0) = Pr(\mathbf{C} = c) = Pr(\mathbf{C} = c \mid \mathbf{M} = m_1)$$

$$\Leftarrow se\ Pr(\mathbf{C} = c \mid \mathbf{M} = m_0) = Pr(\mathbf{C} = c \mid \mathbf{M} = m_1)\ \forall m_0, m_1\ allora$$
$$\exists p\ t.c.\ Pr(\mathbf{C} = c \mid \mathbf{M} = m_i) = p$$
$$Pr(\mathbf{C} = c) = \sum_{m_i \in M} Pr(\mathbf{C} = c \mid \mathbf{M} = m_i) Pr(\mathbf{M} = m_i) =$$
$$p \sum_{m_i \in M} Pr(\mathbf{M} = m_i) = p \cdot 1 = p$$

Nella prima uguaglianza uso il lemma precedente. Nella seconda uguaglianza al passaggio dopo Pr(**C** = c) si ha che Pr(**C** = c | **M** = m_i) = p che è un valore fisso per il cifrario.

Cifrario di Vernam
È possibile costruire un cifrario concreto che sia perfettamente sicuro: il cifrario di Vernam (One-Time Pad):

$K = M = C = \{0, 1\}^n$ Enc(m, k) = m ⊕ k Dec(c, k) = c ⊕ k Pr(**K** = k) = $1 / 2^n$

Il cifrario è corretto in quanto:
Dec(Enc(m, k), k) = (m ⊕ k) ⊕ k = m ⊕ (k ⊕ k) = m ⊕ 0^n = m

Un difetto di questo messaggio è che le *chiavi hanno la stessa lunghezza del messaggio*, questo pone una limitazione molto forte. Inoltre ogni chiave può essere utilizzata una sola volta.

Teorema
Il **cifrario di Vernam** è **perfettamente sicuro**.

Dimostrazione
$$\Rightarrow Pr(\mathbf{C} = c \mid \mathbf{M} = m) = Pr(\mathbf{M} \oplus \mathbf{K} = c \mid \mathbf{M} = m) =$$
$$Pr(m \oplus \mathbf{K} = c) = Pr(m \oplus \mathbf{K} \oplus m = c \oplus m) =$$
$$Pr(\mathbf{K} = c \oplus m) = (\frac{1}{2})^L$$
$$\Leftarrow Pr(\mathbf{C} = c \mid \mathbf{M} = m_0) = (\frac{1}{2})^L = Pr(\mathbf{C} = c \mid \mathbf{M} = m_1)$$

Al punto Pr(m ⊕ **K** ⊕ m = c ⊕ m) avviene una semplificazione per le proprietà dello xor dei due m a sinistra dell'uguale. L indica presumibilmente la lunghezza del messaggio.

Teorema
Se (Gen, Enc, Dec) è uno schema di codifica perfettamente sicuro con *M* e *K* gli spazi dei messaggi e delle chiavi, abbiamo che necessariamente

$$|K| \geq |M|$$

Dimostrazione

Per assurdo sia $\Pi = (Gen, Enc, Dec)$ *perfettamente sicuro ma con* $|K| < |M|$.

Supponiamo che M *assegni a tutti i messaggi la stessa probabilita', fissiamo* $c \in C$ *che abbia probabilita' non nulla.*

$M(c) = \{\hat{m} \mid \hat{m} = Dec_{\hat{k}}(c)$ *per qualche* $\hat{k} \in K\}$

$|M(c)| \leq |K| < |M|$

$Pr(\mathbf{M} = m' \mid \mathbf{C} = c) = 0$

$Pr(\mathbf{M} = m') > 0$

Nella terzultima riga si ha il ≤ in quanto non possono esserci messaggi decifrati da più chiavi rispetto alle chiavi esistenti. Mentre sempre nella stessa riga si ha il < in quanto ci sarà un m' ∈ *M* tale che m' ∉ *M*(c). Da quest'ultima osservazione si formula la penultima riga, mentre l'ultima riga è > 0 in quanto abbiamo una distribuzione uniforme. Tuttavia le ultime due righe sono in netto contrasto con la definizione di sicurezza perfetta di Π, si dimostra così per assurdo il teorema.

Questo, per come abbiamo definito sicurezza perfetta, pone grossi limiti.

Indistinguibilità

Un'ulteriore caratterizzazione si ottiene considerando l'esperimento $PrivK_{A, \Pi}$ ossia una sorta di "gioco" in cui un ipotetico avversario A e uno schema Π = (Gen, Enc, Dec) vengono messi uno contro l'altro:

***PrivK*$_{A, \Pi}$**

$(m_0, m_1) \leftarrow A$;	L'avversario sceglie a suo piacere 2 messaggi (scegliendo *M*)
$k \leftarrow Gen$;	L'esperimento genera la chiave usando il cifrario
$b \leftarrow \{0, 1\}$;	L'esperimento sceglie uno dei due messaggi (a caso)
$c \leftarrow Enc(k, m_b)$;	L'esperimento cifra solo il messaggio scelto (a caso)
$b^* \leftarrow A(c)$;	L'avversario conoscendo il testo cifrato, dirà qual è l'm cifrato
Result: $\neg(b \oplus b^*)$	L'esperimento restituisce 1 se A indovina, 0 altrimenti

$Pr(PrivK_{A, \Pi})$ è la probabilità di successo per l'avversario di trovare quale dei due messaggi è stato cifrato, se l'avversario non riesce ad avere probabilità di successo superiore ad 1/2 allora si considera il cifrario sicuro. Quindi non si hanno più probabilità di successo che tentare a caso.

Definizione

Uno schema di codifica Π = (Gen, Enc, Dec) si dice avere **codifiche perfettamente indistinguibili** sse vale che

$$Pr(PrivK_{A, \Pi}) = 1/2$$

Teorema

Π è **perfettamente sicuro** sse Π ha **codifiche perfettamente indistinguibili**.

Schemi di Codifica a Chiave Segreta

Nella crittografia computazionale, nata alla fine degli anni 70, la sicurezza perfetta viene indebolita, passando ad una definizione che offre garanzie meno forti. Tale indebolimento si sviluppa su due assi:

1. L'impossibilità di forzare lo schema è garantita solo per *avversari efficienti*.

2. Inoltre l'avversario può forzare il sottostante schema, ma la sua *probabilità di successo* dev'essere *molto piccola*.

Approccio Asintotico
I giudizi di sicurezza dipendono da un parametro globale n, detto parametro di sicurezza. Il concetto di efficienza viene catturato tramite gli algoritmi PPT, mentre quello di bassa probabilità di successo viene catturato con il concetto di funzione trascurabile.

Definizione
Un algoritmo probabilistico A si dice **PPT** (probabilistic polynomial time) sse esiste un polinomio *p* che limita superiormente il tempo di calcolo di A *indipendentemente* dalle scelte probabilistiche effettuate da quest'ultimo.

Definizione
Una funzione $f : \mathbb{N} \to \mathbb{R}$ si dice **trascurabile** sse per ogni polinomio $p : \mathbb{N} \to \mathbb{N}$ esiste $N \in \mathbb{N}$ tale che per ogni $n > N$ vale che $f(n) < 1 / p(n)$.

Un esempio di funzione trascurabile è $f(n) = 2^{-n}$ in quanto esponenzialmente piccola, un esempio di funzione non trascurabile è $f(n) = n^{-3}$ in quanto solo polinomialmente piccola.

Lemma
L'insieme di tutte le funzioni trascurabili, chiamato *NGL*, è chiuso per somma, prodotto, e prodotto per un arbitrario polinomio.

Questo approccio fa affidamento sulle risorse limitate dell'avversario, lasciando la possibilità per un cifrario di essere forzato ma solo con probabilità molto bassa.
Dimostrare la sicurezza di uno schema di codifica in questo senso richiederebbe la dimostrazione dell'impossibilità di costruire avversari che soddisfino certi criteri:

$$\forall A \in \text{PPT}.\ \neg \mathbf{BRK}(\Pi, A) \Leftrightarrow \neg \exists A \in \text{PPT}.\mathbf{BRK}(\Pi, A)$$

Tuttavia non si è in grado di dimostrare risultati come i precedenti, ovvero che *NON* esiste un algoritmo tale che riesca a rompere un cifrario.

Occorre modificare leggermente la definizione di schema di codifica Π = (*Gen, Enc, Dec*) imponendo che i tre algoritmi coinvolti siano PPT e che inoltre:
- *Gen* prende in input una stringa nella forma 1^n (1 ripetuto n volte, da interpretare come il parametro di sicurezza) e produce in output una chiave k, che sia tale che $|k| \geq n$. La lunghezza della chiave dev'essere sempre maggiore del parametro, altrimenti la possibilità di provare tutte le chiavi sarebbe comunque possibile.
- *Enc* possa essere genuinamente probabilistico, ossia produrre crittogrammi diversi a partire dalla stessa coppia (m, k).
- *Dec* invece è necessario sia deterministico, altrimenti per lo stesso crittogramma si rischierebbe di produrre due messaggi in chiaro diversi.
Assumiamo, come è ovvio che sia, che lo schema sia almeno corretto, ossia che $\text{Dec}_k(\text{Enc}_k(m)) = m$.
Spesso, Enc_k è definita solo per messaggi di lunghezza pari a l(n), dove n è il parametro di sicurezza con cui k è stata generata. In tal caso diciamo che lo schema di codifica è per messaggi di lunghezza l.

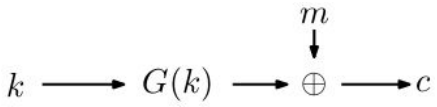
Occorre adattare alle nuove specifiche per i cifrari, la nozione di esperimento che abbiamo visto parlando di sicurezza perfetta:

PrivK$_{A,\Pi}^{eav}$ (n)
$(m_0, m_1) \leftarrow A(1^n)$; La lunghezza dei messaggi dipende da n [l(n)]
if $|m_0| \neq |m_1|$ then Non devono essere distinti per la lunghezza
Result: 0
$k \leftarrow Gen(1^n)$;
$b \leftarrow \{0, 1\}$;
$c \leftarrow Enc(k, m_b)$;
$b^* \leftarrow A(c)$;
Result: $\neg(b \oplus b^*)$

Definizione
Uno schema di codifica Π si dice **sicuro contro attacchi passivi** oppure sicuro rispetto a $PrivK^{eav}$ sse per ogni avversario PPT *A* esiste una funzione $\varepsilon \in NGL$ tale che:

$$Pr(PrivK_{A,\Pi}^{eav}(n) = 1) = 1/2 + \varepsilon(n)$$

Uno schema sicuro conosciuto è One-Time Pad, tuttavia il fatto che la lunghezza della chiave sia uguale a quella dei messaggi è molto limitante. Si vuole trovare un sistema per espandere una chiave di dimensioni limitate fino alla lunghezza del messaggio, al quale poi applicare One-Time Pad.



Prima di tutto è necessario che G sia un algoritmo deterministico, ossia che non utilizzi al proprio interno nessuna forma di scelta casuale, altrimenti a due chiamate diverse possono corrispondere output diversi e quindi risulterebbe impossibile decifrare il crittogramma.
Successivamente bisogna fare in modo che G sia efficientemente calcolabile, altrimenti l'intero schema diventerebbe problematico dal punto di vista computazionale.
Infine si vuole chiaramente che G permetta di espandere la stringa in input, se così non fosse, non avrebbe senso costruire un nuovo schema. Non solo questo, bisogna fare in modo che la stringa G(k) abbia tutte le proprietà che ci aspettiamo da una chiave.

Informalmente, si vuole che G(k) sia *pseudocasuale* ovvero pur non essendo causale, sia indistinguibile da una stringa casuale agli occhi di un avversario *efficiente*.

Definizione (Generatore Pseudocasuale)
Sia $l : \mathbb{N} \to \mathbb{N}$ un polinomio, detto *fattore di espansione* e sia G un algoritmo deterministico che, per ogni input $s \in \{0, 1\}^*$ produca in output una stringa $G(s) \in \{0, 1\}^{l(|s|)}$. Diciamo che G è **generatore pseudocasuale** (GP) sse:
- Per ogni $n \in \mathbb{N}$ vale che $l(n) > n$
- G è polytime
- Per ogni distinguitore D PPT esiste $\varepsilon \in NGL$ tale che:

$$|Pr(D(s) = 1) - Pr(D(G(r)) = 1)| \leq \varepsilon(n)$$

dove s, r sono casuali di lunghezza l(n) e n, rispettivamente

L'output dei generatori pseudocasuali non è casuale in senso stretto: se, per esempio, l(n) = 2n, vi sono soltanto 2^n stringhe (lunghe 2n) prodotte in output da G, mentre le stringhe lunghe 2n sono 2^{2n}. Osserviamo che:

$$2^n / 2^{2n} = 1 / 2^n$$

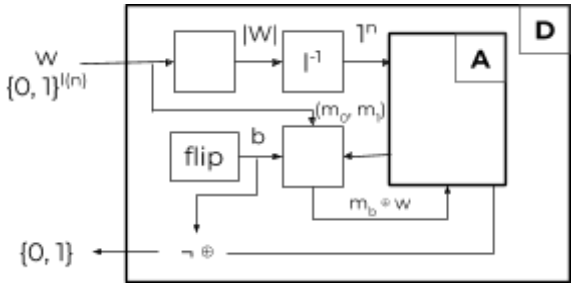
Esiste quindi sempre un distinguitore D che vince contro G con probabilità molto alta, ma tale distinguitore ha complessità esponenziale (2^n tentativi).

Definizione (Schema di Codifica Sicuro Indotto da un GP)
Dato un generatore pseudocasuale G con fattore di espansione l, lo **schema** Π^G è definito come segue:
- *Gen* produce su input 1^n ciascuna stringa lunga n con uguale probabilità, i.e. 2^{-n}.
- *Enc*(m, k) = G(k) $\oplus$ m.
- *Dec*(m, k) = G(k) $\oplus$ c.

La correttezza è dimostrata da: *Dec(Enc(*m, k), k) = G(k) $\oplus$ (G(k) $\oplus$ m) = m.

Teorema
Se G è un generatore pseudocasuale, allora Π^G è **sicuro contro attacchi passivi**.

Dimostrazione
Per contrapposizione dimostriamo che se A vince contro Π^G con probabilità $Pr(PrivK_{\Pi, A}^{eav}(n) = 1) = 1/2 + \varepsilon(n)$ ed ε sia non trascurabile, allora A può essere usato per costruire un distinguitore per G. Un distinguitore prende in input un $w \in \{0, 1\}^{l(n)}$ e restituisce un risultato in {0, 1}, mentre A vuole in input un 1^n e produce una coppia di messaggi.



Distinguiamo due casi:
1. Se a D viene passata una stringa casuale allora siamo in un caso analogo a One-Time Pad e abbiamo:

$$Pr(D(w) = 1) = Pr(PrivK_{OTP, A}^{eav}(n) = 1) = 1/2$$

2. Se a D viene passata una stringa pseudocasuale G(r) allora per ipotesi abbiamo

$$Pr(D(G(r)) = 1) = Pr(PrivK_{\Pi^G, A}^{eav}(n) = 1) = 1/2 + \varepsilon(n)$$

Dove ε si è supposto essere non trascurabile, dunque

$$|Pr(D(w) = 1) - Pr(D(G(R)) = 1)| = |1 / 2 - (1 / 2 + \varepsilon(n))| = |-\varepsilon(n)| = \varepsilon(n).$$

E questo crea un assurdo perché l'ε risultante è di fatto trascurabile, mentre si era assunto non lo fosse.

Tuttavia lo schema Π^G così definito è per messaggi di lunghezza fissa pari ad l. Generalizziamo il concetto ad un caso di lunghezza variabile.

Definizione (GP a Lunghezza Variabile)
Un algoritmo deterministico polytime G è detto **generatore pseudocasuale a lunghezza variabile** se a partire da un seed $s \in \{0, 1\}^n$ e da una stringa nella forma 1^l produce una stringa binaria $G(s, 1^l)$ tale che
- Se $l < l'$ allora $G(s, 1^l)$ è un prefisso di $G(s, 1^{l'})$ (a parità di seed s).
- Per ogni polinomio $p: \mathbb{N} \to \mathbb{N}$, l'algoritmo G_p definito ponendo $G_p(s) = G(s, 1^{p(|s|)})$ è un generatore pseudocasuale (a lunghezza fissa p).

Data questa definizione è facile generalizzare il cifrario Π^G con essa:
- *Enc*(m, k) = G(k, $1^{|m|}$) $\oplus$ m
- *Dec*(c, k) = G(k, $1^{|c|}$) $\oplus$ c

Lemma
Per ogni generatore pseudocasuale G è possibile costruire a partire da G un generatore pseudocasuale a lunghezza variabile H.

Concretamente, i cosiddetti *cifrari di flusso* (es. RC4) non sono veri e propri cifrari (i.e. schemi di codifica), ma di fatto sono progettati per rispettare gli assiomi di un generatore pseudocasuale a lunghezza variabile. Tuttavia non è possibile dimostrare che li soddisfino.

Codifiche multiple
Se tuttavia si usasse la stessa chiave per cifrare più messaggi, gli schemi non rimarrebbero sicuri. Per capire questo passaggio basta passare dell'esperimento $PrivK^{eav}$ a $PrivK^{mult}$. Il secondo è del tutto analogo al primo, solo che l'avversario non genera più solo una coppia di messaggi, bensì una coppia di vettori di messaggi della stessa lunghezza $m_0 = (m_0^1, \ldots, m_0^t)$ e $m_1 = (m_1^1, \ldots, m_1^t)$ da far cifrare. Si richiede come in $PrivK^{eav}$ che per ogni avversario A PPT esista un ε trascurabile tale che:

$$Pr(PrivK_{\Pi, A}^{mult}(n) = 1) = 1/2 + \varepsilon(n)$$

Che è ora la definizione di sicurezza rispetto a $PrivK^{mult}$, tuttavia gli schemi costruiti finora non sono sicuri rispetto a questa definizione.

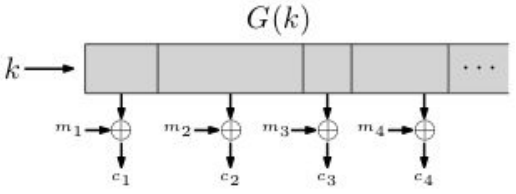
Lemma
Lo schema Π^G non è sicuro rispetto a $PrivK^{mult}$ neanche quando G è *pseudocasuale*.

Intuitivamente ciò è banale, non solo è possibile vincere con probabilità non trascurabile, ma addirittura con probabilità certa. Si pensi infatti ad un avversario che invia come vettori $m_0 = (0^n, 0^n)$ e $m_1 = (0^n, 1^n)$. Banalmente, il ciphertext vector conterrà due elementi uguali nel primo caso e due elementi distinti nel secondo, cosa che permette ad A di capire in modo trivialmente certo quale vettore è stato cifrato. In questo caso, è il determinismo di *Enc* in Π^G a rivelarsi fatale.

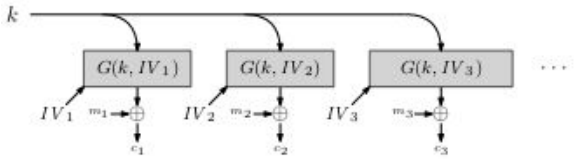
Teorema
Se *Enc* è deterministico, allora qualsiasi schema Π = (*Gen, Enc, Dec*) non può essere sicuro rispetto a $PrivK^{mult}$.

Questo non significa che i cifrari di flusso siano inutili quando usati con codifiche multiple, è sufficiente modificare leggermente la definizione di schema di codifica rendendo *Enc* stateful.

Consideriamo la chiave k come uno stream pseudo-casuale. Ciascun messaggio viene cifrato con una porzione di stream diversa, di fatto aggirando la limitazione di usare sempre la stessa chiave. Lo stato consiste nel ricordarsi la posizione all'interno dello stream. Questa modalità è nota col nome di *Modo Operativo Sincrono*.



Il *Modo Operativo Asincrono* invece richiede generatori pseudocasuali sicuri in senso più forte di quello visto. In particolare, generatori che dipendono non solo da k, ma anche da un initialization vector (IV).



Sicurezza contro attacchi CPA

Chosen-Plaintext Attacks: l'attaccante ha accesso sia in lettura che in scrittura ai messaggi che lo schema di codifica invierà sul canale.

Fino ad ora l'avversario costruiva m_0 ed m_1 ma ricevuto c non poteva far altro che cercare di identificarlo. Nello spirito CPA, ora si supporrà che l'avversario abbia accesso ad un *oracolo* per $Enc_k(\cdot)$. Avere accesso ad un oracolo significa avere la possibilità di invocare una funzione durante qualsiasi momento nell'algoritmo, senza conoscere né la funzione né k. Per il resto, l'esperimento rimane sostanzialmente identico e sarà chiamato $PrivK^{CPA}$. L'oracolo sarà invocato nella seconda fase, quando si analizza c. In generale l'attaccante conosce la funzione *Enc* per il principio di Kerckhoff.

$PrivK^{eav}_{A, \Pi}$ (n)

$k \leftarrow Gen(1^n)$;

$(m_0, m_1) \leftarrow A(1^n, Enc_k(\cdot))$; L'avversario ha sempre accesso all'oracolo Enc_k

$b \leftarrow \{0, 1\}$;

$c \leftarrow Enc_k(m_b)$;

$b^* \leftarrow A(c)$;

Result: $\neg(b \oplus b^*)$

Definizione

Uno schema di codifica Π si dice *sicuro rispetto ad attacchi* CPA (o CPA-sicuro) sse per ogni avversario A esiste una funzione trascurabile ε tale che:

$$Pr(PrivK^{CPA}_{A, \Pi}(n) = 1) \leq 1/2 + \varepsilon(n)$$

Lemma

Ogni schema Π sicuro rispetto a $PrivK^{CPA}$ è sicuro rispetto a $PrivK^{eav}$.

Per dimostrare questo si deve dimostrare che esiste un avversario per $PrivK^{CPA}$ che abbia la stessa probabilità di successo di $PrivK^{eav}$, banalmente questo è $PrivK^{CPA}$ senza l'uso dell'oracolo.

Lemma

Ogni schema Π = (Gen, Enc, Dec) che sia **sicuro rispetto** a **$PrivK^{CPA}$** deve essere tale per cui ***Enc*** è **probabilistico**.

Teorema

Ogni schema di codifica che sia **CPA-sicuro** rimane tale anche in presenza di **codifiche multiple**.

Si utilizza la *pseudocasualità* per produrre schemi di codifica CPA-sicuri, ma non come sono stati visti finora in quanto deterministici. Bisogna passare ad una nozione diversa: la **funzione pseudocasuale**, per la quale servono alcune nozioni preliminari:

- Si useranno *funzioni parziali binarie*, ossia funzioni parziali da {0, 1}* x {0, 1}* a {0, 1}* (in cui una stringa è la chiave e l'altra il messaggio).

- Queste funzioni vogliamo siano parziali, ovvero non definite su tutti i valori del suo dominio, ma solo su alcuni. Una funzione parziale binaria F preserva la lunghezza quando F(k, x) è definita sse |k| = |x| e in tal caso |F(k, x)| = |x|. La funzione restituisce sempre stringhe di lunghezza uguale all'input (|input| = |output|).
- Data una funzione parziale binaria F che preserva la lunghezza, indichiamo con F_k la funzione da $\{0, 1\}^{|k|}$ a $\{0, 1\}^{|k|}$ definita nel modo naturale. Si vuole che scelta k in modo casuale, F_k sia indistinguibile da una funzione veramente casuale.
- Una funzione parziale binaria è efficientemente calcolabile sse esiste un algoritmo polytime che la calcola.

Consideriamo lo spazio delle funzioni da $\{0, 1\}^n$ a $\{0, 1\}^n$, tale spazio è *finito* ad ha cardinalità $2^{n \cdot 2^n}$ (numero finito di funzioni) in quanto si hanno n bit (ovvero 2^n possibili valori di input) che possono produrre OGNUNO a sua un valore di n bit (2^n possibili valori in output) per un totale di $2^{n \cdot 2^n}$. Ha quindi senso pensare alla distribuzione uniforme su tale spazio, la quale assegna probabilità $1 / 2^{n \cdot 2^n}$ a ciascuna tale funzione.

Definizione (Funzione Pseudocasuale)
Data una funzione F parziale binaria, che preserva la lunghezza ed efficientemente calcolabile, diciamo che F è detta **funzione pseudocasuale** (FP) sse per ogni distinguitore D PPT esiste una funzione trascurabile ε tale che:

$$|Pr(D^{F_k(\cdot)}(1^n) = 1) - Pr(D^{f(\cdot)}(1^n) = 1)| \leq \varepsilon(n)$$

Quando si scrive $D^{F_k(\cdot)}$ si sta dando a D l'accesso in forma di oracolo ad $F_k(\cdot)$. Viene passato anche 1^n al distinguitore perché vogliamo sia a conoscenza della lunghezza delle stringhe con cui ha a che fare. Si noti che k è scelta tra tutte le stringhe lunghe n in modo casuale, mentre $f(\cdot)$ è scelta tra tutte le funzioni da $\{0, 1\}^n$ a $\{0, 1\}^n$ in modo casuale.

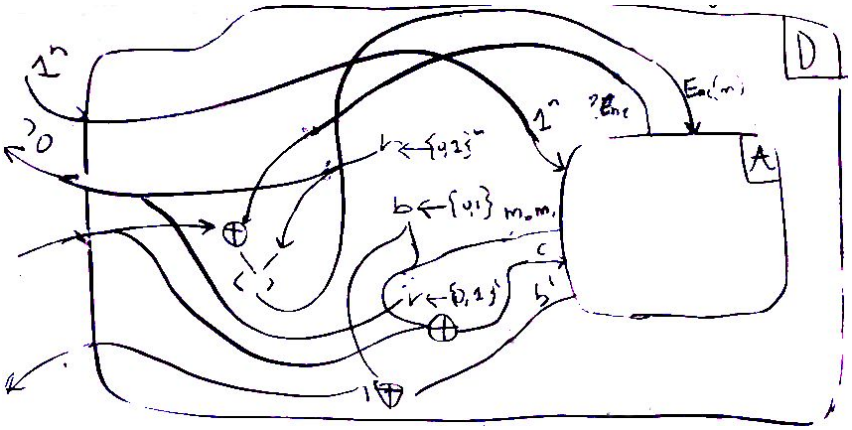
Definizione (Schema Indotto da una Funzione Pseudocasuale)
Data una funzione pseudocasuale F, lo schema Π^F = (*Gen, Enc, Dec*) è definito come segue:
- L'algoritmo *Gen* su input 1^n produce in output ciascuna stringa lunga n con identica probabilità, ossia $1 / 2^n$.
- *Enc*(m, k) è definita come la (codifica binaria della) coppia <r, $F_k(r) \oplus m$> dove r è una stringa casuale lunga |k| bit.
- *Dec*(c, k) ritorna $F_k(r) \oplus s$ ogniqualvolta c è la (codifica binaria della) coppia <r, s>.

Purtroppo anche qui la lunghezza della chiave è uguale alla lunghezza dei messaggi, però si è ottenuta una garanzia maggiore rispetto alla definzione di sicurezza perfetta data all'inizio.

Teorema
Se F è una funzione pseudocasuale, allora Π^F è sicuro contro attacchi CPA.

Dimostrazione
Supponiamo di lavorare con $\hat{\Pi}$ che e' uno "schema di codifica" simile a Π^F ma in cui Gen sceglie una funzione casuale f tra tutte quelle da $\{0,1\}^n$ a $\{0,1\}^n$. Cercheremo di dimostrare che :
$$|Pr(PrivK^{CPA}_{A,\Pi^F}(n) = 1) - Pr(PrivK^{CPA}_{A,\hat{\Pi}}(n) = 1)| \leq \epsilon(n)$$

A partire da un avversario A costruiremo per induzione un distinguitore D per F .



Nel costruire D si vuole "ingannare" A, quando interagisce con D gli si deve dare la visione dell'ambiente uguale all'esperimento.
- Se l'oracolo che "passiamo" a D la funzione pseudocasuale F_k dove k casuale:

(1) $Pr(D^{F_k(\cdot)}(1^n) = 1) = Pr(PrivK^{CPA}_{A,\Pi^F}(n) = 1)$

- Se l'oracolo che "passiamo" a D la funzione casuale $f(\cdot)$ allora:

(2) $Pr(D^{f(\cdot)}(1^n) = 1) = Pr(PrivK^{CPA}_{A,\hat{\Pi}}(n) = 1)$

La disequazione che volevamo dimostrare all'inizio vero perch valgono (1), (2) e l'ipotesi di pseudocasualit di F.
Il fatto che (1) e (2) siano ad una distanza non trascurabile l'ipotesi iniziale.

Ora e' necessario dimostrare che:

$$Pr(PrivK^{CPA}_{A,\hat{\Pi}}(n) = 1) \leq \frac{1}{2} + \frac{q(n)}{2^n}$$

Con le funzioni pseudocasuali non abbiamo la certezza (a differenza dei generatori) che in un punto del dominio non si produca un risultato uguale alla funzione in un altro punto del dominio.

Definiamo allora: **Repeat** = "il challenge-ciphertext c = $< r, s >$ e' tale per cui r e' una, tra le stringhe restituite dall'oracolo (come primo argomento)."

Se **Repeat** vale:

$< r^*, f(r) \oplus m > \sim m$

Challenge-ciphertext

$< r^*, f(r^*) \oplus m_b >$

In questo caso l'avversario vince la partita se riconosce che l'r ricevuto nel ciphertext e' uguale a quello dell'oracolo.

Se **Repeat** *non* vale:
L'unica cosa che l'avversario riesce a fare tirare a caso. Non puo' fare di meglio perche' non ha elementi. La probabilita' e' quindi $\frac{1}{2}$

$$\begin{aligned} Pr(PrivK^{CPA}_{A,\hat{\Pi}}(n) = 1) &= Pr(PrivK^{CPA}_{A,\hat{\Pi}}(n) = 1 \wedge \textbf{Repeat}) + \\ &\quad Pr(PrivK^{CPA}_{A,\hat{\Pi}}(n) = 1 \wedge \neg \textbf{Repeat}) \\ &\leq Pr(\textbf{Repeat}) + \frac{1}{2} \\ &\leq \frac{q(n)}{2^n} + \frac{1}{2} \end{aligned}$$

Messaggi di lunghezza variabile
Il cifrario Π^F è CPA-sicuro ogniqualvolta F è FP, ma può gestire solo messaggi di lunghezza pari alla lunghezza della chiave, soffre quindi delle stesse limitazioni viste per la sicurezza perfetta.
Esiste un modo per generalizzare ogni cifrario CPA-sicuro Π (che può essere anche Π^F ma non solo) per messaggi di lunghezza n (lunghezza chiave) ad un cifrario Π^* per messaggi *nl(n)* dove *l* è un polinomio:

$$Enc^*(k, m_0 || \dots || m_{l(n)}) = Enc(k, m_0) || \cdot || Enc(k, m_{l(n)})$$

Negli schemi CPA sicuri si può applicare lo stesso *Enc* con la stessa chiave a porzioni diverse del messaggio senza che crei problemi in quanto *Enc* è probabilistico.

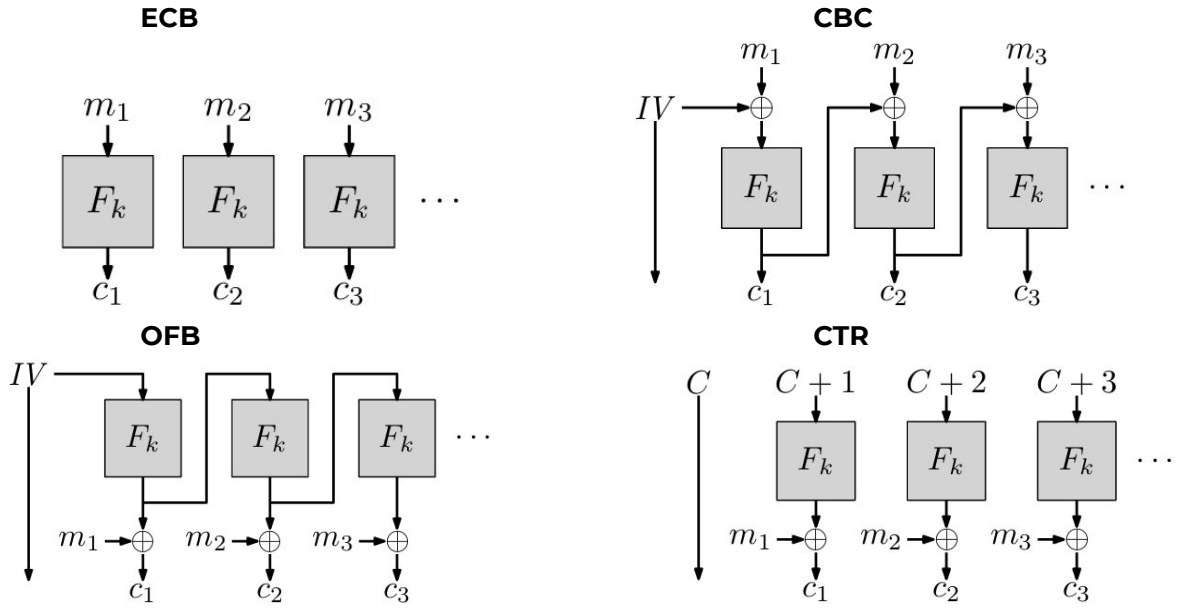
Teorema
Se Π è CPA-sicuro allora anche Π^* è CPA-sicuro (anche per messaggi di lunghezza variabile).

Una **permutazione** in un insieme X non è nient'altro che una funzione biettiva da X a X. Il numero di permutazioni nell'insieme $\{0, 1\}^n$ è $2^n!$, quindi un numero inferiore al numero di funzioni nell'insieme.

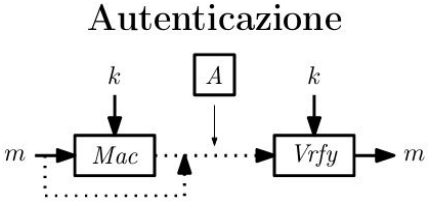
La nozione di permutazione pseudocasuale è data in un modo molto simile a quello di funzione pseudocasuale, richiedendo però che anche l'inversa di F_k sia efficientemente calcolabile. Talvolta però, gli avversari hanno accesso anche all'inversa della funzione (che esiste sempre), oltre che alla funzione stessa. In tal caso ha senso richiedere che:

$$| Pr(D^{F_k(\cdot), F_k^{-1}(\cdot)}(1^n) = 1) - Pr(D^{f(\cdot), f^{-1}(\cdot)}(1^n) = 1) | \leq \varepsilon(n)$$

Ottenendo così la nozione di permutazione fortemente pseudocasuale (si è dato accesso al distinguitore sia ad F che alla sua inversa, avrà quindi due oracoli).
I cifrari a blocco sono spesso pensati per essere permutazioni pseudocasuali.



Schemi di Autenticazione



Si vuole garantire che il ricevente sia sicuro dell'integrità e dell'autenticità del messaggio che riceve.

Message Authentication Codes
Esattamente nello stesso senso in cui i cifrari sono lo strumento atto a garantire la confidenzialità, i *message authentication codes* (MAC) sono strumento atto a risolvere il problema dell'autenticazione.

Un MAC è una tripla Π = (*Gen, Max, Vrfy*) di algoritmi PPT tali che:
- *Gen* prende in input una stringa nella forma 1^n, da interpretare come il parametro di sicurezza e produce in output una chiave k, che possiamo essere tale che $|k| \geq n$.
- L'algoritmo *Mac* prende in input una chiave k e un messaggio, producendo in output un tag *t*.
- L'algoritmo *Vrfy* prende in input una chiave k, un messaggio *m* e un tag *t*, producendo un booleano *b* in output.
Un MAC Π = (*Gen, Mac, Vrfy*) è *corretto* quando *Vrfy*(k, m, *Mac*(k, m)) = m.

In questo caso non si vuole che l'avversario sia in grado di *forgiare* (ovvero produrre un tag valido per) un messaggio di sua scelta *m*, senza conoscere la chiave k. Per questo si assume che:
- L'avversario abbia accesso ad un oracolo per $Mac_k(\cdot)$.
- Ma che una coppia (m, t) ottenuta attraverso l'accesso all'oracolo non sia da considerarsi una forgiatura corretta.
- Che l'avversario debba essere, come al solito, un algoritmo PPT.

Procediamo, come al solito, dando un'opportuna nozione di esperimento:
MaxForge*$_{A,\,\Pi}$*(n)
k ← *Gen*(1^n);
(m, t) ← A(1^n, $Mac_k(\cdot)$);
Q ← {m | A interroga $Mac_k(\cdot)$ su m};
Result: (m ∉ Q AND *Vrfy*(k, m, t) = 1)

Definizione
Un **MAC** Π si dice **sicuro** sse per ogni avversario PPT A esiste una funzione $\varepsilon \in$ NGL tale che:

$$Pr(MaxForge_{\Pi, A}(n) = 1) = \varepsilon(n)$$

Il primo esempio di MAC sicuro si basa su un concetto già conosciuto, quello di Funzione Pseudocasuale (il MAC diventa la FP).

Definizione (MAC Indotto da una Funzione Pseudocasuale)
Data una funzione pseudocasuale *F*, il MAC Π^F = (*Gen, Mac, Vrfy*) è definito come segue:
- L'algoritmo *Gen* su input 1^n produce in output ciascuna stringa lunga n con identica probabilità, ossia $1 / 2^n$.
- *Mac*(k, m) = F_k(m).
- Vrfy(k, m, t) = (F_k(m) ?= t).

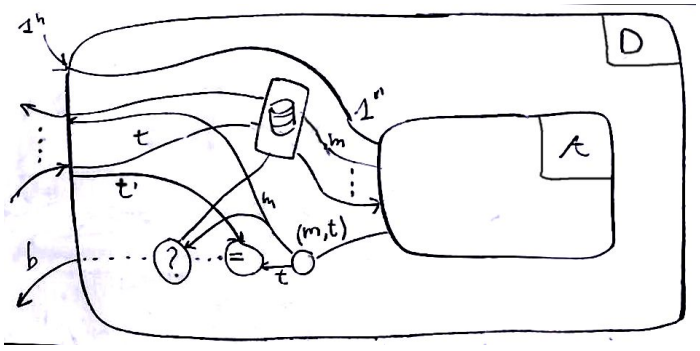
?= restituisce true se l'equivalenza è vera, altrimenti false.

Teorema
Se F è pseudocasuale, allora il *MAC* Π^F è sicuro.

Dimostrazione
E' sufficiente costruire un $\tilde{\Pi}$ come un *Mac* costruito in modo del tutto simile a Π^F ma in cui *Gen* genera una funzione *f* e *Mac* la applica al messaggio in input.

$Pr(MacForge_{A,\tilde{\Pi}}(n) = 1) \leq 2^{-n}$ dimostrata
$|Pr(MacForge_{A,\tilde{\Pi}}(n) = 1) - Pr(MacForge_{A,\Pi^F}(n) = 1)| \leq ngl(n)$

Per assurdo: $Pr(MacForge_{A,\Pi^F}(n) = 1) \leq \epsilon(n)$



Osserviamo che, per come abbiamo costruito D, vale che:
$Pr(D^{F_k(\cdot)}(1^n) = 1) = Pr(MacForge_{A,\Pi^F}(n) = 1)$
$Pr(D^{f(\cdot)}(1^n) = 1) = Pr(MacForge_{A,\tilde{\Pi}}(n) = 1)$

Entrambe queste equazioni valgono per mera definizione di D.

Gestire Messaggi a Lunghezza Variabile
Dato un messaggio $m = m_1 || \ldots || m_n$ si può procedere come segue:
1. *Mac*($\oplus_{i=1}^{n} m_i$). No perché l'avversario può forgiare un $p = p_1 || \ldots || p_n$ dove $\oplus_{i=1}^{n} m_i = \oplus_{i=1}^{n} p_i$ ovvero riesce a trovare un messaggio con uguale XOR dei blocchi.

2. *Mac*(k, m) = $\oplus_{i=1}^{n}$*Mac*(k, m_i). No perché un avversario può forgiare il messaggio $p = m_{\pi(1)} \ldots m_{\pi(n)}$ dove π è una permutazione.

3. Mac(k,m) = $\oplus_{i=1}^{n}$ *Mac*(k, m_i || i). No perchè l'attaccante potrebbe facilmente forgiare altri messaggi.

Soluzione
Dato un MAC Π = (*Gen, Mac, Vrfy*) per messaggi a lunghezza fissa, possiamo costruire un nuovo MAC Π^* = (*Gen, Mac*, Vrfy**) per messaggi a lunghezza variabile come segue:

Mac*(k, m)
$m_1 || \dots || m_d \leftarrow m$;
/* such that $|m_i| = n / 4$ */
$l \leftarrow |m|$;
$r \leftarrow \{0, 1\}^{n/4}$;
for i ← 1 **to** d **do**
  $t_i \leftarrow Mac(k, r||l||i||m_i)$
Result: $(r, t_1, \dots, t_d)$

Vrfy*(k, m, (r, t_1, . . . , t_d))
$m_1 || \dots || m_d \leftarrow m$;
/* such that $|m_i| = n / 4$ */
$l \leftarrow |m|$;
for i ← 1 **to** d **do**
  if $Vrfy(k, r||l||i||m_i) = 0$ **then**
    Result: 0
Result: 1

Teorema
Se Π è sicuro, allora Π* è anch'esso sicuro.

CBC-MAC
La costruzione appena vista, se applicata ad una FP *F*, chiama quest'ultima 4d volte, mentre il tag è lungo 4dn. Si può ovviare a tale inconveniente tramite la costruzione CBC-MAC, sempre basata su una FP *F* ma definita in modo diverso e parametrizzata su l.

Mac^{CBC}(k, m)
$l \leftarrow l(|k|)$;
$m_1 || \dots || m_l \leftarrow m$;
/* such that $|m_i| = |k|$ */
$t_0 \leftarrow 0^n$;
for i ← 1 **to** l **do**
  $t_i \leftarrow F_k(t_{i-1} \oplus m_i)$
Result: t_l

$Vrfy^{CBC}$(k, m, t)
if $l(|k|) \cdot |k| \neq |m|$ **then**
  Result: 0
Result: t ?= $Mac^{CBC}(k, m)$

È cruciale per la sicurezza dello schema che il valore di inizializzazione sia fissato (0^n) in questo caso, e per il principio di Kerckhoff è conosciuto a tutti. C'è questa catena di chiamate alla funzione F_k in cui si manda solo l'ultimo "pezzettino".

La lunghezza dei tag così diventa pari alla lunghezza della chiave, che è molto inferiore alla lunghezza dei messaggi.

Teorema
Se l è un polinomio e F è una FP, allora Π^{CBC} è un MAC sicuro.

Funzioni Hash
Le funzioni hash sono funzioni che comprimono stringhe lunghe in stringhe più corte in modo che ci siano meno collisioni possibile. Una collisione per una funzione hash *H* è una coppia (x, y) tale che $H(x) = H(y)$ con $x \neq y$.
Le collisioni esistono per come sono costruite le funzioni hash, si vuole solo che non solo siano in minor numero possibile, ma in un certo senso impossibili da determinare (anche da parte di avversari costruiti in modo specifico per trovarle).

Più in dettaglio una funzione hash è vista come una funzione $H : \{0, 1\}^* \times \{0, 1\}^* \to \{0, 1\}^*$ in cui: il primo parametro è una chiave s (resa pubblica) e il secondo è una stringa x che la funzione *H* cerca di comprimere.

Definizione (Funzione Hash)
Una *funzione hash* è una coppia di algoritmi PPT (*Gen*, *H*) tali che:

- *Gen* è un algoritmo che prende in input un parametro di sicurezza 1^n e restituisce una chiave s (da cui n possa essere efficientemente calcolato).
- Esiste un polinomio *l* tale che *H*(s, x) restituisce una stringa di lunghezza pari a *l(n)* (dove n è il parametro implicito in s).

La nozione più forte e più restrittiva di funzione hash sicura, si basa sul seguente esperimento:
HashColl $_{A,\ \Pi}$ ***(n)***
s ← *Gen*(1^n);
(x, y) ← A(s);
Result: $(x \neq y) \wedge (H(n) = H(y))$

Definizione
Una funzione hash Π = (*Gen, H*) è detta **resistente alle collisioni** sse per ogni avversario PPT A esiste una funzione trascurabile ε tale che:

$$\Pr(\text{HashColl}_{A,\ \Pi}(n) = 1) \leq \varepsilon(n)$$

Resistenza alla Seconda Preimmagine
Dati s e x, deve essere impossibile per A costruire y ≠ x tale che $H^s(x) = H^s(y)$.

Resistenza alla Preimmagine
Dati s e z = $H^s(x)$ deve essere impossibile per A costruire y (anche y = x) tale che $H^s(y) = z$.

<u>Attacchi Birthday</u>
Si pensi ad attacchi brute force per una funzione hash, cercando la relativa complessità. Consideriamo in tal senso uno scenario in cui la chiave s sia già fissata e in cui l'avversario A voglia trovare una collisione $H^s : \{0, 1\}^* \to \{0, 1\}^l$. L'avversario potrebbe, semplicemente, scegliere in modo casuale q stringhe in {0, 1}*, testare H^s su ognuna di esse, sperando di trovarne due sulle quali H^s restituisce lo stesso valore. Se $q > 2^l$ c'è la certezza di trovare una collisione.
C'è un'osservazione da fare: nel caso in cui l'output di H^s sia uniformemente distribuito è il *caso peggiore* per noi, e per questo possiamo applicare il teorema del compleanno.

Teorema
Dati q valori casuali *scelti uniformemente* in un insieme finito di cardinalità N, la probabilità che due di essi siano identici è q^2 / N.

Se assumiamo che il comportamento di H^s sia più possibile vicino a quello di una funzione casuale (ossia se ci troviamo nel caso peggiore), possiamo quindi concludere che un attacco birthday dove $q = \Theta(2^{l/2})$ avrà probabilità di successo:

$$\Theta\left(\frac{(2^{\frac{\ell}{2}})^2}{2^\ell}\right) = \Theta\left(\frac{2^\ell}{2^\ell}\right) = \Theta(1)$$

È $\Theta(1)$ grazie al teorema del compleanno. Quando una probabilità è una costante (≠ 0) significa che è molto facile amplificarla perché diventi uguale ad 1.

Trasformazione di Merkle-Damgård
Per poter fare in modo che una funzione hash per messaggi di lunghezza fissa possa gestire messaggi di lunghezza arbitraria non si fa altro che applicare la funzione hash più volte a porzioni diverse del messaggio.

Supponiamo che (*Gen, H*) sia una funzione hash per messaggi di lunghezza *p(n) = 2n* (e l'output è lungo n) e costruiamo a partire da essa (*Gen,* H^{MD}) nel modo seguente.

$H^{MD}(s, x)$
B ← round(|x| / n);
$x_1 || \cdots || x_B \leftarrow x$;
/* such that $|x_i| = n$ */
$x_{B+1} \leftarrow |x|$;
$z_0 \leftarrow 0^n$;
for i ← 1 to B + 1 do
  $z_i \leftarrow H(s, z_{i-1} || x_i)$
Result: z_{B+1}

Si inserisce anche la lunghezza del messaggio. Non si parte da un vettore z_0 random se no la funzione sarebbe non deterministica (probabilistica) e bisognerebbe anche tenerne traccia.

Teorema
Se (*Gen, H*) è resistente alle collisioni, allora anche (*Gen, H^{MD}*) lo è.

Il messaggio però non può essere più lungo di 2^n altrimenti non si può rappresentare la sua lunghezza con una stringa lunga al massimo n.

Il metodo più semplice di utilizzo delle funzioni hash è per creare *Mac*, nello specifico perché questi ultimi siano adatti a messaggi di lunghezza variabile.
Dato un MAC Π = (*Gen, Mac, Vrfy*) e una funzione hash (*Gen', H*) definiamo il MAC Π^H = (*Gen^H, Mac^H, $Vrfy^H$*) come segue:
- *Gen^H* su input 1^n ritorna (la codifica di) una coppia (s, k) dove s è il risultato di *Gen'*(1^n) e *k* è il risultato di *Gen*(1^n).
- *Mac^H*((s, k), m) ritorna *Mac_k*(*H^s*(m)).
- Come al solito, *Vrfy*((s, k), m, t) ritorna 1 sse *Mac^H*((s, k), m) = t.

Teorema
Se Π è un MAC sicuro e (Gen', H) è resistente alle collisioni, allora Π^H è sicuro.

Teorema
Se Π = (*Gen, Mac, Vrfy*) è un MAC sicuro, allora Π^* = (*Gen*, Mac*, Vrfy**) è anch'esso sicuro.

Dimostrazione
Sia *Mac**(k, m) <r, $t_1 \ldots t_d$> con t_i = *Mac*(k, r || l || i || m_i)
Si vuole dimostrare che Pr($MacForge_{A, \Pi^*}(n) = 1$) è negligibile.
Repeat = "Lo stesso *r* occorre almeno due volte tra quelli prodotti dall'oracolo in $MacForge_{A,\Pi^*}(n)$."
NewBlock = "Se $< r, t_1, \ldots, t_d >$ e' l'output dell'avversario *A* in $MacForge_{A,\Pi^*}(n)$, allora esiste un *i* tale che t_i non e' ottenuto forgiando una stringa nella forma $r || i || l || m_i$ diverse da tutte quelle in cui *A* chiama l'oracolo."

$$Pr(MacForge_{A,\Pi^*}(n) = 1) =$$
$$Pr(MacForge_{A,\Pi^*}(n) = 1 \wedge Repeat \wedge NewBlock) +$$
$$Pr(MacForge_{A,\Pi^*}(n) = 1 \wedge Repeat \wedge \neg NewBlock) +$$
$$Pr(MacForge_{A,\Pi^*}(n) = 1 \wedge \neg Repeat \wedge NewBlock) +$$
$$Pr(MacForge_{A,\Pi^*}(n) = 1 \wedge \neg Repeat \wedge \neg NewBlock) \leq$$
$$Pr(Repeat) +$$
$$Pr(MacForge_{A,\Pi^*}(n) = 1 \wedge NewBlock) +$$
$$Pr(MacForge_{A,\Pi^*}(n) = 1 \wedge \neg Repeat \wedge \neg NewBlock)$$

[Con i ¬ prima: mi sbarazzo di NewBlock poi mi sbarazzo di Repeat, quello che rimane prima è un Repeat, quello che rimane dopo un NewBlock e infine entrambi i ¬ per sbarazzarsi di quel che rimane]

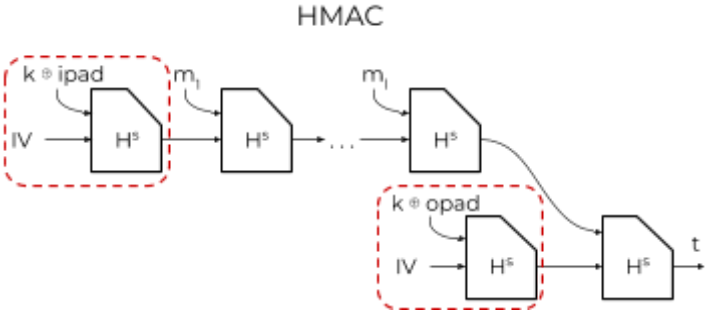
Lemma
$Pr(Repeat)$ *e' negligible*.

Dimostrazione
$Pr(Repeat) = \frac{q^2}{2^{\frac{n}{2}}}$ che e' negligible in quanto q e' un polinomio.

Lemma
$Pr(MacForge_{A,\Pi^*}(n) = 1 \wedge \neg Repeat \wedge \neg NewBlock) = 0$

Quello che manca e' dimostrare che
$Pr(MacForge_{A,\Pi^*}(n) = 1 \wedge NewBlock)$ e' trascurabile
Possiamo costruire un avversario A' per Π a partire da A in modo tale che $Pr(MacForge_{A',\Pi}(n) = 1) = Pr(MacForge_{A,\Pi^*}(n) = 1 \wedge NewBlock)$

Caso di studio: HMAC
Qui si ha bisogno di utilizzare più di un livello di hashing, si avrà un livello in cui porzioni successive del messaggio verranno passate ad H^s, ma c'è bisogno di un passaggio di hash iniziale e due finali. Si hanno due costanti: ipad e opad, poi si ha il vettore di inizializzazione IV e k la chiave. Per garantire la sicurezza del MAC che otteniamo, i due blocchi cerchiati in rosso (inizio e penultimo) è necessario abbiano proprietà di pseudocasualità.



Hash and MAC
$Mac^{HM}((k, s), m) = Mac_k(H^s(m))$

Teorema
Se Π è sicuro e H è resistente alle collisioni, allora Π^{HM} è sicuro.

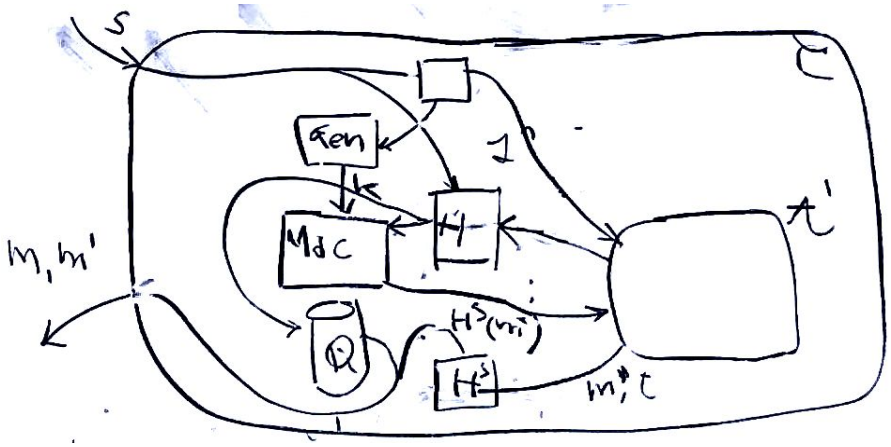
Dimostrazione

Consideriamo un avversario A' per Π^{HM} e sia $MacForge_{A',\Pi^{HM}}(n)$ l'esperimento di nostro interesse.
Consideriamo un evento probabilistico *Coll* definito nel modo seguente: se m^* il messaggio prodotto in output da A' (che possiamo considerare non tra quelli in $\mathbb{Q}$), l'evento ci dice che esiste $m \in \mathbb{Q}$ tale che $H^s(m) = H^s(m^*)$.

$Pr(MacForge_{A',\Pi^{HM}}(n) = 1) =$
$Pr(MacForge_{A',\Pi^{HM}}(n) = 1 \wedge Coll) +$
$Pr(MacForge_{A',\Pi^{HM}}(n) = 1 \wedge \neg\, Coll) \leq$
$Pr(Coll) + Pr(MacForge_{A',\Pi^{HM}}(n) = 1 \wedge \neg\, Coll)$

Lemma
$Pr(Coll)$ trascurabile.

Costruiamo un avversario C per H tale che la probabilita' di successo di C sia $Pr(Coll)$.

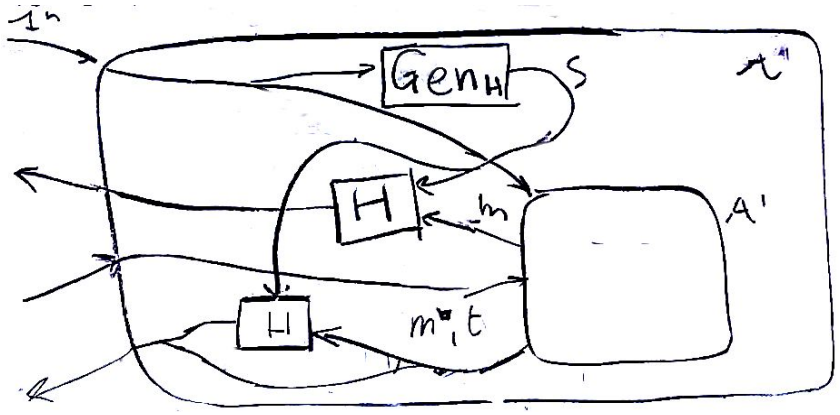


Creiamo noi l'oracolo per A' (tramite Mac, e H).
Analizzando il modo in cui abbiamo definito C, ci rendiamo conto che:
- La visione di A' subroutine di C e' la stessa di A' in $MacForge_{A',\Pi^{HM}}$.
- C produce una collisione esattamente e solo quando vale *Coll*.

$Pr(HashColl_{C,H}(n) = 1) =$ **Pr(Coll)**

Lemma
$Pr(MacForge_{A',\Pi^{HM}}(n) \wedge \neg\, Coll)$ e' trascurabile.

Costruiamo un avversario per Π, chiamiamolo *A*, e partiamo da *A'*.



$Pr(MacForge_{A,\Pi}(n) = 1)$
$Pr(MacForge_{A',\Pi^{HM}}(n) = 1)$
I precedenti non sono uguali perche' m ed m^* possono essere subito diversi, ma dopo l'hash con H si produce un risultato uguale ($Coll$). Se al secondo elemento si aggiunte un $\neg$ $Coll$ varrebbe l'uguaglianza.

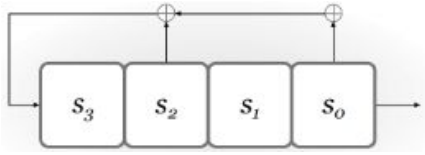
$Pr(MacForge_{A,\Pi}(n) = 1) = Pr(MacForge_{A',\Pi^{HM}}(n) = 1 \wedge \neg Coll)$
Entrambi trascurabili.

Costruzione Concreta di Oggetti Pseudocasuali e Funzioni Hash

Si descriveranno primitive concrete, che seppur non dimostrabilmente pseudocasuali (o resistenti alle collisioni), sembrano avere buone caratteristiche.

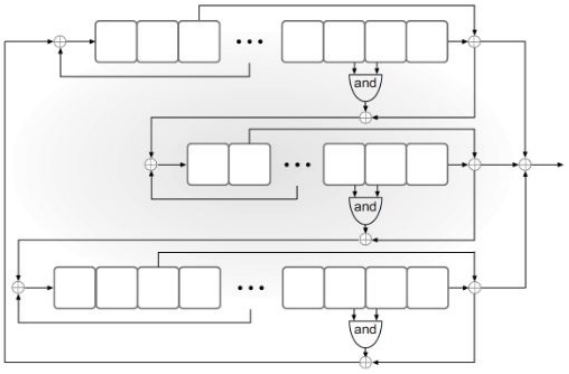
Costruire cifrari di flusso
I cifrari di flusso (o generatori pseudocasuali a lunghezza variabile) sono costituiti da una coppia di algoritmi (*Init, GetBits*) in cui:
- L'algoritmo *Init* inizializza lo stato interno, a partire da una chiave (e opzionalmente da un vettore di inizializzazione IV).
- L'algoritmo *GetBits* produce in output un singolo bit, modificando contemporaneamente lo stato interno.
Applicando iterativamente *GetBits* allo stato ottenuto tramite *Init* si possono ottenere stream di bit arbitrariamente lunghi.

- Registri a scorrimento retroazionato lineare
Sono un particolare tipo di cifrario di flusso, in cui: lo stato è costituito da n bits $s_{n-1}\ s_{n-2} \ldots s_1\ s_0$. *GetBits* calcola un bit *b* come lo XOR di *alcuni* dei bit nello stato, produce in output il valore del bit meno significativo s_0, fa scorrere verso destra gli altri bit, e reinizializza s_{n-1} a b. Quali bit utilizzare per produrre il nuovo stato sono noti e fanno parte dell'algoritmo stesso. La chiave è data dallo stato iniziale e dai coefficienti retroazione.



Questo tipo di cifrario non è sicuro. Prima di tutto bisogna osservare che il comportamento "futuro" di un LFSR è completamente determinato da suo stato. Nel caso in cui tutti gli stati fossero uguali a 0, tutto quello che verrebbe prodotto successivamente sarebbero tutti 0. Per costruzione si evita che gli stati abbiano tutti zeri. Esistono 2^n stati, e un LSDR che cicli sui $2^n - 1$ stati diversi da 0 . . . 0 è detto a *lunghezza massima*, perché il periodo ha la sua lunghezza massima. Gli LSFR a lunghezza massima hanno ottime proprietà statistiche, e sono gli unici ad essere interessanti da un punto di vista crittografico.
La debolezza degli LSFR risiede nel seguente attacco: a partire dai primi 2n output $y_1 \ldots y_{2n}$ è possibile ricostruire completamente lo stato iniziale (ossia $y_1 \ldots y_n$) e i coefficienti di feedback (tramite un sistema di equazioni lineari che si sa risolvere efficientemente). Infatti i primi n bit di output sono lo stato iniziale, e tramite un sistema di equazioni si risale con i successivi n bit di output i coefficienti di feedback (gli XOR effettuati). Quindi si perde la pseudocasualità. Un modo per risolvere la situazione è inserire funzioni non lineari (ad esempio lo XOR si effettua con stati ripetuti) o tra gli stati o sull'elemento di output prima che sia reso noto.

Trivium
Vi sono sempre i registri, ma qui sono 3 blocchi che si intrecciano per generare il nuovo stato e di conseguenza l'output. Questo cifrario è particolarmente adatto ad essere implementato su hardware.



RC4
RC4 è un cifrario pensato per essere implementato via software anziché via hardware. Il suo stato interno consiste in un vettore di byte di lunghezza pari a 256, che rappresenta una permutazione dell'insieme {0, . . . , 255}, assieme a due valori i, j ∈ {0, . . . , 255}, che vanno pensati come indici (puntatori).
In questo cifrario *Init* ha come input una chiave, di 16 bytes, mentre l'output è lo stato iniziale, all'interno viene costruita la permutazione dell'insieme di 256 bytes prima generando gli elementi in ordine e poi effettuando degli swap tra alcuni elementi. Per quanto riguarda invece *GetBits* si prende in input lo stato corrente, mentre in output sarà lo stato della permutazione S in una certa coordinata. All'interno ovviamente viene modificato lo stato, tramite un incremento di i di 1 e di j di S[i] per poi effettuare uno swap di S[i] con S[j].

Init algorithm for RC4

Input: 16-byte key k
Output: Initial state (S, i, j)
(Note: All addition is done modulo 256)

for $i = 0$ to 255:
  $S[i] := i$
  $k[i] := k[i \bmod 16]$
$j := 0$
for $i = 0$ to 255:
  $j := j + S[i] + k[i]$
  Swap $S[i]$ and $S[j]$
$i := 0, \quad j := 0$
return (S, i, j)

GetBits algorithm for RC4

Input: Current state (S, i, j)
Output: Output byte y; updated state (S, i, j)
(Note: All addition is done modulo 256)

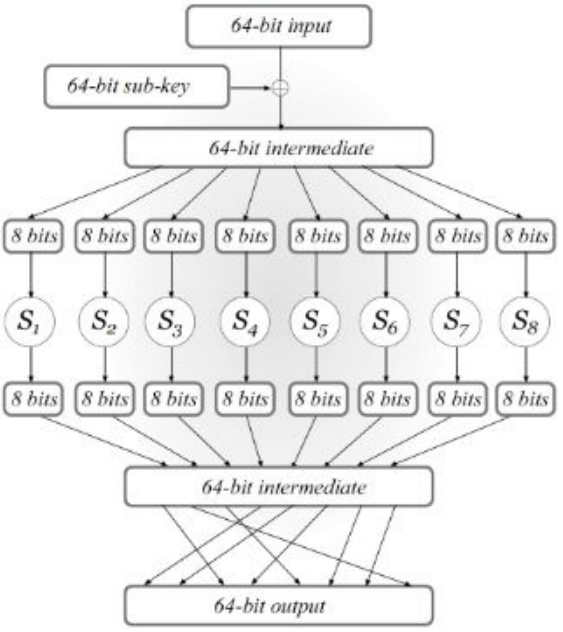
$i := i + 1$
$j := j + S[i]$
Swap $S[i]$ and $S[j]$
$t := S[i] + S[j]$
$y := S[t]$
return $(S, i, j), y$

Tuttavia per RC4 esistono attacchi statistici, ad esempio nei primi byte prodotti in output vi sono più bit probabili che altri. RC4 veniva usato come cifrario di flusso per il sistema di cifratura in WEP nelle reti wifi.

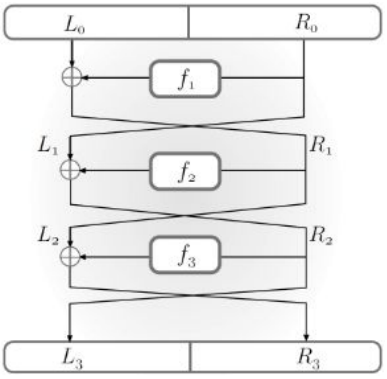
<u>Costruire cifrari a blocco</u>
Ci si occuperà di funzioni nella forma $F : \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^l$, (prende in input chiave lunga n e messaggio lungo l e si produce in output un messaggio pseudocasuale lungo l) in cui n non è necessariamente uguale ad l, inoltre n ed l sono costanti per cui è necessario utilizzare l'approccio concreto.
Gli attacchi contro i cifrari a blocco sono i soliti quattro tipi: ciphertext-only, known-ciphertext, chosen-plaintext e chosen-ciphertext, e quasi sempre gli attacchi *key-recovery*.

- Substitution-Permutation Networks (SPN)
Al messaggio in input viene applicata, iterativamente, una trasformazione, essa stessa ottenuta come composizione di: mixing con la chiave (o parte di essa) (tipicamente tramite XOR), i

cosiddetti S-BOX (trasformazioni che mappano pochi bit (al più 10) in pochi bit) e una permutazione (solitamente fissa).

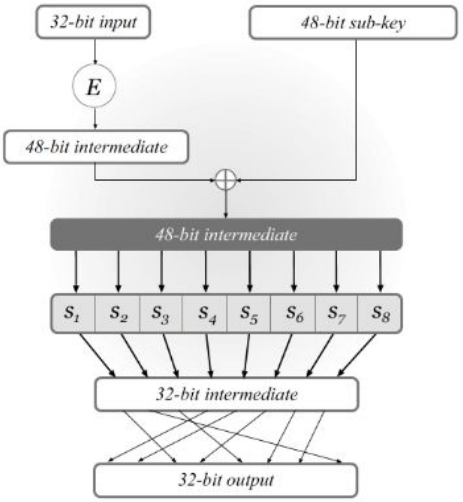


VI sono linee guida per la costruzione delle SPN, che sono necessarie ma non sufficienti per garantire la sufficienza.
- Gli *S-BOX* devono essere *invertibili*: se così non fosse, non ci sarebbe speranza di costruire permutazioni, cosa spesso necessaria.
- Occorre garantire l'*effetto valanga*: una modifica di un bit dell'input di una S-BOX deve propagarsi in almeno due bit dell'output.
Così facendo se il numero di round è sufficientemente alto, una modifica di un bit nel messaggio di ripercuoterà potenzialmente su *tutti i bit* dell'output. Questa caratteristica è necessaria alla pseudocasualità.



- Reti di Feistel
Sono un modello molto simile alle SPN, in queste ad ogni round il sottostante messaggio m viene suddiviso in due sotto-messaggi di uguale lunghezza m_L e m_R dopo il round avremo che il nuovo messaggio $p = p_L \cdot p_R$ sarà tale che $p_L = m_R$ e $p_R = f(m_R) \oplus m_L$ dove f è una funzione che tipicamente dipende dalla chiave, detta *Mengler function*. Questo modo di costruire i round ha l'interessante effetto di permettere ad f di non essere invertibile.

DES
Le chiavi in DES sono lunghe 56 bit, mentre i messaggi sono lunghi 64 bit, in altre parole può essere visto come una funzione: $F_{DES} : \{0, 1\}^{56} \times \{0, 1\}^{64} \to \{0, 1\}^{64}$. DES è strutturato come una rete di Feistel a 16 round (prende in input 32 bit, ovvero la metà del messaggio), in cui una particolare Mengler function *f* prende in input (anche) porzioni diverse della chiave a seconda del round. Le S-BOX di DES non sono invertibili in quanto "rimpiccioliscono" l'output.



Il cuore di DES sono le otto S-BOX, ciascuna è una funzione $\{0, 1\}^6 \to \{0, 1\}^4$. Ogni possibile configurazione $s \in \{0, 1\}^4$ è l'immagine di esattamente quattro configurazioni in input. Le S-BOX sono anche progettate per garantire l'effetto valanga e sono resistenti alla crittografia differenziale.

NOTA: la crittoanalisi differenziale, a differenza di quelle classiche non si basa sul valore dei messaggi ma sulle differenze tra i messaggi. Su messaggi che differiscano di un certo numero di bit, certi valori della chiave sono più probabili di altri. Per questo tipo di analisi si ha bisogno di un oracolo (chosen-plaintext).

Per DES esistono attacchi semplici e molto performanti se si considerano varianti di DES in cui il numero di round è molto basso. Se si considera un DES a 16 round invece l'unico attacco

applicabile rimane brute-force. In conclusione DES non è più da considerarsi sicuro, non per suoi difetti strutturali ma per la lunghezza della propria chiave (troppo corta, facilmente recuperabile con le risorse disponibili oggi).

Una soluzione può essere incrementare la lunghezza della chiave: double encryption, ovvero $DDES_{k\_1,\ k\_2}(m) = DES_{k\_2}(DES_{k\_1}(m))$. Tuttavia questa soluzione è vulnerabile ad attacchi meet-in-the-middle, che hanno complessità nell'ordine di 2^{56}. In questi attacchi si determina per ciascun messaggio e per ciascuna chiave quali siano i possibili crittogrammi che stanno tra la prima cifratura DES e la seconda portando la complessità da 2^{112} a 2^{56}. Si può pensare allora ad una triple encryption, ovvero applicare DES tre volte sulle 3 chiavi, lo stesso si può applicare anche su 2 chiavi, questo elimina l'attacco meet-in-the-middle.
La variante *Triple DES* ha chiavi lunghe 112 bit, in cui viene applicato 3 volte DES ma avvengono anche altre operazioni che lo rendono sicuro (è infatti uno Standard).

AES
Si tratta di una SPN con lunghezza del messaggio pari a 128 bit e lunghezza della chiave di 128, 192 oppure 256 bit. In ogni round, lo stato è visto come una matrice 4 x 4 di byte che inizialmente è pari al messaggio ($4 \cdot 4 \cdot 8 = 128$). Ogni round prevede l'applicazione di quattro trasformazioni:
- *AddRoundKey*: una sotto-chiave lunga 128 bit viene messa in XOR con lo stato.
- *SubBytes*: ogni byte nella matrice viene sostituito con il byte ottenuto applicando una S-BOX fissa.
- *ShiftRows*: ciascuna riga della matrice viene shiftata verso sinistra di un numero di posizioni variabile.
- *MixColumns*: a ciascuna colonna della matrice viene applicata una trasformazione lineare invertibile.
L'attacco concreto più performante rimane quello brute force con ordine 2^{128}.

Costruzione di funzioni hash
Le funzioni hash possono essere viste come funzioni della forma $H : \{0,1\}^* \to \{0,1\}^l$. Quasi sempre queste funzioni sono costruite a partire da una funzione di compressione $C : \{0,1\}^{n+l} \to \{0,1\}^l$ e applicando a quest'ultima una trasformazione simile a quella di Merkle-Damgård.

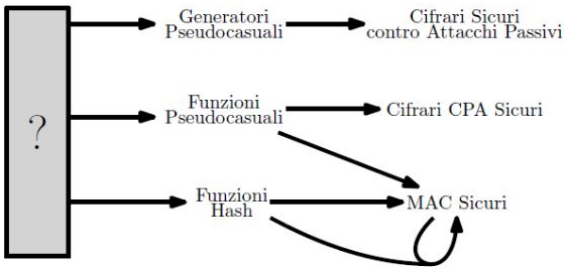
- *Costruzione di Davies-Meyer*
Dato un cifrario a blocco $F : \{0,1\}^n \times \{0,1\}^l \to \{0,1\}^l$ un modo naturale per costruire una funzione di compressione a partire da F è quello di definire $C : \{0,1\}^{l+n} \to \{0,1\}^l$ come segue:

$$C(k \,||\, m) = F_k(m) \oplus m$$

Teorema
Se F è modellata come un *cifrario ideale*, allora la **costruzione di Davies-Meyer** induce una funzione hash resistente alle collisioni, in senso concreto: ogni attaccante di complessità q non può trovare collisioni con probabilità superiore a $q^2 / 2^l$.

Alcuni esempi di funzioni hash sono:
- MD5: l'output è lungo a 128 ed è da considerarsi non sicura.
- SHA1 e SHA2: nel primo l'output è 160 bit e sono disponibili attacchi che richiedono 2^{80} invocazioni della funzione, nel secondo l'output può essere lungo 256 bit o 512 bit (SHA256 e SHA512), queste funzioni sono ottenute tramite la costruzione di Davies-Meyer a partire da cifrari a blocco appositamente definiti (non cifrari a blocco già noti).
- SHA3: supporta output di lunghezza pari a 256 e 512 bit ed è costruita attorno a principi radicalmente diversi dalle costruzioni precedentemente incontrate.

Gli attacchi in questo caso consistono nel trovare collisioni.



Costruzione astratta di oggetti pseudocasuali e funzioni hash

Con questo approccio si costruiscono oggetti pseudocasuali a partire da altre entità, la cui esistenza non è certa, ma è ritenuta molto probabile. In particolare, tali entità sono "problemi difficili" o "operazioni difficili", per i quali si suppone non esistano algoritmi polytime. Crittografia e teoria della complessità si occupano, in tal senso, della dimostrazione che simili algoritmi non esistano. Lo stato delle cose è lo schema a destra.
Dove il punto interrogativo rappresenta per l'appunto l'insieme delle entità su cui fondiamo gli oggetti pseudocasuali della colonna centrale. Vediamo alcune delle entità in questo insieme.

Funzioni One-Way
Informalmente, una funzione è one-way se è facile da calcolare, ma difficile da invertire. Formalmente, ci basiamo su un esperimento:

Invert-A-f(n)
$x = \{0, 1\}^n$;
$y = f(x)$;
$z = A(1^n, y)$;
Result: $(f(z) = y)$;

Definizione
Una funzione $f : \{0, 1\}^* \to \{0, 1\}^*$ si dice **one-way** se esiste un algoritmo deterministico e polytime che calcola f e se per ogni avversario PPT *A* esiste un $\varepsilon \in NGL$ tale che:

$$Pr(Invert_{A, f}(n) = 1) \leq \varepsilon(n)$$

Una sottoclasse delle funzioni one-way sono le permutazioni one-way: funzioni one-way che preservano la lunghezza e che godono dell'interessante proprietà, caratteristica delle permutazioni, che ciascun $y \in \{0, 1\}^*$ determina univocamente un x tale che $f(x) = y$.
Esempi di (presunte) funzioni one-way includono:
- *Moltiplicazione tra naturali*: la funzione $f(x, y) = x \cdot y$ è one way, con opportuni vincoli su x e y e (e.g. non-trivialità, non-parità, etc).
- *Subset-sum problem*: la funzione $f(x_1, \dots, x_n, J) = (x_1, \dots, x_n, \Sigma_{j \in J} x_j)$, con $|x_j| = n$ e J è un sottoinsieme di $\{1, \dots, n\}$. Come ben sappiamo, il problema di determinare J, dati solo $(x_1, \dots, x_n, \Sigma_{j \in J} x_j)$ (i.e. il subset-sum problem) è NP-completo.

Una funzione one-way è tale per cui f(x) non necessariamente cela interamente x. Non è impossibile, secondo la definizione di one-way, ricavare alcune informazioni su x. E.g. Se f(x) è one-way, allora lo è anche $g(x, y) = (x, f(y))$, pur essendo una funzione che "rivela" sempre metà dei propri input.

Definizione
Un predicato *hc* : {0, 1}* → {0, 1} è detto **predicato hardcore** di una funzione *f* se e solo se *hc* è calcolabile in tempo polinomiale e per ogni avversario PPT vale (dove ε è trascurabile):

$$\Pr(A(f(x)) = hc(x)) \leq 1/2 + \varepsilon(n)$$

Teorema Goldreich-Levin
Se esiste una funzione (permutazione) one-way *f*, allora esistono una funzione (permutazione) one-way *g* e un predicato hardcore *hc* per *g*. La funzione *g* si costruisce ponendo g(x, y) = (f(x), y) e il suo hc_G corrisponde a

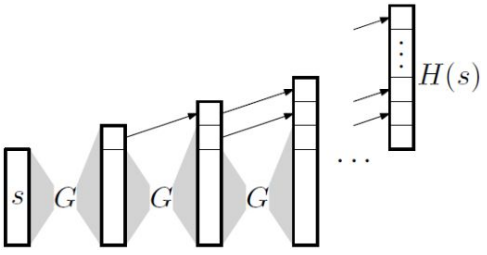
$$hc_G(x, y) = \oplus_{i=1}^{n} x_i \cdot y_i$$

dove $x = x_1 \dots x_n$ e $y = y_1 \dots y_n$. La dimostrazione è non-triviale.

Da permutazioni One-Way a GP
Teorema
Sia f una permutazione one-way e sia *hc* un predicato hardcore per essa. Allora abbiamo che G, definito ponendo G(s) = (f(s), hc(s)) è un **generatore pseudocasuale** con fattore di espansione l(n) = n + 1. I primi |s| bit dell'output sono pseudocasuali per la natura di f, l'ultimo bit è pseudocasuale per come è definito un predicato hardcore.

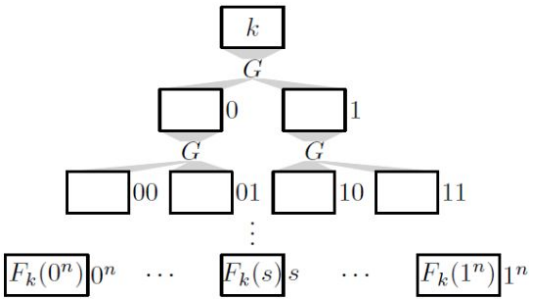
Teorema
Se esiste un generatore pseudocasuale G con fattore di espansione l(n) = n + 1, allora esiste un altro generatore pseudocasuale H con fattore di espansione arbitrario, purché polinomiale.

L'idea alla base della costruzione:



Teorema
Se esiste un generatore pseudocasuale G con fattore di espansione l(n) = 2n, allora esiste una funzione pseudocasuale.

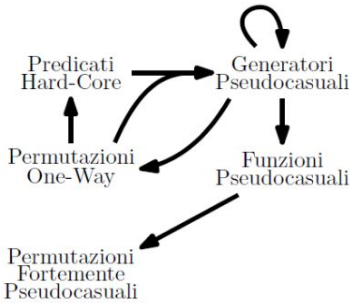
L'idea alla base della costruzione:



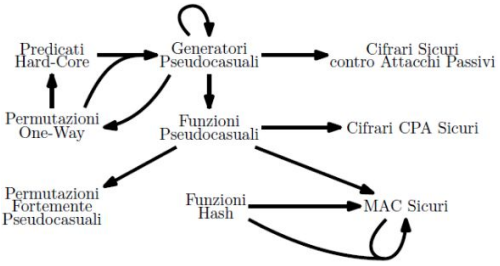
Teorema
Se esiste una funzione pseudocasuale, allora esiste una permutazione fortemente pseudocasuale.

Teorema
Se esiste un generatore pseudocasuale, allora esiste una funzione one-way.

Questo ultimo teorema "chiude il cerchio", ovvero prova che l'esistenza di funzioni one-way, GP, FP e PP sono logicamente equivalenti (nonostante tale esistenza non sia dimostrata).



Il quadro complessivo:



Purtroppo, il lato sinistro di questo schema è esclusivamente teorico.

Algebra, Teoria dei Numeri e Relative Assunzioni

L'insieme $\mathbb{Z}$ è l'insieme dei numeri interi, mentre $\mathbb{N}$ è l'insieme dei numeri naturali. Dati due elementi $a, b \in \mathbb{Z}$, scriviamo $a \mid b$ quando esiste $c \in \mathbb{Z}$ con $ac = b$. Quando $a \mid b$ e a è positivo, diciamo che a è *divisore* di b, che quando a $\notin \{1, b\}$ è detto *fattore* di b. Un numero $p \in \mathbb{N}$ con $p > 1$ è detto *primo* se non ha fattori, in caso contrario *composto*. Dati $n, m \in \mathbb{N}$ dove $m > 1$, indichiamo con $n \bmod m$ il resto della divisione di n per m.

Lemma
Se $n, m \in \mathbb{N}$ e $m > 1$, allora n è **invertibile modulo** m (esiste p tale che $np \bmod m = 1$) ogniqualvolta $\gcd(n, m) = 1$, ossia quando n, m sono primi tra loro (es. 12 e 25).

Dato un numero naturale $N \in \mathbb{N}$, ci chiediamo se sia difficile determinare due interi $p, q \in \mathbb{N}$ tali che $N = pq$. Possiamo quindi definire il seguente esperimento:
wFactor$_A$(n)
$(x, y) \leftarrow \mathbb{N} \times \mathbb{N}$ with $|x| = |y| = n$;
$N \leftarrow x \cdot y$;
$(z, w) \leftarrow A(N)$;
Result: $z \cdot w = N$

Al solito, la nostra assunzione potrebbe stare nel fatto che per ogni A PPT esiste $\varepsilon \in NGL$ tale che:

$$\Pr(\text{wFactor}_A(n) = 1) = \varepsilon(n)$$

L'assunzione che abbiamo appena dato, però, semplicemente non vale. Con probabilità pari a 3/4, il numero N sarà pari, perché basta che uno tra x e y sia pari perché N stesso lo sia. Fattorizzare un numero pari è semplicissimo (si prende N/2 e 2). Occorre garantire che N non sia (in media) banalmente fattorizzabile. Un'idea interessante è quella di modificare *wFactor* in modo che x e y siano sempre e solo numeri primi (rappresentabili in n bit).

Un modo plausibile per generare numeri primi rappresentabili in esattamente n bit è quello di andare per tentativi:
for i ← 1 to t **do**
r ← $\{0,1\}^{n-1}$;
p ← 1 || r; Si aggiunge 1 in testa per assicurarci sia lungo n bit
if p è primo **then**
  Result: p
Result: fail

Tuttavia ci sono due punti su cui soffermarsi:
- Come valorizzare il parametro t? Ci piacerebbe definire t come un polinomio in n, per ragioni di efficienza. vorremmo che per tale valore di t la probabilità di ottenere *fail* fosse trascurabile (in n).
- Come testare la primalità di un numero? Vorremmo poter testare se p è primo in tempo polinomiale.

Teorema
Esiste una costante c tale che per ogni n > 1 il numero di primi rappresentabili in esattamente n bit è almeno pari a $c \cdot 2^{n-1} / n$.

Quindi ad ogni iterata la *probabilità che p sia effettivamente primo* sarà almeno:

$$\frac{c \cdot 2^{n-1}/n}{2^{n-1}} = \frac{c \cdot 2^{n-1}}{n \cdot 2^{n-1}} = \frac{c}{n}$$

Di conseguenza, se $t = n^2 / c$, la probabilità di ottenere *fail* sarà al più pari a (ogniqualvolta n≥c):

$$\left(1 - \frac{c}{n}\right)^t = \left(\left(1 - \frac{c}{n}\right)^{\frac{n}{c}}\right)^n \leq (e^{-1})^n = e^{-n}$$

Per quanto riguarda la primalità esistono algoritmi deterministici per il test di primalità che prendono tempo polinomiale. Inoltre il cosiddetto test di Miller-Rabin è invece probabilistico, ma è PPT.

L'assunzione che stiamo cercando di definire sarà parametrizzata su un algoritmo, chiamato *GenModulus* che, su input 1^n, produce in output una tripla (N, p, q) dove *N = pq* e *p, q* siano primi ad *n* bits. L'esperimento *wFactor* diventa il seguente:
Factor*$_{A, GenModulus}$*(n)
(N, p, q) ← *GenModulus*(1^n);
(r, s) ← A(N);
Result: r·s = N

Diciamo che la fattorizzazione è difficile rispetto a GenModulus sse per ogni algoritmo A che sia PPT esiste una funzione trascurabile $\varepsilon \in NGL$ tale che:

$$\Pr(\text{Factor}_{A, \text{GenModulus}}(n) = 1) = \varepsilon(n)$$

Quest'assunzione è sufficiente a derivare una funzione one-way, ma non a dimostrare la sicurezza di schemi a chiave pubblica.

Teoria dei Gruppi
Un *gruppo* è una struttura algebrica $(\mathbb{G}, \circ)$ dove $\circ$ è un'operazione binaria che sia associativa, con identità *e* e in cui ogni $g \in \mathbb{G}$ abbia un inverso g^{-1}. Un gruppo finito $(\mathbb{G}, \circ)$ è detto avere ordine pari a $|\mathbb{G}|$. Un gruppo si dice abeliano se $\circ$ è un'operazione commutativa. L'operazione binaria è spesso indicata:
- Con il simbolo di addizione + ; in tal caso il gruppo si chiama additivo, e se $m \in \mathbb{N}$ si può usare la notazione:

$$mg = \underbrace{g + \cdots + g}_{m \text{ volte}}$$

- Con il simbolo di moltiplicazione · ; il tal caso il gruppo si chiama moltiplicativo, e se $m \in \mathbb{N}$ si può scrivere:

$$g^m = \underbrace{g \cdot \ldots \cdot g}_{m \text{ volte}}$$

Il calcolo di mg o di g^m può essere eseguito tramite un numero di operazioni polinomiale in |m|, ossia logaritmico in *m*. I fattori possono essere calcolati velocemente, ad esempio $g^{2\wedge 3} = ((g^2)^2)^2$, l'elevamento al quadrato è un'operazione semplice.

Teorema
Se $(\mathbb{G}, \cdot)$ ha ordine m, allora per ogni $g \in \mathbb{G}$, esiste un g tale che $g^m = 1_{\mathbb{G}}$.

Corollario
Se $(\mathbb{G}, \cdot)$ ha ordine $m > 1$, allora per ogni $g \in \mathbb{G}$ e per ogni i, $g^i = g^{[i \bmod m]}$.

Gruppi Finiti (esempi)
L'insieme $\mathbb{Z}_N = \{0, \ldots, N-1\}$ è un gruppo se la sottostante operazione è l'addizione modulo *N* ovvero la mappa che ad (*a, b*) fa corrispondere *a + b mod N*. L'identità è 0; L'inverso di $n \in \mathbb{Z}_N$ è N−n.
L'insieme $\mathbb{Z}_N$ diventa un gruppo con la moltiplicazione modulo *N* quando: eliminiamo 0 (che non è invertibile) dal gruppo; ed *N* è primo. Ciò garantisce che ogni $1 < n < N$ sia invertibile modulo N.
Se consideriamo $\mathbb{Z}_N$ con *N* composto, c'è un modo per rendere questo insieme un gruppo rispetto alla moltiplicazione modulo *N*: basta considerare $\mathbb{Z}^*_N \subseteq \mathbb{Z}_N$ definito come $\{n \in \mathbb{Z}_N \mid gcd(n, N) = 1\}$.
Tutti i gruppi considerati sono *abeliani*.
Un esempio di $\mathbb{Z}^*_N$ è $\mathbb{Z}^*_6 = \{1, 5\}$

La funzione che ad ogni naturale $N > 1$ associa la cardinalità di $\mathbb{Z}^*_N$ è detta **funzione di Eulero** ed è indicata con Φ:

$$\Phi(N) = |\mathbb{Z}^*_N|$$

Chiaramente $1 \leq \Phi(N) < N$. Ma possiamo dire qualcosa di più:
- Se N è un numero primo p, allora $\Phi(N) = p - 1$ perché ogni n compreso tra 1 e p − 1 è primo con p, ovvero gcd(n, p) = 1.
- Se N è il prodotto di due primi p e q, allora $gcd(a, N) \neq 1$ precisamente quando $p \mid a$ oppure $q \mid a$.
Quindi:

$$\mathbf{\Phi(N)} = N - 1 - (p - 1) - (q - 1) = pq - p - q + 1 = p(q - 1) - 1(q - 1) = \mathbf{(p - 1)(q - 1)}$$

Teorema

Sia N > 1. Per ogni naturale $e > 0$, definiamo $f_e : \mathbb{Z}^*_N \to \mathbb{Z}^*_N$ ponendo $f_e(x) = x^e \bmod N$. Se $\gcd(e, \Phi(N)) = 1$, allora f_e è una permutazione. Inoltre se d è l'inverso di e (modulo $\Phi(N)$), allora f_d è l'inversa di f_e.

Osserviamo come dati *e* ed *N*, il valore di $f_e(x)$ sia efficientemente calcolabile a partire da x, è sufficiente applicare l'algoritmo di esponenziazione.
Osserviamo anche dati *e* e $\Phi(N)$, l'inverso *d* di e modulo $\Phi(N)$ sia efficientemente calcolabile. L'inversione modulo un qualunque intero è un problema trattabile.
Infine, osserviamo come dato *N* prodotto di due primi p e q il valore di $\Phi(N)$ non sia invece facilmente calcolabile. Lo sarebbe se si potesse (efficientemente) fattorizzare N.

Assunzione RSA
Questa assunzione predica la *difficoltà di invertire* $f_e : \mathbb{Z}^*_N \to \mathbb{Z}^*_N$ quando *N* è il prodotto di due primi *p* e *q* e $\gcd(e, \Phi(N)) = 1$. A differenza dell'assunzione sulla fattorizzazione, il problema che stiamo assumendo difficile diventa facile se oltre ad *N* si conoscono anche *p* e *q*, oppure se si conosce un inverso *d* di *e* (modulo $\Phi(N)$). L'assunzione RSA è parametrizzata su una routine GenRSA che, dato un input 1^n ritorna:
- Un naturale N che sia prodotto di due primi *p* e *q* con $|p| = |q| = n$.
- Un naturale *e* tale che $\gcd(e, \Phi(N)) = 1$.
- Un naturale *d* tale che $ed \bmod \Phi(N) = 1$.

L'esperimento *RSAInv* è parametrizzato su un avversario e sulla routine GenRSA:
$RSAInv_{A,GenRSA}(n)$
$(N, e, d) \leftarrow GenRSA(1^n)$;
$y \leftarrow \mathbb{Z}^*_N$;
$x \leftarrow A(N, e, y)$;
Result: $x^e \bmod N = y$

Diciamo che il problema RSA è difficile rispetto a GenRSA sse per ogni avversario A che sia PPT esiste $\varepsilon \in NGL$ tale che:

$$Pr(RSAInv_{A, GenRSA}(n) = 1) = \varepsilon(n)$$

Come abbiamo già avuto modo di argomentare, se A riuscisse a fattorizzare *N*, riuscirebbe a calcolare $\Phi(N)$ e quindi a calcolare *d*, invertendo di conseguenza f_e.

Per costruire GenRSA potremmo per esempio procedere come segue:
GenRSA(n)
$(N, p, q) \leftarrow GenModulus(1^n)$;
$M \leftarrow (p - 1)(q - 1)$;
$e \leftarrow \{1, \ldots, M\}$ tale che $\gcd(e, M) = 1$;
$d \leftarrow e^{-1} \bmod M$;
Result: (N, e, d)

Se GenRSA è costruita in questo modo a partire da GenModulus, è possibile dimostrare che dall'Assunzione RSA segue l'Assunzione sulla Fattorizzazione dei Numeri Primi.

Gruppi Ciclici

Consideriamo un gruppo moltiplicativo finito $(\mathbb{G}, \cdot)$, un suo elemento $g \in \mathbb{G}$ e costruiamo: $\langle g \rangle = \{g^0, g^1, g^2, \ldots\} \subseteq \mathbb{G}$
Sappiamo che $g^m = 1_{\mathbb{G}}$, per cui possiamo certo scrivere che $\langle g \rangle = \{g^1, \ldots, g^m\}$.
Ci potrebbe però essere un $i < m$ tale che $g^i = 1_{\mathbb{G}}$. Per ovvie ragioni $\langle g \rangle = \{g^1, \ldots, g^i\}$, e quindi $\langle g \rangle$ contiene al più i elementi. In realtà ne ha esattamente i; se $1 \leq k < j < i$, allora:

$$g^j = g^k \Rightarrow g^j \cdot g^{-k} = 1_{\mathbb{G}} \Rightarrow g^{j-k} = 1_{\mathbb{G}}.$$

L'ordine di $g \in G$ è il più piccolo naturale i tale che $g^i = 1$, ossia la cardinalità di $\langle g \rangle$.
Un gruppo finito $(\mathbb{G}, \cdot)$ si dice ciclico se esiste $g \in \mathbb{G}$ con $\langle g \rangle = \mathbb{G}$. Un tale g si dice generatore di $\mathbb{G}$.
Un esempio: in $(\mathbb{Z}_8, +)$ ci sono elementi come 1 che hanno ordine dell'elemento pari ad 8, mentre un numero come 4, per arrivare all'identità, che qui è 0, basta addizionarlo a se stesso 2 volte (4^2), quindi non è detto che tutti gli elementi di un gruppo finito abbiano ordine pari all'ordine del gruppo, solo alcuni, o a volte nessuno. Inoltre non è detto che tutti gli elementi di $\mathbb{G}$ generino $\mathbb{G}$ stesso, solo alcuni, a volte nessuno. Non troviamo nessun elemento in $\mathbb{Z}_8$ il cui ordine non divida 8. Se l'ordine m è primo, gli interi i che dividono l'ordine sono 1 ed m stesso.

Lemma
Se $\mathbb{G}$ ha ordine m e $g \in \mathbb{G}$ ha ordine i, allora $i \mid m$.

Se così non fosse, avremmo che $m = ik + j$ con $j < i$. Ma quindi:
$g^j = g^{j+ik-ik} = g^{m-ik} = g^m(g^i)^{-k} = 1_{\mathbb{G}}(1_{\mathbb{G}})^{-k} = 1_{\mathbb{G}}$

Teorema
Se $\mathbb{G}$ ha ordine primo allora $\mathbb{G}$ è ciclico e ogni $g \in G$ con $g \neq 1_{\mathbb{G}}$ genera $\mathbb{G}$.
Assunzione del Logaritmo Discreto
Se $\mathbb{G}$ è un gruppo moltiplicativo ciclico, allora esiste una corrispondenza biunivoca "naturale" tra $\mathbb{G}$ e $\mathbb{Z}_{|\mathbb{G}|}$. Ad ogni $h \in \mathbb{G}$ si può mettere in corrispondenza un unico $x \in \mathbb{Z}_{|\mathbb{G}|}$, ossia quello tale per cui $g^x = h$. Chiamiamo tale x il logaritmo discreto di h rispetto a *g*, che scriviamo $log_g h$.
Il problema del logaritmo discreto è semplicemente il problema di calcolare $\log_g h$ dati un gruppo ciclico $\mathbb{G}$, un generatore g per $\mathbb{G}$ e un elemento casuale h. L'esperimento con cui formalizzeremo l'assunzione del logaritmo discreto è parametrizzato, al solito, da una routine GenCG che dato 1^n costruisce un gruppo $\mathbb{G}$, di ordine q con $|q| = n$ e un generatore $g \in \mathbb{G}$.

L'esperimento DLog è definito come segue:
$\text{DLog}_{A, \text{GenCG}}(n)$
$(\mathbb{G}, q, g) \leftarrow \text{GenCG}(1^n)$;
$h \leftarrow \mathbb{G}$;
$x \leftarrow A(\mathbb{G}, q, g, h)$;
Result: $g^x = h$

Al solito, diciamo che l'assunzione del logaritmo discreto vale rispetto a GenCG sse per ogni avversario A PPT esiste $\varepsilon \in NGL$ tale che:

$$\Pr(\text{DLog}_{A, \text{GenCG}}(n) = 1) = \varepsilon(n)$$

Assunzione Computazionale di Diffie-Hellmann
Dato un gruppo ciclico $\mathbb{G}$ e un generatore $g \in \mathbb{G}$ per esso, definiamo la funzione $DH_g : \mathbb{G} \times \mathbb{G} \to \mathbb{G}$ come segue:

$$DH_g(h, j) = g^{(\log_g h)\cdot(\log_g j)}$$

Osserviamo come: $DH_g(g^x, g^y) = g^{x \cdot y} = (g^x)^y = (g^y)^x$
Il problema CDH consiste nel calcolare efficientemente DH_g dati un gruppo $\mathbb{G}$ e un generatore *g* per esso (prodotti tramite GenCG).

L'assunzione CDH (rispetto a GenCG) vale quando il problema CDH è difficile (rispetto allo stesso GenCG). Ogni algoritmo efficiente per il logaritmo discreto induce un algoritmo efficiente per CDH. Basta calcolare i logaritmi di *h* e *j*, moltiplicare i risultati ottenuti ed elevare *g* per il prodotto.

Informalmente, il problema DDH consiste nel distinguere $DH_g(h, j)$ da un arbitrario elemento del gruppo $\mathbb{G}$, dati ovviamente h e j. Formalmente, diciamo che il problema DDH è difficile (o che l'assunzione DDH è valida rispetto a GenCG) sse per ogni A PPT esiste ε trascurabile tale che:

$$|Pr(A(\mathbb{G}, q, g, g^x, g^y, g^z) = 1) - Pr(A(\mathbb{G}, q, g, g^x, g^y, g^{xy}))| \leq \varepsilon(n)$$

dove $(\mathbb{G}, q, g)$ è il risultato di $GenCG(1^n)$ e $x, y, z \in \mathbb{Z}_q$ sono casuali.
Ogni algoritmo efficiente per CDH induce trivialmente un algoritmo efficiente per DDH. Basta che A calcoli $DH_g(h, j)$ dove h e j sono il quarto e quinto parametro. Il risultato va poi confrontato con il sesto parametro.

Assunzioni di Diffie-Helmann su Gruppi Specifici
Prima di tutto è da osservare come l'utilizzo di gruppi con un numero primo di elementi è preferibile, questo perché: testare se un elemento è o meno un generatore è triviale, e perché è possibile dimostrare che se $\mathbb{G}$ è un gruppo di ordine primo q con $|q| = \Theta(2^n)$ allora:

$$Pr(DH_g(h, j) = y) = 1/q + \varepsilon(n)$$

Prendiamo poi in considerazione i gruppi $\mathbb{Z}_p^*$ dove *p* è un numero primo. Un algoritmo GenCG che generi gruppi di questo tipo esiste ed è efficiente. L'assunzione del logaritmo discreto vale per questo gruppo. DDH non si crede essere difficile per questi gruppi. Esiste però un diverso algoritmo GenCG' che restituisce un sottogruppo di $\mathbb{Z}_p^*$ e per il quale anche DDH si crede essere difficile.

Dalla Fattorizzazione alle Funzioni One-Way
Sia data una funzione GenModulus che utilizzi al più p(n), dove p è polinomio, bit casuali su input di lunghezza n, costruiamo un algoritmo che calcoli una funzione $f_{GenModulus}$ come segue:
- l'input è una stringa *x*;
- calcoliamo prima di tutto un intero n tale che $p(n) \leq |x| \leq p(n+1)$;
- calcola(N, p, q) come il risultato di $GenModulus(1^n)$ usando come bit casuali quelli in *x*, che sono sufficienti.
- restituisci N.

Osserviamo a questo punto come le seguenti distribuzioni sono identiche per ogni $m \in N$, il risultato N di $f_{GenModulus}(x)$ dove $x \in \{0, 1\}^m$ è scelta in modo casuale; il risultato N di $GenModulus(1^n)$ dove $p(n) \leq m \leq p(n + 1)$.

Teorema
Se la fattorizzazione è difficile rispetto a GenModulus, allora $f_{GenModulus}$ è una funzione one-way.

La Rivoluzione a Chiave Pubblica

Le principali limitazioni della crittografia a chiave simmetrica sono: la distribuzione delle chiavi, lo storage delle chiavi (dati U utenti sarebbe necessario memorizzare U^2 chiavi) e la gestione dei sistemi aperti (nuovi utenti che possono aggiungersi).

Per risolvere i primi due problemi si può usare un un sistema *Key Distribution Center (KDC)* in cui tutti gli utenti condividono una chiave segreta con il KDC, ogni volta che un utente A vuole comunicare con un utente B, invia uno speciale messaggio al KDC in cui richiede il rilascio di

una session-key che verrà mandata sia ad A che a B e al termine della sessione verrà cancellata. Le chiavi da salvare sono solo una (con il KDC) ma il sistema rappresenta un singolo point-of-failure.

Un protocollo per la distribuzione delle chiavi di sessione è Needham-Schroeder (NS):
1. A: Invia (A, B) al KDC
2. KDC: $k \leftarrow Gen$; Invia ($Enc(k_A, k)$, $Enc(k_B, k)$) ad A
3. A: Invia $Enc(k_B, k)$ a B

La rivoluzione avvenne quando Diffie e Hellman osservarono che è assolutamente plausibile uno scenario asimmetrico, in cui ci siano una chiave di cifratura, usata dal mittente, ed una chiave di decrittazione, usata dal destinatario. Sorprendentemente, la sicurezza dello schema di codifica deve valere anche contro avversari che conoscano la chiave di cifratura, che è anche detta chiave pubblica.

Diffie-Hellman proposero tre primitive a chiave pubblica:
- Crittografia a chiave pubblica (per la segretezza)
- Firma digitale (per l'autenticazione)
- Protocollo interattivo per lo scambio di chiavi

Protocolli per lo Scambio di Chiavi
Un protocollo per lo scambio delle chiavi è un insieme di regole Π che descrive in che modo due parti A e B si debbano scambiare un certo numero di messaggi. Al termine del protocollo A ha costruito una chiave k_A, mentre B ha costruito k_B. Il protocollo è corretto se $k_A = k_B$. La sicurezza di un protocollo è formulata, tramite un esperimento:

$KE_{A, \Pi}^{eav}(n)$
(transcript, k) $\leftarrow \Pi(1^n)$;
$b \leftarrow \{0, 1\}$;
if b = 0 then
  $k^* \leftarrow \{0, 1\}^n$
else
  $k^* \leftarrow k$
$b^* \leftarrow A(transcript, k^*)$;
Result: $\neg(b \oplus b^*)$

Come al solito Π è sicuro sse per ogni A PPT esiste $\varepsilon \in NGL$ tale che:
$Pr(KE_{A, \Pi}^{eav}(n) = 1) = 1/2 + \varepsilon(n)$
Il protocollo di Diffie-Hellman per lo scambio di chiavi è definito come segue:
1. A:
   $(\mathbb{G}, q, g) \leftarrow GenCG(1^n)$
   $x \leftarrow \mathbb{Z}_q$
   $h_1 \leftarrow g^x$
   Invia $(\mathbb{G}, q, g, h_1)$ a B
2. B:
   $y \leftarrow \mathbb{Z}_q$
   $h_2 \leftarrow g^y$
   Invia h_2 a B
   Output h_1^y
3. A:
   Output h_2^x

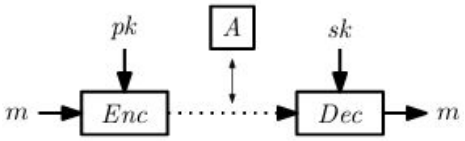
$k_B = h_1^y = (g^x)^y = g^{xy}$
$k_A = h_2^x = (g^y)^x = g^{xy}$

Teorema
Se DDH è difficile relativamente a GenCG, allora Π è **sicuro** (nel senso di KE^{eav}).

Esistono attacchi che l'esperimento KE^{eav} non cattura, ovvero attacchi di impersonificazione, ed attacchi man-in-the-middle (per questi Diffie-Hellman è noto essere insicuro).

Schemi di Codifica a Chiave Pubblica

Si ha una coppia di chiavi (*pk, sk*) dove *pk* è una chiave pubblica, viene usata dal mittente nella codifica dei messaggi, *sk* è una chiave segreta.



Alcuni svantaggi della chiave asimmetrica sono che questi schemi tendono ad essere ordini di grandezza meno performanti degli schemi a chiave simmetrica, ed occorre distribuire le chiavi pubbliche su canali autenticati (altrimenti un attaccante potrebbe mettersi in mezzo e fornire la sua chiave).

La definizione di schema di codifica Π = (*Gen, Enc, Dec*) deve essere opportunamente modificata:
- *Gen*: prende in input una stringa nella forma 1^n e produce in output una coppia di chiavi (pk, sk), tali che $|pk|, |sk| \geq n$ e tali che n possa essere inferito da pk o sk.
- *Enc*: prende in input un messaggio m e una chiave pubblica pk e produce in output un crittogramma.
- *Dec*: può essere probabilistico, prende in input un crittogramma c e una chiave segreta sk e produce in output un messaggio, oppure uno speciale simbolo ⊥.

Assumiamo che lo schema sia corretto, questa volta in senso probabilistico: deve esistere una funzione trascurabile ε tale che per ogni coppia (pk, sk) prodotta da $Gen(1^n)$ e per ogni n:

$$\Pr(Dec_{sk}(Enc_{pk}(m)) \neq m) \leq \varepsilon(n)$$

Spesso, Enc_k è definita solo per messaggi di lunghezza pari a n, oppure su tutto lo spazio $\{0,1\}^*$.
La nozione di esperimento va opportunamente modificata:
$PubK_{A,\Pi}^{eav}(n)$
(pk, sk) ← Gen(1^n);
(m_0, m_1) ← A(1^n, pk);
if $|m_0| \neq |m_1|$ then
  Result: 0
b ← {0, 1};
c ← *Enc*(k, m_b);
b^* ← A(c);
Result: ¬(b ⊕ b^*)

Definizione
Uno schema di codifica a chiave pubblica Π si dice sicuro contro attacchi passivi sse per ogni avversario PPT A esiste una funzione $\varepsilon \in NGL$ tale che:

$$\Pr(PubK_{\Pi,A}^{eav}(n) = 1) = 1/2 + \varepsilon(n)$$

l fatto che A abbia accesso a pk implica che A possa crittare qualunque messaggio, anche senza l'accesso ad oracoli. Inoltre dati pk e c = $Enc_{pk}(m)$, è sempre possibile ricostruire m avendo a disposizione tempo arbitrario.

Teorema
Se Π è sicuro contro attacchi passivi, allora è CPA-sicuro.

Teorema
Non esistono cifrari asimmetrici che siano sicuri in senso perfetto.

Sappiamo che ogni avversario passivo, avendo a disposizione pk, è di fatto anche attivo. Di conseguenza, tante proprietà che abbiamo visto valere nel caso simmetrico e per attacchi CPA, valgono anche qui.

Teorema
Nessuno schema a chiave pubblica in cui *Enc* sia deterministico può essere sicuro rispetto a $PubK^{eav}$.

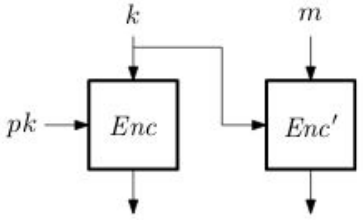
Molti schemi di codifica a chiave pubblica concreti hanno *Enc* deterministico, e questo ha conseguenze nefaste.

Codifiche Multiple
Analogamente a quanto visto nel caso simmetrico, possiamo parlare di sicurezza in presenza di codifiche multiple. Basta definire un nuovo esperimento $PubK^{mult}$ in cui l'avversario produca non una coppia di messaggi (m_0, m_1) ma una coppia di tuple di messaggi (m_0, m_1) dove $m_0 = (m_0^1, \ldots, m_0^t)$, $m_1 = (m_1^1, \ldots, m_1^t)$, e $|m_0^j| = |m_1^j|$. Al solito, uno schema di codifica a chiave pubblica Π si dice sicuro rispetto a codifiche multiple sse per ogni A PPT esiste ε con:

$$Pr(PubK_{\Pi, A}^{mult}(n) = 1) = 1/2 + \varepsilon(n)$$

Teorema
Se uno schema di codifica Π è sicuro rispetto $PubK^{eav}$, allora è sicuro rispetto a $PubK^{mult}$.

Codifica Ibrida
Con la codifica ibrida non si fa altro che cercare di mettere insieme i pregi delle codifiche a chiave pubblica e a chiave privata. Dati Π = (*Gen, Enc, Dec*) a chiave pubblica e Π' = (*Gen', Enc', Dec'*) a chiave privata, possiamo costruire Π^{Hy} in cui la codifica abbia più o meno l'aspetto seguente:



Nel definire la codifica ibrida, faremo l'assunzione che Gen' restituisca una stringa casuale in $\{0, 1\}^n$ e Π includa $\{0, 1\}^n$ tra lo spazio dei messaggi. Formalmente, lo schema Π^{Hy} è definito nel modo seguente a partire da Π e Π':

$Gen^{Hy}(1^n)$	$Enc^{Hy}(pk, m)$	$Dec^{Hy}(sk, (c, d))$
Result: $Gen(1^n)$	$k \leftarrow \{0,1\}^n$; $c \leftarrow Enc_{pk}(k)$; $d \leftarrow Enc_k(m)$; **Result**: (c, d)	$k \leftarrow Dec_{sk}(c)$; $m \leftarrow Dec_k(d)$; **Result**: m

Teorema
Se Π è CPA-sicuro e Π' ha codifiche indistinguibili, allora Π^{Hy} è sicuro.

La codifica ibrida porta due vantaggi principali:
1. Il *tempo di codifica*: supponiamo che la codifica della chiave richieda tempo α e che la codifica del messaggio richieda tempo β per ogni bit. Il tempo medio impiegato da Enc^{Hy} per ogni bit sarà quindi, per messaggi lunghi t, pari a $TIME(t) = (\alpha + \beta t) / t$ e osserviamo che con t che tende a infinito il tempo tende a β:

$$\lim_{t\to\infty} \frac{\alpha + \beta t}{t} = \beta$$

2. La *lunghezza dei crittogrammi*: al crescere di |m|, la lunghezza di (c, d) è lineare.

Schema di Codifica RSA
Textbook RSA

$Gen(1^n)$	$Enc((N, e), m)$	$Dec((N, d), c)$
$(N, e, d) \leftarrow GenRSA(1^n)$; **Result**: ((N, e),(N, d))	$c \leftarrow m^e \bmod N$; **Result**: c	$m \leftarrow c^d \bmod N$; **Result**: m

La correttezza dello schema discende dal fatto che se la coppia ((N, e), (N, d)) è ottenuta tramite *Gen*, allora f_d è l'inversa di f_e.

Tuttavia Textbook RSA <u>insicuro</u> rispetto alla nostra definizione, basti osservare che *Enc* sia deterministico. Vale però una nozione di sicurezza molto debole: data (N, e) chiave pubblica e $c = m^e \bmod N$, non è possibile determinare il messaggio m nella sua interezza, almeno quando vale l'*Assunzione RSA*. Da un punto di vista teorico, occorrerebbe garantire che $m \in \mathbb{Z}_N^*$, anche quando $m \in \mathbb{Z}_N$, codifica e decodifica funzionano. Si può inoltre dimostrare che $\phi(N)/N$, vista come funzione di n è nella forma $1 - \varepsilon(n)$.

Padded RSA
Questa versione <u>sicura</u> di RSA è costruita come segue:

$Gen(1^n)$	$Enc((N, e), m)$	$Dec((N, d), c)$
$(N, e, d) \leftarrow GenRSA(1^n)$; **Result**: ((N, e),(N, d))	$r \leftarrow \{0,1\}^{\lvert N\rvert - l(n) - 1}$; $c \leftarrow (r \,\|\|\, m)^e \bmod N$; **Result**: c	$m \leftarrow LSB(c^d \bmod N, l(n))$; **Result**: m

Dove l è una funzione tale che $|m| \leq l(n) \leq 2n - 2$ e LSB ritorta i bit meno significativi. Occorre scegliere l(n) sufficientemente piccolo, meno che lineare.

Teorema
Se vale l'*Assunzione RSA* rispetto a GenRSA e se $l(n) = O(\lg n)$, allora *Padded RSA* è **sicuro** rispetto ad attacchi passivi.

Schema di Codifica Elgamal
Questo schema può essere dimostrato sicuro a partire dall'*Assunzione DDH*.

L'osservazione da cui partire è il fatto che fissati un due elementi m, c ∈ $\mathbb{G}$ di un gruppo finito, la probabilità che un elemento casuale k ∈ $\mathbb{G}$ sia tale per cui m·k = g è pari a 1 / |$\mathbb{G}$|. Tutto questo si può facilmente dimostrare osservando che:

$$Pr(m \cdot k = c) = Pr(k = m^{-1} \cdot c) = 1 / |\mathbb{G}|$$

Siamo in altre parole in presenza di una situazione simile a quella per OTP.
Formalmente, lo schema di Elgamal è definito nel modo seguente:

Gen(1^n)
($\mathbb{G}$, q, g) ← GenCG(1^n);
x ← $\mathbb{Z}_q$;
sk ← ($\mathbb{G}$, q, g, x);
pk ← ($\mathbb{G}$, q, g, g^x);
Result: (sk, pk)

Enc((G, q, g, h), m)
y ← $\mathbb{Z}_q$;
Result: (g^y, h^y·m)

Dec((G, q, g, x), (c, d))
Result: d / c_1^x

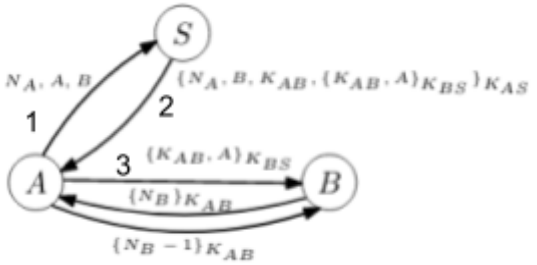
La correttezza dello schema è facile da dimostrare: $\frac{d}{c_1^x} = \frac{h^y \cdot m}{g^{yx}} = \frac{(g^x)^y \cdot m}{g^{xy}} = m$

Teorema
Se vale l'*Assunzione DDH* relativamente a GenCG, allora lo schema di Elgamal è **sicuro**.

La Crittografia Formale

I limiti della *crittografia computazionale* sono essenzialmente due: ragionare in modo probabilistico diventa difficile appena gli scenari diventano anche solo leggermente più complessi; e quando il numero di parti coinvolte cresce, è difficile capire anche solo come declinare il concetto di efficienza, centrale nell'approccio computazionale. Spesso i protocolli che vorremmo dimostrare sicuri hanno le seguenti caratteristiche: molteplici parti coinvolte, molteplici round di interazione e le primitive crittografiche (codifica, autenticazione) vengono utilizzate come subroutine.

Esempio: protocollo di **Needham-Schroeder**



N_A e N_B sono i cosiddetti "nonce", ovvero valori casuali generati rispettivamente da A e B.
Gli attacchi contro NS sono essenzialmente due:
1. Se un attaccante avesse a disposizione una "vecchia" chiave di sessione K_{AB}, potrebbe fare il replay del messaggio 3. B risponderebbe generando un nonce N_B, ma l'attaccante potrebbe decodificare il messaggio e produrre N_B−1. Così l'attaccante impersonifica A.
2. In assenza di B nel messaggio 2, un attaccante C potrebbe intercettare il messaggio 1, modificare la seconda componente facendola diventare C. Da quel punto in poi potrebbe impersonificare B senza che A se ne accorga.

In entrambi questi attacchi centrali, ciò che rende possibile l'attacco non è una fragilità nelle primitive crittografiche utilizzate, ma un errore di natura logica nella costruzione del protocollo stesso.

Il modello formale (o modello di Dolev-Yao) è un modello alternativo rispetto a quello computazionale:

	Computazionale	**Formale**
Messaggi	Stringhe binarie	Espressioni
Avversario	Algoritmi efficienti	Processi arbitrari
Attacco	Evento di probabilità non trascurabile	Evento possibile

Espressioni
Le espressioni del modello formale vanno pensate come alberi di derivazione e non come la relativa codifica binaria. La peculiarità rispetto all'approccio computazionale è che si suppone che conoscere un'espressione non necessariamente implichi conoscere le sue sotto-espressioni. Un'espressione potrebbe per esempio essere $\{K_{AB}, A\}K_{BS}$. Un avversario potrebbe benissimo conoscere l'espressione. Ma se non conosce K_{BS}, l'espressione gli appare come un'entità imperscrutabile. In tal caso non c'è alcuna possibilità che l'avversario riesca a ricostruire, ad esempio, K_{AB}. Questa è la principale differenza con il modello computazionale.

Di modelli formali non ne esiste solo uno ma molti, ciò che è comune a tutti i modelli formali è la loro semplicità. La sicurezza dei protocolli si può dimostrare senza assunzioni. Dati un protocollo, decidere se esista un avversario è un problema affrontabile anche con tecniche automatiche.

Un semplice modello formale
Partiamo dall'insieme $\mathbb{B} = \{0, 1\}$ dei booleani, e da un insieme $\mathbb{K}$ di simboli di chiave. Gli elementi di $\mathbb{K}$ non devono essere confusi con le stringhe binarie: sono simboli atomici, senza una struttura interna. Le espressioni del modello non sono nient'altro che le espressioni prodotte tramite la seguente grammatica:

$$M, N ::= K \mid i \mid (M, N) \mid \{M\}_K$$

dove $i \in \mathbb{B}$ e $K \in \mathbb{K}$.
Osserviamo come non ci sia ambiguità nelle espressioni. Ad esempio, $\{M\}_K$ e (N, L) sono sempre diverse.

Implicazione tra espressioni
Un concetto centrale nel modello formale è l'implicazione tra espressioni. Informalmente, un'espressione M *implica* N quando conoscere M permette di ricostruire N. Indichiamo che M implica N con $M \vdash N$. Le regole formali sono le seguenti:

$$\frac{}{M \vdash 0} \qquad \frac{}{M \vdash 1} \qquad \frac{M \vdash N \quad M \vdash L}{M \vdash (N, L)} \qquad \frac{M \vdash (N, L)}{M \vdash N}$$

$$\frac{M \vdash (N, L)}{M \vdash L} \qquad \frac{M \vdash N \quad M \vdash K}{M \vdash \{N\}_K} \qquad \frac{M \vdash \{N\}_K \quad M \vdash K}{M \vdash N}$$

Nel modello formale, il sistema diventa un insieme di processi che interagiscono scambiandosi messaggi. L'*avversario* non è nient'altro che uno dei possibili processi. L'avversario interagisce

con il protocollo interferendo nella comunicazione (può leggere i messaggi scambiati e può inviare uno qualunque tra i messaggi "implicati" dai messaggi che ha precedentemente letto). Talvolta, si suppone che l'avversario possa non solo interagire con una sessione del protocollo, ma che possa farne partire molte, interferendo con le diverse sessioni.

Implicazione ed equivalenze
La relazione di implicazione è un buon modo di modellizzare le capacità dell'avversario nel modello formale. Se E è un insieme di espressioni, $\{M \mid \exists N \in \mathcal{E}.N \vdash M\}$ è ciò che l'avversario riesce a calcolare a partire da E. Ad ogni passo, l'avversario potrà calcolare qualunque espressione tra quelle in tale insieme ed utilizzarla per costruire un attacco.

Allo scopo di confrontare il modello formale e quello computazionale, vale la pena parlare di **equivalenze** tra espressioni. Due espressioni si considerano equivalenti se sono indistinguibili di fronte ad un avversario il cui compito sia quello di "separarle".
Ad esempio, le due espressioni

$$(0, \{0\}_{K1}) \qquad (0, \{1\}_{K2})$$

sono da considerarsi equivalenti.

Patterns
Il modo in cui un avversario "vede" un espressione è catturato dalla nozione di **pattern**:

$$P, Q ::= K \mid \mathrm{i} \mid (P, Q) \mid \{\mathrm{P}\}_K \mid \square$$

I pattern cui corrisponde un espressione dipende però dall'insieme di chiavi T di cui l'avversario dispone:

$$\begin{aligned} p(K, \mathcal{T}) &= K \\ p(i, \mathcal{T}) &= i \\ p((M, N), \mathcal{T}) &= (p(M, \mathcal{T}), p(N, \mathcal{T})) \\ p(\{M\}_K, \mathcal{T}) &= \begin{cases} \{p(M, \mathcal{T})\}_K & \text{if } K \in \mathcal{T} \\ \square & \text{otherwise} \end{cases} \end{aligned}$$

Il modo in cui un avversario vede un espressione *M* è: $pattern(M) = p(M, \{K \in \mathbb{K} \mid M \vdash K\})$

Due espressioni M e N sono **equivalenti**, e scriveremo $M \equiv N$, sse pattern(M) = pattern(N), ovvero se le due espressioni sono indistinguibili agli occhi di ogni avversario.
L'equivalenza va poi indebolita, ponendo:

$$M \cong N \Leftarrow\Rightarrow M \equiv N\sigma$$

dove σ è una biiezione su $\mathbb{K}$. Alcuni esempi sono:

$$\begin{array}{lll} 0 \cong 0 & \{0\}_K \cong \{1\}_K & K_1 \cong K_2 \\ 1 \ncong 0 & \{0\}_K \cong \{K\}_K & (K, \{0\}_K) \ncong (K, \{1\}_K) \end{array}$$

Per capire con il modello formale sia un'astrazione del modello computazionale, bisogna prima di tutto capire a cosa corrisponde un'espressione *M* nel modello computazionale, dato uno schema di codifica Π = (Gen, Enc, Dec). Π corrisponderà ad una *famiglia di distribuzioni*, parametrizzata su un parametro di sicurezza n.
Prima di tutto, ad ogni *chiave* $K \in \mathbb{K}$ che occorre in *M* faccio corrispondere la chiave ottenuta tramite $Gen(1^n)$, poi, a 0 e 1 faccio corrispondere una stringa che codifichi tale valore booleano.
Le *coppie* (*N, L*) in *M* saranno codificate opportunamente come stringhe binarie.

Un *crittogramma* $\{N\}_K$ che occorre in M sarà trattata invocando *Enc*.
La famiglia di distribuzioni cui corrisponde *M* è indicata con $[M]_\Pi$.

Definizione
Due famiglie di distribuzioni $\mathbf{D} = \{D_n\}_{n\in N}$ ed $\varepsilon = \{E_n\}_{n\in N}$ sono dette computazionalmente indistinguibili sse per ogni avversario PPT A esiste una funzione trascurabile $\varepsilon \in NGL$ t.c.:

$$|Pr(A(1^n, D_n) = 1) - Pr(A(1^n, E_n) = 1)| \leq \varepsilon(n)$$

In tal caso scriviamo $\mathbf{D} \sim \varepsilon$.

Un'espressione *M* si dice aciclica sse per ogni sottoespressione $\{N\}_K$ di M, la chiave K non occorre in N.

Teorema (Abadi & Rogaway)
Se Π è sicuro e *M, N* sono acicliche, allora $M \cong N$ implica $[M]_\Pi \sim [N]_\Pi$.