Sicurezza

Alessio Marini, 2122855

Appunti presi durante il corso di **Sicurezza** nell'anno **2025/2026** del professore Emiliano Casalicchio.
Gli appunti li scrivo principalmente per rendere il corso più comprensibile **a me** e anche per imparare il linguaggio Typst. Se li usate per studiare verificate sempre le informazioni 🙏.

Contatti:
alem1105
marini.2122855@studenti.uniroma1.it

September 27, 2025

Indice

1. CIA - Key Objective / Requirements

Sicurezza Informatica - NIST

Le misure che garantiscono **confidenzialità, integrità e disponibilità** di un sistema informatico nel sua interezza quindi hardware, software, firmware, le informazioni che memorizza, processa e comunica

Queste tre proprietà vengono indicate nella **triade CIA** e significano:
- **Confidenzialità**: Garantire che le informazioni private non vengano divulgate a persone non autorizzate e garantire anche che ogni utente abbia il controllo su quali dei suoi dati personali vengono memorizzati nel sistema o a chi vengono inviati. L'utente deve quindi essere in grado di gestire la sua **privacy**.
- **Integrità**: Assicurare che i dati o il sistema vengano modificati solo in modo autorizzato.
- **Disponibilità**: Garantire l'accesso affidabile alle informazioni e ai servizi, non devono esserci interruzioni del servizio.

La triade è in costante conflitto con usabilità e costi del sistema, il lavoro di un esperto di sicurezza è quindi quello di trovare il giusto compresso fra queste variabili.

Nei sistemi moderni, ai tre pilastri della triade, si aggiungono:
- **Autenticità**: La garanzia che un messaggio o un utente provengano effettivamente dalla fonte dichiarata.
- **Accountability**: La capacità di tracciare in modo univoco le azioni di un'entità, è utile per indagini forensi, isolare i guasti e la **nonrepudiation** ovvero quando qualcuno nega di aver compiuto un'azione.

1.1. Levels of Impact

Possiamo classificare l'impatto di eventuali violazioni:
- *Basso*: Danni minori o perdite finanziare limitate, il sistema continua comunque a funzionare anche se magari in modo ridotto.
- *Moderato*: Danni significativi alle capacità operative e perdite finanziare importanti.
- *Alto*: Il sistema non funziona più, danni finanziari enormi e ci sono rischi catastrofici o letali per le persone.

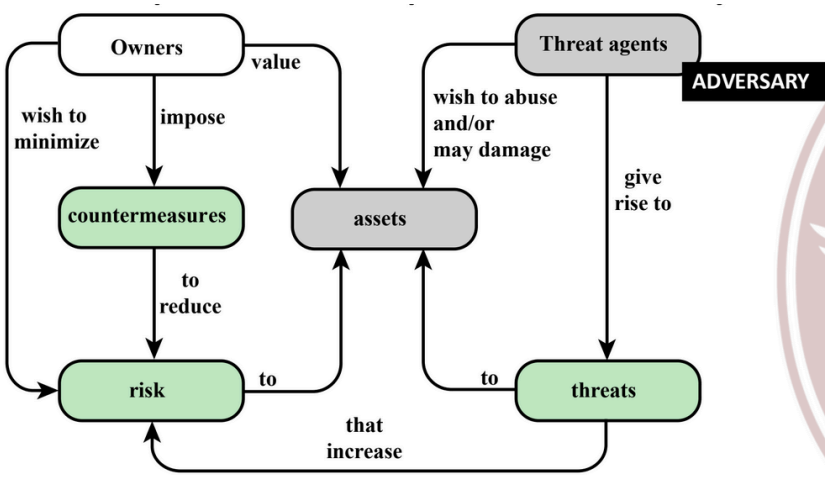
1.2. Computer Security Challenges

La sicurezza informatica presenta diverse sfide che la rendono complessa:
1. Non è semplice per chi si trova alla prime armi e spesso le procedura di sicurezza risultano controintuitive
2. Richiede una gestione di **informazioni riservate** e anche una decisione su dove posizionare le difese.
3. Dagli utenti viene vista come **un ostacolo** e dai manager come uno spreco di risorse fino a quando non subiscono un attacco.
4. Spesso viene aggiunta alla fine dell'implementazione di un sistema come un «ripensamento» invece di essere implementata dall'inizio.

Dilemma del Difensore

Chi difende deve eliminare **tutte le debolezze** di un sistema (praticamente impossibile), mentre a chi attacca ne basta una sola.

1.3. Un modello per la Sicurezza



Nel diagramma sono indicati:

- **Asset (Risorsa)**: È quello cha ha un valore nel sistema (Hardware, software, Dati ecc...)
- **Vulnerabilità (di un asset)**: Una debolezza nel sistema che può essere sfruttata. Ne esistono di diversi tipi:
  - **Corrupted**: Dati alterati
  - **Leaky**: Dati trapelati
  - **Unavailable**: Sistema non disponibile
- **Threat**: Una qualsiasi «situazione» che potrebbe causare danni di qualsiasi tipo.
- **Attack**: Quando un *Threat Agent* ovvero un avversario sfrutta attivamente una vulnerabilità.
- **Countermeasure**: Tecniche o dispositivi utili a prevenire gli attacchi o a ridurne gli effetti.

Possiamo anche calcolare il rischio grazie alla formula:

$$R = P \times D$$

Dove:

- P: Probabilità che l'evento si verifichi
- D: Impatto dell'evento

Nota

L'obiettivo delle contromisure **non è azzerare il rischio** dato che sarebbe troppo costoso o comunque impossibile, ma è quello di portarlo ad un livello accettabile definito **Rischio Residuo**.

2. Classificazione degli Attacchi

La prima distinzione che possiamo fare è quella della provenienza dell'attacco, o dall'interno (**insider**) o dall'esterno (**outsider**). Gli attacchi si dividono poi in:

- **Passivi**: L'attaccante osserva e raccoglie informazioni senza alterare il sistema, questi sono molto difficile da rilevare e l'unica difesa è la **prevenzione**. Ne esistono di due tipi:
  - *Release of Message Contents*: Leggere il contenuto di un messaggio
  - *Traffic Analysis*: Anche se il messaggio è cifrato, l'attaccante osserva chi sta parlando con chi e guardando anche la dimensione del messaggio, la frequeunza e altri dati può ricavare qualche informazione.
- **Attivi**: L'attaccante altera i dati o il funzionamento del sistema:
  - *Replay*: Catturare un dato e reinviarlo successivamente per ottenere accesso.
  - *Masquerade*: Fingersi qualcun altro
  - *Modification of Messages*: Alterare parti legittime di un messaggio o ritardarlo.
  - *Denial of Service (DoS)*: Interrompere il servizio di un sistema sovraccaricandolo di richieste.

Vediamo adesso le conseguenze e i tipi di attacco che le producono (secondo lo standard RFC4949):

- **Unauthorized Disclosure**: Vengono divulgati dati da persone non autorizzate, attacchi possibili:
  - *Exposure*: Esposizione diretta
  - *Interception*
  - *Inference*: Deduzione di dati da altri segnali
  - *Intrusion*
- **Deception**: Vengono fornite informazioni ad un utente non autorizzato
  - *Masquerade*: Fingersi un'altra persona
  - *Falsification*: Alterari i dati per ingannare
  - *Repudiation*: Negare di aver compiuto un'azione
- **Disruption**: Interruzione delle funzionalità del sistema:
  - *Incapacitation*: Disabilitare un componente
  - *Corruption*: Alterare i dati che permettono il funzionamento del sistema
  - *Obstruction*: Ostacolare i servizi
- **Usurpation**: Viene preso il controllo del sistema da parte di utenti non autorizzati.
  - *Misappropriation*: Prendere il controllo logico o fisico di una risorsa
  - *Misuse*: Utilizzare un componenete per svolgere operazioni dannose.

2.1. Requisiti di Sicurezza

Le contromisure alle vulnerabilità e minacce possono essere classificate e categorizzate in differenti modi, possiamo classificarle in funzione dei requisiti funzionali, il FIPS 200 le divide in:

- **Requisiti di Sicurezza Tecnici**: Ad esempio controllo degli accessi, autenticazione e integrità del sistema.
- **Funzionali**: Formazione del personale, valutazione dei rischi, sicurezza fisica
- **Misti**: Gestione della configurazione, risposta agli incidenti e protezione dei media.

2.2. Principi fondamentali di Design per la sicurezza

Questi principi sono fondamentali per chi progetta architetture di rete o sviluppa software:

- **Economy of Mechanism**: I sistemi di sicurezza devono essere piccoli e semplici, più codice scrivi e più bug introduci.
- **Fail-safe default**: L'accesso deve basarsi sui permessi e non sulle esclusioni, se qualcosa va storto il sistema deve bloccarsi in uno stato sicuro negando l'accesso.
- **Complete Mediation**: Ogni singolo accesso a una risorsa deve essere verificato senza fidarsi ciecamente della cache.
- **Open Design**: La sicurezza non deve basarsi sulla segretezza del codice ma sulla forza dell'algoritmo o delle chiavi.
- **Separation of Privilige**: Richiedere più condizioni per sblcocare un'azione (2FA).
- **Least Privilege**: Utenti e processi devono avere solo i permessi strettamente necessari per fare il loro lavoro.
- **Layering**: Usare ostacoli multipli e sovrapposti, se l'attaccante buca il firewall deve trovare altri blocchi.

Esempio Fail-Safe

Prendiamo questo esempio in pseudocodice:

```c
DWORD dwRet = IsAccessAllowed(...);
if (dwRet == ERROR_ACCESS_DENIED) {
  // Security check failed
} else {
  // Security check OK
}
```

Questo codice non va bene perché se per qualsiasi motivo la funzione `IsAccessAllowed()` fallisce e ritorna un dato non previsto noi stiamo garantendo l'accesso al sistema.

2.3. Superfici e Alberi di Attacco

Per difendere un sistema è utile capire come possiamo attaccarlo.

Definiamo come **Attack Surface** l'insieme di tutte le vulnerabilità raggiungibili in un sistema, si divide in:

- *Rete*: Porte aperte e server esposti
- *Software*: Bug nel codice
- *Umana*: Social engineering o errore del personale

Il nostro obiettivo è quello di ridurre il più possibile questa superficie.

Per analizzare le debolezze del sistema possiamo costrutire un **albero di attacco** ovvero un albero che rappresenta le minacce al nostro sistema:

- Il nodo radice è l'obiettivo dell'attaccante
- I nodi intermedi sono i sotto-obiettivi
- Le foglie sono gli attacchi che può fare per colpire un sotto-obiettivo.

I sotto-obiettivi possono essere in AND o OR fra di loro.

Questi alberi sono utilizzati dai team di sicurezza per capire qual è il percorso «più economico» o «più facile» per un attaccante, questo permette quindi di posizionare le difese (layering) dove sono più necessarie.

Esempio di albero:

2.4. Computer Security Strategies

Come si gestisce quindi la sicurezza a livello aziendale?

- **Specification / Policy (Cosa Fare?)**: Le regole che indicano cosa deve fare il sistema (es. lunghezza minima di una password). Queste devono bilanciare il costo della sicurezza col costo di un eventuale attacco.
- **Implementation / Mechanisms (Come Farlo?)**: Le tecnologie usate per prevenire, rilevare, rispondere e ruperare i dati.
- **Assurance ed Evaluation (Funziona?)**: Il livello di confidenza che i meccanismi implementati rispettino effettivamente le policy. Questo si ottiene tramite test, certificazioni e analisi.

3. Cryptographic Tools

3.1. Symmetric Encryption

È la tecnica di crittografia universale per garantire la **confidenzialità** dei dati. Il mittente e il destinatario, per scambiarsi messaggi, devono condividere la stessa chiave segreta K. Questa tecnica, per essere sicura, richiede:

- Un algoritmo crittografico forte.
- Che la condivisione della chiave sia avvenuta in modo sicuro.

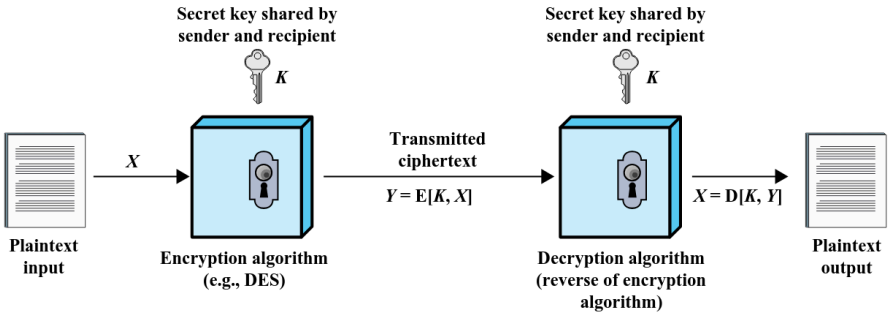


Figure 2.1 Simplified Model of Symmetric Encryption

I principali tipi di attacco a queste tecniche sono:

- **Crittoanalisi**: Sfruttare la natura matematica dell'algoritmo o se si conoscono alcune caratteristiche del testo non cifrato per dedurre la chiave.
- **Brute-Force**: Si provano tutte le chiavi possibili finché non si ottiene un testo sensato. In media è necessario provare almeno la metà di tutte le chiavi possibili.

La distrubuzione delle chiavi

Il vero problema di questa tecnica è la distribuzione della chiave, infatti, come fanno due persone a scambiarsi la chiave in modo sicuro se il canale su cui parlano non è protetto? Questo problema verrà risolto dalla **crittografia asimmetrica**.

Tra i principali algoritmi per la crittografia simmetrica troviamo:

- **DES (Data Encryption Standard)**: Utilizza blocchi da 64-bit e una chiave da 56-bit. Anche se è l'algoritmo più studiato la sua chiave da 56-bit lo rende molto debole contro i moderni processori di oggi.
- **3DES (Triple DES)**: Ripete il DES tre volte usando 2 o 3 chiavi diverse portando quindi la lunghezza effettiva della chiave a 112 o 168 bit. I problemi principali sono che è molto lento e usa ancora blocchi piccoli da 64-bit.
- **AES (Advanced Encryption Standard)**: Creato per sostituire il 3DES. È molto più efficiente, utilizza blocchi da 128-bit e chiavi da 128, 192 o 256-bit.

L'AES è più efficiente perchè utilizza un'architettura chiamata **Rijndael** che applica le sue trasformazioni all'intero blocco di 128-bit simultaneamente organizzandolo in una matrice, questa struttura è ottimizzata per eseguire calcoli paralleli ed è perfetta per come lavorano i

processori moderni. Questo rende l'algoritmo più veloce rispetto ad un DES o 3DES che invece utilizzano la **Rete di Feistel**, questa divide il blocco a metà e, a ogni passaggio, cripta solo una metà alla volta incrociandola con l'altra.

Tempi medi richiesti per rompere un algoritmo:

Key size (bits)	Cipher	Number of Alternative Keys	Time Required at 10^9 decryptions/s	Time Required at 10^{13} decryptions/s
56	DES	$2^{56} \approx 7.2 \times 10^{16}$	2^{55} ns = 1.125 years	1 hour
128	AES	$2^{128} \approx 3.4 \times 10^{38}$	2^{127} ns = 5.3×10^{21} years	5.3×10^{17} years
168	Triple DES	$2^{168} \approx 3.7 \times 10^{50}$	2^{167} ns = 5.8×10^{33} years	5.8×10^{29} years
192	AES	$2^{192} \approx 6.3 \times 10^{57}$	2^{191} ns = 9.8×10^{40} years	9.8×10^{36} years
256	AES	$2^{256} \approx 1.2 \times 10^{77}$	2^{255} ns = 1.8×10^{60} years	1.8×10^{56} years

Oggi però i **computer quantistici** minacciano gravemente gli algoritmi con chiave pubblica, infatti questi si basano su problemi matematici come la fattorizzazione che un QC può risolvere facilmente. Oggi si cercano quindi algoritmi **Quantum - resistant**.

3.2. Type of Symmetric Encryption

Ovviamente dobbiamo cifrare file più grandi di un qualsiasi blocco, per questo esistono diverse tipologie di cifratura:

- **Block Cipher**: Cifra il testo elaborando un blocco alla volta, può riutilizzare le chiavi ed è il più comune.
- **Stream Cipher**: Cifra il flusso di dati continuamente, byte alla volta, generando un flusso pseudocasuale, è molto più veloce e usa meno codice.

Un metodo di cifratura che utilizza il cifrario a blocchi è il **ECB**:

- **Electronic Codebook (ECB)**: È la modalità base per usare i cifrari a blocchi su messaggi lunghi. Ogni blocco viene cifrato singolarmente con la stessa chiave, il problema è che se il testo in chiaro ha delle regolarità allora queste appariranno anche nel testo cifrato.

3.3. Message Authentication

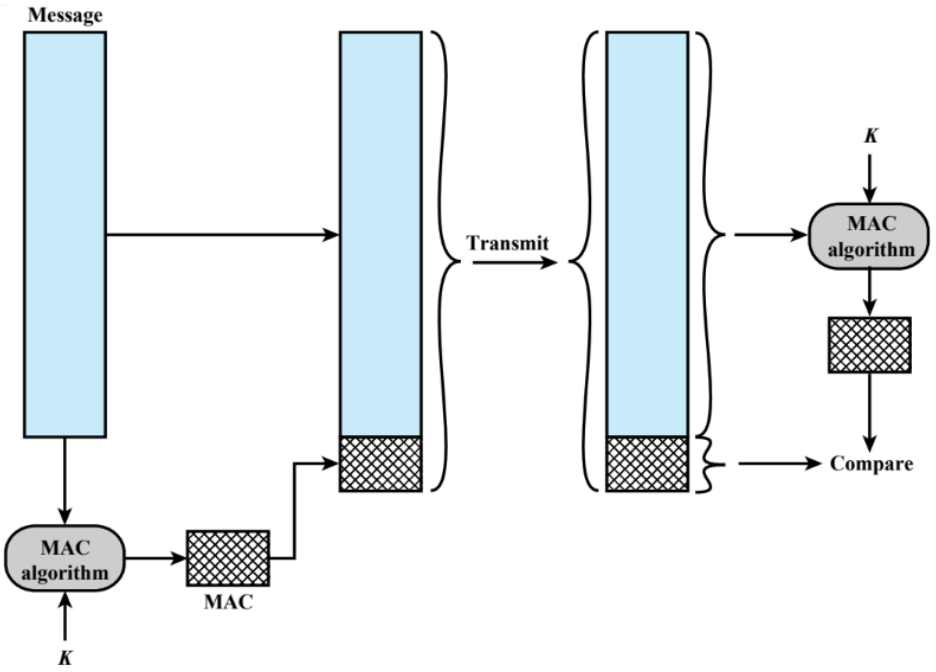
La cifratura dei messaggi protegge dalla lettura non autorizzata ma non dalla manomissione del messaggio ovvero gli attacchi attivi. Con l'autenticazione dei messaggi garantiamo:

- Il contenuto non è stato alterato
- Il messaggio proviene da una fonte legittima

Per garantire che il messaggio non sia stato alterato si utilizza un **MAC (Message Authentication Code)**. Attraverso la chiave K e il messaggio M si calcola un «tag»:

$$\text{MAC} = F(K, M)$$

Se il ricevente, ricalcolando il MAC con la stessa chiave segreta, ottiene lo stesso risultato, ha la garnazie che il messaggio non sia stato modificato.



Ovviamente, anche qui, la chiave deve essere in possesso solo delle persone fidate.

Possiamo inoltre distinguere due casi:
- **Without Confidentiality**: Il messaggio è autenticato ma non cifrato, viene soltanto aggiunto il TAG al messaggio.
- **With Confidentiality**: Il messaggio è autenticato e cifrato, quindi anche il tag viene cifrato.

Quando è utile avere autenticazione ma non confidenzialità?
- Sistemi broadcast o molto carichi
- Protocolli di rete, l'header del pacchetto IP non deve essere alterato ma non abbiamo bisogno di nasconderlo dato che i router devono leggerlo.

3.4. Funzioni di Hash

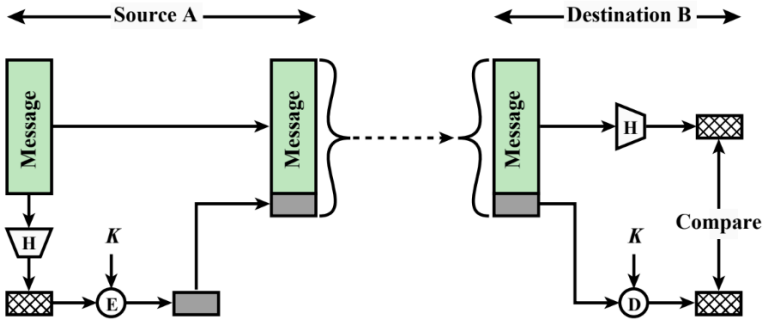
Una funzione Hash $H(M)$ è un'alternativa al *MAC* che non richiede una chiave in input, prende un messaggio di dimensione variabile e restituisce un'impronta di dimensione fissa (*digest*). Esistono 6 proprietà da ricordare per le hash function. Data una funzione *H*:

1. H può essere applicata a blocchi di qualsiasi dimensione.
2. H produce un output di lunghezza fissa.
3. $H(x)$ è relativamente semplice da calcolare per ogni x in input.
4. Per ogni codice hash h dato, deve essere difficile calcolare un x tale che $H(x) = h$. Una funzione con questa proprietà si dice **one-way** o **pre-image resistant**.
5. Per ogni blocco x dato, deve essere difficile trovare un y con $y \neq x$ tale che $H(y) = H(x)$. Una funzione con questa proprietà si dice **second preimage resistant** o **weak collision resistant**.
6. Deve essere difficile trovare una coppia (x, y) tale che $H(x) = H(y)$. Una funzione che soddisfa questa proprietà si dice **collision resistant** o **strong collision resistant**.

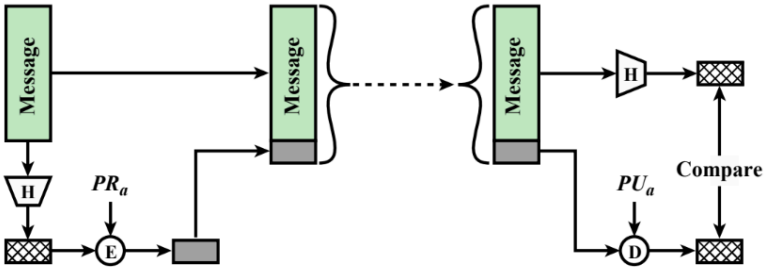
Le proprietà 1,2 e 3 sono necessarie soltanto per le hash function utilizzare per message authentication mentre 4, 5 e 6 sono generali ma comunque vanno rispettate se vogliamo avere una buona funzione di hash.

Possiamo utilizzare l'hash a scopo di autenticazione in diversi modi:

- **One-Way Hash function + Symmetric Encryption**

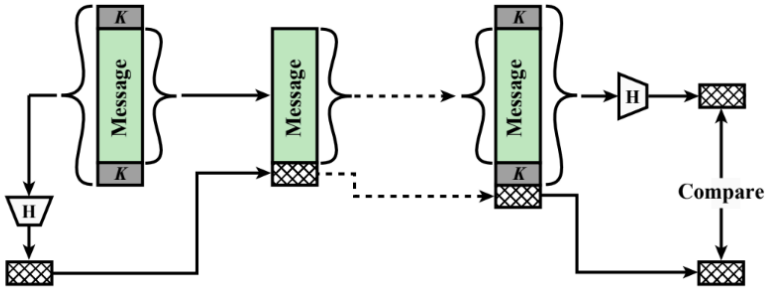


- **One-Way Hash function + Public Key Encryption**



  Qui il vantaggio è che non è richiesta la distribuzione delle chiavi in modo sicuro.

- **One-Way Hash function + Secret Value (no encryption)**



  Non abbiamo bisogno di crittografia, risparmiando quindi computazione. Chi invia il messaggio deve calcolare $\mathrm{MD}_M = H(K \mid M \mid K)$ e inviare $\mathrm{MD}_M \mid M$, a questo punto il destinatario, che conosce K, deve calcolare $H(K \mid M \mid K)$ e verificare MD_M

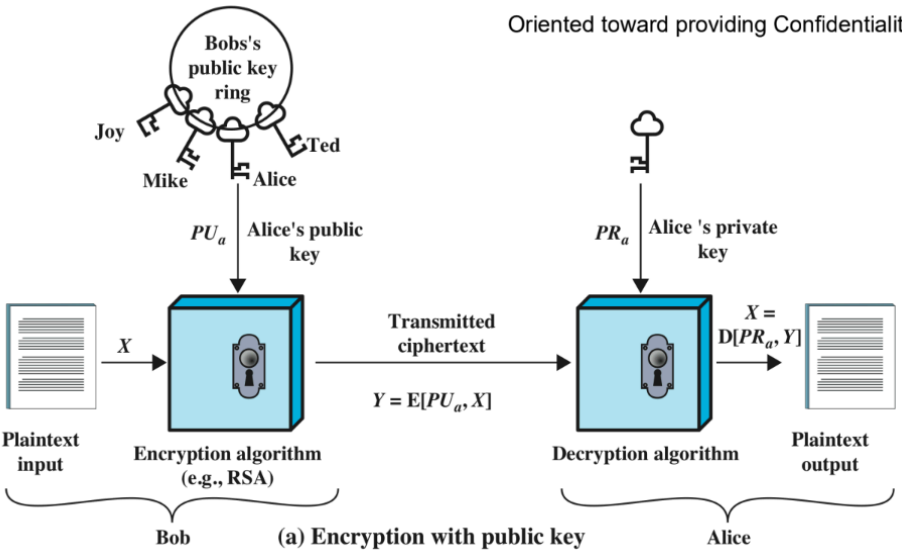
Lo standard più utilizzato è lo **SHA** anche se oggi è considerato debole e si utilizza **SHA-2 o SHA-3**.

3.5. Crittografia a Chiave Pubblica (Asimmetrica)

È un algoritmo di crittografia proposto da **Diffie e Hellman** nel 1976, risolve principalmente il problema della distribuzione della chiave. Viene utilizzata una coppia di chiavi collegate fra loro matematicamente, una **pubblica** e una **privata**. Queste garantiscono:

- **Confidenzialità**: Io cifro con la chiave pubblica di Bob e solo Bob con la sua chiave privata sarà in grado di decifrare il messaggio.

- **Autenticazione**: Cifro l'hash con la mia chiave privata, adesso tutti possono decifrarlo con la mia chiave pubblica garantendo quindi che sono stato io a generare quel messaggio.



Tra i principali algoritmi a chiave asimmetrica troviamo:

- **RSA**: Utilizzato per la cifratura, firme digitali e scambio di chiavi.
- **Diffie-Hellman**: Serve *solo* per permettere a due entità di generare una chiave simmetrica condivisa attraverso un canale sicuro.
- **DSS (Digital Signature Standard)**: Garantisce firma digitale con SHA-1
- **ECC (Elliptic Curve)**: Sicuro quanto RSA ma usa chiavi più corte, è quindi più veloce e leggero.

Algorithm	Digital Signature	Symmetric Key Distribution	Encryption of Secret Keys
RSA	Yes	Yes	Yes
Diffie-Hellman	No	Yes	No
DSS	Yes	No	No
Elliptic Curve	Yes	Yes	Yes

Vero vantaggio della chiave asimmetrica

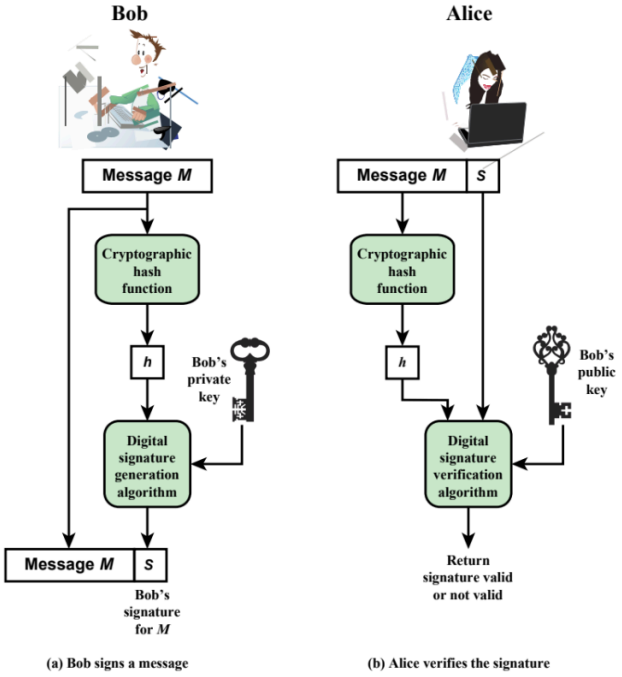
La crittografia asimmetrica è **computazionalmente lentissima** rispetto a quella simmetrica, nel mondo reale infatti viene si utilizzata ma soltanto all'inizio delle comunicazioni per scambiare in modo sicuro una chiave segreta temporanea simmetrica, dopo questo scambio il resto della comunicazione avviene con cifratura simmetrica tramite AES.

Si vede più avanti

3.6. Digital Signatures

Per firma digitale intendiamo gli algoritmi che ci permettono di verificare l'autenticità del mittente di un messaggio, la sua integrità e la non repudiabilità.

L'idea è quella di creare un pacchetto di dati che accompagna il messaggio. Per fare questo vengono utilizzati algoritmi di cifratura e i 3 più importanti sono *DSA*, *RSA* e *Algoritmi a curva ellittica ECDSA*.



(a) Bob signs a message (b) Alice verifies the signature

Il mittente deve:
- Calcolare l'hash del messaggio.
- Cifriamo l'hash appena ottenuto.
- Inviare il messaggio in chiaro insieme all'hash cifrato.

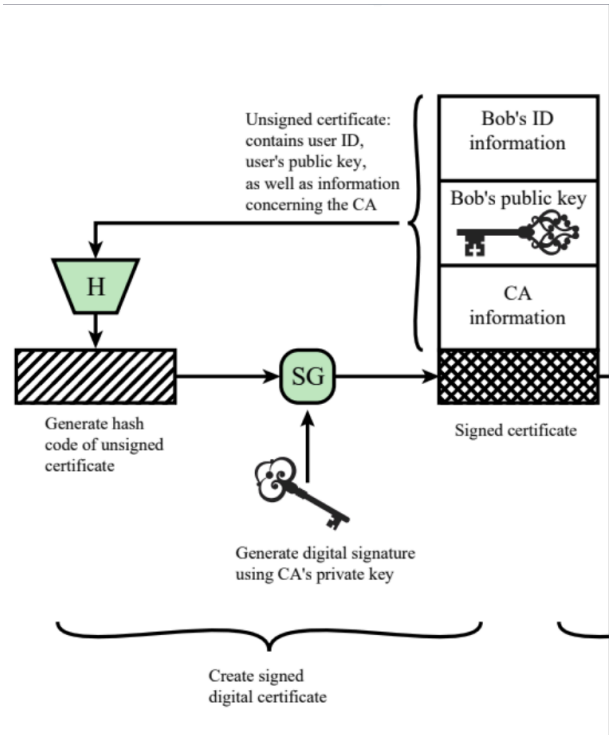
Il destinatario, per leggere il messaggio deve:
- Effettuare l'hash del messaggio in chiaro.
- Decifrare tramite la chiave pubblica del mittente l'hash ricevuto.
- Verificare che i due hash combacino.

Il problema principale in questo tipo di algoritmi è che qualcuno potrebbe essersi finto Bob quando ha pubblicato la sua chiave pubblica. Come risolviamo questo problema? utilizziamo un **certificato a chiave pubblica**.

I certificati a chiave pubblica sono composti da:
- La chiave pubblica da distribuire.
- Informazioni sul possessore di questa chiave (ID)
- Informazioni sul verificatore dell'identità che chiamiamo **Certification Authority (CA)** (Aruba, Infocert, PosteItaliane ...), ovvero un'entità fidata che si occupa di verificare l'appartenenza delle chiavi.
- Periodo di validità di questo certificato.

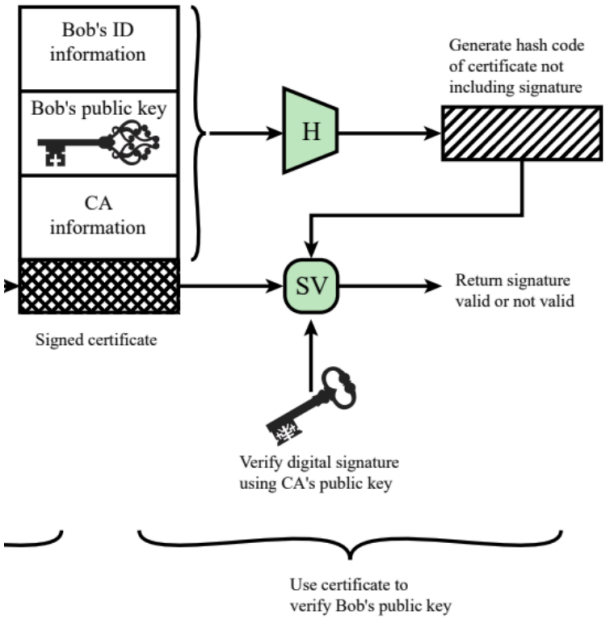
Ma come si crea un certificato?
1. L'utente crea la coppia di chiavi, una pubblica e una privata.
2. Prepara un certificato non verificato che include il suo ID e la sua chiave pubblica.
3. Fornisce questo certificato ad una CA in **modo sicuro**.

4. La CA crea il certificato seguendo questi passaggi:
   - Usa una funzione hash su tutto il certificato.
   - Genera una firma digitale utilizzando la propria chiave privata.
5. La CA allega la firma digitale al certificato non verificato.
6. Invia il certificato verificato all'utente



Adesso il certificato potrà essere pubblicato dall'utente e chiunque può verificarlo utilizzando la chiave pubblica della CA, nello specifico:

1. L'utente calcola l'hash del certificato (senza la firma).
2. Tramite la chiave pubblica della CA decifra la firma e verifica che combacia con l'hash ottenuto.



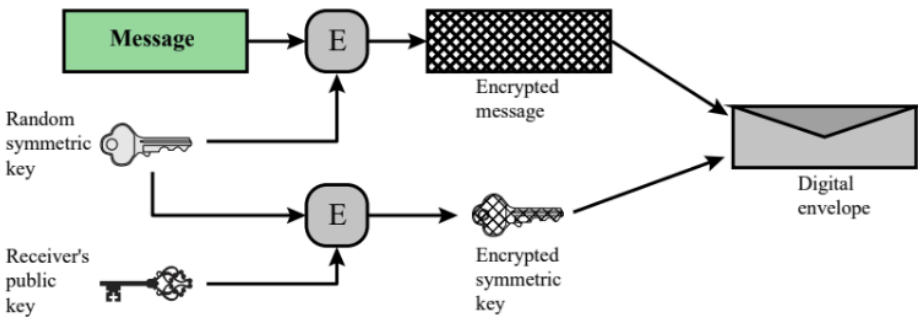
Lo standard per la formattazione dei certificati è X.509

3.7. Digital Envelope

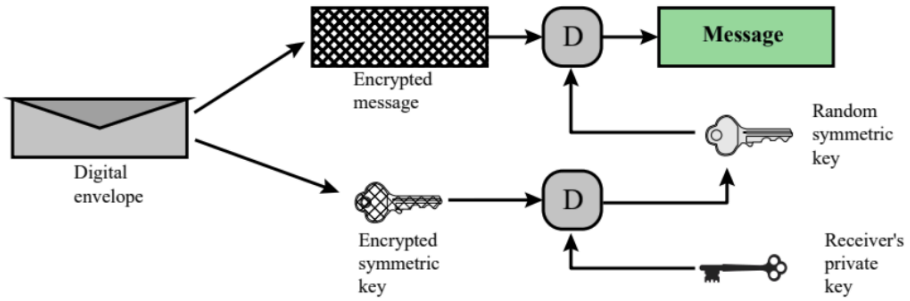
Utilizzare algoritmi a chiave asimmetrica è molto pesante dal punto di vista delle prestazioni. Per allegerire il sistema ma mantenere comunque una trasmissione sicura viene utilizzata una chiave asimmetrica per scambiare una chiave simmetrica che verrà poi utilizzata per il resto della conversazione.

Questo metodo di cifratura non viene utilizzato solo per questo ma in generale per scambiare messaggi cifrati con chiave simmetrica.

I passaggi da seguire sono:
1. Si prepara il messaggio da inviare.
2. Generiamo una chiave simmetrica casuale che verrà usata soltanto in questa comunicazione.
3. Si cifra il messaggio utilizzando la chiave simmetrica appena generata.
4. Si cifra la chiave simmetrica utilizzando la chiave pubblica del destinatario.
5. Si allega la chiave cifrata al messaggio cifrato e si invia.



Quindi il destinatario per leggere il messaggio dovrà semplicemente decifrare prima la chiave simmetrica e poi con questa il messaggio. Ovviamente per garantire autenticità si possono utilizzare i certificati visti prima.



(b) Opening a digital envelope

3.8. Random Numbers

Vengono usati moltissimo nel mondo della crittografia in algoritmi a chiave pubblica o cifratura a flusso ma anche per generare token per sessioni o altro ...

Quali caratteristiche devono soddisfare?

- **Randomness**
  - **Uniform Distribution**: La frequenza con cui un determinato numero viene generato deve essere *circa* la stessa per ogni numero.
  - **Independence**: Non deve essere possibile prevedere la sequenza di numeri che vengono generati. (È difficile da provare)
- **Unpredictability**
  - Ogni numero è statisticamente indipendente dagli altri numeri della sequenza. Un attaccante non deve essere in grado di predirre i numeri futuri basandosi su quelli usciti.

Generare numeri completamente casuali in informatica è molto difficile e per questo si generano, invece, dei numeri detti **pseudocasuali** ovvero delle sequenze di numeri che soddisfano dei **test di randomicità**. Per generare questi numeri ci si basa su delle fonti non deterministiche o in generale eventi fisici, ad esempio gli **Hardware RNG** ovvero delle statistiche della macchina come la temperatura, il jitter dei circuiti o i quantum effects.

4. Cifratura Simmetrica

È conosciuta come il metodo convenzionale di cifratura, ha **5 componenti** principali:
- **Plaintext**: Il testo in chiaro da dare all'algoritmo.
- **Encryption Algorithm**: Algoritmo che effettua *sostituzioni e traformazioni* al plaintext.
- **Secret Key**: Anche questa viene data in input all'algoritmo e da questa dipendono tutte le trasformazioni effettuate nell'algoritmo.
- **Ciphertext**: Il testo di output dell'algoritmo, per ogni testo se utilizziamo chiavi diverse dobbiamo ottenere risultati diversi.
- **Dectryption Algorithm**: L'algoritmo di cifratura inverso, dato il ciphertext e la chiave deve fornire in output il plaintext.

4.1. Cryptographics Systems

Sono meccanismi che utilizzano algoritmi, chiavi e protocolli per proteggere la **confidenzialità, l'integrità, l'autenticazione** e la **non ripudiabilità**.

Vengono classificati su 3 *dimensioni*

- Tipo di operazioni utilizzate per cifrare il plaintext
  - **Sostituzione**: Prende un insieme di byte e li rimpiazza con altri byte.
  - **Trasposizione**: I byte del testo vengono mischiati fra di loro.

  Queste sono indipendenti dalla chiave e vengono ripetute un certo numero di volte (*round*) sul testo. Nell'effettuare queste operazioni non dobbiamo **perdere informazione**, questo significa che le operazioni devono essere invertibili.
- Tipo di chiave utilizzata:
  - **Simmetrica** o **asimmetrica**
- Modo in cui viene trattato il testo in chiaro:
  - **Block**: Viene processato un blocco alla volta. Quindi da un blocco in input otteniamo un blocco in output.
  - **Stream**: L'input viene continuamente processato, ogni elemento produce un elemento in output.

4.1.1. Block Cipher

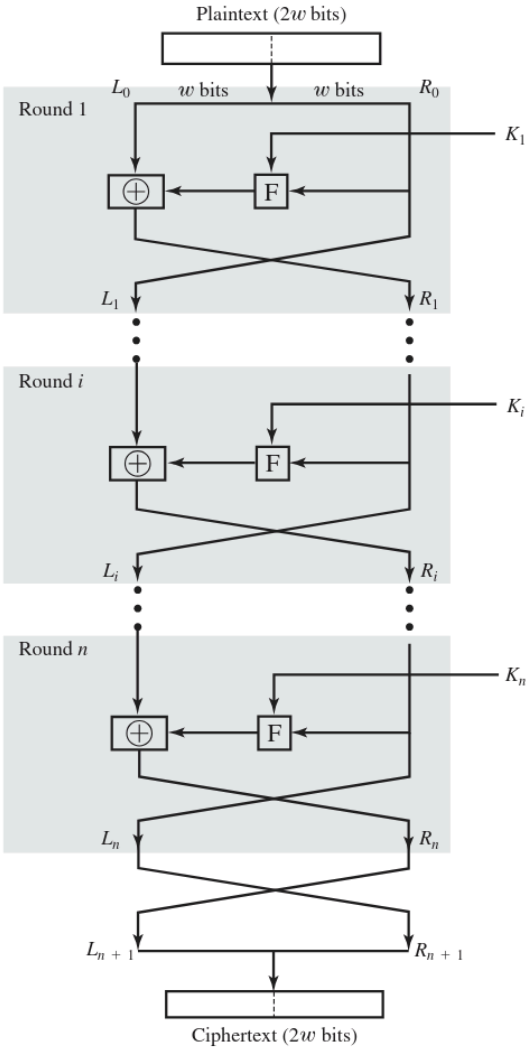
I meccanismi di cifratura simmetrici a blocchi consistono in una **serie di round** composti da **sostituzioni e permutazioni** basati su una chiave.

Una delle strutture più famose è la **Feistel Cipher Structure** utilizzata anche nell'algoritmo DES.

In questa struttura dobbiamo tenere a mente che:
- È divisa in round e ogni round effettua le stesse operazioni.
- Si parte con una chiave ma poi a partire da questa vengono generate più chiavi, nello specifico una diversa per ogni round.

A ogni round il testo viene diviso in due parti uguali, la funzione *F* prende in input la parte destra ed effettua delle sostituzioni, l'output della funzione viene poi messo in *XOR* con la

parte sinistra del testo. A questo punto viene effettuata una sostituzione tra le due parti ovvero mettiamo quella sinistra a destra e quella di destra a sinistra, in questo modo il round successivo effettua le stesse operazioni perché si troverà a destra la parte non cifrata e a sinistra quella cifrata.



Per progettare una struttura di cifrario a blocchi dobbiamo considerare diversi parametri:

- **Block Size**: Se il blocco è grande ho più sicurezza ma meno velocità nelle operazioni, un tradeoff ottimale è 128-bit.
- **Key Size**: Più è grande e più è alta la sicurezza ma riduciamo la velocità delle operazioni. Un tradeoff ottimale è 128 - 256-bits.
- **Numero di round**: Aumenta la sicurezza ad aumentare il numero di round. Anche qui dobbiamo trovare un giusto tradeoff che è 16 round.
- Algoritmo di **generazione delle sottochiavi**: Più è complesso, più è difficile fare **cryptanalysis**
- **Funzione** utilizzata nei **round**: Stesso ragionamento dell'algoritmo di generazione delle sottochiavi.
- **Velocità dell'algoritmo** di cifratura e decifratura.
- L'algoritmo deve essere **facile da analizzare**: Serve ad individuare meglio le vulnerabilità però le componenti dell'algoritmo devono essere tali che anche se conosciute da chi fa cryptanalysis sia impossibile invertire le operazioni senza la chiave.

Cryptanalysis

Processo utilizzato per cercare di scoprire il testo in chiaro o la chiave, sfruttando soltanto la conoscenza dell'algoritmo di cifratura, conoscendo quest'ultimo ed i suoi punti deboli si cerca di risalire al testo in chiaro.

4.1.1.1. DES - Data Encryption Standard

Il DES era un algoritmo complesso da analizzare.

Il **processo di cifratura** era composto da:
- Block cipher con blocchi da 64-bits e chiave da 56-bits.
- Utilizzava una variante della struttura Feistel a 16 rounds.
- Dalla chiave a 56 bit venivano generate 16 sottochiavi

Il **processo di decifratura** era semplicemente l'inverso della cifratura, questo ha in input il ciphertext e poi utilizza in ordine inverso rispetto alla cifratura le sottochiavi generate dalla chiave.

Per i moderni processori di oggi la chiave a 56-bits è considerata obsoleta e si è passati al 3DES.

4.1.1.2. 3DES

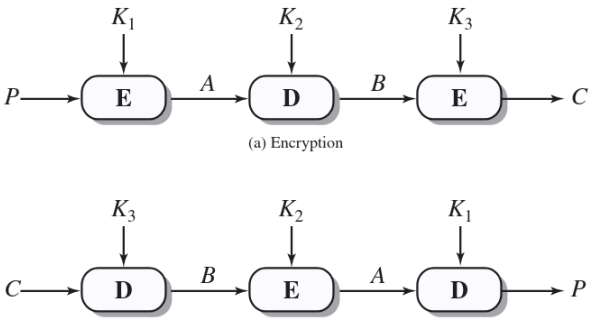
- Si estende la chiave fino a 168-bits che veniva suddivisa in 3 chiavi da 56-bits
- Si cifra con le tre chiavi in modo sequenziale, ovvero applicare il DES 3 volte.
- Per decifrare si fa l'opposto.

Nello specifico definiamo come:
- E funzione di cifratura
- D funzione di decifratura

Per effettuare una cifratura con 3DES si applica:
- $C = E(K_3, D(K_2, E(K_1, P)))$

Mentre per la decifratura:
- $P = D(K_1, E(K_2, D(K_3, C)))$

Graficamente:



Utilizzare la funzione di decifratura come secondo step in 3DES non offre vantaggi in termini di crittografia ma permette la retrocompatibilità con il DES classico, o meglio permette a chi usa 3DES di decriptare anche messaggi cifrati con DES.

Per migliore sicurezza si è passati poi all'AES.

4.1.1.3. AES - Advanced Encryption Standard

AES utilizza blocchi da 128-bits e chiavi da 128, 192 o 256-bits, negli esempi più avanti la assumiamo da 128. È strutturato in questo modo:

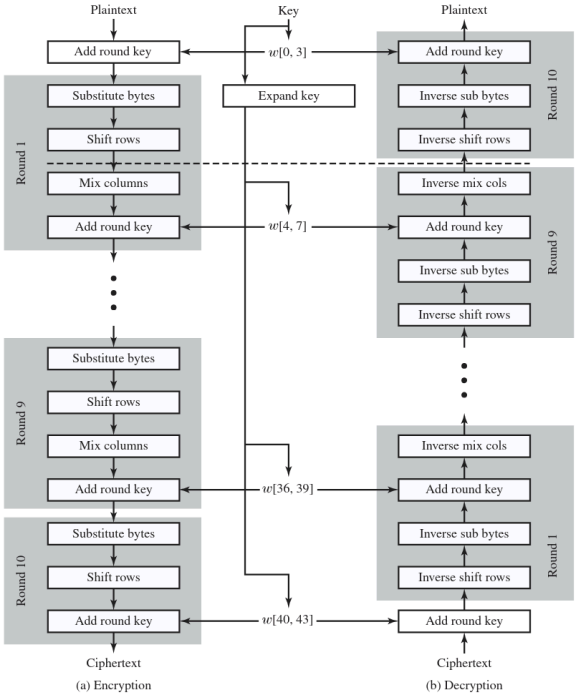
Vuole come input un blocco da 128-bit, questo blocco viene rappresentato come una matrice di byte. Durante l'esecuzione dell'algoritmo, questo blocco viene copiato nello **State array** che viene modificato ad ogni stage di cifratura o decifratura. Dopo l'ultimo stage l'array viene copiato in una matrice di output. Anche la chiave viene rappresentata in una matrice di bytes, questa chiave viene estesa in un array di words, ogni words è composta da 4 bytes e l'array in totale contiene 44 words, ovviamente aumentando i bit della chiave aumentiamo anche le words nell'array.

L'ordine usato nelle matrici è per colonne, quindi ad esempio i primi 4 bytes del plaintext occuperanno la prima colonna della matrice:

B1	B5	B9	B13
B2	B6	B10	B14
B3	B7	B11	B15
B4	B8	B12	B16

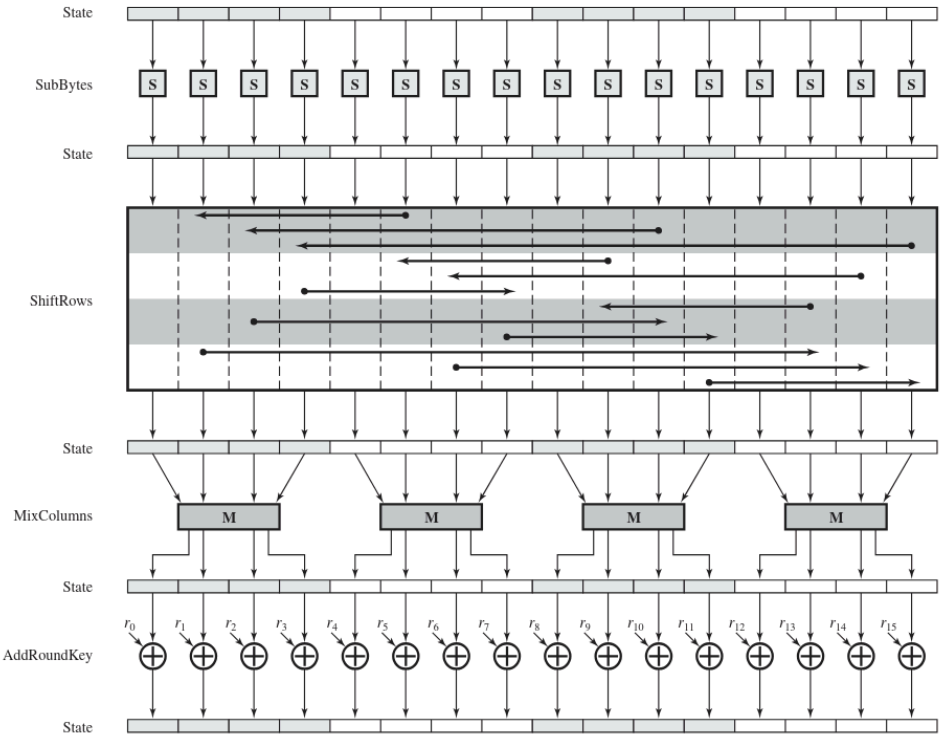
In questo algoritmo i round:

- Non seguono architettura Feistel.
- La chiave iniziale viene estesa in un array di 44 words da 32bit, 4 words (128 bit) costituiscono una chiave utilizzabile in un round.
- Vengono utilizzati 4 diversi stage, 1 di permutazione e 3 sostituzioni:
  - **Substitute bytes**: Utilizza delle tabelle per fare sostitutizioni byte-a-byte.
  - **Shift Rows**: Permutazione effettuata sulle righe.
  - **Mix Columns**: Sostituzione che altera ogni byte in una colonna andandoli a smistare sulle restanti colonne.
  - **Add Round Key**: Un semplice *XOR* del blocco corrente con la chiave del round corrente.



La struttura è relativamente semplice sia per la cifratura che la decifratura. La cifratura inizia con una *Add Round Key* seguita da 9 round contenenti tutti gli stage visti sopra, seguiti infine da un round di tre stage.

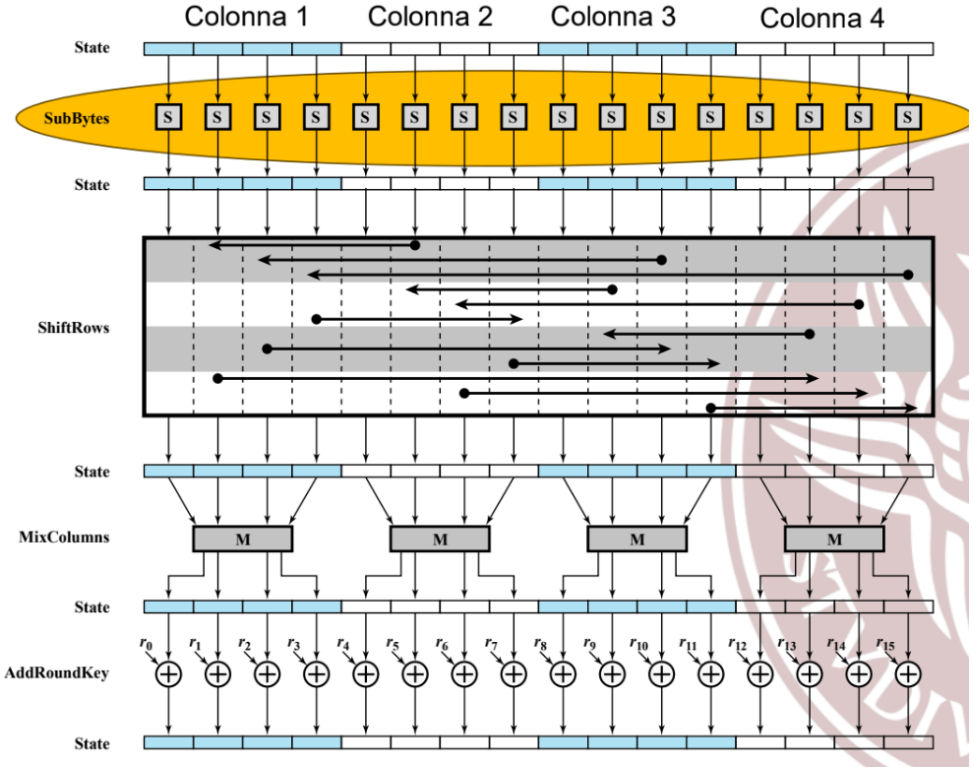
Nella figura sono rappresentati tutti gli stage di un round:



- Solo lo stage *Add Round Key* utilizza la chiave e per questo viene svolto all'inizio e alla fine della cifratura, tutti gli altri stage applicati all'inizio o alla fine sono invertibili senza conoscenza della chiave e non aggiungerebbero sicurezza.

- Anche gli altri sono utili però perchè servono a creare confusione nei dati, ma come detto prima da soli non bastano perchè non utilizzano la chiave.
- L'algoritmo di decifratura utilizza le funzioni inverse degli stage e per la chiave basta fare lo *XOR* come per la cifratura, inoltre va anche utilizzata la chiave estesa ma in ordine inverso.
- Dato che tutti gli stage sono invertibili, noteremo che allo stesso *livello verticale* nello *state array* avremo sempre lo stesso array.

Vediamo adesso i principali passaggi:



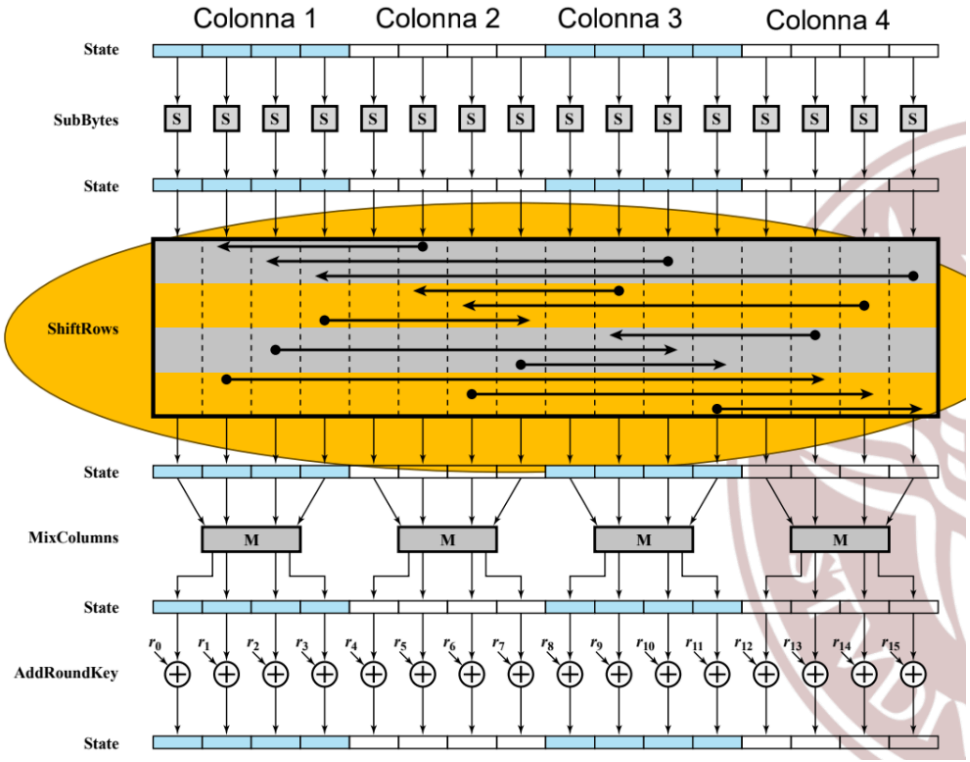
1. Effettuiamo una **Substitute Bytes**, una sostituzione byte a byte del blocco attraverso una tabella. Ovviamente esiste anche la matrice per la decifratura e queste due devono essere progettate in modo che ci sia poca correlazione fra byte in input e di output. Inoltre non deve essere possibile definire una relazione matematica per capire l'output dall'input.

La tabella è nella forma:

		y															
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
x	0	63	7C	77	7B	F2	6B	6F	C5	30	01	67	2B	FE	D7	AB	76
	1	CA	82	C9	7D	FA	59	47	F0	AD	D4	A2	AF	9C	A4	72	C0
	2	B7	FD	93	26	36	3F	F7	CC	34	A5	E5	F1	71	D8	31	15
	3	04	C7	23	C3	18	96	05	9A	07	12	80	E2	EB	27	B2	75
	4	09	83	2C	1A	1B	6E	5A	A0	52	3B	D6	B3	29	E3	2F	84
	5	53	D1	00	ED	20	FC	B1	5B	6A	CB	BE	39	4A	4C	58	CF
	6	D0	EF	AA	FB	43	4D	33	85	45	F9	02	7F	50	3C	9F	A8
	7	51	A3	40	8F	92	9D	38	F5	BC	B6	DA	21	10	FF	F3	D2
	8	CD	0C	13	EC	5F	97	44	17	C4	A7	7E	3D	64	5D	19	73
	9	60	81	4F	DC	22	2A	90	88	46	EE	B8	14	DE	5E	0B	DB
	A	E0	32	3A	0A	49	06	24	5C	C2	D3	AC	62	91	95	E4	79
	B	E7	C8	37	6D	8D	D5	4E	A9	6C	56	F4	EA	65	7A	AE	08
	C	BA	78	25	2E	1C	A6	B4	C6	E8	DD	74	1F	4B	BD	8B	8A
	D	70	3E	B5	66	48	03	F6	0E	61	35	57	B9	86	C1	1D	9E
	E	E1	F8	98	11	69	D9	8E	94	9B	1E	87	E9	CE	55	28	DF
	F	8C	A1	89	0D	BF	E6	42	68	41	99	2D	0F	B0	54	BB	16

I primi 4 bit vengono usati per la coordinata *x* mentre gli ultimi 4 per la *y*. Ad esempio il byte con rappresentazione esadeciamale *95* viene tradotto in *2A* ovvero il byte presente alla riga 9, colonna 5.

A questo punto salviamo il risultato nello state array e passiamo all'operazione successiva:



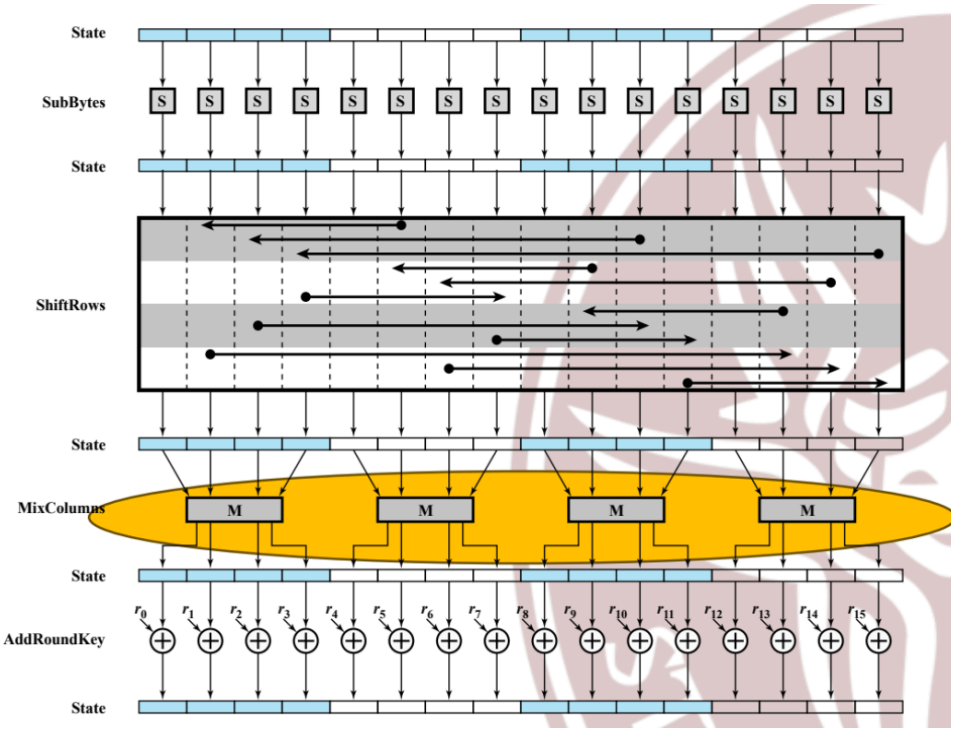
2. Effettuiamo uno **Shift Rows**, nello specifico:

- La prima riga non viene modificata.
- La seconda viene spostata di un byte a sinistra.
- La terza di 2 byte a sinistra.
- La quarta di 3 byte a sinistra.

Gli shift sono circolari quindi chi arriva al limite sinistro «sbuca» all'estremità destra. Questa trasformazioni si assicura che i 4 byte di una colonna vengano smistati in 4 colonne diverse.

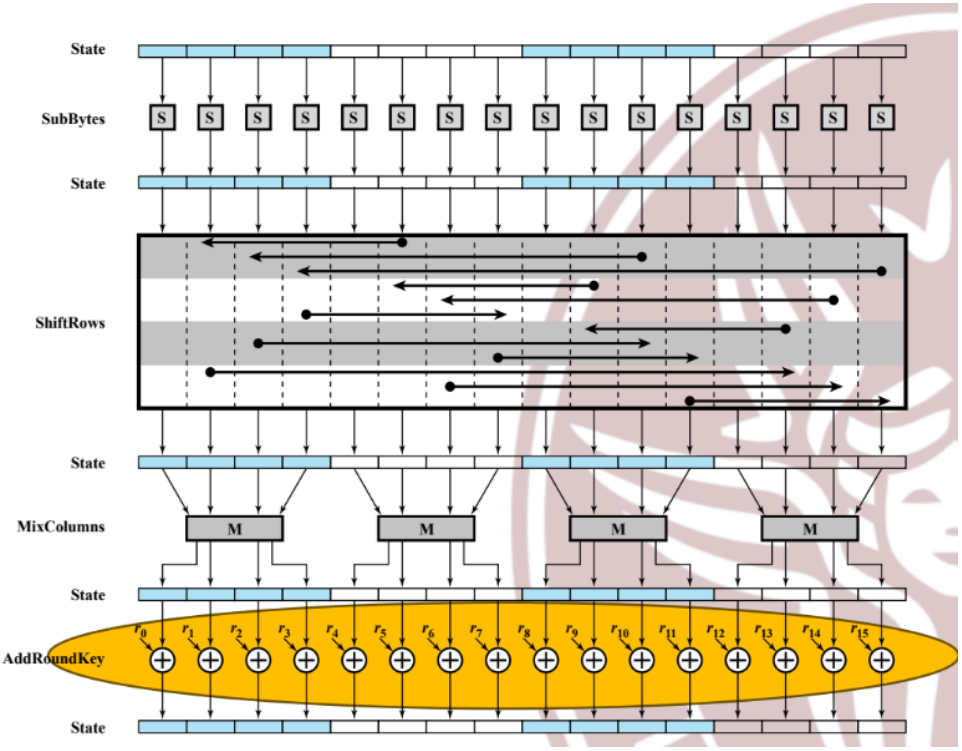
Una volta finito salviamo il risultato nello state array.

3. A questo punto, possiamo passare alla **Mix Column**:



In questa operazione prendiamo in input i 4 byte di una colonna e otteniamo in output i nuovi 4 byte, non ci deve essere perdita di informazione, questo significa che l'**output dipende dall'input**.

4. Come ultima operazione facciamo l'**Add Round Key**:



Per quanto riguarda l'algoritmo di espansione della chiave:

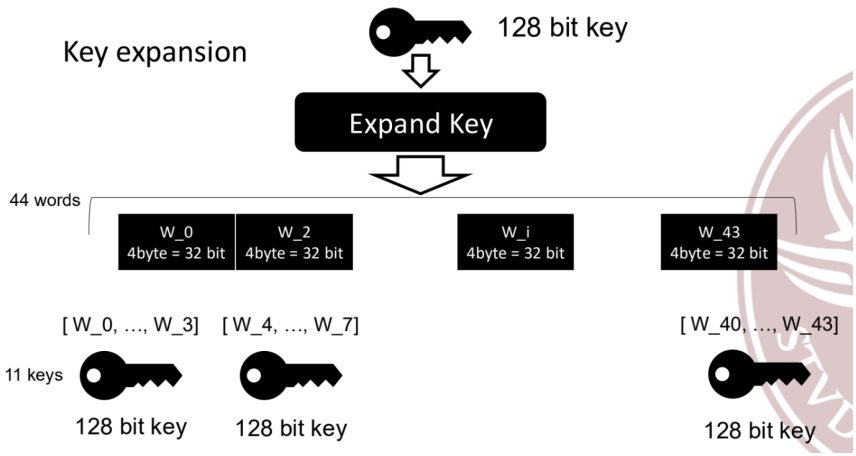
- Ha come input la chiave composta da 4 words, ovvero 16bytes (128-bits).
- Restituisce come output un vettore da 44 words ovvero 156bytes, questo è sufficiente per garantire il numero di chiavi necessario per tutti i round.

Queste bastano per tutto l'algoritmo dato che le prime 4 word vengono usate per il primo *Add Round Key* e le restanti 40 nei successivi 10 round. Ovviamente all'aumentare della lunghezza della chiave aumentano anche i round.

Gli steps sono:
1. Viene copiata la chiave nelle prime 4 words dell'array esteso.
2. Il resto viene riempito con 4 words alla volta dove la word `word[i]` dipende da `w[i-1]` e `w[i-4]`.

Esistono diversi complessi algoritmi *finite-field* per generare questa chiave estesa.

Nello specifico otteniamo:



4.1.2. Stream Ciphers

La cifratura a flusso va a cifrare i byte continuamente, dobbiamo generare sequenze di chiavi che cambiano continuamente, non possiamo cifrare ogni byte con la stessa chiave.

Data una chiave, viene generata sempre la stessa sequenza di chiavi per i successivi byte.

Seed

Viene utilizzato lo stesso concetto dei seed per i generatori di numeri, con un generatore se utilizziamo lo stesso seed otteniamo in output la stessa sequenza di numeri.

4.1.2.1. RC4

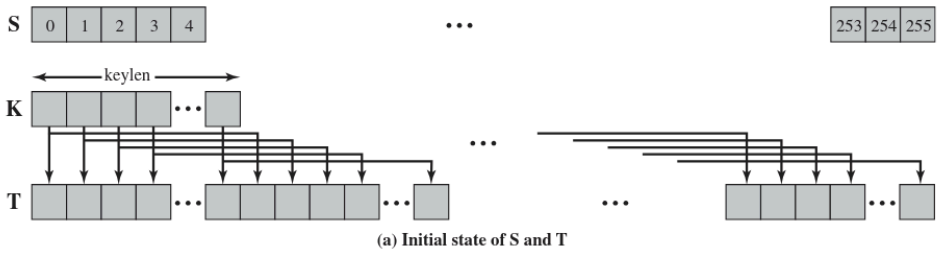
È un algoritmo estremamente semplice, utilizza una chiave di lunghezza variabile da 1 a 256bytes (da 8 a 2048bits) che viene usata per inizializzare lo **state array** S di 256-byte. In ogni momento dell'algoritmo avremo che S contiene una permutazione di tutti i numeri a 8 bit ovvero di tutti i numeri da 0 a 255. Per la cifratura e decifratura, un byte k viene generato da S selezionando una delle 255 entries, ad ogni valore k selezionato viene effettuata una nuova permutazione dei valori di S.

Si compone di diverse fasi

- **Inizializzazione**: Viene inizializzato un array S con tutti i valori da 0 a 255 in modo che `S[0] = 0, S[1] = 1, ..., S[255] = 255`.

Viene anche inizializzato anche un vettore T, se la lunghezza della chiave K è 256bytes allora K viene trasferita su T, altrimenti per una chiave di lunghezza *keylen* bytes, i primi *keyleb* bytes di K vengono copiati in T e poi K viene ripetuta tante volte quanto necessario per riempire T. Possiamo riassumere questa operazione con il seguente pseudocodice:

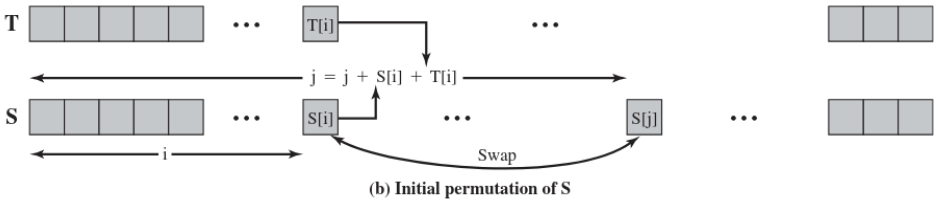
```
1  for i = 0 to 255 do
2  S[i] = i;
3  T[i] = K[i mod keylen]
```



(a) Initial state of S and T

Viene poi effettuata la **prima permutazione**: Per fare questa utilizziamo T, scorriamo tutto S e per ogni `S[i]` lo swappiamo con un altro byte in S basandoci sullo schema fornito da `T[i]`. Nello specifico:

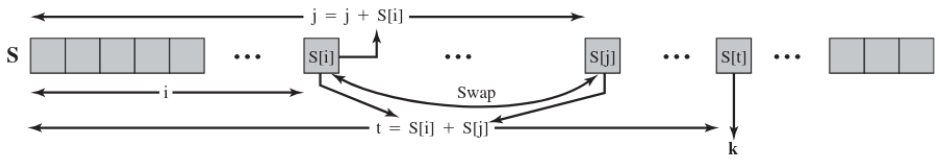
```
1  j = 0;
2  for i = 0 to 255 do
3    j = (j + S[i] + T[i]) mod 256;
4    Swap (S[i], S[j]);
```



(b) Initial permutation of S

Iniziamo la **generazione delle chiavi**: A questo punto la chiave non è più necessaria e la generazione avviene ciclando su S e per ogni elemento `S[i]` lo swappiamo con un altro elemento di S basandoci sulla corrente configurazione di S. Se raggiungiamo l'elemento in posizione 255 torniamo a 0. Possiamo usare il seguente pseudocodice:

```
1  i, j = 0;
2  while (true)
3    i = (i + 1) mod 256;
4    j = (j + S[i]) mod 256;
5    Swap (S[i], S[j]);
6    t = (S[i] + S[j]) mod 256;
7    k = S[t];
```



(c) Stream generation

Per cifrare effettuiamo uno *XOR* del valore k con il successivo byte del plaintext mentre per decifrare usiamo sempre lo *XOR* del valore k con il byte successivo del ciphertext.

4.2. Block Cipher Mode Operation

Gli algoritmi con chiave simmetrica e a blocchi processano un blocco alla volta, quindi quando il testo è più lungo è necessario spezzarlo in più blocchi. Possiamo applicare i block cipher in 5 **modes of operation** individuate dal NIST:

Mode	Description	Typical Application
Electronic Code book (ECB)	Each block of 64 plaintext bits is encoded independently using the same key.	• Secure transmission of single values (e.g., an encryption key)
Cipher Block Chaining (CBC)	The input to the encryption algorithm is the XOR of the next 64 bits of plaintext and the preceding 64 bits of ciphertext.	• General-purpose block-oriented transmission • Authentication
Cipher Feedback (CFB)	Input is processed s bits at a time. Preceding ciphertext is used as input to the encryption algorithm to produce pseudorandom output, which is XORed with plaintext to produce next unit of ciphertext.	• General-purpose stream-oriented transmission • Authentication
Output Feedback (OFB)	Similar to CFB, except that the input to the encryption algorithm is the preceding DES output.	• Stream-oriented transmission over noisy channel (e.g., satellite communication)
Counter (CTR)	Each block of plaintext is XORed with an encrypted counter. The counter is incremented for each subsequent block.	• General-purpose block-oriented transmission • Useful for high-speed requirements

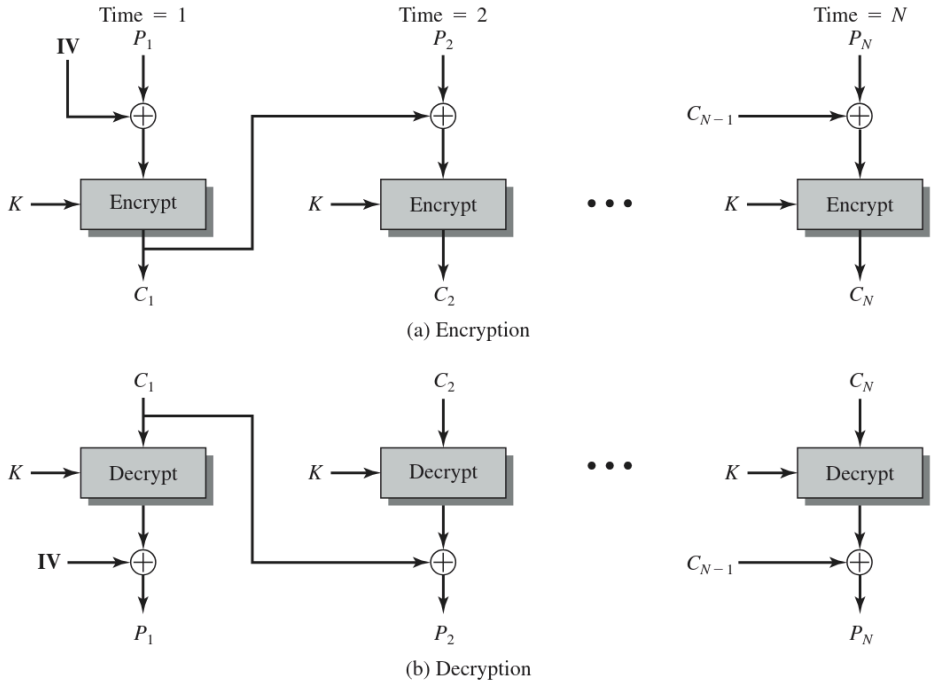
4.2.1. Electronic Codebook Mode (ECB)

È il metodo più semplice, infatti se i blocchi hanno dimensione b bits allora processiamo il testo b bits alla volta cifrandoli sempre con la stessa chiave. Per messaggi molto lunghi questa tecnica non è molto sicura infatti se ci sono ripetizioni nel testo queste avranno la stessa cifratura; oppure se il messaggio è molto strutturato potrebbe essere possibile effettuare cryptanalysis, ad esempio se ogni pagina inizia con gli stessi header abbiamo già delle combinazioni che possiamo sfruttare.

Per risolvere questo problema abbiamo bisogno di una tecnica che, anche se troviamo blocchi uguali, produce una cifratura diversa.

4.2.2. Cipher Block Chaining Mode (CBC)

In questa modalità di cifratura a blocchi l'input dell'algoritmo è lo *XOR* del blocco in plaintext corrente e il precedente blocco cifrato, viene comunque utilizzata la stessa chiave.



Per la decifratura ogni blocco cifrato viene passato all'algoritmo, il risultato viene messo in *XOR* con il precedente blocco cifrato per produrre il blocco in chiaro.

Per capire come funziona e soprattutto dimostrare il funzionamento possiamo scrivere:

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

Dove $E[K, X]$ è la funzione di cifratura effettuata sul blocco X utilizzando la chiave K. Continuiamo la dimostrazione scrivendo:

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

Per produrre il primo blocco cifrato abbiamo bisogno di un **initilization vector** (IV) con cui effettuare lo *XOR* con il primo blocco di plaintext. Per la decifratura facciamo lo stesso ragionamento quindi il vettore IV viene messo in *XOR* con l'output dell'algoritmo di decifratura per recuperare il primo blocco di plaintext. Il vettore IV deve essere noto a mittente e destinatario e per avere più sicurezza possiamo proteggere anch'esso con una chiave. Possiamo farlo mandando il vettore usando cifratura ECB. Perché è importante proteggere IV? Perché anche se un attaccante non conosce la chiave ma riesce a leggere e modificare IV può modificare il primo blocco del messaggio, infatti:

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

Adesso usiamo la notazione $X[j]$ per indicare il bit in posizione j:

$$P_1[i] = IV[i] \oplus D(K, C_1)[i]$$

E utilizzando la proprietà dello *XOR*:

$$P_1[i]' = IV[i]' \oplus D(K, C_1)[i]$$

Dove la notazione con <“> indica il complemento, questo significa che se un attccante può modificare i bit in IV allora può modificare anche i bit nel primo blocco senza conoscere la chiave.

4.2.2.1. Propagazione degli errori

In ECB se avviene un errore di trasmissione nel testo cifrato, solo il corrispondente blocco in chiaro viene compromesso. In CBC l'errore si propaga, se avviene un errore in C_k allora corrompe sicuramente anche P_k e P_{k+1}, ma corrompe anche i successivi?

No, supponiamo C_k corrotto e notiamo come il blocco in chiaro P_{k+2} dipenda soltanto da C_{k+1} e C_{k+2}:
- $C_k' \neq C_k$ corrotto
- $P_k = C_{k-1} \oplus D(K, C_k)$ corrotto
- $P_{k+1} = C_k \oplus D(K, C_{k+1})$ corrotto
- $P_{k+2} = C_{k+1} \oplus D(k, C_{k+2})$ non corrotto

Se invece c'è un errore nella versione originale di P_k? Attraverso quanti blocchi si propaga l'errore? Che effetti nota il destinatario? Un errore in P_k corrompe C_k ma dato che C_k è l'input per calcolare C_{k+1} anche quest'ultimo viene corrotto. Questo errore viene portato avanti in modo indefinito su tutti i blocchi cifrati successivi a C_k. Alla fine della ricezione però l'algoritmo di decifratura ricostruisce il messaggo corretto per tutti i blocchi ad eccezione del blocco contentente l'errore.

Cifratura (sempre corrotti)

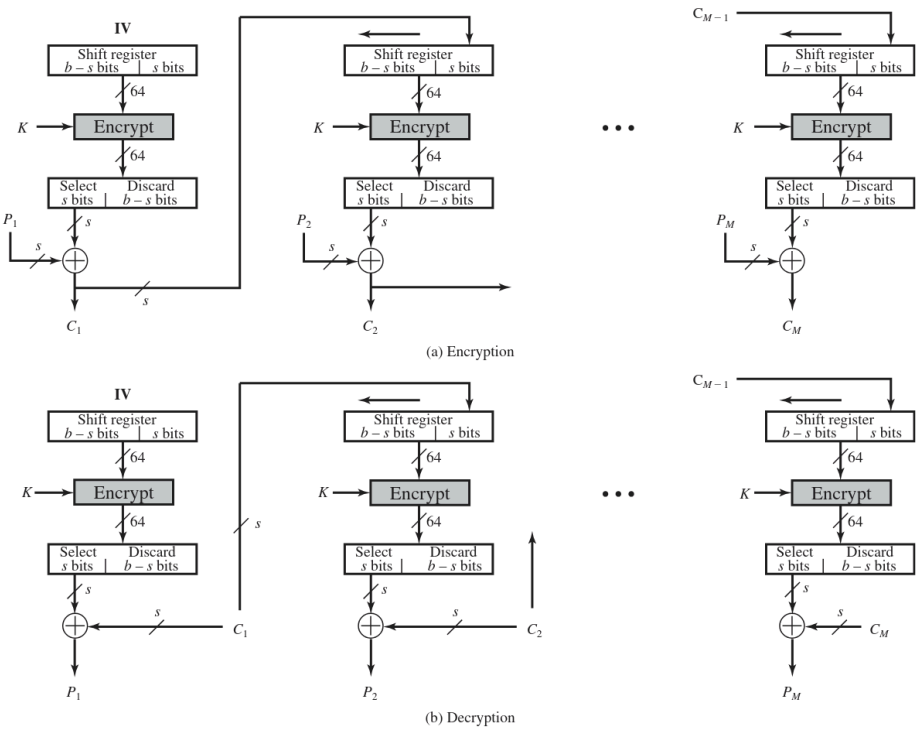
- $P_k' \neq P_k$
- $C_k = E(K, [C_{k-1} \oplus P_k'])$
- $C_{k+1} = E(K, [C_k \oplus P_{k+1}])$
- $C_{k+2} = E(K, [C_{k+1} \oplus P_{k+2}])$

Decifratura

- $P_k = C_{k-1} \oplus D(K, C_k) = C_{k-1} \oplus C_{k-1} \oplus P_k'$ corrotto
- $P_{k+1} = C_k \oplus D(K, C_{k+1}) = C_k \oplus C_k \oplus P_{k+1}$ ok
- $P_{k+2} = C_{k+1} \oplus D(K, C_{k+2}) = C_{k+1} \oplus C_{k+1} \oplus P_{k+2}$ ok

4.2.3. Cipher Feedback Mode (CFB)

Con questa modalità possiamo convertire qualsiasi block cipher in uno stream cipher.

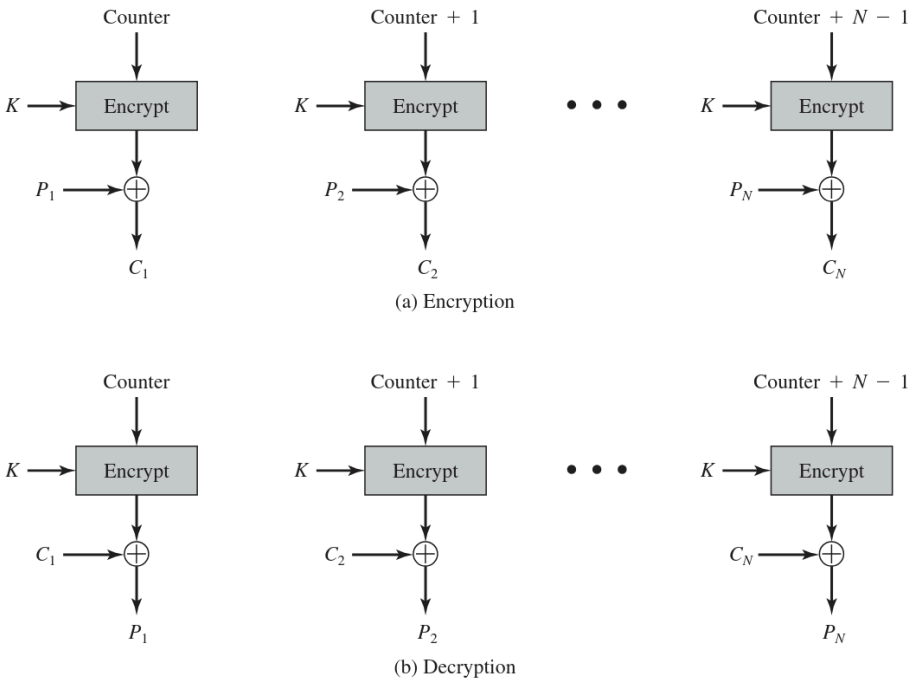


Nell'immagine sopra viene assunta come unità di trasmissione s, di solito sono 8 bits. Anche qui il plaintext viene concatenato nella funzione, quindi il testo cifrato di ogni unità di testo in chiaro è in funzione dell'unità precedente.

La funzione di cifratura prende in input uno shift di b bit che per la prima unità viene applicato ad un vettore IV, i s bit più significativi del risultato della funzione vengono messi in *XOR* con la prima unità di plaintext P_1 per produrre la prima unità di testo cifrato C_1 che verrà trasmessa. Il contenuto dello shift register viene shiftato a sinistra di s bits e C_1 viene piazzato negli s bits più a destra del registro. Questo processo continua finché non vengono cifrate tutte le unità di plaintext.

Per la decifratura viene utilizzato lo stesso schema, l'unica differenza è che le unità di testo cifrato ricevute vengono messe in *XOR* con l'output che viene prodotto dalla funzione di cifratura.

4.2.4. Counter Mode (CTR)



Viene utilizzato un *counter* che corrisponde alla grandezza dei blocchi del plaintext, l'unico requisito è che il valore counter sia diverso per ogni blocco plaintext che viene cifrato. Di solito questo valore viene inizializzato e poi incrementato ogni volta di 1. Per la cifratura viene cifrato il counter e poi messo in *XOR* con il blocco in chiaro, per la decifratura viene usato il processo inverso.

Questa tecnica presenta diversi vantaggi:
- **Hardware Effiency**: Questa tecnica può essere utilizzata in parallelo su più blocchi.
- **Software Efficiency**: Praticamente uguale a quella hardware, abbiamo molte opportunità per sfruttare il parallelismo ad esempio con processori che utilizzano istruzioni SIMD.
- **Preprocessing**: La cifratura non dipende dall'input, questo significa che se abbiamo abbastanza memoria possiamo precalcolarci dei valori per poi effettuare solo gli *XOR* quando arriva l'input.
- **Random Access**: L'i-esimo blocco del testo in chiaro o del testo cifrato può essere processato senza dipendenze temporali da altri blocchi.
- **Provable Security**: È dimostrato essere sicuro quanto le altre tecniche di block cipher.
- **Simplicity**: A differenza di ECB e CBC, il CTR ha bisogno soltanto di implementare un algoritmo di cifratura e non di decifratura.

4.3. Cryptanalysis

Il processo con il quale un attaccante ricava delle informazioni su un messaggio o una chiave. La strategia dell'attaccante dipende da:
- Algoritmo di cifratura
- Informazioni a disposizione dell'attaccante (cosa viene inviato nel messaggio o altro)

Ci sono vari attacchi in base a cosa si conosce:
- Ciphertext only

- Known plaintext
- Chosen plaintext
- Chosen ciphertext
- Chosen text

Supponiamo che l'attaccante conosca sempre l'algoritmo di cifratura utilizzato (non sempre vero nella realtà)

Ciphertext only

L'attaccante conosce soltanto il testo cifrato, può provare attacchi **bruteforce** sulla chiave, provare a capire dei pattern del test se, ad esempio, conosce in che lingua è scritto il messaggio originale e quindi capire quali sono le parole più ripetute.

Known plaintext

L'attaccante conosce uno o più messaggi in chiaro e la loro cifratura, oppure alcuni pattern presenti nel testo in chiaro. Con queste conoscenze l'attaccante potrebbe essere in grado di dedurre la chiave in base a come viene trasformato il testo.

Chosen plaintext

L'attaccante è in grado di accedere al sistema e inserire un suo messaggio che presenta dei pattern in grado di rilevare la struttura della chiave. Ovviamente deve essere in grado di ottenere il messaggio cifrato risultante

Chosen ciphertext

L'attaccante ha accesso al sistema per decifrare, seleziona un testo cifrato e lo invia al sistema, analizza il risultato per capire informazioni sulla chiave.

Chosen text

L'attaccante ha accesso sia alla cifratura che alla decifratura, può far cifrare messaggi scelti da lui e anche decifrare.

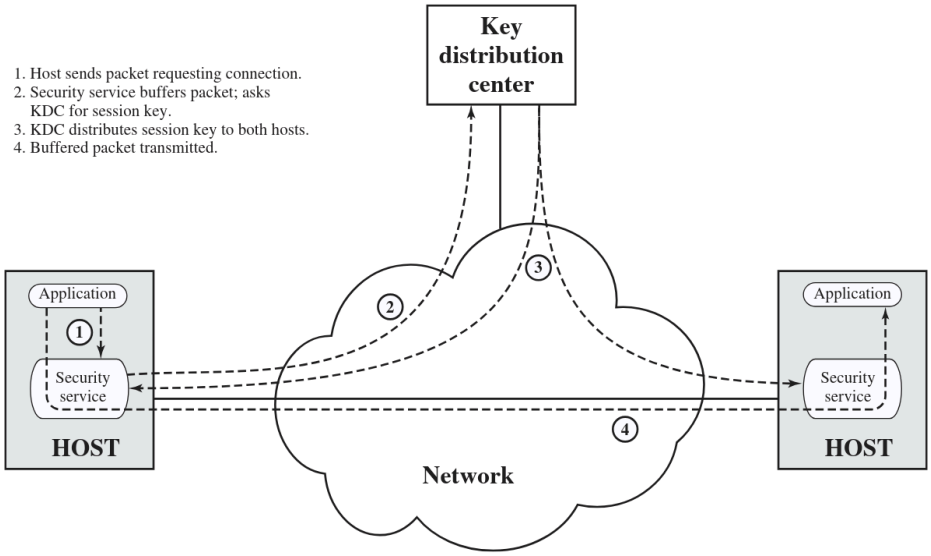
Computationally Secure Encryption Schemes

La cifratura è computazionalmente sicura se:
- Il costo per romperla è maggiore al valore delle informazioni che si ricavano.
- Il tempo per romperla supera il tempo utile di quell'informazione.

4.4. Key Distribution

Abbiamo visto che questi algoritmi funzionano con chiave simmetrica, questo significa che mittente e destinatario devono condividerla in modo sicuro. Ci sono diversi modi per farlo, supponiamo le due parti siano A e B:
- A potrebbe portare fisicamente la chiave a B.
- Qualcuno di esterno potrebbe selezionare e portare la chiave a B.
- Se A e B già avevano una chiave per parlare potrebbero usarla per trasmettere la nuova chiave.
- Se A e B hanno una connessione sicura con C questo potrebbe inviare la chiave al loro posto.

Le prime due opzioni prevedono uno scambio fisico, la terza è possibile ma se un attaccante entra in possesso di una chiave allora avrà anche tutte le successive. L'opzione 4 rimane la più attuabile:



In questo schema ignoriamo la cifratura nei collegamenti, potrebbe esserci o no, identifichiamo però due tipi di chiave:

- **Session Key**: Quando due *end systems* vogliono comunicare stabiliscono una connessione logica, per tutta la durata della sessione tutti i dati vengono cifrati con una one-time session key, alla fine della sessione la chiave viene distrutta.
- **Permanent Key**: Questa chiave viene utilizzata per distribuire le sessions key.

Questa configurazione consiste in:

- **Key Distribution Center:** Il Key Distribution Center (KDC) determina quali sistemi sono autorizzati a comunicare fra di loro, quando due di loro ottengono l'autorizzazione questo gli invia una session key per la connessione.
- **Security Service Module (SSM)**: Esegue la cifratura e ottiene le session key degli utenti.

1. Un host invia una richiesta per parlare con un altro host
2. il SSM salva il pacchetto e chiede al KDC l'autorizzazione
3. Tra SSM e KDC avviene una comunicazione cifrata usando una master key condivisa solo fra loro, se il KDC approva la comunicazione genera una session key e la invia ai due SSM utilizzando una chiave permanente per ogni SSM.
4. L'SSM può adesso stabilire la connessione tra i due sistemi e rilasciare il pacchetto di connessione.

Tutti i dati che vengono scambiati adesso saranno cifrati dall'SSM utilizzando la session key.

5. Cifratura a Chiave Asimmetrica e Message Authentication

5.1. Secure Hash Functions

Abbiamo già visto precedentemente le proprietà che dovrebbero essere rispettate da una buona hash function, adesso vediamo diverse funzioni con un focus sul gruppo delle **Secure Hash Algorithms (SHA)**.

5.1.1. Simple Hash Functions

Tutte le funzioni di hash operano nel seguente modo:
- L'input viene visto come una sequenza di blocchi da n-bits.

L'input viene processato un blocco alla volta in modo iterativo, alla fine dell'operazione si ottiene un risultato da n-bit. Per capire questo meccanismo si può prendere come esempio una delle funzioni più semplici ovvero con lo *XOR*, possiamo processarlo nel seguente modo:

$$C_i = b_{i1} \oplus b_{i2} \oplus ... \oplus b_{im}$$

Dove:
- C_i: L'i-esimo bit dell'hash code con $1 \leq i \leq n$
- m: Numero dei blocchi da n bits in input
- b_{ij}: i-esimo bit del j-esimo blocco
- $\oplus$: *XOR* operation

Quindi ogni blocco viene messo in XOR con il successivo e alla fine otteniamo un valore grande quanto la grandezza dei blocchi.

Possiamo vedere questo processo graficamente con la seguente tabella:

	Bit 1	Bit 2	• • •	Bit *n*
Block 1	b_{11}	b_{21}		b_{n1}
Block 2	b_{12}	b_{22}		b_{n2}
	• • •	• • •	• • •	• • •
Block *m*	b_{1m}	b_{2m}		b_{nm}
Hash code	C_1	C_2		C_n

Questa operazione non fa altro che produrre un bit di parità per ogni bit ed è conosciuta come **longitudinal redundancy check**, è abbastanza efficace per check di integrità. Però c'è da notare che la probabilità che un errore nei dati non porti ad un cambiamento nell'hash code è di 2^{-n}. Più i dati sono formattati e più perde di efficacia, ad esempio in quasi tutti i normali file di testo il primo bit di ogni ottetto è sempre 0, quindi se viene usato un valore hash da 128-bit (grandezza del risultato) invece di un'efficacia di 2^{-128} avremo 2^{-112}. Infatti dato che un ottetto è fatto da 8bit se abbiamo un hash value da 128 significa che abbiamo 16 ottetti (16 x 8), ma sappiamo che il primo bit di ognuno è 0 quindi ci saranno 16 bit bloccati a 0 che non cambieranno mai.

Per migliorare questo algoritmo possiamo applicare un **1-bit circular shift**, o rotazione, sull'hash value dopo che ogni blocco viene processato, possiamo riassumere l'operazione in:
1. Impostiamo l'hash value da n-bit a zero.
2. Processiamo ogni blocco successivo nel seguente modo:

- Ruotiamo il valore hash corrente a sinistra di 1-bit.
- *XOR* del blocco con il valore hash.

Questo serve a «randomizzare» l'input ed abbassare il numero di *regolarità* all'interno del testo, nonostante questo garantisca buona integrità dei dati non garantisce sicurezza. Questo perché lo *XOR* è bidirezionale e facilissima da invertire, un attaccante potrebbe:

1. Intercettare un messaggio vero e il suo Hash Value valido
2. Scrivere un messaggio a suo piacimento
3. Calcolare l'hash di questo messaggio con lo stesso algoritmo e ottenere un hash falso
4. Aggiunge in fondo al messaggio un falso blocco di bit senza senso
5. Poiché lo XOR è prevedibile l'attaccante può calcolare matematicamente l'esatto valore che questo blocco extra deve avere per far si che sommandolo al risultato finale diventi magicamente indentico all'hash value vero.

Cifrare tutto non è la soluzione in questo caso, era stata proposta una soluzione dove oltre allo XOR applicato a blocchi da 64bits veniva cifrato tutto il messaggio con CBC, il procedimento era il seguente:

- Dato un messaggio in blocchi da 64bits $X_1, X_2, ..., X_N$ definiamo l'hash code C come lo XOR blocco a blocco di tutti i blocchi e con l'hash code messo come ultimo blocco:

$$C = X_{N+1} = X_1 \oplus X_2 \oplus ... \oplus X_N$$

Poi viene cifrato il messaggio intero e inserito l'hash code per produrre il messaggio cifrato $Y_1, Y_2, ..., X_{N+1}$. Sono stati scoperti diversi modi in cui il testo cifrato può essere manipolato e non rilevato dall'hashcode. Per esempio (seguendo le regole del CBC):

$$X_1 = IV \oplus D(K, Y_1)$$
$$X_i = Y_{i-1} \oplus D(K, Y_i)$$
$$X_{N+1} = Y_N \oplus D(K, Y_{N+1})$$

Ma sappiamo che X_{N+1} è l'hash code:

$$\begin{aligned} X_{N+1} &= X_1 \oplus X_2 \oplus ...X_N \\ &= [IV \oplus D(K, Y_1)] \oplus [Y_1 \oplus D(K, Y_2)] \oplus ... \oplus [Y_{N-1} \oplus D(K, Y_N)] \end{aligned}$$

Dato che i termini nell'equazione precedente possono essere messi in XOR in qualsiasi ordine, significa che l'hash code non cambia se il testo cifrato viene permutato.

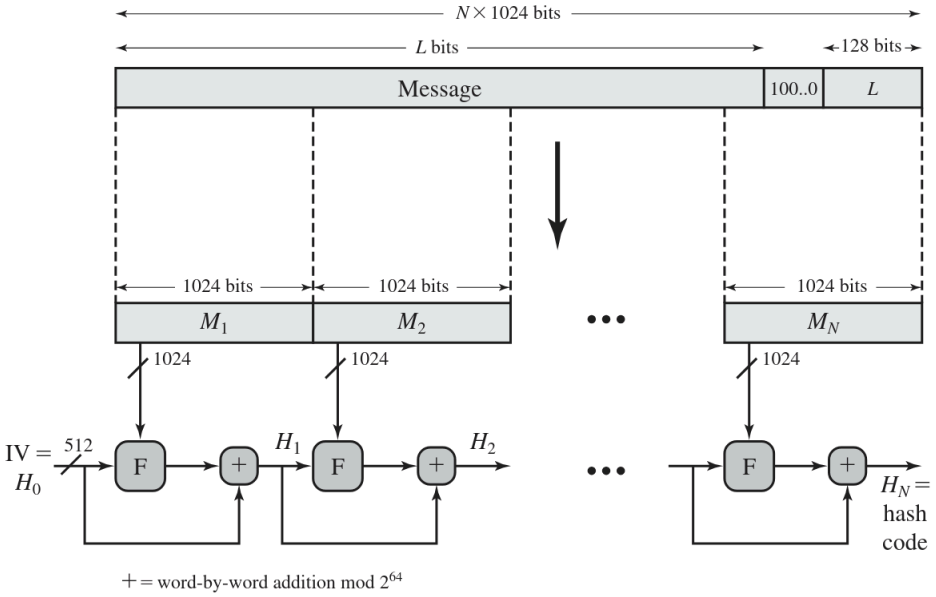
5.1.2. The SHA Secure Hash Function

Esistono diverse versioni di SHA:

	SHA-1	SHA-256	SHA-384	SHA-512
Message digest size	160	256	384	512
Message size	$< 2^{64}$	$< 2^{64}$	$< 2^{128}$	$< 2^{128}$
Block size	512	512	1024	1024
Word size	32	32	64	64
Number of steps	80	64	80	80
Security	80	128	192	256

Nel corso ci concentriamo su SHA-512. L'algoritmo prende in input un messaggio con una lunghezza massima di 2^{128} bits e produce un output di 512-bit chiamato *digest*. L'input viene processato in blocchi da 1024-bit.

Graficamente possiamo vedere il processo:



Consiste in:

1. **Append padding bits**: Al messaggio viene aggiunto un padding in modo che la lunghezza sia congruente a 896 modulo 1024, il padding viene sempre aggiunto anche se il messaggio è alla lunghezza desiderata. Il numero di bit di padding quindi si trova nel range di 1-1024 bits e il padding consiste in un singolo bit impostato a 1 seguito da tutti i bit necessari impostati a 0.
2. **Append Length**: Viene aggiunto alla fine un blocco da 128bit, questo viene trattato come un intero senza segno a 128bit e contiene la lunghezza del messaggio originale (prima dell'aggiunta del padding).

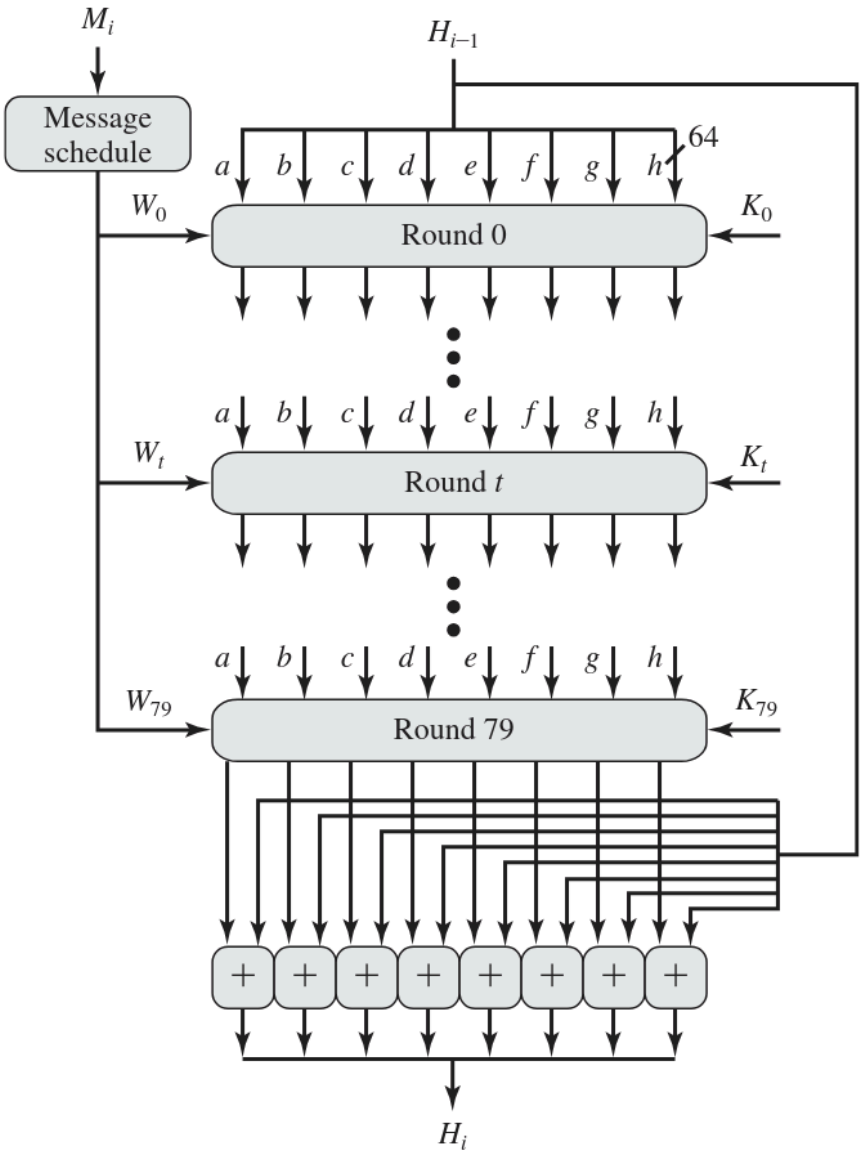
Dopo questi 2 passaggi otteniamo un messaggio che ha una lunghezza multiplo di 1024bits. Nell'immagine il messaggio viene rappresentato come una sequenza di blocchi da 1024bits $M_1, M_2, ..., M_N$ e la lunghezza totale è quindi $N \times 1024$bits.

3. **Initialize hash buffer**: Un buffer da 512bit viene utilizzato per memorizzare i risultati intermedi e finale della funzione di hash. Il buffer può essere rappresentato come 8 registri da 64bits inizializzati a questi valori esadecimali:

$$a = 6A09E667F3BCC908 \quad e = 510E527FADE682D1$$
$$b = BB67AE8584CAA73B \quad f = 9B05688C2B3E6C1F$$
$$c = 3C6EF372FE94F82B \quad g = 1F83D9ABFB41BD6B$$
$$d = A54FF53A5F1D36F1 \quad h = 5BE0CD19137E2179$$

Questi valori sono salvati in formato **big-endian** ovvero dove il bit più significativo si trova nel posizione più a sinistra. I valori sono stati ottenuti prendendo i primi 64bits delle parti frazionarie delle radici quadrate dei primi 8 numeri primi.

4. **Process message in 1024-bit (128-word) blocks**: È il cuore dell'algoritmo, consiste in 80 rounds e nella figura precedente viene indicato come *F*. La logica invece viene illustrata con questa figura:



Ogni round prende come input il buffer da 512bit e aggiorna il suo contenuto. Come input del primo round abbiamo il valore hash intermedio H_{i-1}, ogni round t utilizza un valore a 64bit W_t derivato dal blocco corrente (1024bit) M_i. Ad ogni round si utilizza anche una costante K_t dove $0 \leq t \leq 79$ (una per ogni round), queste costanti sono i primi 64bits delle parti frazionarie

delle radici cubiche dei primi 80 numeri primi. Le costanti forniscono un set di patterns a 64bit randomizzati che dovrebbe rimuovere ogni regolarità presente nel testo in input. Le operazioni eseguite in un round sono *shift circolari* e operazioni booleane basate su *AND, OR, NOT e XOR*. L'output dell'80-esimo round viene sommato insieme all'input del primo round H_{i-1} e viene prodotto H_i, la somma viene fatta in modo indipendente per ognuna delle 8 words del buffer usando un'addizione modulo 2^{64}.

5. **Output**: Dopo che tutti gli N blocchi sono stati processati abbiamo l'output da 512bit.

L'algoritmo SHA-512 ha la proprietà che ogni bit dell'hash code è una funzione di ogni bit in input, le ripetizioni molto complesse della funzione *F* producono risultati molto «disordinati» ed è molto difficile che due messaggi scelti a caso, anche simili, abbiano lo stesso hash code. In ogni caso trovare due messaggi con lo stesso hash code è nell'ordine delle 2^{256} operazioni, mentre trovare un messaggio dato un hash code è nell'ordine delle 2^{512}.

5.2. HMAC

Vediamo l'approccio con hash code per l'**autenticazione dei messaggi**, questa tecnica è motivo di interesse perché:

- Le funzioni hash generalmente vengono eseguite più velocemente rispetto agli algoritmi di crittografia come il *DES*.
- Sono molto più diffuse librerie di codice per funzioni hash.

Funzioni hash come *SHA-1* però non sono state progettate per essere usate come MAC e non possono essere usate in questo modo direttamente perché non si basano su una chiave segreta. Ci sono stati diversi tentativi per introdurre una chiave negli algoritmi hash esistenti e quello che ha ricevuto più supporto è l'**HMAC**, questo è stato anche usato per implementare MAC nelle connesioni internet in vari protocolli.

Obiettivi dell'HMAC:

- Utilizzare, senza modifiche, le esistenti funzioni hash.
- Permettere la sostituzione della funzione hash scelta con altre più sicure.
- Preservare le performance originali della funzione hash.
- Utilizzare le chiavi in modo semplice.
- Avere un'ottima comprensione della «forza» della funzione hash che stiamo usato, il vantaggio infatti è che se la funzione è sicura allora lo è anche HMAC.

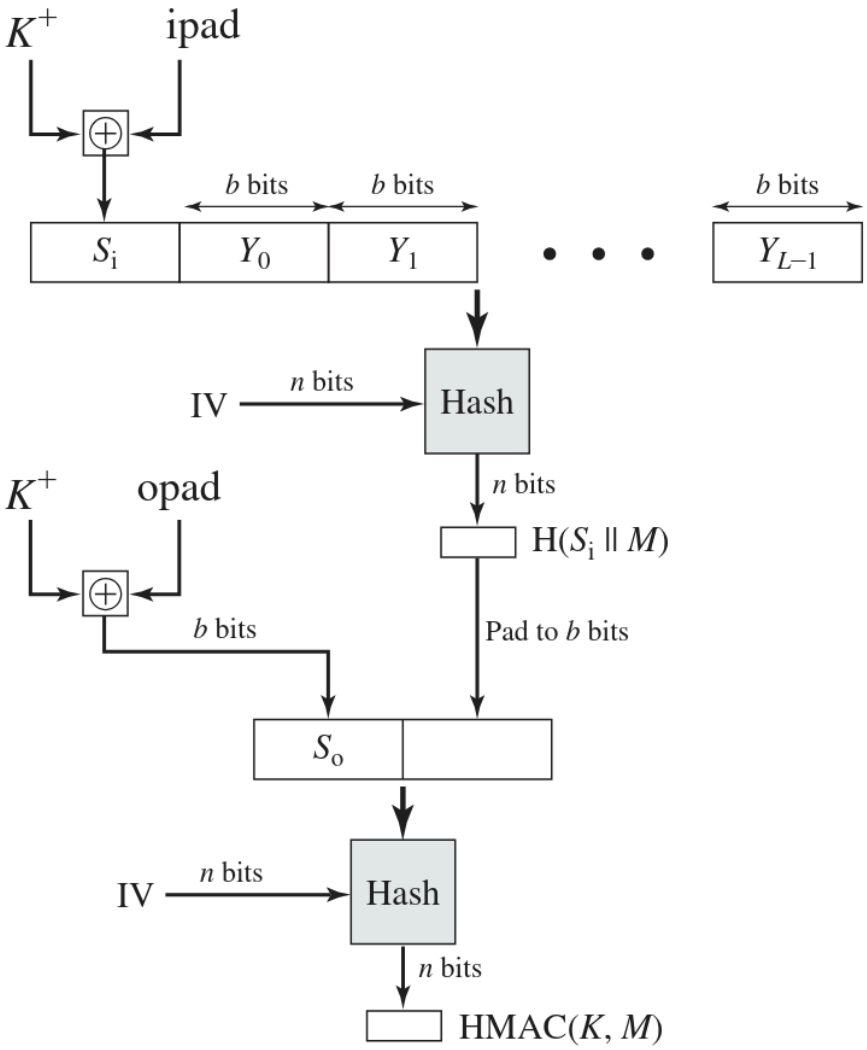
HMAC tratta la funzione hash come una *«black box»*, questo ha due benefici, il primo è che parte dell'algoritmo è già pronta senza modifiche mentre il secondo è che possiamo appunto sostituire facilmente una funzione con un'altra, ad esempio se su quella in uso vengono scoperte delle debolezze.

Vediamo come funziona l'algoritmo, definiamo i seguenti termini:

- H - funzione hash (SHA).
- M - Messaggio input di HMAC (incluso il padding specificato nella funzione hash)
- L - Numero di blocchi in M
- Y_i - Blocco i-esimo di M con $0 \leq i \leq L-1$
- b - Numero di bits in un blocco
- n - Lunghezza dell'hashcode prodotto dalla funzione hash

- K - Chiave segreta, se la lunghezza della chiave è maggiore di b, allora la chiave viene data in input alla funzione hash per produrre una chiave da n-bit, una lunghezza consigliata è $\geq n$.
- K^+ - K con un padding di zeri a sinistra in modo che la lunghezza sia b bits.
- ipad - 00110110 (36 in esadecimale) ripetuto $\frac{b}{8}$ volte
- opad - 01011100 (5C in esaedcimale) ripetuto $\frac{b}{8}$ volte

Graficamente abbiamo:



La formula formale dell'HMAC è:

$$\text{HMAC}(K, M) = H[(K^+ \oplus \text{opad}) \parallel H[k^+ \oplus \text{ipad}] \parallel M]$$

Che possiamo descrivere in questi passaggi:

1. Aggiungi zeri a sinistra di K per creare una stringa lunga b chiamata K^+
2. *XOR* di K^+ con *ipad* per produrre il blocco da b-bit S_i.
3. Aggiungere alla fine di S_i M
4. Applicare H allo stream generato dallo step 3.
5. *XOR* di K^+ con *opad* per produrre il blocco da b-bit S_o
6. Aggiungere alla fine del blocco S_o il risultato dello step 4.

7. Applicare H allo stream generato dallo step 6 per ottenere il risultato finale.

Da notare che lo *XOR* con *ipad* significa invertire una metà dei bit di K e anche con *opad* ma con un set di bit diverso. Passare S_i e S_o all'algoritmo di hash significa infatti creare due chiavi pseudocasuali da K. HMAC dovrebbe avere un tempo d'esecuzione simile alla funzione hash per lunghi messaggi, infatti aggiunge 3 esecuzione della funzione base.

5.2.1. Secutity of HMAC

HMAC fa affidamento quasi interamente sulla funzione hash che utilizza, è stato provato infatti che la probabilità di successo di un attacco contro HMAC da parte di un attaccante che può leggere l'output di un messaggio generato da un utente è equivalente a uno dei seguenti attacchi alla funzione hash:

1. L'attaccante è in grado di calcolare un output della compression function anche con IV random, segreto e sconosciuto all'attaccante.
2. L'attaccante trova collissioni nella hash function anche con IV randomico e segreto.

Nel primo attacco, possiamo vedere la compression function come la hash function applicata ad un singolo blocco da *b*-bit, per questo attacco l'array IV è rimpiazzato da un segreto casuale di n bits. Un attacco di questo tipo alla funzione hash richiede un attacco di forza bruta sulla chiave che è nell'ordine di 2^n, o il *birthday attacck* che è un caso particolare del secondo attacco. Nel secondo attacco, l'attaccante è in ricerca di due messaggi M e M' che producono lo stesso hash quindi $H(M) = H(M')$, questo è il *birthday attacck* detto prima. Per questo tipo di attacco siamo nell'ordine di $2^{\frac{n}{2}}$ per una lunghezza dell'hashcode di n.

Con questi dati potremmo pensare che ad esempio l'algoritmo MD5 non sia più sicuro dato che un numero di tentativi di 2^{64} oggi è abbastanza fattibile, quindi MD5 renderebbe HMAC non sicuro? **No**, questo perché HMAC utilizza una chiave segreta *K*, per calcolare l'HMAC di un messaggio non basta il messaggio stesso ma serve anche questa chiave. Quindi MD5 da solo è debole perché l'attaccante potrebbe mettersi a genere hashcode finché non trova una collisione, ma se usato con HMAC allora l'attacco deve diventare **online** ovvero si deve mettere in ascolta e sperare di trovare due messaggi con lo stesso hash, per farlo dovrebbe catturare circa 2^{64} messaggi, anche con una rete a 1Gbps che trasmette senza mai fermarsi ci vorrebbero comunque 150.000 anni di intercettazioni continue, inoltre questo presuppone che la chiave non venga **mai cambiata** per lo stesso tempo.

5.3. Authenticated Encryption (AE)

NON HO TROVATO LE PAGINE DEL LIBRO SU QUESTO PARAGRAFO, DEVO CAPIRLO E SCRIVERLO MEGLIO

Sono sistemi di cifratura che garantiscono sia confidenzialità che integrità. (Message encryption + message authentication) Sono implementati attraverso block cipher, il primo è stato **OCB AE (2014)**:

- Approvato come standard in *IEEE 802.11* wireless LAN e incluso in *MiniSec*.

Poi abbiamo **OCB** che è simile a ECB:

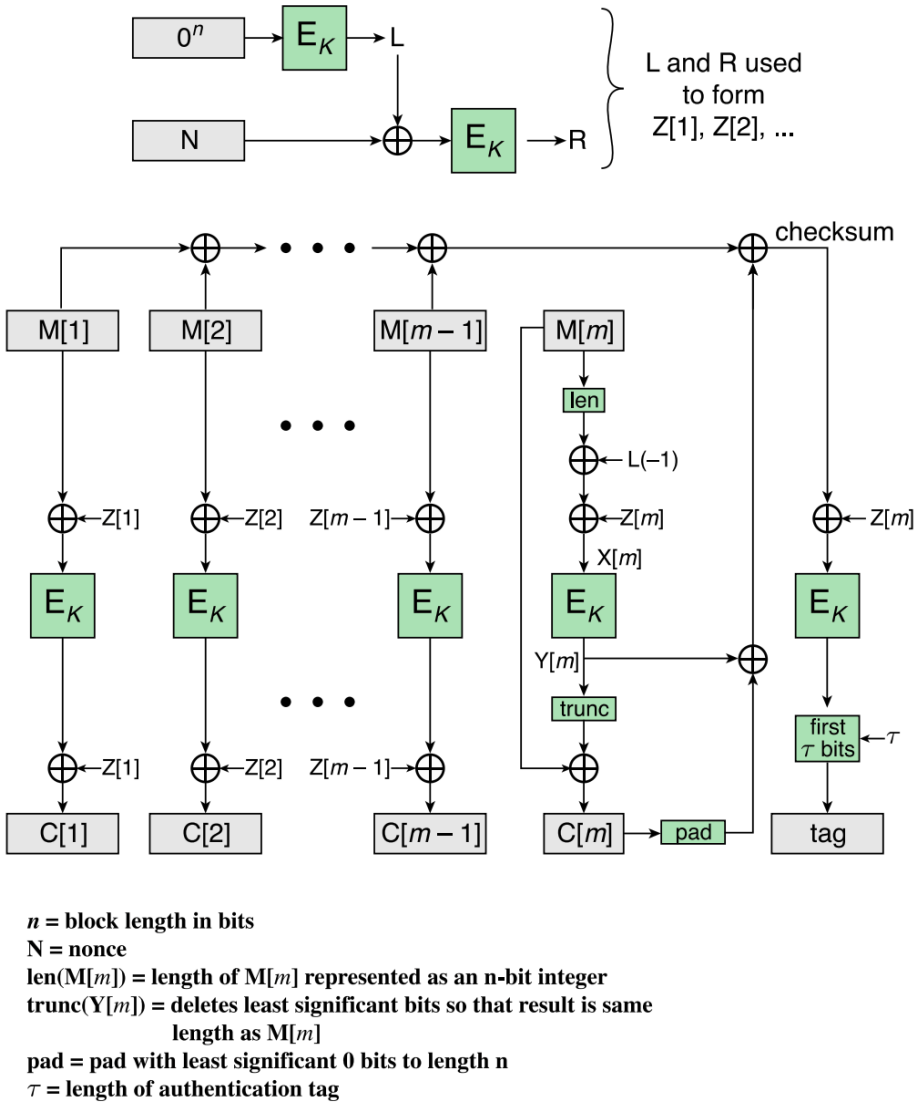
- Ha la stessa debolezza di ECB quindi stesso messaggio con stessa chiave produce lo stesso risultato, non è quindi sicuro per messaggi lunghi.

- Introduce un nuovo vettore Z, che viene utilizzato per effettuare *XOR* con ogni blocco $M[i]$ nel seguente modo:

$$C[i] = E_k(M[i] \oplus Z[i]) \oplus Z[i]$$

- AES è usato come algoritmo di cifratura.
- M viene diviso in blocchi da n-bits.
- Il numero totale di blocchi è $m = \lceil \frac{\text{len}(M)}{n} \rceil$
- N, un valore arbitrario di n-bit chiamato **nonce**
  - Se vengono cifrati più messaggi con la stessa chiave allora sarebbe meglio cambiare questo valore per ogni messaggio.
  - Ogni valore differente di N produce un set diverso di $Z[i]$.

Graficamente abbiamo:



n = block length in bits
N = nonce
len(M[m]) = length of M[m] represented as an n-bit integer
trunc(Y[m]) = deletes least significant bits so that result is same length as M[m]
pad = pad with least significant 0 bits to length n
τ = length of authentication tag

Calcolo di $Z[i]$:

- $L[0] = L = E_k(0^n)$
- $R = E_k(N \oplus L)$
- $L[i] = 2 \cdot L[i-1]; 1 \leq i \leq m$; dove la moltiplicazione è un'operazione nei *campi finiti*.
- $Z[1] = L \oplus R$

- $Z[i] = Z[i-1] \oplus L[\text{ntz})i]; 2 \leq i \leq m$ con ntz(i) indichiamo il numero di 0 meno significativi in i.

Operazioni nei campi finiti

Servono a non far «esplodere» il risultato, se il risultato sfora il numero massimo che i bit possono contenere allora questo risultato viene riavvolto ovvero, se ad esempio abbiamo come risultato 260 e come valore massimo 255 allora il risultato sarà invece 4 perché una volta raggiunto 256 riniziamo a contare da 0.

Vediamo adesso come avviene il calcolo di $C[m]$, della checksum e del tag:
- $\text{len}(M[m]) \leq n$ bit
- $C[m]$ viene calcolato in modo diverso da $C[i = \{1, m-1\}]$
- $C[m]0^*$ è uguale a $C[m]$ con un padding fino a raggiungere una lunghezza di n.
- $L[-1] = L \cdot 2^{-1}$
- La checksum è in funzione di M e K
- Viene prodotto un Tag di lunghezza **tau**
- **tau** determina il livello di autenticazione, più è alto **tau** maggiore è il livello di autenticazione.

Algoritmo di cifratura:

1. Partizionare M in $M[1], ..., M[m]$
2. $L = L(0) = E_K(0^n)$
3. $R = E_k(N \oplus L)$
4. for $(i = 1; i \leq m)$ $L(i) = 2 \cdot L(i-1)$
5. $L(-1) = L \cdot 2^{-1}$
6. $Z[1] = L \oplus R$
7. for $(i = 2; i \leq m)$ $Z[i] = Z[i-1] \oplus L(\text{ntz}(i))$
8. for $(i = 1; i \leq m-1)$ $C[i] = E_K(M[i] \oplus Z[i]) \oplus Z[i]$
9. $X[m] = \text{len}(M[m]) \oplus L(-1) \oplus Z[m]$
10. $Y[m] = E_K(X[m])$
11. $C[M] = M[m] \oplus (\text{first len}(M[m]) \text{ bits of } Y[m])$
12. Checksum = $M[1] \oplus ... \oplus M[m-1] \oplus C[m]0^* \oplus Y[m]$
13. Tag = $E_K(\text{Checksum} \oplus Z[m])[\text{first } \tau\text{bits}]$

Algoritmo di decifratura:

1. Partizionare M in $M[1], ..., M[m]$
2. $L = L(0) = E_K(0^n)$
3. $R = E_K(N \oplus L)$
4. for $(i = 1; i \leq m)$ $L(i) = 2 \cdot L(i-1)$
5. $L(-1) = L \cdot 2^{-1}$
6. $Z[1] = L \oplus R$
7. for $(i = 2; i \leq m)$ $Z[i] = Z[i-1] \oplus L(\text{ntz}(i))$
8. for $(i = 1; i \leq m-1)$ $M[i] = D_K(C[i] \oplus Z[i]) \oplus Z[i]$
9. $X[m] = \text{len}(M[m]) \oplus L(-1) \oplus Z[m]$

10. $Y[m] = E_K(X[m])$
11. $M[m] = (\text{first len}(C[m]) \text{ bits of } Y[m]) \oplus C[m]$
12. Checksum = $M[1] \oplus ... \oplus M[m-1] \oplus C[m]0^* \oplus Y[m]$
13. Tag' = $E_K(\text{Checksum} \oplus Z[m])[\text{first } \tau\text{bits}]$

5.4. RSA Publick-Key Encryption

RSA è un block cipher dove il plaintext e il testo cifrato sono interi tra 0 ed $n-1$ per un n. Cifratura e decifratura sono nella seguente forma, dato un blocco di testo in chiaro M e un blocco di testo cifrato C:

$$C = M^e \bmod n$$
$$M = C^d \bmod n = (M^e)^d \bmod n = M^{ed} \bmod n$$

Sia mittente che destinatario devono conoscere i valori di n ed e, inoltre solo il destinatario conosce il valore di d. Questo algoritmo ha come chiave pubblica $PU = \{e, n\}$ e come chiave privata $PR = \{d, n\}$, per essere un buon algoritmo di cifratura a chiave pubblica deve soddisfare i seguenti requisiti:

1. Deve essere possibile trovare valori di e, d, n tali che $M^{ed} \bmod n = M$ per ogni $M < n$.
2. Deve essere relativamente facile calcolare M^e e C^d per ogni valore $M < n$.
3. Deve essere praticamente impossibile determinare d dato e ed n.

Riguardo al primo requisito, significa che dobbiamo trovare una relazione nella forma

$$M^{ed} \bmod n = M$$

Per soddisfare questa equazione, dato che usiamo numeri interi, é necessario soddisfare:

$$e \cdot d \bmod \Phi(n) = 1$$

Dove con $\Phi(n)$ indichiamo la **funzione toziente di Eulero**, questa rappresenta il numero di interi positivi piú piccoli di n che sono «relativamente primi» a n cioé che non hanno divisori in comune con n a parte 1. Calcolare questa funzione per numeri grandi é si molto semplice ma praticamente impossibile dato che andrebbero controllati tutti i numeri prima di n per vedere se sono primi rispetto ad esso. É qui che sono d'aiuto i numeri primi, infatti la matematica ci dice che se prendiamo $n = p \cdot q$ con p e q primi allora possiamo calcolare $\Phi(n) = (p-1)(q-1)$, infatti se un numero é primo non condivide divisori con nessuno dei numeri che lo precedono e quindi $\Phi(q) = q - 1$. Quindi se conosciamo i due numeri che hanno «creato» n allora calcolare $\Phi(n)$ si calcola subito altrimenti ci vorrebbero tantissimi anni.

Adesso utilizziamo quindi $\Phi(n)$ per creare le chiavi:

- L'esponente pubblico e, lo scegliamo a caso e lo usiamo per cifrare le chiavi. L'unica regola é che non deve avere divisori in comune con $\Phi(n)$. La chiave pubblica che diamo a tutti é (e, n).
- L'esponente privato d lo scegliamo quindi tale che rispetti $e \cdot d \bmod \Phi(n) = 1$, quindi un numero che moltiplicato per e ci dia come resto 1 se diviso per $\Phi(n)$.

Per cifrare il messaggio dobbiamo prendere il messaggio M, elevarlo per e e dividerlo per n prendendo il resto:

$$C = M^e \bmod n$$

Per decifrare, riceviamo C lo eleviamo per d. Matematicamente stiamo facendo:

$$(M^e)^d \bmod n = M^{ed} \bmod n$$

Grazie al fatto che abbiamo costruito e e d usando la funzione di Eulero possiamo dire che elevare alla potenza e e poi alla potenza d sono due operazioni che si annullano lasciandoci il messaggio originale M.

Passaggi da eseguire

Prima di tutto generiamo le chiavi

1. Selezioniamo p, q primi e diversi fra loro
2. Calcoliamo $n = p \cdot q$
3. Calcoliamo $\Phi(n) = (p-1)(q-1)$
4. Selezioniamo un intero e con M.C.D.$(\Phi(n), e) = 1; 1 < e < \Phi(n)$
5. Calcoliamo d sapendo che $d \cdot e \bmod \Phi(n) = 1$
6. Creiamo la chiave pubblica $KU = \{e, n\}$
7. Creiamo la chiave privata $KR = \{d, n\}$

Poi possiamo cifrare un messaggio (un intero)

- Testo in chiaro M con $M < n$
- Testo cifrato é dato da $C = M^e \bmod n$

Per decifrare

- Dal testo cifrato C otteniamo $M = C^d \bmod n$

Esempio di utilizzo

1. Selezioniamo due numeri primi $p = 17, q = 11$
2. Calcoliamo $n = pq = 187$
3. Calcoliamo $\Phi(n) = (p-1)(q-1) = 160$
4. Selezioniamo e tale che é primo e minore rispetto a $\Phi(n) = 160$, ad esempio $e = 7$.
5. Determiniamo d tale che $d \cdot e \bmod 160 = 1$ e $d < 160$, in questo caso $d = 23$

Otteniamo come chiavi chiave pubblica $\{7, 187\}$ e come chiave privata $\{23, 187\}$ Vogliamo inviare il numero 88:

1. Lo cifriamo calcolando $88^7 \bmod 187$ e otteniamo 11
2. Per decifrare calcoliamo $11^{23} \bmod 187$ e riotteniamo 88

5.4.1. Sicurezza dell'RSA

Esistono 4 diversi modi per attaccare l'algoritmo RSA:

- **Brute Force**: Provare tutte le possibili chiavi private.
- **Mathematical Attacks**: Ci sono diversi approcci possibili, tutti equivalenti per quanto riguarda gli sforzi da fare per fattorizzare il prdotto di due numeri primi.
- **Timing Attacks**: Questi dipendono dal tempo di esecuzione dell'algoritmo di decifratura.

- **Chosen ciphertext attacks**: Sfruttano le proprietá dell'RSA. Non affronteremo questo argomento.

La difesa contro il primo tipo di attacchi é uguale a molti altri algoritmi ovvero usare uno spazio delle chiavi enorme, in questo caso maggiore é il numero di bit di d maggiore sará il tempo necessario a «trovarlo», in questo caso peró visti i calcoli necessari, maggiore é la grandezza della chiave e maggiore sará il tempo necessario per cifrare e decifrare.

5.4.1.1. Il problema della fattorizzazione

Ci sono 3 approcci per attaccare RSA matematicamente:
- Fattorizzare n nei suoi due fattori primi, questo permette di calcolare subito $\Phi(n)$ e determinare anche d.
- Determinare $\Phi(n)$ direttamente senza determinare prima p, q, come prima ci permette di calcolare subito d
- Determinare direttamente d senza trovare prima $\Phi(n)$.

Il problema principale é quello di fattorizzare n nei suoi due fattori primi, infatti determinare $\Phi(n)$ é equivalente a fattorizzare n. Possiamo quindi vedere i tempi di fattorizzazione di n come «benchmark» della sicurezza di RSA. Per un n grande con grandi fattori primi, la fattorizzazione é un problema difficile ma non quanto lo era prima. Nella seguente tabella sono misurati i tempi in MIPS-years ovvero un anno in cui un processore che puó eseguire un milione di istruzioni al secondo processa senza interruzioni. Il numero di cifre fa riferimento al numero di cifre della chiave.

	Is the Person Claimed	Is Not the Person Claimed
Test is Positive	True Positive = a	False Positive = b
Test is Negative	False Negative = c	True Negative = d

Col tempo sono stati scoperti diversi algoritmi per la fattorizzazione dei numeri ma in generale, per ora, una chiave nel range di 1024-2048 bits è ritenuta sicura. Inoltre i ricercatori hanno dato qualche restrizione aggiuntiva per rendere meno semplice la fattorizzazione di n:
- p e q devono avere una lunghezza simile ma non troppo piccoli.
- Sia $(p-1)$ che $(q-1)$ devono avere un fattore primo grande, questo perché se hanno solo fattori primi piccoli possono essere ricavati subito da un algoritmo chiamato **Pollard's $p-1$**
- Il M.C.D.$(p-1, q-1)$ deve essere piccolo, questo perché se é grande significa che genera meno chiavi possibili.

Inoltre é stato dimostrato (**Michael Wiener**) che se $e < n$ e $d < n^{\frac{1}{4}}$ allora d puó essere facilmente determinato.

5.4.1.2. Timing Attacks

Come detto prima si basano sul tempo di esecuzione dell'algoritmo di decifratura, **Paul Kocher** ha dimostrato che in alcuni casi é possibile determinare la chiave privata solamente tenendo traccia di quanto tempo ci mette il computer a decifrare il messaggio.

Per usare questo attacco si indovina la chiave bit a bit, procedendo da sinistra verso destra:
- L'attaccante crea un messaggio cifrato e lo invia, questo messaggio é costruito in modo tale che quando il computer arriva al bit che lui sta cercando di indovinare, se quel bit é 1 allora l'operazione sará lentissima mentre se é 0 allora sará normale.
- Si cronometra il tempo totale che impiega il computer a rispondere
- Se il tempo é stato estremamente lungo allora puó impostare quel bit a 1 altrimenti a 0. Si passa al bit successivo.

Esistono tre difese contro questo tipo di attacchi:
- **Constant Exponentiation Time**: Si forza il computer ad impiegare sempre lo stesso tempo massimo per ogni bit, ovviamente andiamo a peggiorare le prestazioni.
- **Random Delay**: Si aggiunge del «rumore» facendo fare delle piccole pause casuali al processore, il difetto é che l'hacker puó inviare il messaggio tante volte e fare una media dei tempi di risposta rilevando i tempi reali.
- **Blinding**: Consiste nel modificare il messaggio ricevuto per imbrogliare il cronometro dell'hacker.

In alcuni RSA é implementato il Blinding nel seguente modo:
1. Genera un segreto casuale r tra 0 e $n-1$.
2. Calcola $C' = C(r^e) \bmod n$ dove e é l'esponente pubblico.
3. Calcola $M' = (C')^d \bmod n$ come il classico RSA
4. Calcola $M = M' \cdot r^{-1} \bmod n$

Ovvero:
1. Viene scelto un numero casuale
2. Lo cifriamo e moltiplichiamo per il messaggio ricevuto.
3. Decifriamo questo valore ottenuto andando a imbrogliare il cronometro dell'hacker
4. Moltiplichiamo il valore ottenuto con l'inverso del numero inventato prima per ottenere il messaggio originale.

Questa operazione di blinding diminuisce le performance di un 2-10%.

5.5. Diffie-Hellman

L'algoritmo permette a due utenti di scambiarsi una chiave segreta in modo sicuro.

Vengono scelti due elementi pubblici:
- Un numero primo q
- $\alpha < q$ e α deve essere una radice primitiva di q

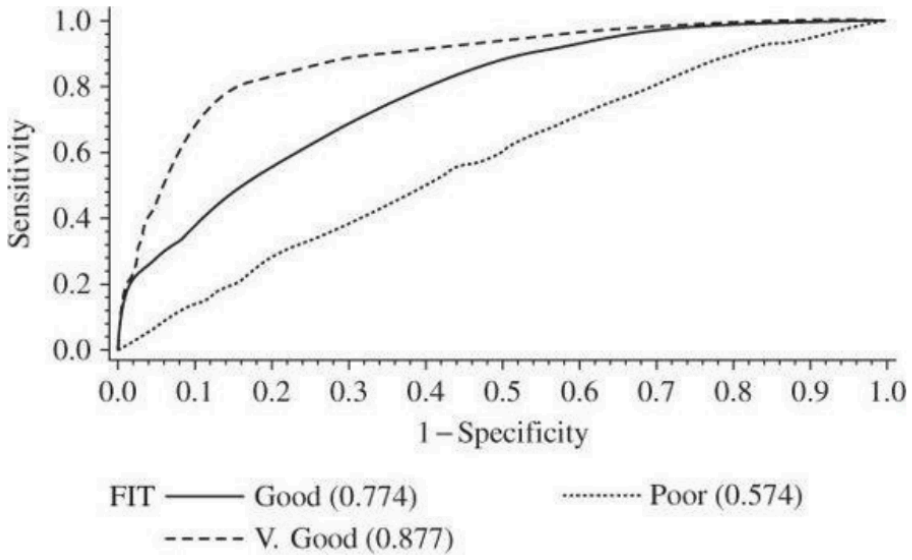
Una radice primitiva modulo n é un numero g che elevato a potenza riesce a generare da solo tutti i numeri possibili di quel modulo. Ad esempio se $n = 7$ allora 3 é una radice primitiva infatti elevando 3 a tutte le potenze da 1 a 6 e calcolando il resto della divisione con 7 otteniamo tutti i numeri da 1 a 6. Non tutti i numeri possiedono radici primitive, cioé avviene solo per $2, 4, p^k$ o $2p^k$ dove p é un primo dispari.
- L'utente A genera le sue chiavi, prima seleziona una chiave privata X_A t.c. $X_A < q$ e poi calcola la chiave pubblica $Y_A = \alpha^{X_A} \bmod q$
- L'utente B genera le sue chiavi nello stesso modo quindi $X_B < q$ e $Y_B = \alpha^{X_B} \bmod q$
- Devono scambiarsi la loro chiave pubblica e poi possono calcolare la chiave in comune:

- A la puó calcolare eseguendo $K = (Y_B)^{X_A} \mod q$
- B la puó calcolare eseguendo $K = (Y_A)^{X_B} \mod q$

- Questa chiave K é il valore che si sono «scambiati» e che possono usare per le loro future comunicazioni.

5.5.1. Sicurezza del Diffie-Hellman

Prendiamo questo esempio dove un algoritmo utilizza Diffie-Hellman per scambiare le chiavi:



Questo tipo di comunicazione non é sicura contro attacchi del tipo **man-in-the-middle**. Alice e Bob devono scambiarsi la chiave e Darth é l'avversario:

1. Darth prepara l'attacco generando due chiavi private casuali X_{D1} e X_{D2} e genera anche le corrispettive chiavi pubbliche.
2. Alice trasmette la sua chiave pubblica a Bob.
3. Darth intercetta Y_A e trasmette Y_{D1} a Bob, inoltre calcola anche $K2 = (Y_A)^{X_{D2}} \mod q$
4. Bob riceve Y_{D1} e calcola $K1 = (Y_{D1})^{X_B} \mod q$
5. Bob trasmette Y_B
6. Darth intercetta Y_B e trasmette Y_{De} ad Alice, inoltre calcola anche $K1 = (Y_B)^{X_{D1}} \mod q$
7. Alice riceve Y_{D2} e calcola $K2 = (Y_{D2})^{X_A} \mod q$

A questo punto Alice e Bob sono convinti di essersi scambiati le chiavi ma invece tutte le loro future comunicazioni saranno lette da Darth che potrá anche modificare i messaggi. Questo algoritmo é vulnerabile a questo tipo di attacchi perché non c'é autenticazione del mittente, é possibile quindi risolvere questi problemi utilizzando firme digitali o certificati a chiave pubblica.

6. User Authentication

Il processo di autenticazione per gli utenti avviene in due step:
- **Identification**: Identificare la persona
- **Authentication**: Provare che questa persona è realmente chi dice di essere.

Ovviamente utilizziamo il termine persona ma questa potrebbe tranquillamente essere un altro sistema.

L'autenticazione può basarsi su:
- **Something the individual knows**: Ad esempio una password, risposte a delle domande o un PIN
- **Something the individual possesses**: Chiavi elettroniche, smart cards, ci si riferisce a questi strumenti come *token*
- **Something the individual is**: Dati biometrici come impronta digitale o retina.
- **Something the individual does**: Si basa sempre su dati biometrici ma includono azioni come ad esemmpio parlare, scrivere o il ritmo di scrittura.

6.1. Authentication with something you know

Le password offrono sicurezza nell'autenticazione ma questa viene compromessa dal comportamento umano, le principali difficoltà nel loro utilizzo sono:
- Per garantire la massima sicurezza andrebbe utilizzata una password diversa per ogni «oggetto» al quale accediamo ma è scomodo.
- Se qualcuno di non autorizzato viene a conoscenza della password sarà necessario cambiarla e informare altri utenti autorizzati del cambio avvenuto.
- La password potrebbe andare persa e gli amministratori di sistema dovranno fornirne un'altra.

In alcuni studi hanno indicato i 12 step che, di solito, un attaccante usa per violare una password:
- Inserire una password vuota
- Password uguale allo user ID
- Se è derivata dal nome dell'utente
- Contenuta in un piccolo dizionario di password
- Contenuta in un dizionario completo Inglese
- Contenuta in un dizionario non inglese
- Contenuta in un dizionario inglese considerando anche lettere grandi o sostituzioni con numeri, ad esempio O-0, E-3…
- Ottenuta tramite brute force con tutte le combinazioni alfanumeriche
- Ottenuta tramite brute force con tutte le combinazioni dal set di caratteri completo.

Il principali ostacolo è quello del tempo richiesto per indovinare la password, infatti qualsiasi password può essere indovinata e quello che ci protegge è il numero di tentativi richiesti.

I principali attacchi alle password sono:
- **Dictionary Attack**: Sono delle raccolte di parole e frasi provenienti ad esempio da film, serie TV, mitologia ecc… che spesso vengono utilizzate anche come password. Questi dizio-

nari non vengono soltanti usati da parte degli attaccanti ma anche dagli amministratori di sistema per individuare utenti che inseriscono password deboli.

Per indovinare una password si può provare anche a rimpicciolire il set di possibilità andando ad analizzare l'utente, infatti è molto probabile che una password sia un dato personale come ad esempio il nome di un animale domestico o che derivi da queste informazioni.

Per verifica un utente il sistema operativo controlla che ID e password forniti dall'utente combacino all'interno del suo database, questa tabella potrebbe essere letta da utenti malintenzionati e per questo i sistemi operativi non salvano le password in chiaro ma una versione cifrata di queste. Quindi quando un utente prova ad autenticarsi il sistema operativo applica la stessa funzione di cifratura all'input e verifica che combaci con quello memorizzato da lui. (non sempre si usa la cifratura ma comunque operazioni simili.) Questo ovviamente non ci protegge da attacchi brute force ma è possibile renderli più difficili ad esempio inserendo un tempo di attesa tra una password e un'altra.

- **Rainbow Table**: Cifrare le password ha comunque una vulnerabilità ovvero che password identiche avranno la stessa password cifrata, quindi se un utente si impossessa di una tabella con password cifrata può comunque capire chi sta utilizzando la stessa password, anche se non sa quale sia.

Per risolvere questo problema si utilizza il **salt** ovvero aggiungere dei dati insieme alla password prima di cifrarla, questi dati possono essere ad esempio il momento di iscrizione o altri dati randomici.

Vediamo adesso come costruire password forti e resistenti anche ad **exhaustive attack** ovvero quando l'attaccante prova tutte le combinazioni possibili:
- Utilizzare anche i numeri oltre alle lettere a-z e usare anche varianti maiuscole, in questo modo portiamo a 62 il set di caratteri disponibili per ogni carattere della password.
- Altra roba ma non credo serva per il corso.

6.2. Authentication Based on Biometrics: Something you are

Qui utilizziamo caratteristiche biometriche dell'utente per autenticarlo, ad esempio:
- Impronta digitale
- Geometria della mano
- Retina
- Voce
- Scrittura
- Riconoscimento facciale

Uno dei vantaggi rispetto alle password è che i dati biometrici non possono essere persi, rubati, dimenticati e sono sempre disponibili. Questi però presentano anche alcuni difetti:
- Sono molto invasivi e non tutti sono d'accordo sul condividere questo tipo di dati.
- Hanno costi di implementazione molto alti.
- Possono diventare un *single point of failure*, infatti con le password se questa viene dimenticata o altro possiamo cambiarla o resetterla ma questo non avviene per i dati biometrici, se il sensore legge male le impronte allora l'utente rimane bloccato.

- Tutti i sensori implementano dei *threshold* per l'accettazione dei dati e anche funzionalità per riconoscere volti o altro anche se questi sono inclinati o altre variazioni.
- Possono verificarsi *falsi positivi* ovvero quando un utente viene accettato quando non dovrebbe e *falsi negativi* ovvero quando un utente viene rifiutato quando non dovrebbe. Spesso ridurre i falsi positivi porta ad un aumento dei falsi negativi e viceversa.

Un sistema dobrebbe avere un rate di falsi positivi dello 0.001% e un rate di falsi negativi dell'1%. Anche queste piccole percentuali se usate con tantissimi utente potrebbero però portare a problemi.

I test di screening devono essere in grado di valutare il grado di efficacia dei loro *matching*, devono quindi poter determinare se stiano identificando efficacemente le persone ricercate senza danneggiare coloro che non lo sono. Quando un sistema confronta un dato in suo possesso con un dato che sta misurando sta effettuando quello che chiamiamo *dichotomus test*: o c'è un match o non c'è. Possiamo descrivere questo test usando un Reference Standard, lo standard di riferimento è l'insieme di regole che determina quando un test positivo corrisponde a un risultato positivo. L'obiettivo è evitare due tipi di errori: i falsi positivi e i falsi negativi.

Possiamo misurare il successo dell'analisi usando 4 misure standard: **sensitivity, prevalence, accuracy e specificity**, per vedere come funzionano assegnamo delle variabili ai match in questo modo:

	Is the Person Claimed	Is Not the Person Claimed
Test is Positive	True Positive = a	False Positive = b
Test is Negative	False Negative = c	True Negative = d

- Sensitivity: È il grado di sensibilità con il quale il sensore fa match o no. Rappresenta la proporzione di risultati positivi tra tutti i possibili match corretti e si calcola come:

$$\frac{a}{a+c}$$

- Specificity: Misura la proporzione di risultati negativi fra tutte le persone non ricercate in quel momento, si calcola come:

$$\frac{d}{b+d}$$

- Accuracy: Misura la sensibilità con la quale il test identifica correttamente le analisi, si misura come:

$$\frac{a+d}{a+b+c+d}$$

- Prevalence: Misura quanto è comune una certa condizione, si misura come:

$$\frac{a+c}{a+b+c+d}$$

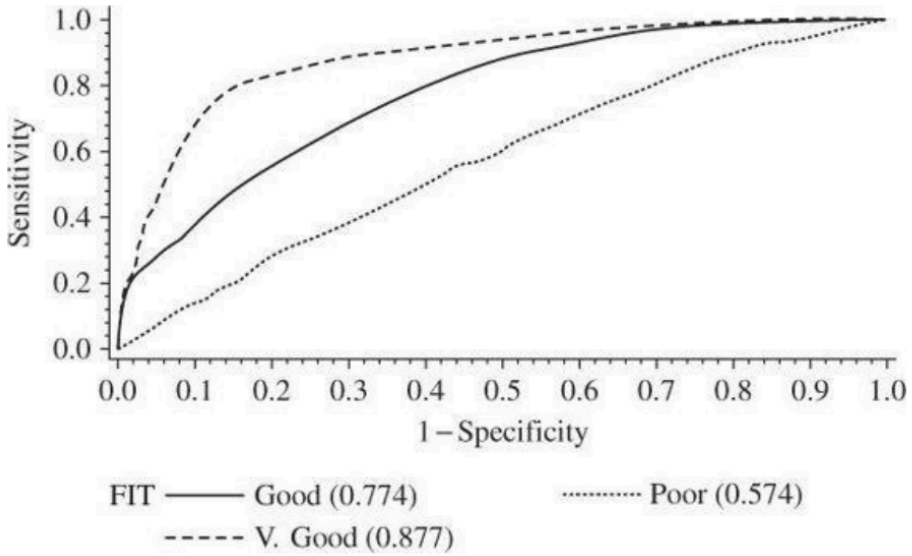
Sensitivity e specificity sono inversamente proporzionali, se una sale allora l'altra scende, per questo infatti non possiamo semplicemente dire che vogliamo rimuovere i falsi positivi, questo porterebbe ad un aumento dei falsi negativi, va trovato il giusto bilanciamento fra i due. Per aiutarci calcoliamo il *positive predictive value* del test, un numero che ci dice quante volte un match positivo rappresenta effettivamente l'autenticazione della persone cercata in quel momento. Questo si calcola come:

$$\frac{a}{a+b}$$

Con lo stesso ragionamento possiamo calcolare il *negative predictive value* come:

$$\frac{d}{c+d}$$

Il **receiver operating characteristic (ROC) curve** è una rappresentazione grafica del trade-off tra falsi negativi e falsi positivi, di base questo mostra il rate di falsi positivi (1 - specificity) sull'asse delle x e il rate dei veri positivi (sensitivity oppure 1 - rate dei falsi negativi) sull'asse y. Vogliamo cercare di rimanere il più a sinistra e il più in alto possibile, questo significa avere tanti veri positivi e un basso rate di falsi positivi. Nel grafico più l'area sotto la curva è grande più avviene quello scritto prima:



L'amministratore di sistema, in base all'utilizzo che verrà fatto di questo test deciderà il giusto compromesso tra i due valori.

6.3. Authentication Based on Tokens: Something you have

Si basa appunti su degli oggetti che possediamo, possiamo dividere i token in:

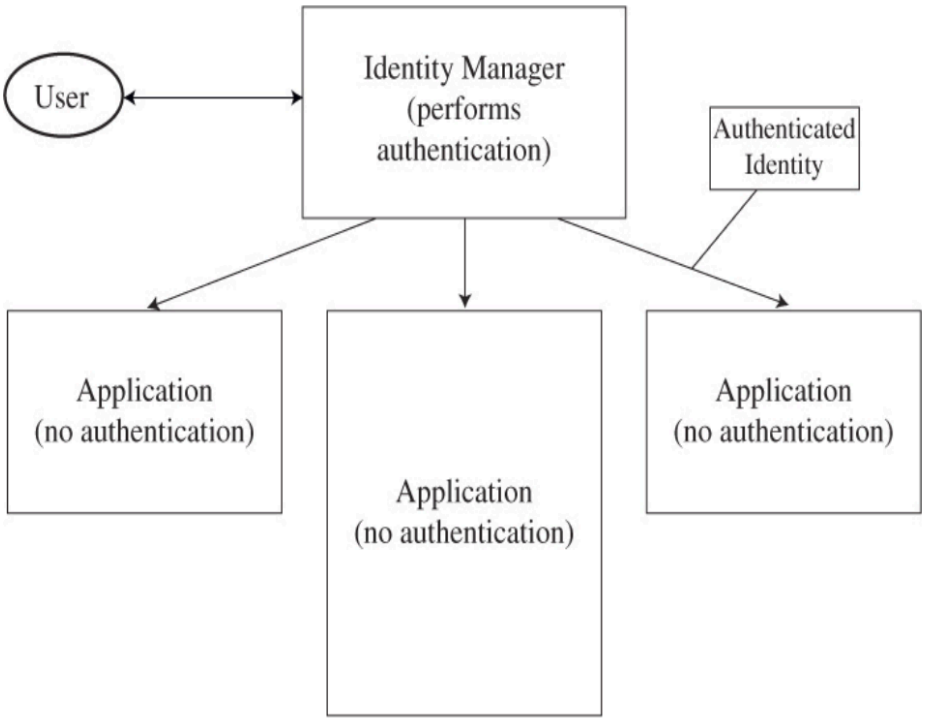
- **Token passivi**: Sono semplici oggetti che non fanno nulla, foto è un esempio di token passivo che appunto contiene un'informazione che non cambia mai.
- **Token attivi**: Possono cambiare o anche interagire con l'ambiente, ad esempio un abbonamento elettronico per il trasporto pubblico che va usato sui tornelli.

Un'altra differenza è:

- **Static token**: I dati che contengono rimangono fissi. Questi sono deboli ad un tipo di attacco chiamato **skimming** ovvero utilizzare un dispositivo per copiare i dati di autenticazione del token ed inviarli all'attaccante.
- **Dynamic Token**: I dati cambiano nel tempo, ad esempio esistono dei dispositivi che generano dei codici a intervalli di tempo e questi codici vanno utilizzati per effettuare il login. Questo tipo di token elimina il problema dello skimming.

6.4. Federated Identity Management (FIM)

È l'unione di diversi sistemi di autenticazione e identificazione, invece di mantenere diversi profili utente questi sistemi mantengono un singolo profilo con un solo metodo di autenticazione e i sistemi collegati a questo sistema non richiederanno autenticazione.



Ad esempio lo SPID fa parte di questi sistemi.

Un approccio simile invece è quello del **Single Sign On** questo permette ad un utente di accedere a diversi servizi utilizzando un solo set di credenziali, una volta loggati il sistema si «fida» e possiamo accedere a tutti i servizi collegati. Ad esempio quando facciamo il login con Google / Apple.

Infine troviamo la **multifactori authentication** che combina più autenticazioni prima di dare l'accesso all'utente. Esempio classico è quando facciamo login con email e password ma poi ci viene chiesto un codice inviato sul nostro telefono. Sicuramente avere più fattori di autenticazione aiuta la sicurezza ma più che il numero di questo, quello che protegge è la sicurezza dei sistemi adottati. Possiamo avere anche un'autenticazione a 100 fattori ma se sono tutti uguali e deboli sono più inutili di un singolo fattore più sicuro.

6.5. Authentication Security Issues

- **Client Attack**: L'attaccante fa finta di essere un utente autorizzato, cerca di ottenere i dati senza accedere all'host.
- **Host Attack**: È un attacco ai file dell'host dove sono memorizzati i dati di login degli utenti come ad esempio le password.
- **Eavesdropping Attack**: Un attaccante cerca di ottenere la password «studiando» l'utente
- **Trojan Horse Attack**: Un software o un dispositivo mascherato da applicazione autorizzata che in realtà ha lo scopo di rubare dati sensibili come password.
- **Replay Attack**: Un attaccante cattura la richiesta di un utente e la ripete.
- **Denial of Service Attack**: Disabilitare i sistemi andandoli a sovracaricare di richieste, ad esempio mandare offline il sistema di autenticazione con migliaia di richieste.