Università degli studi di Modena e Reggio Emilia

Dipartimento di Ingegneria Enzo Ferrari

Crittografia Applicata

Anno Accademico 2023/24

Indice

Capitolo 1

Introduzione

Crittologia: l'arte delle scritture segrete, può essere divisa in 3 macro argomenti:

1. **crittografia**: come **trasformare** messaggi per proteggerne il contenuto (l'informazione).
2. **steganografia**: come **nascondere** messaggi per evitare che venga individuato (ex. ***least significant bit steganography***).
3. **crittoanalisi**: come **analizzare** messaggi e rivelarne l'informazione.

1.1 Crittografia Classica

La sicurezza della crittografia classica si basa unicamente sulla **segretezza** del **metodo** (noto solo al *sender* e al *receiver*), considerava come una tipologia di attacco quello **passivo** (*read only*). Basandosi su questi concetti il suo utilizzo in applicazioni reali è molto limitato (nel senso moderno), considerando la comunicazione in termini di scambio di informazioni in linguaggio naturale.
Alcuni esempi: **scytale** (*transposition cipher*), **caesar cipher** (*shift cipher*) e **vigenere cipher**.

1.2 Crittografia Moderna

1.2.1 Encryption only

La crittografia moderna si basa su due principi:

1. **Kerckhoffs principles**:
   - gli algoritmi devono essere pubblici.
   - la sicurezza del metodo si deve basare sulla **segretezza** della **chiave**.
   - uno schema deve essere "praticamente", se non "matematicamente" indecifrabile
2. **Shannon principles**:
   - **confusione**: ogni bit del crittogramma deve dipendere da più bit della chiave, oscurando, però, la correlazione tra le due.
   - **diffusione**: se viene cambiato un singolo bit del testo in chiaro, allora almeno la metà dei bit del crittogramma devono cambiare, e viceversa.

Nella crittografia moderna lo spazio delle chiavi deve essere sufficentemente ampio per evitare una ricerca esaustiva su di esso, in più, nessuna informazione (né del *plaintext*, né della *key*) deve poter essere estrapolata dal *ciphertext*. Viene detto che il *ciphertext* deve essere **indistinguibile** da una sequenza di bit *random*.

1.2.2 Extended and applied settings

Gli avversari (i crittoanalisti) non sono più unicamente **passivi**, ma bisogna modellare delle tipologia di avversari che siano capaci anche di **interagire** con i nostri sistemi e **manipolare** dei messaggi.
Per ognuna di queste modellazioni è necessario **provare la sicurezza** dei sistemi andando a **definire** delle attività (tramite la comprensione e modellare cosa è "sicuro") e **costruendo** attività (progettandole e provandone la veridicità).

Bisogna definire le **primitive**, gli **schemi**, i **protocolli** e le *applicazioni* che vengono utilizzati, andandoli ad analizzare separatamente e completa.
⇒ può essere necessario costruire schemi e protocolli modellati su misura per applicazioni reali inerenti ad un certo caso d'uso: **Crittografia Applicata**.

1.3 Crittografia Applicata

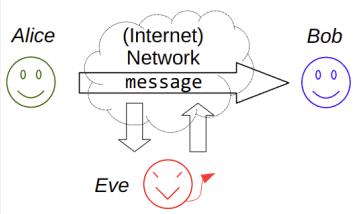
È un layer di astrazione che può essere (quasi) direttamente mappato all'interno di una soluzione per un caso d'uso reale (quindi tecniche "pratiche"). Siccome analizziamo **soluzioni pratiche** bisogna gestire possibili errori dovuti ad **implementazioni** o ***deployment*** errati. I protocolli sicuri assumono che un attaccante tenti di accedere alle informazioni in transito (violazione della **confidenzialità**) o cerchi di impersonificare un mittente (violazione dell'**autenticazione**).
Una delle sicurezze che deve garantire un protocollo sicuro è la **confidenzialità**.

Quando si studi/analizza/progetta un protocollo crittografico è necessario identificare:

- **system model**: descrive lo scenario ("idealmente") di utilizzo, andando a definire: gli attori **legittimi**, la **tipologia di protocollo** utilizzata, le **informazioni** possedute dagli attori legittimi, e altre informazioni sullo scenario applicativo.

  Esempio

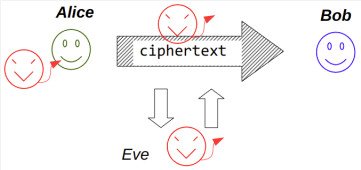
  Un protocollo sicuro che ha come **scopo** proteggere le informazioni scambiati tra due attori: **Alice** e **Bob** quando un possibile attaccante **Eve** può accedere direttamente all'informazione tramite il canale fisico.

- **threat modelling** (modellazione della tipologia di attaccante): abbiamo identificato degli attori legittimi, ma quanto sono affidabili? Modelliamo il protocollo sulla base dell'attaccante.
  1. che operazioni può effettuare sui dati: solo lettura, modifica, inserimento o eliminazione dei dati.
  2. qual è la superficie di attacco e cosa può provare a fare: ha accesso ad alcune funzionalità (cifrazione/decifrazione), che tipologia conoscenza (*white/gray/black box*), quanti tentavi si hanno: adattivo o meno.

  Esempio

  Cosa può fare **Eve** per compromettere la comunicazione: leggere, manipolare le informazioni in transito, compromissione di un attore legittimo.

- **security guarantees**: quale aspetto di sicurezza vogliamo garantire: **confidenzialità**, **integrità** (autenticazione), **disponibilità**, **non ripudio** (è anche presente il concetto di ***forward security***).
- **cryptography settings**: le due classi principali sono: crittografia **simmetrica** (le funzioni di *encrypt* e *decrypt* utilizzano lo stesso **segreto**) e crittografia **asimmetrica** (sono presenti due differenti **chiavi**, uno utilizzabile durante la funzione di *encrypt* - *public* - e l'altro utilizzato durante la funzione di *decrypt* - *secret*).
- **security assumptions of a proposed scheme**

Alcuni **protocolli**:

1. ***Secure key exchange protocol*** (scambio sicuro di chiavi): Alice e Bob non hanno nessuna **chiave**, ne vogliono ottenere una **sicura** e **condivisa** comunicando su un canale sincrono e non sicuro.
2. ***Secure storage***: gli algoritmi di crittografia possono aggiungere protezione su dati conservati in canali protetti, l'avversario ha avuto accesso ai dati dopo aver sconfitto le difese iniziali, negli scenari di storage possiamo andare ad analizzare due tipologie di dati: "*data at rest*" (è lo standard e la *best-practice*) oppure "*data in use*" (continua ricerca soprattuto per i dispositivi mobili).
3. ***(Identity) Authentication***: (***challenge-response protocols***) ovvero dimostrare il possesso di un segreto senza mandarlo.
4. ***Password Protection***: gli schemi crittografici possono "proteggere" le password in caso di *data breaches*, non solo usando l'hash (anche se aggiunto il *salt*), ma anche tecniche per prevenire il ***brute-forcing*** attraverso **ASICs** (**Application-specific integrated circuit**).

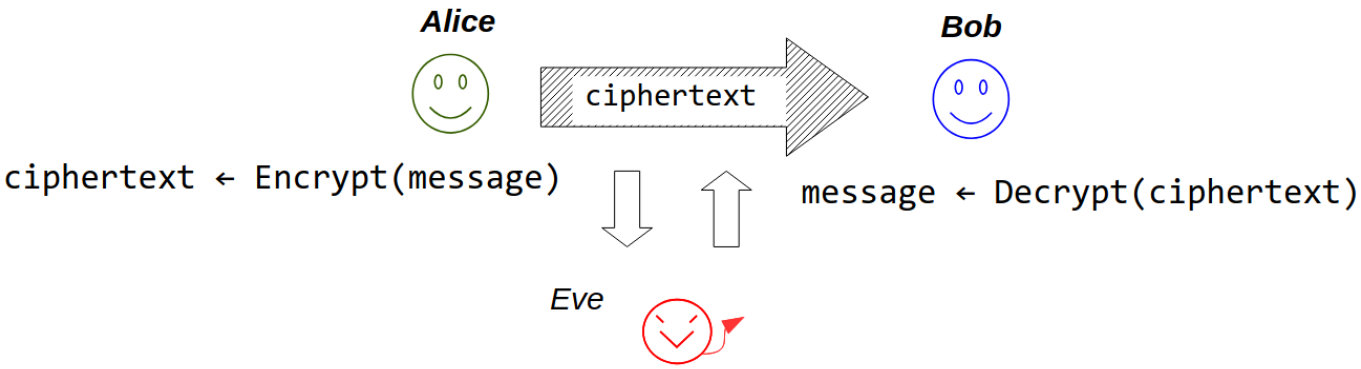
Per riassumere quanto visto fino ad adesso, possiamo dire che la **crittografia applicata** è l'insieme di molti ***system models*** (comunicazioni sincroni, messaggi di gruppi asincroni, *disk encryption*, ...), molti ***security guarantees*** (*information security*), integrità e autenticazione, molti **type of attackers** (passivi e attivi oppure online e offline). La **crittografia applicata** mira a dimostrare la sicurezza sfruttando "strati inferiori":

- la sicurezza dei **protocolli** viene ridotta alla sicurezza sugli schemi.
- la sicurezza degli **schemi** viene ridotta alla sicurezza delle primitive.
- la sicurezza delle **primitive** viene ridotta alla robusteza di problemi matematici.

Per progettare un **protocollo di comunicazione sicuro** è necessario utilizzare un approccio modulare basato su uno *stack* di altri "oggetti", un approccio tipico per lo *stack* è quello che ogni *layer* ha un determinato compito: il protocollo o lo schema è modificato su certi ***cryptographic settings*** e la sicurezza viene provato contro uno specifico tipo di avversario (***security models***).

Capitolo 2

Crittografia Simmetrica



Le garanzie di sicurezze che si cercano di mantenere sono:

- **confidenzialità**: Eve non può accedere a nessuna della informazioni sul messaggio.
- **autenticazione**: Bob può verificare se il messaggio non è stato inviato da Alice, viene anche chiamata ***data origin authenticity*** nel contesto della comunicazioni e implica anche la protezione contro modifiche illegittime (**integrità**).

La sicurezza non esiste in natura è quindi necessario idearla e modellarla, questa prima parte prende il nome di *Definitional Activity*. È comunque importante ricordare che le **definizioni** possono essere **errate** principalmente per errori nella modellazione o nello sviluppo software, ma anche perché non si è stato in grado di modellare quello che era invece richiesto. Un altro errore che si può essere portati a fare è quello di utilizzare in maniera errata certe definizioni ad esempio al di fuori del contesto per cui era stata definita.

Definitional Activity: permette di descrivere che cosa l'avversario può **fare** e cosa può **vedere**. Esistono molteplici modi per definire la sicurezza in maniera più formale, uno tra questi è la ***simulation-based security*** dove viene definita una funzione ideale che soddisfa la definizione di security e poi dimostrare che la funzione costruita si comporti come quella ideale.

First Adversary Model

Quindi modelliamo e identifichiamo le casistiche e tipologie di un attaccante.

Attack Model: Passive Eavesdropper (EAV)

Ha capacità di lettura dei soli *ciphertext* e non è capace di **scegliere nulla**

Security Goal: Indistinguishably

L'avversario non può distinguere il *ciphertext* da sequenza di caratteri random.

Modellando in questo modo il nostro avversario è possibile osservare che non viene descritto nulla sul nascondere la lunghezza del *plaintext*, infatti per questa prima modellazione l'avversario può vedere la lunghezza del *plaintext*.

Nota: la crittografia non ha come obiettivo quello di nascondere la lunghezza del testo in chiaro, nel caso in cui questa informazione fosse confidenziale, è necessario proteggerla a livello applicativo.

Le capacità di un avversario vengono espresse e descritte tramite degli algoritmi chiamati **esperimenti**, che vengono eseguiti da un'entità chiamata ***challenger*** (che per semplicità andiamo ad identificare nell'attore onesto).

Come prima andiamo ad analizzare ***IND-EAV***, il *challenger* va a scegliere un messaggio **m** che viene scelto con la stessa probabilità tra:

- dati random: $m \leftarrow \{0,1\}^n$
- un messaggio generato attraverso la cifrazione $m = \text{Encryption}(p)$, dove **p** può essere scelto nello stesso modo di prima $p \leftarrow \{0,1\}^n$

All'avversario viene fornito **m** e deve scegliere se è un messaggio randomico o se è l'output dell'*encryption*, l'avversario vince l'esperimento se la sua decisione è corretta.

IND-EAV: Perfect & Computational Indistinguishably

Andremo a discutere due tipologie di sicurezza:

1. ***perfect***: la probabilità dell'avversario di vincere l'esperimento è del **50%**, viene anche chiamata ***Unconditional Security*** o ***Informatition Theoretic Security***.
2. ***computational***: la probabilità dell'avversario di vincere l'esperimento è **50%** più una **quantità trascurabile**.

Qualunque tipologia di schema **praticabile** garantisce **sicurezza computazionale**, e se capace di essere sicuro contro un'**esperimento IND-EAV** viene detto ***IND-EAV secure***.

2.1 Sicurezza Incondizionata & One-Time Pad

XOR: gli schemi di crittografia moderni sono progettati per **dati binari**. L'operazione base per la crittografia simmetrica è lo **XOR**.

$$c = m \oplus k$$

m	k	c
0	0	0
0	1	1
1	0	1
1	1	0

Nota: lo **XOR** può essere anche modellato come la somma bit per bit modulo 2: $c_i = (m_i \oplus k_i) \bmod 2$

Lo XOR viene scelto perché dato un certo **m**, se **k** viene scelta in maniera randomica la probabilità di **c** di essere **0** o **1** è $\mathbf{p = 0.5}$.
In questo modo sapere **c** non da informazioni su **m** e quindi **c** è indistinguibile da una successioni di bit random: $\{0,1\}^n$

One-Time Pad - Vernam's Cipher: è un algoritmo di crittografia che esegue un XOR bit a bit tra il testo in chiaro e la chiave, le due lunghezze devono essere uguali e la chiave devere essere random. $c_i = m_i \oplus k_i \; \forall i \in \{0, ..., n\}$ dove n è la lunghezza del testo in chiaro.
Per la decifrazione bisogna utilizzare la stessa chiave: $m = c \oplus k = (m \oplus k) \oplus k = m \oplus (k \oplus k) = m$.
Anche se **OTP** è **incondizionatamente sicuro** non è praticabile realmente in quanto la generazione della chiave per testo arbitrario è computazionalmente onerosa ed è un algoritmo completamente **malleabile**. Gli schemi di crittografia oggi usati sono **computazionalmente sicuri**.

Nota sulla randomicità in crittografia: la randomicità in crittografia è differente da quella "statistica", ovvero una **distribuzione uniforme di 0 e 1** (che è necessaria ma non sufficiente), ma deve essere ***unpredictable***, in modo tale che anche osservando una sequenza, più o meno lunga di bit, non sia possibile predirre il bit successivo.

2.2 Sicurezza Computazionale: Security Level e Key Sizes

Gli schemi crittografici moderni hanno come parte dei requisiti i ***Kerckhoffs principle*** e necessitano uno **spazio delle chiavi largo** abbastanza per prevenire attacchi di ricerca esaustiva. Inoltre lo schema deve essere progettato in modo che si possa prevenire crittanalisi sul crittogramma, quindi nessuna informazione deve essere ottenuta dal crittogramma indipendentemente dal tipo di dato e deve essere sicura contro l'esperimento IND-EAV.

Le condizioni necessarie avere degli schemi computazionalmente sicuri sono:

- gli schemi utilizzati devono essere computazionalmente sicuri.
- definiamo F_k come una **PRF - *Pseudo-Random Function*** con una chiave fissa **k** scelta randomicamente.
  - la **chiave** deve essere "**corta**" (ma lunga abbastanza per resistere ad attacchi a forza bruta).
  - deve essere capace di cifrare grandi moli di dati.
  - data la **chiave** le funzioni di ***encryption*** e ***decryption*** devono essere **efficenti**.
  - senza la **chiave** la probabilità di rompere lo schema crittografico deve essere **trascurabile**.

È necessario tradurre in termini algoritmici **efficenti** e **trascurabile**. Alice e Bob che usano la funzione di *encryption* e *decryption* con la chiave devono essere capaci di eseguire gli algoritmi con costo *efficient*, quindi il **costo computazionale** e **di memorizzazione** sono **polinomiali** sui parametri di sicurezza. Eve, che non conosce la chiave deve operare in maniera attraverso algoritmi **inefficienti**.
→ se il costo dell'attacco diverge da quello degli attori legittimi, è possibile scegliere i parametri di sicurezza appropriati in modo tale che la probabilità di completare correttamente l'algoritmo si molto piccola: **trascurabile**.

Se fissiamo come probabilità di successo per definire un attacco a *brute force* **inefficiente** 10^{-6}, identifichiamo il valore di **N** per funzioni che hanno costo computazionale diverso, per quali valori di $\mathbf{n} > \mathbf{N}$ le probabilità di sucesso sono inferiori?

Costo di Esecuzione		**Probabilità di Successo**		***Threshold*, b = 2**
$\mathbf{O}(b^n)$	→	$\mathbf{O}(b^{-n})$	→	**N = 20**
$\mathbf{O}(b^{\sqrt{n}})$	→	$\mathbf{O}(b^{-\sqrt{n}})$	→	**N = 400**
$\mathbf{O}(b^{\log n})$	→	$\mathbf{O}(b^{-\log n})$	→	**N = 32**

La conoscenza del costo dell'attacco più noto determina il valore del parametro di sicurezza, tra gli altri, la **dimensione della chiave**, identificato dal valore **N**.

$$\exists N \mid f(n) < \tfrac{1}{p(n)},\ \forall n < N$$

Esempio

Definiamo il costo di cifrazione $c_{enc}(n) = n$ mentre il costo dell'attacco $c_{attack} = n^2$ dove n è la lunghezza della chiave.

Negli anni 2000 l'*encryption* utilizzava una chiave a 64bit e impiegava 1ms, mentre l'attacco a forza bruta, impiegava 2 anni. Dopo 10 anni, nel 2010, con la stessa chiave la cifrazione impiegava 0.1ms e il suo brute force 2 mesi.

Aumentando la lunghezza della chiave, raddoppiandola, la fase di cifrazione impiegava 0.2ms, mentre quella di *brute force* passava da 2^{64} a $2^{128} \simeq 10^{20}$ mesi.

Grazie a nuove scoperte vengono trovati algoritmi che **indeboliscono** o **compromettono** il cifrario. Ad esempio alcuni schemi vengono pubblicamente violati pochi anni dopo la loro scoperta come gli schemi crittografici della famiglia *rc* o *sha1*. È anche possibile che schemi standard vengano indeboliti attraverso *backdoor*, parametri deboli o "particolari" e implementazioni deboli.

Efficient function* → *polynomial

Il costo (computazionale e memorizzativo) sono polinomiali rispetto ad un certo parametro di sicurezza n, algoritmi di *encryption* costano al massimo:

$$p(n) := a \cdot n^x$$

Negligible function* → *smaller that any inverse polynomial

Esiste un valore di **N** tale che la funzione sia minore di qualsialsi funzione polinomiale:

$$\exists N \text{ t.c. } f(n) < \tfrac{1}{p(n)},\ \forall\ n < N$$

PseudoRandom functions

Definiamo una **funzione ideale** che soddisfa computazionalmente l'esperimento di sicurezza IND-EAV, nel caso di crittografia a chiave privata, questo tipo di funzione si chiama ***(keyed) family of PseudoRandom Function (PRF)***.

$$\mathbf{F} \ : \ \mathbf{K} \times \mathbf{P} \mapsto \mathbf{C}$$

Dove:

- **K** è uniformemente scelto da $\{0,1\}^{Lk(n)}$
- **P** è il *plaintext* scelto arbitrariamente da $\{0,1\}^{Lp(n)}$
- **C** soddisfa computazionalmente **IND-EAV**, dove la "quantità trascurabile" è espressa dalla funzione **negl(n)**

Uno schema di crittografia deve essere **funzionale**. Definiamo **F** come la **PRF** allora **F** si definirà computazionalmente sicura se:

- lo spazio della chiavi è "**piccolo**", ma grande a sufficienza per resistere ad attacchi basati su ricerca esaustiva. Quindi $Lk(n)$ **deve** essere una funzione efficiente.
- ha la capacità di generare in output grande quantità di dati ***pseudorandom*** (è sicuro per IND-EAV).
- il costo di computazione di **F** è **efficiente**.
- senza la chiave, la probabilità di rompere lo schema crittografico è **trascurabile**, il costo di calcolare $\mathbf{F}^{-1}$ è **inefficiente**.

Concrete parameters for acceptable security guarantees

Gli schemi di crittografia (simmetrica) moderni vengono considerati computazionalmente sicuri, tali schema possono essere violati se si dispone di abbastanza tempo e abbastanza risorse.
Il ***Security Level*** dello schema è la media del numero di operazioni necessarie per rompere lo schema: gli standard stabiliscono dei valori tali che la quantità di tempo e risorse necessaria per calcolare tale quantità di operazioni è *unfeasible*.

- 80-bit di sicurezza $\rightarrow$ 2^{80} operazioni in media per rompere lo schema (insicuro dal 2010).
- 112-bit di sicurezza $\rightarrow$ 2^{112} operazioni in media per rompere lo schema (insicuro dal 2030).
- 128-bit di sicurezza $\rightarrow$ 2^{128} operazioni in media per rompere lo schema (stimata la sicurezza per ogni scenario successivo).

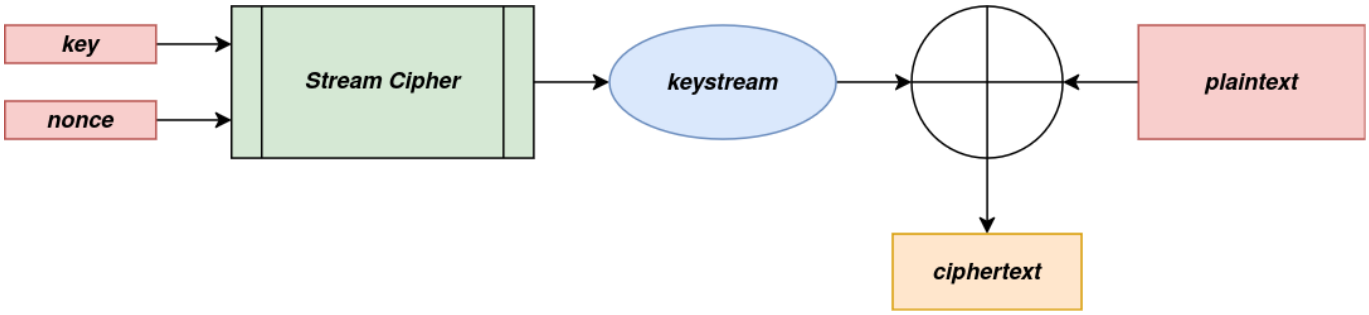
Nei moderni **schemi di crittografia simmetrica**, la **lunghezza della chiave** definisce il **livello di sicurezza**, in quanto l'attacco *best-known* è basato sull'indovinare il segreto. A differenza, negli **schemi di crittografia asimmetrica** dove invece è presente solo una correlazione, in quanto dipende dagli attacchi noti alla matematica sottostante.
Software e librerie **dovrebbero** implementare configurazioni **sicure** di **default** e aggiornate se necessarie

Asymmetric Cryptography & Quantum Computers (PQC)

Si stima che gli attuali standard di crittografia asimmetrica saranno efficacemente violati dai computer quantistici nei prossimi decenni. Nell'ultimo decennio, sono stati ipotizzati e analizzati a fondo nuovi problemi cosiddetti "***post-quantum hard problems***", ovvero che non possono essere risolti in modo efficiente nemmeno con un computer quantistico.

2.3 Stream Cipher

Il primo approccio per implementare una funzione "reale" che approssimi la funzione ideale PRF. Gli *stream cipher* sono ***Deterministic Pseudorandom Bitstring Generators (DPBG)***. Gli schemi a flusso sono funzioni **deterministiche** che prendono in input ***small random input*** e generano come output dati che non possono essere distinti (***indistinguishably***) da dati random e non su cui non si può fare una previsione (***unpredictable***), il dato **pseudo-random** viene chiamato ***keystream***, la funzione di cifrazione richiede xor*are* il *keystream* con il messaggio, infatti gli *stream cipher* cercano di approssimare il **OTP**



Il ***nonce*** è un valore non confidenziale che **deve** essere **univoco** per ogni operazioni di *encryption*. Uno dei più popolari cifrari a flusso è ***ChaCha20***, utilizzo:

```
1    $ echo Hello! | openssl chacha20 -pbkdf2 -k pippo -a -e | \
2        openssl chacha20 -pbkdf2 -k pippo -a -d
```

Questa tipologia di schema è **malleabile** ed è vulnerabile a **riutilizzo della chiave**, sia in caso di messaggi diversi che nel caso di due stati diversi di un certo file (dipende dal contesto)

$$c_1 = m_1 \oplus \text{DPGB}(k) \qquad c_2 = m_2 \oplus \text{DPGB}(k)$$

$$c_1 \oplus c_2 = (m_1 \oplus \text{DPGB}(k)) \oplus (m_2 \oplus \text{DPGB}(k)) = (m_1 \oplus m_2) \oplus (\text{DPGB}(k) \oplus \text{DPGB}(k)) = m_1 \oplus m_2$$

In questo modo la *keystream* viene generata in modo che dipenda unicamente dalla chiave, se noi andiamo a ad utilizzare un altro valore (***nonce***) è possibile rimuovere la vulnerabilità di riutilizzo della chiave. Definendo k_i la *keystream* nell'istante i-esimo avremo:

$$k_i = \text{DPBG}(k, n)$$

Dove:

- **k** è la chiave privata della comunicazione.
- **n** è il **nonce**, il problema permane se nessuno dei due viene aggiornato.

Trasparent Data Encryption (TDE)

Nel caso in cui si voglia cifrare un disco, si vuole non inficiare lo spazio totale che si ha, quindi per evitare che il nonce venga salvato per ogni porzione scritta su disco normalmente si tende ad utilizzare informazioni che esistono già.
Il contesto non può essere utilizzato in quanto nel tempo non cambia mai quindi si è comunque affetti da *key reuse*.

2.4 Block Cipher & Modes of Operation

Un **cifrario a blocchi** è una famiglia di permutazioni pseudo-causali con chiave [***keyed family of pseudorandom permutation (PRP)***].

$$\mathbf{F} \ : \ \{0,1\}^{Lk} \times \{0,1\}^{Lb} \mapsto \{0,1\}^{Lb}$$

Dove **Lb** è la lunghezza del blocco da cifrare, mentre **Lk** è la lunghezza della chiave. Nei *block cipher* sia la funzione F che F^{-1} sono **efficenti**.

Esempio: consideriamo un *block cipher* ideale che utilizza una permutazione dell'alfabeto, per mappare ogni carattere, che è noto in crittografia classica come ***substitution cipher***.

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

La mappatura delle lettere è la chiave del nostro cifrario e la lunghezza della chiave è pari a:

$$26 \cdot \log_2 26 \simeq 112 \text{ bits}$$

Esempio: *Ideal Block Cipher* con *block size* 2

In questo caso andiamo a mappare ogni possibile valore di due bit $2^2 = 4$ con ogni possibile permutazione dei suoi valori $4! = 24$ e utilizziamo l'**indice** come chiave che va a selezionarci la permutazione associata.

Indice	**00**	**01**	**10**	**11**
0	00	01	10	11
1	00	01	11	10
2	00	10	01	11
3	00	10	11	01
4	00	11	01	10
5	00	11	10	01
6	01	00	10	11
7	01	00	11	10
8	01	10	00	11
9	01	10	11	00
10	01	11	00	10
11	01	11	10	00
12	10	00	01	11
13	10	00	11	01
14	10	01	00	11
15	10	01	11	00
16	10	11	00	01
17	10	11	01	00
18	11	00	01	10
19	11	00	10	01
20	11	01	00	10
21	11	01	10	00
22	11	10	00	01
23	11	10	01	00

Una tabella è un ***block cipher* ideale**, quindi se volessimo generalizzare nel caso di una tabella a n bit, quanti bit servirebbero per memorizzarla:

- trasferendo unicamente la permutazione unica del blocco a n bit avremo bisogno di $n \cdot 2^n$ bit, il che è un costo esponenziale.
- trasferendo l'intera tabella, avremo bisogno di $n \cdot 2^n \cdot 2^n!$ bit il che la renderebbe impraticabile da gestire.

Encryption schemes basati su *Block cipher*

I *block cipher* sono **primitive** crittografiche che possono essere utilizzate per costruire degli **schemi di crittografia simmetrica** (*block cipher* $\neq$ *encryption scheme*).

I *block cipher* possono essere utilizzati direttamente con *encryption scheme* solo **in certi casi particolari** (**KEM**), negli altri casi per costruire uno schema crittografico è necessario un ulteriore

algoritmo chiamato ***operations modes*** (*encryption mode*).

- **AES**: è un *block cipher* che ha una ***block size*** di 128bits e una ***key size*** che può variare a 128bits, 192bits, 256bits.
- **AES-128**: una particolare implementazione di **AES** con la *key size* a 128bits.
- **AES-128-CBC**: è un ***encryption scheme*** che si basa su un *block cipher* **AES-128** usato in combinazione con l'*operations mode* **CBC**.

Real Block Ciphers: distribuire un algoritmo invece che una tabella è molto più "facile". Possibili *block cipher*:

block cipher	***block size***	***key size***	**supporto hardware**	**deprecato**
DES	64 bits	56 bits	si	si
AES	128 bits	128, 192, 256 bits	si	no
3DES	64 bits	56 bits x 3	si	ni

AES-NI mette a disposizione nuove instruzioni hardware per la crittografia tramite **AES** è diffuso per tutte le CPU x86. **ARMv8 Cryptography Extensions** che invece non è disponibile su CPU ARM vecchie o di "fascia bassa".

Un ***block cipher*** deve essere progettato basandosi su due proprietà:

1. **Diffusione**: ogni bit del testo in chiaro influsci su ogni bit del crittogramma (durante la modifica).
2. **Confusione**: i pattern all'interno del testo in chiaro non influenzano i pattern del testo cifrato.

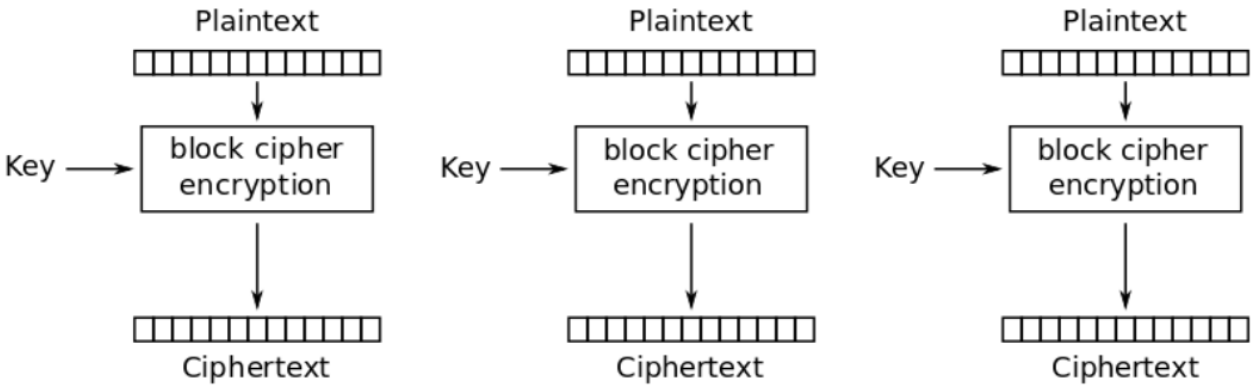
Nota: i ***block ciphers*** possono lavorare come **primitive** all'interno di altri blocchi di crittografia, ad esempio **ChaCha20** e **SHA2** utilizzano al loro interno dei *block cipher*.

Block cipher Operation Modes

Abbiamo detto che *block cipher* sono primitive non schemi di crittografia, infatti sono utilizzabili unicamente su dati che hanno come lunghezza massima la *block size*. Un *block cipher* è **deterministico**, il che lo rende vulnerabile a ***frequency attack***, è però possibile creare uno *symmetric encryption scheme* utilizzando un ***modes of operations***. È Possibile suddividerli in famiglie in base al loro comportamento:

- costruire un ***stream cipher*** da un *block cipher*.
- cifrare blocco per blocco per ridurlla la **malleabilità**.
- combinare la **riservatezza** con altre operazioni crittografiche, ad esempio garantire **integrità**: ***authenticated encryption***.

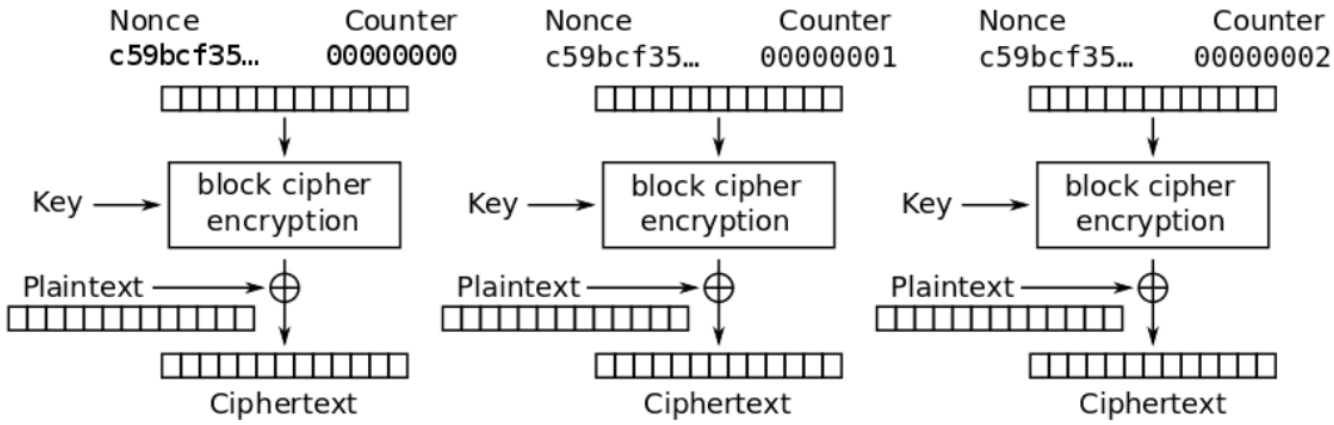
Mode of Operation - Electronic CodeBook (ECB)



Viene usato direttamente il *block cipher*, è necessario utilizzare un informazione con lunghezza multipla della *block size* il dato viene diviso in n blocchi e successivamente cifrati tramite il blocco e gli n blocchi di *ciphertext* vengono concatenati, è molto inefficiente e **vulnerabile** (***ECB Penguin***).

In alcuni casi particolari viene comunque utilizzato, soprattuto per sviluppare e mantenere del codice.

Mode of Operation - Counter Mode (CTR)



Permette di costruire un ***stream cipher*** da un *block cipher*, dove il blocco viene utilizzato come funzione per generare bit pseudo-casuali (***stream key***). È molto utilizzato nelle comunicazioni, è importante l'uso del ***nonce***.

È importante andare ad analizzare il ***nonce***, infatti sappiamo che $len_n + len_c = b$ dove **b** è la *block size* come bilanciamo la **lunghezza** del ***nonce*** (len_n) e la **lunghezza** del ***counter*** (len_c).

Ipotizziamo di utilizzare **AES** come *block cipher* e fissiamo $len_n = 64$ avremo problemi di **ricorsione statistica** dopo:

$$2^{len_c} \cdot 2^{\log_2 b} = 2^{64} \cdot 2^7 = 2^{71}$$

Andando ad aumentare $len_n = 96$ avremo invece problemi di ricorsione dopo 2^{39} bytes $\simeq$ 64gb. La lunghezza del ***nonce*** va a modificare il **numero di messaggi** prima che ci siano problematiche

legate alla crittoanalisi statistica, mentre la lunghezza del ***counter*** va a modificare la **grandezza del messaggio** prima che al suo interno possano esserci problematiche legate alla crittoanalisi statistica.

Mode of Operation - Cipher Block Chaning (CBC)

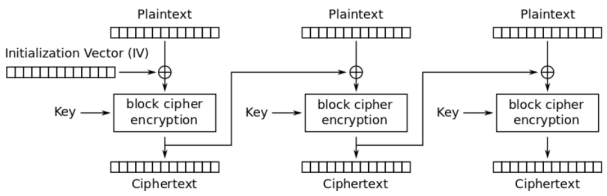


Figura 2.1: **CBC *encryption***

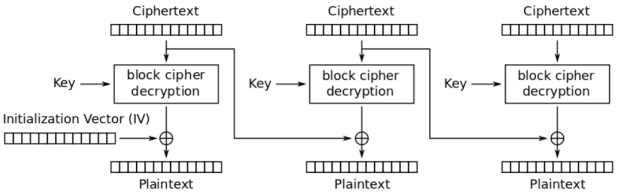


Figura 2.2: **CBC *decryption***

Il **CBC** usa il blocco cifrato precedente ottenendo così un **effetto a valanga** (*avalanche effect*) questo porta ad avere a parità di blocchi di testo in chiaro differenti blocchi di crittogramma. Aumenta, inoltre, la resistenza contro attacchi del tipo di ***frequency analysis*** questo non rimuove l'utilità di modifica nel tempo della chiave (***re-keying*** è comunque importante dopo una certa quantità di dati cifrati).

Introduce l'utilizzo di un ***Initialization Vector*** per abilitare ***multi-message security*** (molto simile al *nonce* ma si basa su assunzioni di sicurezza diversi). L'**IV** è un dato random non c'è bisogno di mantenerlo segreto.

Al contrario del **CTR** che è vulnerabile (come l'**OTP**) a **malleabilità** il **CBC** è abbastanza resistente, assumiamo che un attaccante abbia un ***decryption oracle*** e posso modificare l'n-esimo blocco avremo controllo di modifica dell'n-esimo bit dell'$(n+1)$-esimo blocco. Il **CBC** al contrario è molto più sensibile a problematiche legate a disturbi o malfunzionamenti della rete. Altre grande differenza è che a differenza del **CTR** che è **parallelizzabile** per entrambi le funzioni (*encryption* e *decryption*) il **CBC** no.

Re-Keying è la pratica per cui dopo Δ messaggi inviati la chiave simmetrica deve cambiare, ma questa pratica è in mano al protocollo e non allo schema.

Multi-message Security

Dobbiamo considerare che gli *encryption scheme* come un *tool* ***general purpose***, normalmente ogni informazione può essere trasmessa come dati binario a lunghezza variabile e bisogna essere capaci di cifrare messaggi differenti con la stessa chiave senza che l'attaccante sia capace di distinguere se stiamo cifrando due messaggi uguali o differenti. Per questo motivo si è deciso di utilizzare o il ***nonce (n)*** o il ***initialization vector (iv)***, anche se è un informazione pubblica

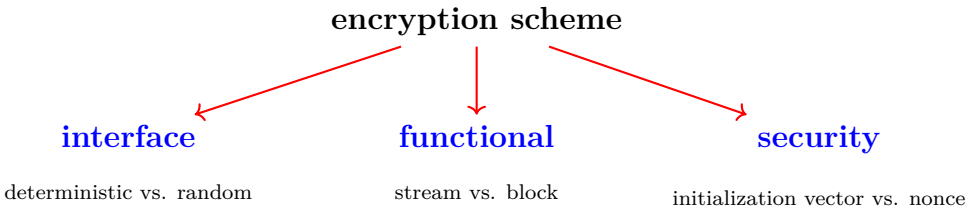
devono essere scelti da un attore "onesto" non dall'attaccante, se così non fosse ci possono essere attacchi potenziali a strutture simmetriche. È anche importante ricordare che i *block cipher* sono deterministici e che quindi utilizzando (nel caso del **CBC**) lo stesso **IV**, stesso *plaintext* e stessa **chiave** avremo lo stesso ***ciphertext***.

Esistono altre ***mode of operations*** oltre a **ECB**, **CTR**, **CBC** che affronteremo più avanti e che avranno a che fare con garanzie di **autenticazione**.

2.5 Opeation Framework for Symmetric Encryption

Developer view to symmetric encryption

Operations Framework ↔ ***interface*** anche note come **API** che vengono utilizzate per interagire con uno **schema** o **protocollo**.



Alcune interfacce non espongono un **API** che non richiede **IV** o **nonce**, ma lo generano internamente, quindi a pari input possono esistere più output. Questo è stato scelto per evitare che nel "mondo reale" **IV** e **nonce** vengono spesso confusi ed esposti nelle interfacce in maniera invertibile, questa cosa è molto pericolosa in quanto hanno requisiti differenti:

- nel caso del *nonce* è fondamentale l'**unicità**.
- nel caso del *iv* è fondamentale la **randomicità**.

In alcuni casi è possibile utilizzare un *nonce* per generare un *iv*.

Il *nonce* a volte ci serve generali "a mano" infatti generandolo in maniera random esiste la possibilità dopo un certo numero n di iterazioni della cifratura la possibilità di duplicarlo. Generandoli "a mano", ad esempio con un **contatore**, si ha la garanzia di utilizzare tutti i possibili valori dello spazio, ad esempio considerando un *nonce* a 96 bit, se utilizzo un contatore il *nonce* avrà tutti i possibili valori da 0 a $2^{96} - 1$ prima di avere un duplicato. Il **NIST - *National Institute of Standard Technology*** definisce la probabilità massima con la quale può avvenire una duplicazione del *nonce* andando a basarsi sulla lunghezza dello stesso: $p = 2^{-32}$.

- se uso un contatore (***statefull***) allora ho 2^{96} combinazioni da utilizzare.
- se uso un random (***stateless***) allora ho 2^{32} combinazioni prima di trovarmi nella condizione di ***nonce* duplicati**.

SIV - Syntetic Initialization Vector: è un modo per ottenere metodi alternativi il *nonce* dall'*iv* e viceversa, oppure è possibile cambiare schema di cifratura.

Encryption & Padding

Siccome bisogna essere capaci di cifrare dati che hanno una lunghezza non multipla della *block size*, abbiamo la necessità di aggiungere informazioni fittizie per raggiungere la lunghezza richiesta: ***padding***. L'unico requisito è che sia distinguibile dal contenuto reale.
PKCS#7: aggiunge n volte il valore n per tutto il numero di bytes che mancano, nel caso in cui i dati sono già in lunghezza multipla della *block size* allora bisogna aggiungere un blocco intero di padding.

Nel caso di ***stream cipher*** le interfacce sono simili a quello del **CBC**, non è necessario fare *padding*, ma bisogna fare i conti con il ***nonce*** in quanto deve essere un valore unico, mai usato fino a quel momento e che non utilizzerò più tardi per cifrare dei dati con la stessa chiave.

Nonce vs Initialization Vector

Sono normalmente adottate in base ai requisiti di sicurezza di un *encryption scheme*:

- ***nonce***: utilizzato per schemi che richiedono **unicità** → ***stream cipher***.
- **IV**: utilizzato per schemi che richiedono **randomizzazione** ed **non predicibilità** → ***block encryption***.

È comunque buona prassi accertarsi che la descrizione fornita dalla libreria sia congrua all'utilizzo che se ne fa.

Ma cosa succede se il *nonce* viene riutilizzato?
Dipende dallo schema ma bisogna cercare di evitarlo in quanto permette di fare la *disclosure* di informazioni in quanto lo schema diventerebbe deterministico e ci permetterebbe di rilevare se due testi cifrati (*frequency attacks*), ma anche rompere completamente la sicurezza dello schema come ad esempio rivelare la chiave o rivelare il valore del *plaintext*.

Alcuni casi d'uso necessitano, però, la capacità di essere resistenti al *nonce reuse* (***nonce reuse resistance***) ad esempio:

- dispositivi *low-power* che necessitano di scambiare messaggi più piccoli → *nonce* più piccoli → alta probabilità di collisione.
- librerie in cui manca l'implementazione di tale controllo.

Abbiamo definito dei requisiti di sicurezza sugli schemi crittografici che possono essere: un *nonce* univoco o un *iv* randomico e/o non predicibile, ma se il nostro sistema non è capace di crearli, ad

esempio perché la fonte di randomicità del nostro dispositivo non ha un *poll* abbastanza ampio? È possibile utilizzare un *nonce* e utilizzando un ***block cipher*** trasformarlo in un *iv*.
È importante rimarcare che la **segretezza** di *nonce* ed *iv* non è richiesta, vengono normalmente concatenati come prefisso del *ciphertext*.

Probabilistic vs Deterministic Encryption

L'unica differenza è la presenza esplicita del *nonce/iv* negli input della funzione di *encryption*.

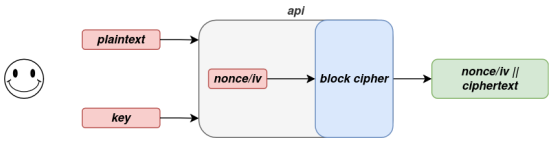


Figura 2.3: Probabilistic Encryption
Nata per correggere errori di uso improprio delle primitive di crittorafia, infatti si assumeva che chi le utilizzava avesse una buona conoscenza delle basi crittografiche. A pari input l'ouput varia.

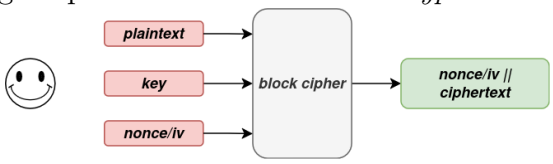


Figura 2.4: Deterministic Encryption
Alcune librerie non sono in grado di scegliere in maniera corretta i valori in base ai requisiti di sicurezza impostati per via dell'architettura o dell'ambiente, ad esempio *low-level device* (**IOT**).

2.6 How much data can we encrypt?

Ogni *cipher* e ogni *mode of operation* permette di cifrare un certo numero di messaggi, che è proporzionale alla gestione e dimensione del *nonce/iv*, ma anche una certa quantità di dati, fissati una chiave e un *nonce/iv* è importante capire quanti dati possiamo cifrare, utilizzare le raccomandazione degli *standard* (calcolate utilizzando statistiche e calcoli approfonditi).

- *block cipher* utilizzati come *stream cipher*: **AES-128-CTR** normalmente il testo da cifrare per generare la *stream key* viene splittato in 96bits di *nonce* e 32bits di *counter*.
- *stream cipher* ad esempio **chacha20** che è simile ad un *block cipher* utilizzato in **CTR mode** ne esistono due varianti:
  - 64bits *nonce* e 64bits *counter* lo schema originale.
  - 96bits *nonce* e 32bits *counter* la variante **IETF** per il protocollo **TLS**.
- **xchacha20** utilizza 192bits *nonce* e 64bits *counter*

Capitolo 3

Hash Function & MAC

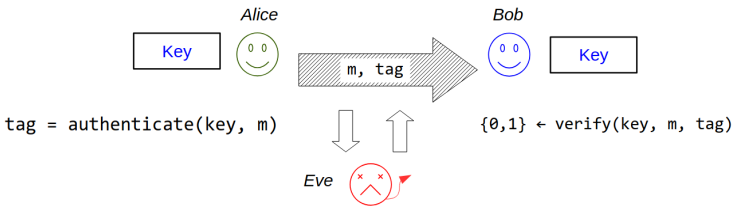
Abbiamo visto che le configurazioni di sicurezza della crittografia simmetrica sono: **confidenzialità**, **integrità** e **autenticità**. Abbiamo anche definito che Alice e Bob condividono la conoscenza di un'unica chiave per la comunicazione. Con i *block cipher* siamo riusciti a ottenere **confidenzialità**.

Integrity: è possibili identificare dal ricevente di un messaggio se quel messaggio è stato modificato durante la trasmissione. Consideriamo una comunicazione tra Alice e Bob, dove il messaggio verrà inviato da Alice, prima di farlo verrà calcolato a ***small-sized digest*** che rappresenta il dato. In questo modo qualunque modifica al dato o al ***digest*** può essere verificata - in questo caso non c'è bisogno della chiave - $d = H(m)$ dove:

- **d** è il *digest* della funzione.
- **H** è una funziona che ritorna una sequenza di byte che rappresenta il dato.
- **m** è il messaggio.

Alice a quel punto invia a Bob $m||d$ in questo modo, Bob può ricalcolarsi $d' = H(m)$ e accettare il messaggio da Alice - quindi verificarne l'**integrità** - se e solo se $d == d'$.

Authenticity Guarantess: il destinatario del messaggio può controllare se il mittente è un mittente legittimo - ovvero qualcuno che ha accesso alla chiave segreta simmetrica - è possibile *bindare* una qualche informazione (metadato) per rendere l'informazione identificativa, ma quasto è utile solamente nella crittografia asimmetrica.



L'integrità dell'informazione è condizione necessaria per l'autenticità, se l'attaccante può modificare il dato, allora può anche impersonificare il mittente del dato, violando l'autenticità.

È spesso confusa l'integrità del dato con la sua autenticità, nel contesto della sicurezza informatica noi cerchiamo l'autenticità - e quindi implicitamente l'integrità - per questo motivo andremo ad analizzare ***authenticated encryption***. È però fondamentale differenziare le due proprietà in quanto utilizzano schemi di crittografia differenti:

- ***Integrity*** utilizzo delle **funzioni hash**.
- ***Authenticity*** utilizza i ***Message Authentication Code - MAC***, per ora andremo ad analizzarli nell'ambito della crittografia simmetrica.

Hash function & cryptographic integrity guarantees

Andiamo, velocemente, ad analizzare quelle situazioni in cui è richiesta ***integrità*** fuori dai requisiti di crittografia: come ad esempio modifiche al dato per errori di trasmissione o per guasti - normalmente scaturite da fenomeni fisici - per i quali sono presenti molteplici algoritmi, tra i quali: ***parity***, **CRC**, ***checksum***. In questo caso è possibile modellare la tipologia di attaccante come un **attaccante irrazionale** (perdita di bit randomici o a raffica).
Nei settaggi di crittografia moderna, l'attaccante è sempre **razionale** e conosce gli **algoritmi crittografici** e si comporta di conseguenza, grazie al requisito di ***cryptographic integrity-protection*** anche se noti i dati di base non deve essere capace di trovare una soluzione al problema nonostante l'assenza di una chiave condivisa. Anche in questo caso è presente la sicurezza computazione ma viene applicata in maniera differente.

3.1 Hash Function

Cryptography Hash Function: una funzione hash H viene definita come:

$$H : \{0,1\}^* \mapsto \{0,1\}^n$$

ovvero è possibile mappare una quantità arbitraria di bit - nelle moderne funzioni hash $*$ è così ampio che può anche considerato a ∞ - in una sequenza fissata (piccola) - che è definita dall'algoritmo. L'ouput di H viene chiamato ***digest*** - le funzioni di hash esistono anche per scopi non prettamente crittografici. La dimensione di **n** è scelta in maniera tale che sia altamente improbabile che due input differenti generino lo stesso output, in questo modo il *digest* è un informazione "piccola" che rappresenta univocamente l'informazione contenuto nel dato. È possibile paragonare il comportamento ad una **funzione di compressione pseudorandom**

$$m_1 \neq m_2 \longleftrightarrow d_1 \neq d_2$$

Il comportamente delle funzioni hash - **PRF** - può essere applicato anche a circostante non crittografiche, ad esempio: **MD5** è una funzione hash deprecata, ma viene utilizzata per identificare in maniera univoca i commit su git. È possibile anche utilizzarle come **primitive** per costruire blocchi per altri schemi crittografici, ad esempio: alcuni **MAC** si basano su delle *hash function* ed anche alcune ***Key Derivation Function - KDF***.

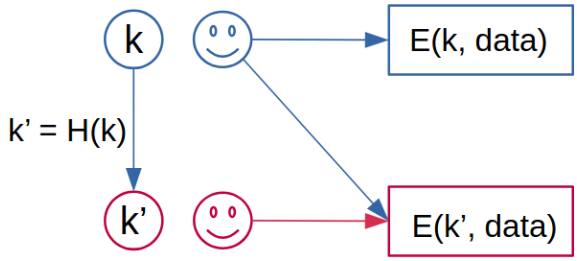
In base al tipo di applicazione che stiamo utilizzando bisogna che la funzione hash sia resistente a diversi ***attack model***, tutti quanti, se no viene definita **deprecata**.

Attack Models Differenti

- ***One Way*** anche nota come ***first pre-image resistance***, deve essere **efficiente** calcolare $H(m) = d$, ma **inefficiente** calcolare la funzione opposta, ovvero risalire a $m = H^{-1}(d)$
- ***Second pre-image Collision Resistant*** ovvero dato un messaggio m_1 è **inefficiente** trovare un messaggio m_2 tale che $H(m_1) = H(m_2)$
- ***Collision Resistant*** è **inefficiente** trovare una coppia di messaggi - **arbitrari** - m_1 e m_2 tale che $H(m_1) = H(m_2)$

stronger attack models

First pre-Image Resistance: dato un *digest*, calcolarsi il dato. È tipicamente associato a **garanzie di confidenzialità** - può essere applicato a schemi di **KDF**.

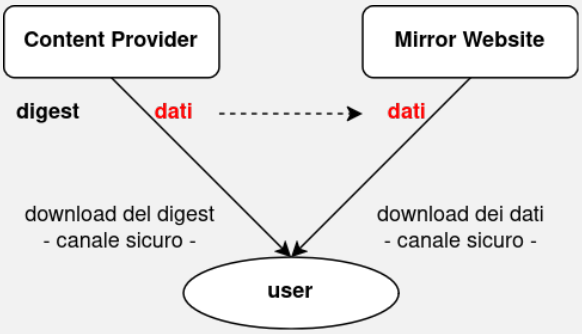


Dato in input una chiave, ottengo in output un'altra chiave $k' = H(k)$ in questo caso cosa dovrebbe rompere Eve per riuscire a decifrare? In questo contesto anche se H **non** è *collision resistance* non è di interesse infatti anche se si trovasse un k'' t.c. $k' = H(k'') \rightarrow k \neq k''$ e quindi Eve non riuscirebbe a rompere lo schema crittografico di cifratura, l'unico modo per Eve di ottenere il dato in chiaro è riuscire a trovare una funzione H^{-1} t.c. $k = H^{-1}(k')$ e utilizzarla per decifrare le informazioni di Alice - ci basta: ***One Way***.
L'unica limitazione è che se l'input k è debole e quindi vulnerabile a *brute-force* allora sarebbe possibile eseguire una ricerca esaustiva nell'insieme delle chiavi - ***HKDF***.

Second pre-Image Collision Resistance: dato un d e un m tale che $d = H(m)$ trovare un valore m' t.c. $d = H(m')$ è tecnicamente richiesta per gli ***authentication schemes*** ad esempio nel

contesto di ***Keyed Hash Function*** - per l'offuscamento di password su DB - infatti riuscendo o a trovare H^{-1} o trovando un'altro valore m' che generi lo stesso *digest* è possibile bypassare il controllo.

Esempio: **Download di dati**

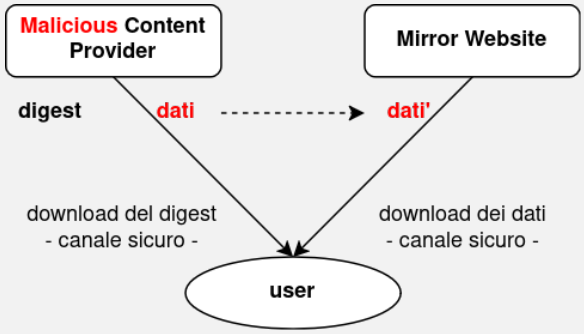


Quando un sito mette a disposizione dei dati per il download - ad esempio un **.iso*, normalmente il contenuto per il download è disponibile su un *server mirror*, quindi il *provider* si calcola il *digest* $d = H(m)$ e poi fa l'upload del contenuto sul *server mirror*. L'utente si scarica il contenuto dal *server mirror* e scarica il *digest* dal *provider*, successivamente calcola l'hash del dato scaricato $d_u = H(m_d)$ se e solo se $d_u = d$ accetta il dato scaricato.

In questo caso la funzione hash H deve rispettare la "normativa" di sicurezza *second pre-image collision resistance* in quanto il messaggio iniziale - la nostra **.iso* - viene fornita dal *server mirror*. L'attaccante vincerebbe se riuscisse a trovare un messaggio m_a t.c. $H(m_d) = H(m_a)$ e contemporaneamente quel messaggio dovrebbe contenere un *payload* malevolo. Solo in quel caso l'utente lo scaricherebbe il messaggio e la funzione di verifica - chiamata ***verify*** - andrebbe a buon fine e quindi l'utente accetterebbe il nuovo messaggio.

Collision Resistance: riuscire ad ottenere, arbitrariamente, due messaggi m_1 e m_2 tali che $H(m_1) = H(m_2)$ è la più forte come garanzia di sicurezza, nel caso dovesse mancare, la funzione hash sarebbe molto facile da violare.

Esempio



Il *provider* è malevolo e riesce a trovare due messaggio m e m' tali che $H(m) = H(m')$ a questo punto fa l'upload del messaggio malevolo sul *server mirror*, l'utente che si scarica riesce a verificare la correttezza del *digest*.

Hash Function Security Parameters: il **parametro di sicurezza** delle funzioni *hash* è la **lunghezza del *digest*** in quanto rappresenta le prossibili combinazioni di n bit, ovvero i possibili valori prodotti dalla funzione *hash* (2^n). Come discusso nel paragrafo di sicurezza computazionale, il valore di n deve essere scelto considerando l'attacco noto più efficiente, analizzando anche il numero di operazioni richieste per trovare una collisione.

Una funzione hash sicura è quello in cui l'attacco più efficiente per trovare una collisione è il ***Birthday Attack*** che richiede $2^{n/2}$ ovvero la probabilità di trovare una collisione è del 50% su N ($\sqrt{N}$)

Alcune delle più popolari funzioni hash:

- **md5**: ad oggi deprecata, molto facile trovare delle collisioni.
- **sha1**: anche questa ad oggi deprecata, vi sono trovate delle collisioni. 160bit di *digest*.
- **sha2**: sha1 "potenziata" ha diverse lunghezze del *digest* in base alla versione: **sha224**, **sha256**, **sha384** e **sha512**.
- **sha3**: è stata implementata con una primitiva differente rispetto a sha1 e sha2, è stata standardizzata ufficialmente nel 2015, ha le stesse sottoversioni del sha2.
- **blake2**: utilizza come primitiva **chacha** non è ancora stata standardizzata dal NIST ma è molto popolari in certi ambienti, ad esempio, quello del *software open source*.

Sia le funzioni ***hash*** che le funzioni ***mac*** vengono definite come funzioni con un solo input, ma è possibile concatenare più input, ma bisogna farlo in maniera sicura.

$$H('builtin'||'security') = H('built'||'insecurity') = H('builtinsecurity')$$

Concatenare i risultati cercando di rendere univoco l'output:

- utilizzando **caratteri speciali** per concatenare, se possibile: considersiamo che il carattere ; non sia ammesso come input allora sarà possibile concatenare gli input come: $d = H('builtin' \,||\, ';' \,||\, 'security')$
- **esplicitando** la **lunghezza dell'input**: $d = H('7buildinsecurity')$

3.2 Message Authentication Code

Il destinatario può rilevare se il messaggio non è stato mandato da un mittente legittimo, che nel caso lo si andasse a definire all'interno della crittografia simmetrica, identificherebbe colui che conosce il segreto condiviso. Il **MAC** è una funzione che ha due input: la chiave e il message e come output un ***tag***.

$$\textbf{tag} = \textbf{MAC}(key,\ message)$$

La funzione ***verify*** è simile a quella delle funzioni hash, ma include come input anche la chiave simmetrica.

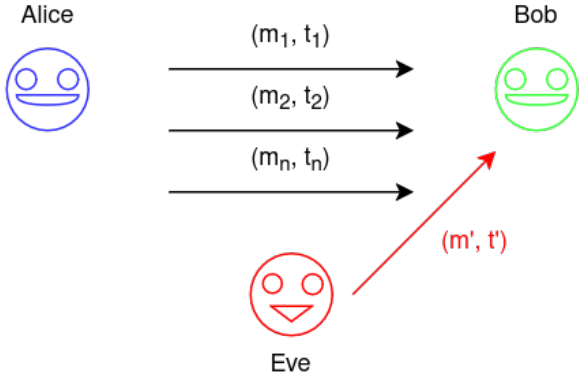
$$\{0,1\} \leftarrow \textbf{verify}(key,\ message,\ tag)$$

Il **MAC** permette a Bob di verificare che il messaggio è stato generato da Alice.

Attack Models for MAC

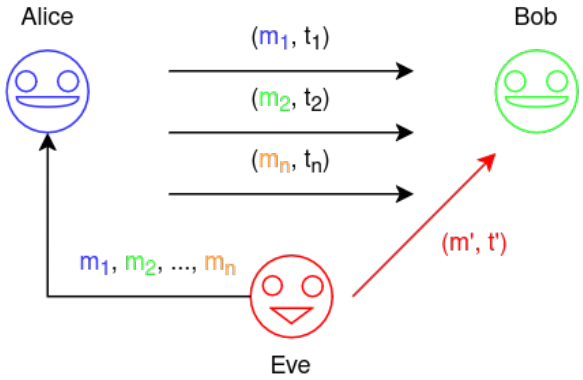
Existential Forgery for Passive Eavesdropper

L'attaccante osserva coppie (m_i, t_i) trasmesse tra gli attori benevoli (Alice e Bob) ed è capace di *forgiare* una nuova coppia (m', t') mai inviata e l'attacco ha successo nel caso in cui il messaggio venga accettato.



Existential Forgery for Chosen Message

In questo caso l'attaccante controlla i messaggi che vengono autenticati, quindi sceglie n volte un messaggio m_i e osserva il tag generato t_i a quel punto se riesce a generare una nuova coppia (m', t') non ancora inviata che verrà accettata l'attacco avrà avuto successo.



MAC: Security Parameters:

- la **lunghezza della chiave**: l'attacco più efficiente deve essere quello di *brute force*.
- la **lunghezza del *tag***: permette di determinare la dimensione del dato o il numero di messaggi autenticabili con la stessa chiave (dipende dal tipo di ***MAC scheme*** che si vuole utilizzare). Ricordiamo che il *tag* è ***overhead*** sul messaggio e quindi è possibile minimizzarlo ma aggiungendo a livello di protocollo altri fattori di sicurezza.

La scelta del **MAC** dipende dai requisiti nei quali va applicato. Disponibilità software (presenza di librerie), disponibilità hardware (accelleratore hardware) e l'abilità di soddisfare requisiti degli schemi (creare *nonce*). Ogni tipologia di **MAC** offre un diverso *trade-offs* in termini di: lunghezza e numero di messaggi, modelli di attacco e lunghezza del tag consentita.

Consideriamo i **MAC** all'interno di una situazione di ***Replay Attack***. Abbiamo detto che il **MAC** può garantire che il *tag* sia stato creato con una certa **chiave simmetrica**, ma nel contesto in cui siamo Eve re-invia lo stesso messaggio che ha inviato Alice a Bob, che è valido, perché creato da Alice - se Alice e Bob fossero due banchieri e Alice inviasse un messaggio con scritto "Aggiungi 1000 euro nell'account di Carlo". Con queste tipologie di attacco utilizzando la crittografia è difficile arginarle, è possibile però metterci una "pezza" lato architetturale (a livello **trasporto** o **applicazione**) ad esempio aggiungendo un **contatore univoco** al messaggio:

$$\mathbf{m} = (\text{id=n, data}), \text{tag}$$

Se consideriamo che la comunicazione sia *full-duplex* - ***Reflection Attacks*** - la possibilità che il messaggio venga inviato al mittente dello stessa e che venga verificato il *tag* non è nulla, quindi si è aggiunto un bit di direzione del messaggio:

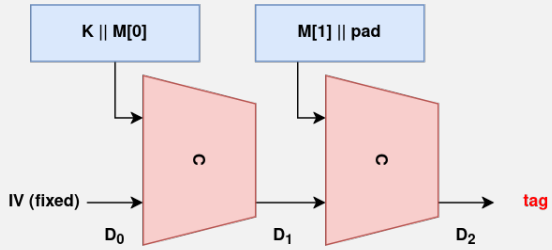
$$\mathbf{m} = (\text{id=n, dir=val, data}), \text{tag}$$

Il bit di direzione è aggiunto come **metadato** per un verifica ulteriore da parte del destinatario. È anche possibile utilizzare una difesa diversa, gestire la comunicazione *full-duplex* come due **comunicazioni sicure *half-duplex*** quindi utilizzare due **chiavi differenti** entrambe condivise, ma se invia Alice, verrà utilizzata k_1, se invia Bob k_2.

MAC & Hash function

Consideriamo un *tag* generato in questo modo: **H**(**key** || **message**), ovvero ci calcoliamo l'hash di una stringa generata concatenando la chiave con il messaggio. Si potrebbe pensare che se la funzione hash è *collision resistance* un'attaccante non riuscirebbe a calcolare un *digest* la cui pre-immagine è (parzialmente) sconosciuta. In verità questo tipo di costrutto è vulnerabile al ***length extension attack*** in quanto molte funzioni hash si basano sulla primitiva di ***Merkle-Damgard*** (non vale per **sha3**)

Merkle-Damgard Design



È la primitiva che viene utilizzata da *hash function* come **md5**, **sha1**, **sha2**. Considerando la figura, vediamo che:

- **K** è il segreto.
- **M[0]** e **M[1]** è pubblico.
- la costruzione del **pad** è pubblica.
- D_2 è pubblico (***tag***)

In questa circostanza l'attaccante vince se riesce a generare un *tag* valido per un certo messaggio arbitrario senza conoscere la chiave **K**. Noi (come attaccante) sappiamo che il *tag* è generato **H**(**key** || **message**) dove **K** è un segreto con una data lunghezza **l** fissata, noi vogliamo calcolare un certo *tag'* che venga generato **H**(**key** || **message** || **message'**).

1. Consideriamo di utilizzare la stessa *hash function*, e di ripristinare il suo stato interno come, quindi ponendo come nostro **IV** il *tag* generato dalla computazione legittima, ma è necessario impostare anche il punto iniziale da dove andare a calcolare il **pad'** e deve essere pari a $len(K) + len(M) + len(pad)$.
2. Inviamo come messaggio **M** || **pad** || **M'** e come tag quello appena calcolato ***tag'***.
3. il destinatario calcolerà **H**(**K** || **pad** || **M** || **pad** || **M'**) e produrrà lo stesso *tag'*

HMAC: è un MAC che viene costruito con alla base una funzione hash (alcune volte chiamato ***keyed-hash function***), è necessario che mantenga due requisiti funzionali:

- il **MAC** deve essere sicuro fintanto che la primitiva hash su cui è costruito è *collision resistance*.
- per molti degli scenari presi in considerazione è sufficiente una *second pre-image collision resistance*

Viene definito come:

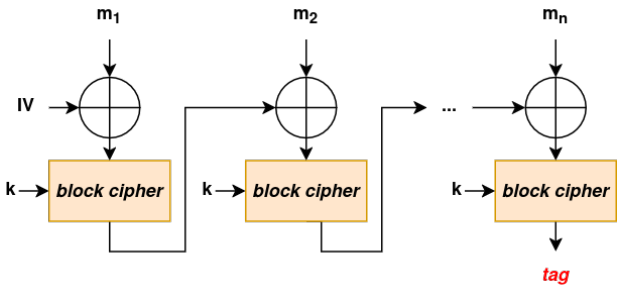
$$\mathbf{HMAC}(\mathbf{K}, \mathbf{M}) = \mathbf{H}((\mathrm{Kp} \oplus \mathrm{opad}) \,||\, \mathbf{H}((\mathrm{Kp} \oplus \mathrm{ipad}) \,||\, (\mathrm{m})))$$

Dove:

- **K** è un segreto a lunghezza variabile, mentre **Kp** è la sua versione *zero-padded*.

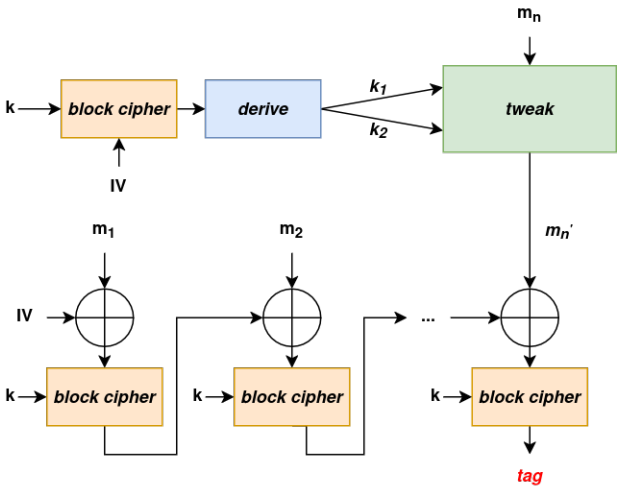
- **opad**(***outer padding***) e **ipad**(***inner padding***) sono costanti con un elevata distanza di Hamming (ovvero che le posizioni di bit diversi sono elevate), utilizzate per consentire alle due funzioni hash di utilizzare chiavi diverse.

Andiamo a considerare i **MAC** costruiti su *block cipher*, il primo e il più semplice per costruire un **MAC** basato su *block cipher* fu quello di utilizzare la **CBC** *mode of operation* e utilizzare come *tag* il testo cifrato dell'ultimo blocco.



È possibile *forgiare* un messaggio, da parte di un attaccante, infatti dato **(m, t)** e **(m', t')**, un CBC-MAC producerà **t'** per un messaggio costruito come: **m** || **t** $\oplus$ **m'** || **m'**.

CMAC è il successore del **CBC-MAC** utilizza come primitiva **AES-SIV**(**CTR-CMAC**), lo schema è molto simile ma aggiunge due operazioni sull'ultimo blocco prima di utilizzarlo. Lo schema deriva due chiavi k_1 e k_2 dalla chiave originale e una delle due viene *xorata* con l'ultimo blocco del messaggio m_n (***tweak block***) prima di utilizzare il blocco del **CMAC**.



UHF - Universal Hash Function si riferisce ad una ***keyed hash function*** che prende due input: una chiave **k** e un messaggio **m**. Questa funzione garantisce che la probabilità - che dati due messaggi distinti m_1 e m_2 - di avere $UHF(m_1) = UHF(m_2)$ è **trascurabile**. Un'implementazione popolare per una funzione **UHF** è quella basata su **polinomi modulo p** (con **p** primo):

- scegliamo un primo **p**
- consideriamo il messaggio m come un vettore di interi in $\mathbb{Z}_p$ con un massimo di n elementi: $m = [m_1, m_2, ..., m_n]$
- la funzione hash **H** viene calcolata utilizzando gli elementi di m come coefficienti del polinomio $F(k)$:

$$H(k, M) = \sum_{i=1}^{n} (m_i \cdot k^i) \mod p$$

One-Message Polynomial evaluation MACs: è possibile costruire un **MAC** partendo da delle **UHF**, consideriamo il messaggio $m = m_1, m_2, ..., m_n$

$$mac(m, k, r) = r + H(m, k) = r + \sum_{i=1}^{n} (m_i \cdot k^i) \mod p$$

In questo caso il **MAC** è un polinomio di grado n, **p** è fissato ed è pubblico, se m_i è 128bit allora $p > 128$. **k** e **r** vengono scelti randomicamente:

- **k** lavora come la chiave per la computazione, può essere utilizzata per più messaggi.
- **r** ha la stessa funzionalità di un ***nonce*** random, e bisogna utilizzarlo assolutamente per un unico messaggio, in questo caso **r** è **segreto**.

Nel caso in cui **r** venisse ripetuto per due *tag*:

$$\mathbf{t_1} = mac(m_1, k, r) = r + H(m_1, k)$$
$$t_1 - r - H(m_1, k) = 0$$
$$\mathbf{t_2} = mac(m_2, k, r) = r + H(m_2, k)$$
$$t_2 - r - H(m_2, k) = 0$$

$$t_1 - \cancel{r} - H(m_1, k) = t_2 - \cancel{r} - H(m_2, k)$$
$$\sum_{i=1}^{n} (m_{1,i} \cdot k^i) + t_1 = t_2 + \sum_{i=1}^{n} (m_{2,i} \cdot k^i)$$
$$\sum_{i=1}^{n} ((m_{1,i} - m_{2,i}) \cdot k^i) + t_1 - t_2 = 0 \mod p$$

In questo modo la chiave **k** è tra le radici del polinomio $m_2 - m_1 + t_1 + t_2$. È possibile estendere *one-message poly MACs* ad un *multi-message* utilizzando o una **PRF** oppure **PRP** (ad esempio un

block cipher), in questo modo, invece che utilizzare un valore di **r** random possiamo calcolarlo partendo da una chiave e un *nonce*:

$$mac(m, k = \langle k_1, k_2 \rangle, n) = AES(n, k_1) + H(m, k_2)$$

Se k_1 è segreta, un avversario, non sarebbe in grado di calcolarsi **AES**(n, k_1) anche se il valore n fosse pubblico e non randomico, ma utilizzare lo stesso *nonce* più volte esporrebbe k_2 e quindi romperebbe lo schema.

GMAC è basato su **AES-GCM** e in questo caso i rischi per la sicurezza in caso di errori di utilizzo sono molto più elevati a causa del comportamento lineare della funzione. Richiede un ***unpredictable nonce***, riutilizzare lo stesso *nonce* per firmare messaggi diversi permette la ***key recovery***, in questo caso è molto importante utilizzare la variante **SIV** - molto più robusti a discapito delle *performance* - come primivita dello schema (**AES-GCM-SIV**). La probabilità di successo in caso di attacco **aumenta** all'**aumentare della dimensione del messaggio**.

Non possiamo effettuare un'analisi a *black-box* per definire come la dimensione del *tag* influenzi il **livello di sicurezza**. Il dimensionamento richiede la conoscenza dello schema e le analisi sono parecchio complesse. Alcuni esempi:

- **HMAC** e **CMAC**: è possibile mandare un numero di messaggi pari alla radice quadrata della dimensione del *digest*
- **GHASH** (e **GMAC**): il numero dei messaggi è lineare con la dimensione del *digest*, ma diminuisce rispetto alla dimensione di ogni messaggio.
- **Poly1305**: rappresenta un *trade-off*, alto numero di messaggi dimensionalmente limitati.