

Università degli studi di Modena e Reggio Emilia

Dipartimento di Ingegneria Enzo Ferrari

Crittografia Applicata

Anno Accademico 2023/24

Indice

| | | |
|----------|--|-----------|
| 1 | Introduzione | 1 |
| 1.1 | Crittografia Classica | 1 |
| 1.2 | Crittografia Moderna | 2 |
| 1.2.1 | Encryption only | 2 |
| 1.2.2 | Extended and applied settings | 2 |
| 1.3 | Crittografia Applicata | 3 |
| 2 | Crittografia Simmetrica | 5 |
| 2.1 | Sicurezza Incondizionata & One-Time Pad | 7 |
| 2.2 | Sicurezza Computazionale: Security Level e Key Sizes | 7 |
| 2.3 | Stream Cipher | 11 |
| 2.4 | Block Cipher & Modes of Operation | 12 |
| 2.5 | Operation Framework for Symmetric Encryption | |
| | Developer view to symmetric encryption | 17 |
| 2.6 | How much data can we encrypt? | 19 |
| 3 | Hash Function & MAC | 20 |
| 3.1 | Hash Function | 21 |
| 3.2 | Message Authentication Code | 25 |
| 4 | Security Guarantees & Attack Model | |
| | for Symmetric Encryption | 31 |
| 5 | Encryption schemes with integrity guarantees | 38 |
| 5.1 | IND-CCA2 Authenticated Encryption | 38 |
| 5.2 | IND-CCA1 Encryption Schemes | |
| | length-preserving scheme for disk encryption | 40 |
| 5.3 | XTS Operation Mode | |
| | CCA1 narrow block encryption | 42 |

| | | |
|----------|---|-----------|
| 5.4 | Wide Block Encryption | 43 |
| 6 | Randomness | 44 |
| 7 | Derived Schema of Symmetric Crypto | 47 |
| 7.1 | SHA3 Derived Schemes | 47 |
| 7.2 | Key Derivation Function | 49 |
| 7.3 | Hash Table Flooding & small tag MAC's | 53 |
| 7.4 | Commitment Schemes | 55 |
| 7.5 | Authenticated Data Structure | 56 |
| 7.6 | Synthetic Initialization Vectors | 57 |

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 sufficientemente 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**).

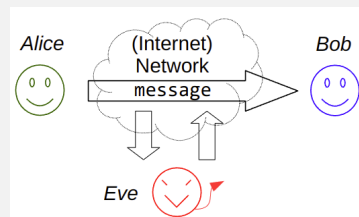
Una delle sicurezze che deve garantire un protocollo sicuro è la **confidenzialità**.

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

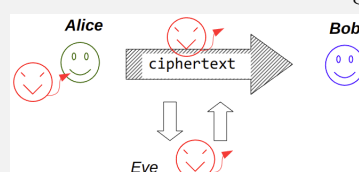
Un protocollo sicuro che ha come **scopo** proteggere le informazioni scambiati tra due attori: **Alice** e **Bob** quando un possibile attaccante **Eve** può accedere direttamente all'informazione tramite il canale fisico.



- **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 soprattutto 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)**.

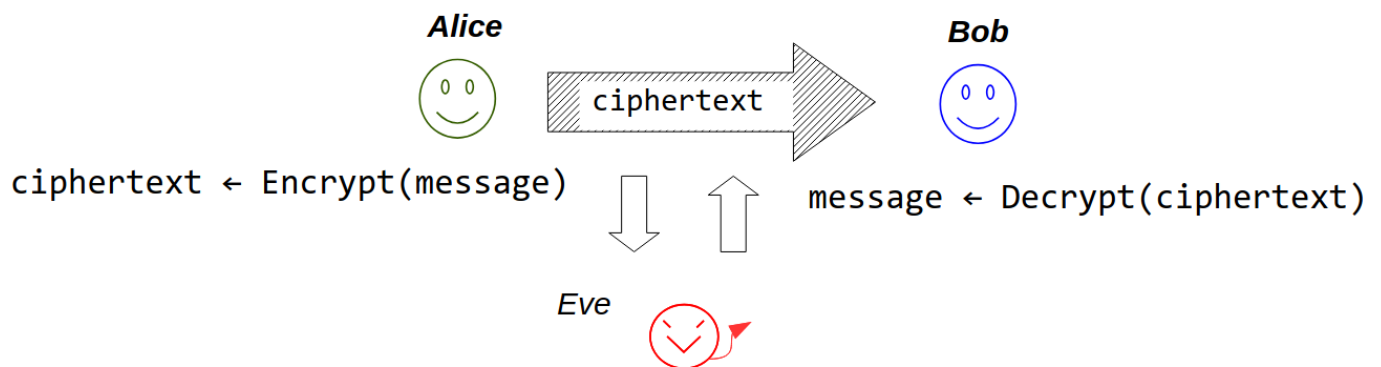
Per riassumere quanto visto fino ad adesso, possiamo dire che la **crittografia applicata** è l'insieme di molti **system models** (comunicazioni sincroni, messaggi di gruppi asincroni, *disk encryption*, ...), molti **security guarantees** (*information security*), integrità e autenticazione, molti **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 robustezza 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 provata 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 delle 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 **Information 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 **p = 0.5**.

In questo modo sapere **c** non dà informazioni su **m** e quindi **c** è indistinguibile da una successione 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 deve essere random. $c_i = m_i \oplus k_i \forall i \in \{0, \dots, n\}$ 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 predire 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 \mathbf{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 **efficienti**.
 - senza la **chiave** la probabilità di rompere lo schema crittografico deve essere **trascurabile**.

È necessario tradurre in termini algoritmici **efficienti** 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 \mathbf{N} per funzioni che hanno costo computazionale diverso, per quali valori di $\mathbf{n} > \mathbf{N}$ le probabilità di successo sono inferiori?

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

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 \mathbf{N} .

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

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 \rightarrow **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 \rightarrow **smaller than any inverse polynomial**

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

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

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:

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

Uno schema di crittografia deve essere **funzionale**. Definiamo \mathbf{F} come la **PRF** allora \mathbf{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 \mathbf{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.

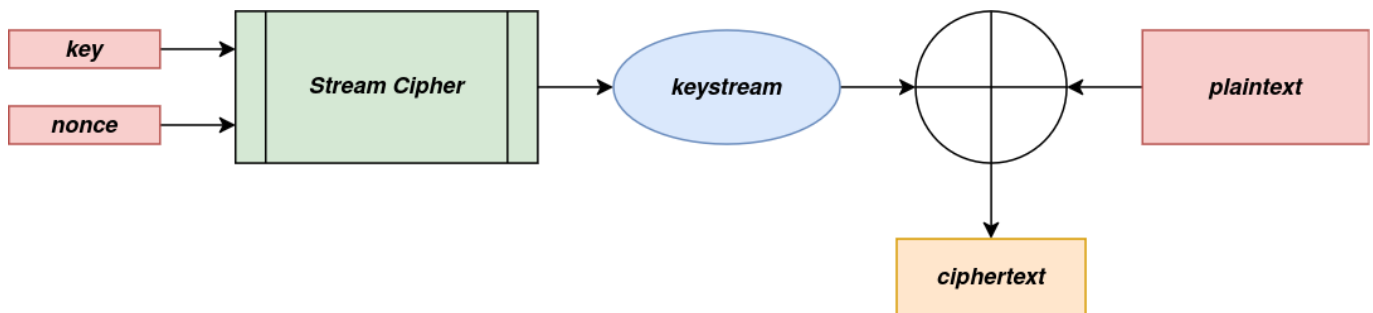
Software e librerie **dovrebbero** implementare configurazioni **sicure** di **default** e aggiornate se necessarie

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 **xorare** 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)

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

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

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.

Transparent 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)*].

$$\mathbf{F} : \{0, 1\}^{L_k} \times \{0, 1\}^{L_b} \mapsto \{0, 1\}^{L_b}$$

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 **efficienti**.

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*:

| <i>block cipher</i> | <i>block size</i> | <i>key size</i> | 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 istruzioni 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 influisce 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.

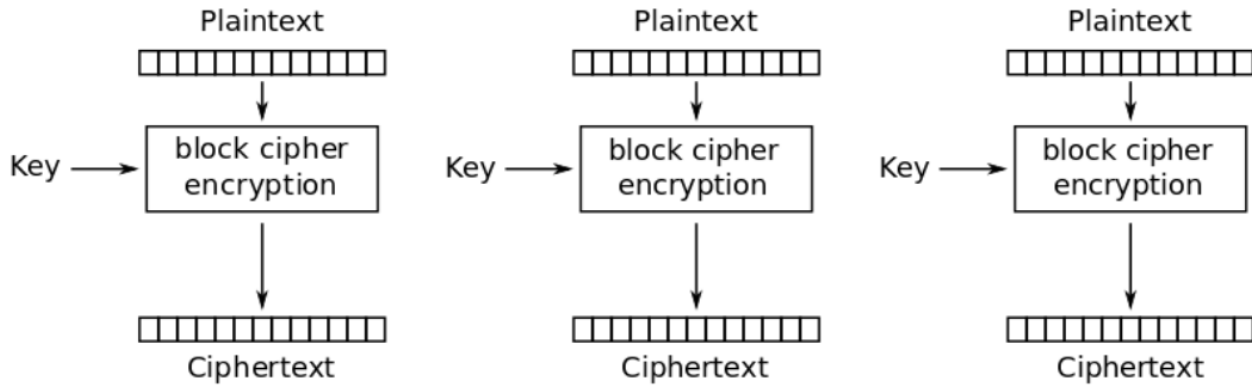
Nota: i *block ciphers* possono lavorare come **primitive** all'interno di altri blocchi di crittografia, ad esempio **ChaCha20** e **SHA2** utilizzano al loro interno dei *block cipher*.

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 ridurre la **malleabilità**.
- combinare la **riservatezza** con altre operazioni crittografiche, ad esempio garantire **integrità**: **authenticated encryption**.

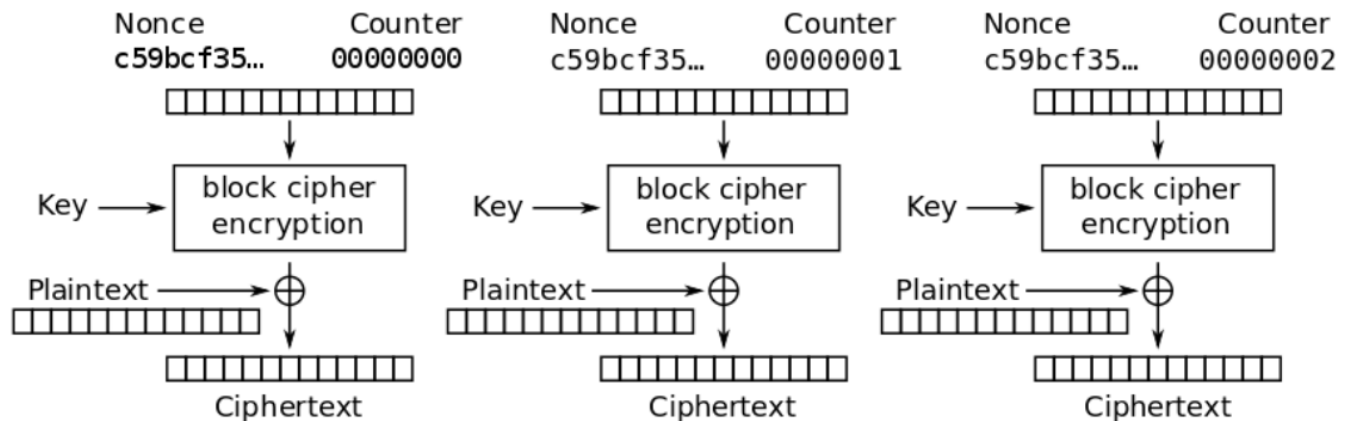
Mode of Operation - Electronic CodeBook (ECB)



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, soprattutto 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 $\simeq 64$ gb. 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 Chaining (CBC)

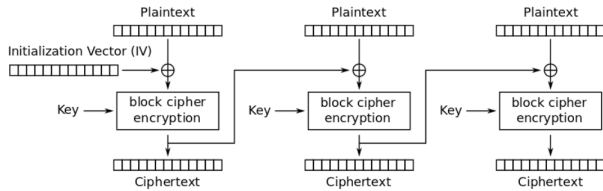


Figura 2.1: CBC *encryption*

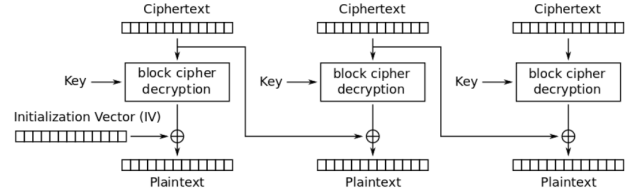


Figura 2.2: CBC *decryption*

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

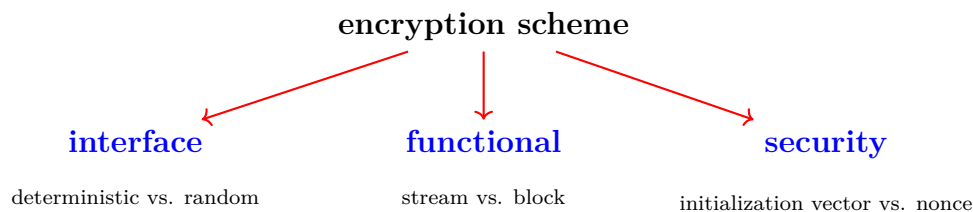
devono essere scelti da un attore “onesto” non dall’attaccante, se così non fosse ci possono essere attacchi potenziali a strutture simmetriche. È anche importante ricordare che i *block cipher* sono deterministici e che quindi utilizzando (nel caso del **CBC**) lo stesso **IV**, stesso *plaintext* e 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 Operation Framework for Symmetric Encryption

Developer view to symmetric encryption

Operations Framework \leftrightarrow *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*.

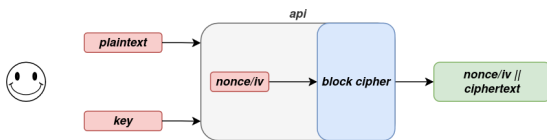


Figura 2.3: Probabilistic Encryption

Nata per correggere errori di uso improprio delle primitive di crittografia, infatti si assumeva che chi le utilizzava avesse una buona conoscenza delle basi crittografiche. A pari input l'output varia.

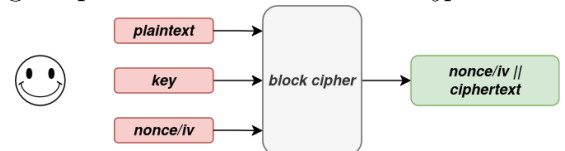


Figura 2.4: Deterministic Encryption

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:

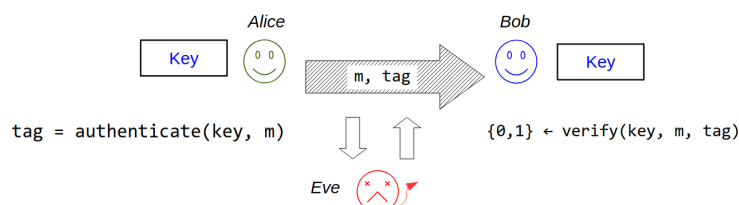
confidenzialità, **integrità** e **autenticità**. Abbiamo anche definito che Alice e Bob condividono la conoscenza di un'unica chiave per la comunicazione. Con i *block cipher* siamo riusciti a ottenere **confidenzialità**.

Integrity: è possibile 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 funzione 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 Guarantee: 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 questo 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 computazionale 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'output 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 comportamento delle funzioni hash - **PRF** - può essere applicato anche a circostanze 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**.

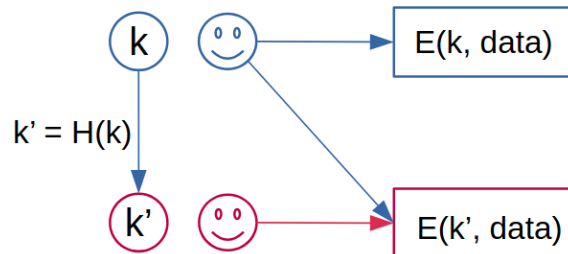
In base al tipo di applicazione che stiamo utilizzando bisogna che la funzione hash sia resistente a diversi *attack model*, tutti quanti, se no viene definita **deprecata**.

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**.



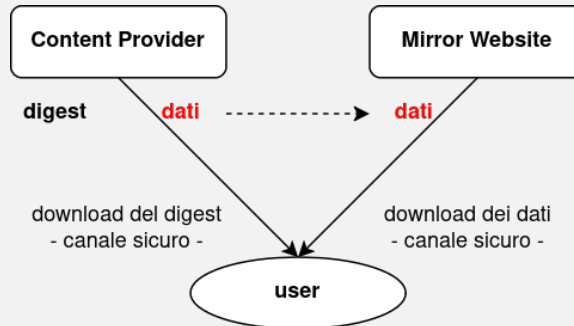
Dato in input una chiave, ottengo in output un'altra chiave $k' = H(k)$ in questo caso cosa dovrebbe rompere Eve per riuscire a decifrare? In questo contesto anche se H **non** è *collision resistance* non è di interesse infatti anche se si trovasse un k'' t.c. $k' = H(k'') \rightarrow k \neq k''$ e 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.

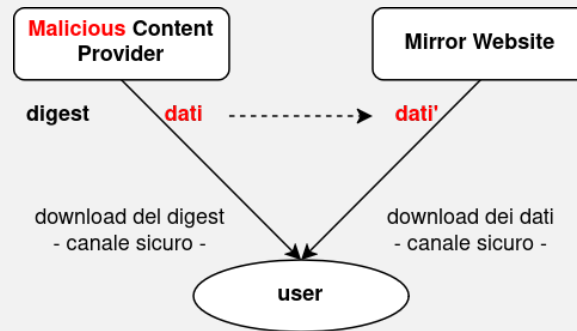
Esempio: Download di dati



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 messaggi 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 possibili 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.

$$H('builtin' || 'security') = H('builtin' || 'insecurity') = H('builtininsecurity')$$

Concatenare i risultati cercando di rendere univoco l'output:

- utilizzando **caratteri speciali** per concatenare, se possibile: consideriamo che il carattere ; non sia ammesso come input allora sarà possibile concatenare gli input come:

$$d = H('builtin' || ';' || 'security')$$

- esplicitando la lunghezza dell'input: $d = H('7builtinsecurity')$

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*.

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

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

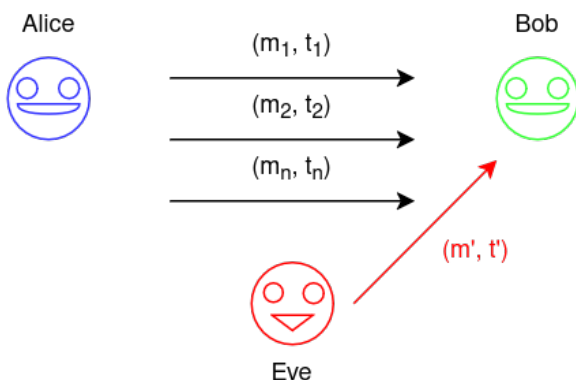
$$\{0, 1\} \leftarrow \text{verify}(\text{key}, \text{message}, \text{tag})$$

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

Attack Models for MAC

Existential Forgery for Passive Eavesdropper

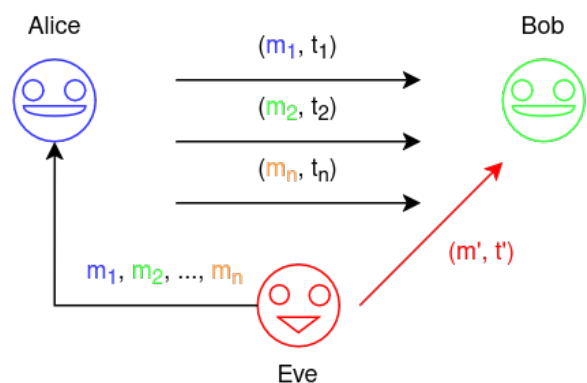
L'attaccante osserva coppie (m_i, t_i) trasmesse tra gli attori benevoli (Alice e Bob) ed è capace di *forgiare* una nuova coppia (m', t') mai inviata e l'attacco ha successo nel caso in cui il messaggio venga accettato.



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 (acceleratore 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:

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

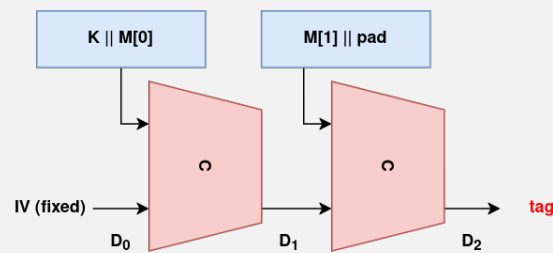
Se consideriamo che la comunicazione sia *full-duplex* - **Reflection Attacks** - la possibilità che il messaggio venga inviato al mittente dello stessa e che venga verificato il *tag* non è nulla, quindi si è aggiunto un bit di direzione del messaggio:

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

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: $\mathbf{H}(\text{key} \parallel \text{message})$, 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 $H(\text{key} \parallel \text{message})$ dove K è un segreto con una data lunghezza l fissata, noi vogliamo calcolare un certo **tag'** che venga generato $H(\text{key} \parallel \text{message} \parallel \text{message}')$.

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 $\text{len}(K) + \text{len}(M) + \text{len}(\text{pad})$.
2. Inviemo come messaggio $M \parallel \text{pad} \parallel M'$ e come tag quello appena calcolato **tag'**.
3. il destinatario calcolerà $H(K \parallel \text{pad} \parallel M \parallel \text{pad} \parallel M')$ 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:

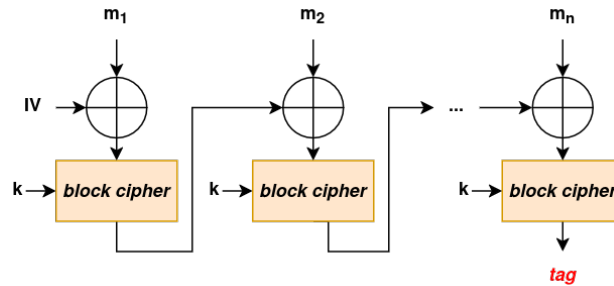
$$\text{HMAC}(K, M) = \text{H}((K_p \oplus \text{opad}) \parallel \text{H}((K_p \oplus \text{ipad}) \parallel (m)))$$

Dove:

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

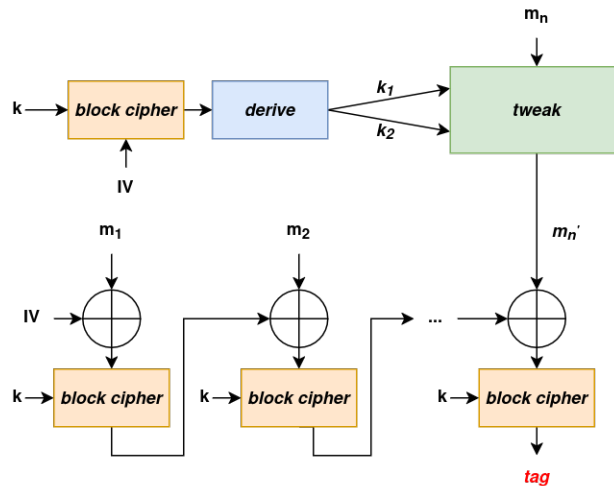
- **opad**(*outer padding*) e **ipad**(*inner padding*) sono costanti con un'elevata distanza di Hamming (ovvero che le posizioni di bit diversi sono elevate), utilizzate per consentire alle due funzioni hash di utilizzare chiavi diverse.

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 (\mathbf{m}, \mathbf{t}) e $(\mathbf{m}', \mathbf{t}')$, un **CBC-MAC** produrrà \mathbf{t}' per un messaggio costruito come: $\mathbf{m} \parallel \mathbf{t} \oplus \mathbf{m}' \parallel \mathbf{m}'$.

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:

$$m = [m_1, m_2, \dots, m_n]$$
- 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 $m = m_1, m_2, \dots, m_n$

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

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*:

$$t_1 = mac(m_1, k, r) = r + H(m_1, k)$$

$$t_1 - r - H(m_1, k) = 0$$

$$t_2 = mac(m_2, k, r) = r + H(m_2, k)$$

$$t_2 - r - H(m_2, k) = 0$$

$$t_1 - r - H(m_1, k) = t_2 - r - H(m_2, k)$$

$$\sum_{i=1}^n (m_{1,i} \cdot k^i) + t_1 = t_2 + \sum_{i=1}^n (m_{2,i} \cdot k^i)$$

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

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 \mathbf{r} random possiamo calcolarlo partendo da una chiave e un *nonce*:

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

Se k_1 è segreta, un avversario, non sarebbe in grado di calcolarsi $\text{AES}(n, k_1)$ 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 primitiva 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.

Capitolo 4

Security Guarantees & Attack Model

for Symmetric Encryption

Ogni schema crittografico garantisce sicurezza contro un certo attaccante a condizione che siano soddisfatti certi requisiti:

- dobbiamo decidere quale modello si adatta allo schema considerato.
- dobbiamo considerare i vincoli funzionali e non.

Lo scopo finale dell'attaccante è quello di violare la **confidenzialità**, per ora abbiamo solo considerato un attaccante che non ha alcuna conoscenza **EAV**, ma a volte, l'attaccante può anche:

- avere **conoscenza** aggiuntiva sul *plaintext*.
- violare l'**integrità** di un vettore di attacco per ottenere delle informazioni.
- **interagire** con le parti fidate.

È importante ricordare che la confidenzialità nella crittografia moderna è modellata come **indistinguibilità** da una sequenza random. Infatti l'avversario vince nel caso in cui è capace di riconoscere un crittogramma da un messaggio random con una probabilità maggiore del $50\% + \text{negl}(n)$

Non è possibile enumerare tutti i possibili attacchi di uno scenario reale, quello che si prefigge la crittografia moderna è considerare alcuni modelli formali contro i quali la sicurezza degli schemi crittografici è comprovata. Il che ci permette di progettare il protocollo o l'applicazione andando a scegliere il modello più adatto al proprio scenario. Gli schemi di sicurezza sono dimostrati sicuri in presenza di determinati modelli di attacco.

Attack Model

I modelli di attacco vengono pensati basandosi su due criteri di misura: **knowledge** - che cosa sa l'avversario del nostro sistema - **attack surface** - quali interfacce espone il nostro sistema che anche un avversario potrebbe utilizzare.

La crittografia formale moderna chiama queste interfacce **oracoli**, e normalmente andiamo a definirne di due tipologie:

- **encryption oracle**: l'attaccante può dare **input** all'oracolo e gli verrà ritornato un dato cifrato in **output** che corrisponde a $E(\text{input})$.
- **decryption oracle**: è il duale dell'*encryption oracle*.

Un vincolo aggiuntivo che si può aggiungere alla superficie di attacco quando si considera la distribuzione di un applicativo nel mondo reale è la tipologia di computazione che deve eseguire un attaccante: **online** - l'interazione è con l'oracolo - e **offline** - non è necessario consultare l'oracolo.

Attack Models Differenti

- **Eavesdropper - EAV** anche nota come **Ciphertext-Only Attack - COA** l'avversario è passivo e non ha conoscenza sui crittogrammi che riesce ad intercettare.
- **Known-Plaintext Attack - KPA** in questo caso l'avversario ha la conoscenza della coppia (p_i, c_i)
- **Chosen-Plaintext Attack - CPA** l'attaccante ha accesso ad una *encryption interface* e quindi è in grado di cifrare messaggi arbitrari.
- **Non-Adaptive Chosen-Ciphertext Attack - CCA1**: l'avversario ha accesso ad una *decryption interface*, ma accesso limitato al risultato e limite sul numero di *query* che può eseguire.
- **Adaptive Chosen-Ciphertext Attack - CCA2**: l'avversario ha accesso ad una *decryption interface* e anche al risultato e quindi modificare l'attacco in base ai risultati.

stronger attack models

Come detto in precedenza la **confidenzialità** in crittografia moderna equivale a dire che il crittogramma prodotto è **indistinguibile** da un *random bitstring*. Associando **IND** ad un modello di attacco andiamo a definire una **garanzia di sicurezza** dello schema preso in considerazione, ad esempio dire che il nostro schema di crittografia è **IND-CPA** significa che lo schema è **indistinguibile da un random** contro un modello di attacco **Chosen-Plaintext Attack**.

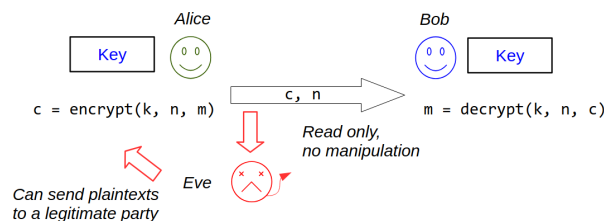
IND-EAV (Indistinguishably under Eavesdropper Attack): è la tipologia di attacco più debole, in quanto un attaccante non sa nulla. Eve è un attaccante passivo che osserva crittogrammi senza eseguire alcuna modifica su di essi e senza interagire con le parti fidate, ma soprattutto senza alcuna conoscenza del testo in chiaro.

Semantic Security Games on IND-COA

Tutti i *security games* per la confidenzialità sono basati sul concetto di **indistinguibilità**, un algoritmo eseguito dall'attaccante che ha come scopo di distinguere informazioni dal *plaintext*. Nel modello di attacco **COA**, l'avversario legge i crittogrammi in transito e prova a indovinare il bit successivo. Riuscirà a vincere il gioco solo se ha un *distinguisher* che indovina il bit corretto con una probabilità > 0.5 , nei moderni modelli di attacco, l'avversario non deve necessariamente ottenere la chiave di cifrazione o il contenuto effettivo del testo in chiaro affinché uno schema si consideri "rotto".

IND-KPA (Indistinguishably under Know-Plaintext Attack): Eve è un attaccante passivo che osserva i crittogrammi senza eseguire alcuna modifica su di essi e senza interagire con le parti fidate, ma può conoscere vecchie corrispondenze tra m e c o informazioni sullo spazio dei testi in chiaro, come la distribuzione statistica.

IND-CPA (Indistinguishably under Chose-Plaintext Attack): è la sicurezza minima richiesta da tutti i moderni crittosistemi, l'attaccante viene modellato per avere accesso ad un *encryption oracle*.

**Security Games**

1. il **challenger** genera una chiave segreta.
2. l'**adversary** sceglie due messaggi m_0 e m_1 della stessa lunghezza e gli invia al **challenger**.
3. il **challenger** genera randomicamente un bit $b \leftarrow \{0, 1\}$ e genera $c = E(m_b)$ e lo invia all'avversario.
4. l'**adversary** analizza il crittogramma e può scegliere se ripetere gli step dal secondo o passare al quinto.
5. l'avversario espone come output un valore $b \leftarrow \{0, 1\}$ in base a quale messaggio - se m_0 o m_1 - è il corrispondente di c .

L'*adversary* vince il gioco se è capace di riconoscere con una probabilità $> 0.5\% + \text{negl}(l)$ a quale messaggio corrisponde il crittogramma ricevuto.

Qualunque schema **deterministico** è intrinsecamente vulnerabile a un attacco del tipo

IND-CPA, in quanto basterebbe all'attaccante ripetere lo stesso messaggio per un più iterazioni

consecutive e vincere, infatti schemi crittografici sono sicuri sotto la modellazione **Distinct Chosen-Plaintext Attack - DCPA** che obbligano l'*adversary* a scegliere una coppia di messaggi diversi per ogni iterazione del *security game*. Crittografia deterministica è spesso implementata fornendo *nonce* o *IV* fissi (costanti) allo schema, cercando di evitare schemi che sono vulnerabili contro *re-use* di *nonce* o *IV* oppure preferendo schemi che utilizzano **SIV**.

Deterministic Encryption with Associated Data - DAEAD framework.

IND-CPA e il requisito di imprevedibilità dell'IV in CBC

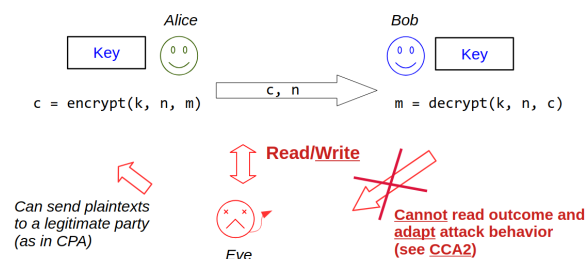
- *adversary*: sceglie un m che ha lunghezza pari alla *block size* del *block cipher* e lo invia.
- *challenger*: sceglie un *IV* e calcola il *ciphertext* c , ritornando la coppia (c, iv) .

Facciamo ora un'**assunzione**: l'avversario conosce l'*IV* successivo - **predictable IV**.

L'avversario può calcolarsi un messaggio m' come $m' = (m \oplus iv \oplus iv')$ dove iv' è l'*IV* successivo che utilizzerà il *challenger*. Se il *challenger* cifrerà il messaggio

$$c' = E(iv', m') = E(m' \oplus iv', k) = E(m \oplus iv \oplus iv' \oplus iv', k) = E(m \oplus iv, k) = c$$

IND-CCA1 (Indistinguishably under Chosen-Ciphertext Attack) permette di difendersi contro attacchi attivi sul crittogramma. L'attaccante è capace di manipolare il *ciphertext* al *decryption oracle* (gli input dell'attaccante non sono adattivi per via della definizione data).



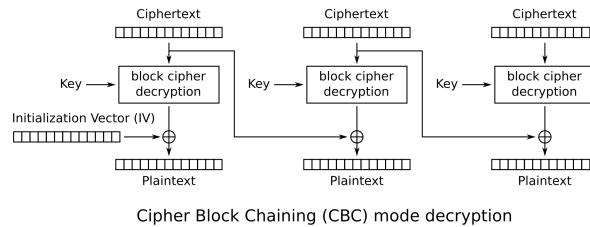
IND-CCA coinvolge anche **modifiche al crittogramma** e **attacchi all'integrità** per violare la **confidenzialità**.

Malleability si riferisce alla capacità di un'attaccante di ricostruire quali bit del crittogramma sono influenzati dalla modifica di un bit nel testo in chiaro.

Un schema crittografico basato su **stream cipher** viene definito **fully malleable** in quanto una modifica nel testo cifrato in una certa posizione c_i porta ad una modifica nel testo in chiaro nella medesima posizione p_i , ad esempio:

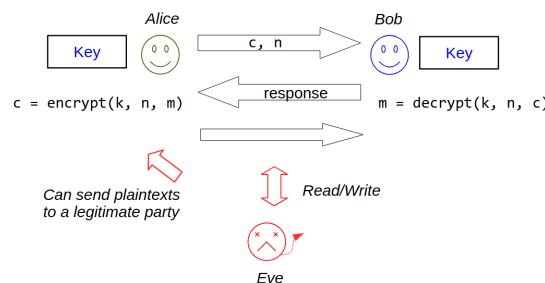
1. *flipping* il **bit di direzione** in uno protocollo di comunicazione per eseguire un **reflection attack**.
2. *flipping* il numero di versione per effettuare un **downgrade attack**.

Per ottenere una certa garanzia di **integrità** (senza utilizzare **MAC**) è necessario ricorrere a modalità di crittografia basate su *block cipher*. Andiamo ad osservare il comportamento di uno schema crittografico basato su un qualunque *block cipher* ma che abbia come *mode of operation* **CBC**, in questo caso analizzando lo schema per decifrare un crittogramma generato, ad esempio, con **AES-CBC** è fattibile modificare l'*IV*.



Assumiamo che l'attaccante conosca il testo originale p e voglia che in decifrazione venga generato il testo p' , nel **primo blocco**, allora sarebbe possibile calcolarsi un IV tale che $IV' = (IV \oplus p \oplus p')$ e sostituire l' IV originale con quello generato, in quel caso il primo blocco sarebbe p' e i restanti blocchi rimarrebbero invariati.

IND-CCA2 (Indistinguishably under Adaptive Chosen-Ciphertext Attack) difende da attacchi attivi iterati e potenzialmente illimitati (l'attaccante è ancora **polinomiale**), in cui l'attaccante adatta i suoi attacchi in base alla risposta di Bob. L'attaccante invia un testo cifrato manipolato al *decryption oracle*.



Padding Oracle Attack

Consideriamo un attacco alle funzioni di **decifrazione** e **unpadding**, l'attacco ha successo se riusciamo a generare un crittogramma che decifrandolo ha la struttura del *padding* corretto. Gli attacchi al *padding* mirano a manipolare un blocco di un crittogramma $c[i]$ per manipolare il blocco successivo di *plaintext* $p[i + 1]$. È simile alla manipolazione dell' IV ma in questo caso andiamo a considerare gli ultimi due blocchi del *ciphertext*, modificare il penultimo blocco per manipolare l'ultimo. In questo caso non abbiamo la certezza di come manipolare il crittogramma, ma possiamo *guessare* - questo comporta che, in caso, di un unico tentativo ci siano probabilità trascurabili di successo, se avessimo più tentativi rientriamo nei **CCA2 attacks**.

```

1 import string
2
3 for character in string.printable:
4     c[-2][-1] = character
5     p[-1] = c[-2] ⊕ D(c[-1])
6
7     if p[-1] has valid padding:
8         attack ok
9     else:
10         attack failed

```

Fissiamo \mathbf{X} come il *forged ciphertext* e con \mathbf{X}_n l' n -simo **LSB**, \mathbf{P} è il *plaintext* originale e \mathbf{T} l'output della funzione di decifrazione.

1. impostiamo $n = 1$ e $\mathbf{X}_n = 0$
2. eseguiamo la *decryption query* con $D(iv = X, ciphertext = V)$ e osserviamo la risposta:
 - **errore**: il crittogramma decifrato non ha il *padding* valido, quindi $\mathbf{X}_{n+} = 1$ e ripetiamo dal passo 2.
 - **ok**: attacco andato a buon fine (tentativi massimo: 256, 128 in media), procediamo al punto 3.
3. segniamo \mathbf{X}_n come $\mathbf{X}_{n,n}$
4. se $n = 16$ procediamo al passo 5, se no definiamo

$$\mathbf{X}_{m,(m+1)} = [\mathbf{X}_{m,m} \oplus m \oplus (m+1)] \forall m \in [1, n]$$

e $n+ = 1$ e ripetiamo dal passo 2.

5. in fine possiamo decifrare il *plaintext* originale, nell'ultimo blocco, calcolando:

$$P = C \oplus X \oplus [16]$$

Dove $[16]$ è un blocco di padding nello standard **PKCS#7**

Questo attacco funziona perché:

- \mathbf{V} non viene mai modificato e quindi anche \mathbf{T} che è sconosciuto.
- la decifratura non modificata esegue: $T \oplus C = P$.
- il *padding oracle* ci permette di calcolare \mathbf{X} in modo tale che $(T \oplus \mathbf{X}) = [16]$.
- $T = (\mathbf{X} \oplus [16])$ e $P = (C \oplus T) = (C \oplus C \oplus [16])$

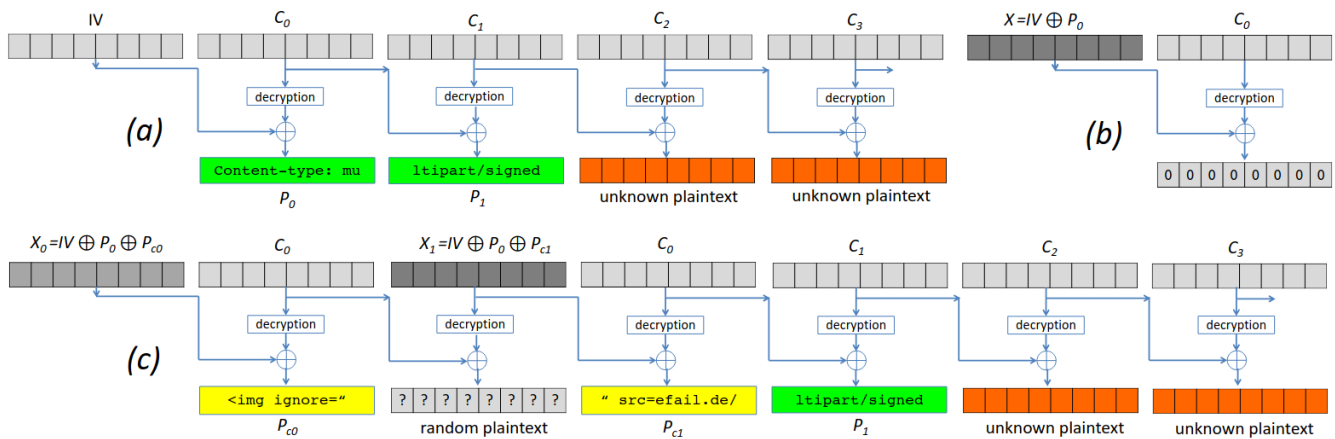
In questo caso non riuscire a sfruttare la **malleabilità** del sistema implicherebbe l'impossibilità di effettuare questo tipo di attacco, infatti la probabilità di indovinare il **padding** di dimensione n sarebbe in media 2^{8n+1} che è **inefficiente** in n . Siccome però in questo caso siamo capaci di sfruttare la malleabilità l'algoritmo abbassa il suo costo a $2^{8-1} \cdot n$ tentativi per un padding di dimensione n . Per **AES** sono necessarie all'incirca $\simeq 2000$ iterazioni per decifrare un blocco, è possibile decifrare l'intero crittogramma andando a **troncare** il crittogramma all'ultimo blocco.

In alcuni casi potrebbe essere necessario gestire i falsi positivi del caso 2, al passaggio n potremmo trovare un padding di dimensione $n' > n$

Authenticated Encryption: tutti quelli schemi di crittografia che sono resistenti a modelli di attacco **IND-CCA2** verranno chiamati *authenticated encryption scheme* ovvero schemi che non sono malleabili e che quindi nel caso di modifica, questa verrà rilevata, internamente utilizzano schemi di crittografia insieme a funzioni **MAC**, alcuni esempi:

- **AES-GCM**: utilizza **AES-CTR** insieme ad un **GMAC**.
- **ChaCha20Poly1305**: utilizza **ChaCha20** insieme ad un autenticatore **Poly1305**.
- **AES-CCM**: utilizza **AES-CTR** insieme ad un **CMAC**.

Un esempio è quello di **EFail vulnerability**.



Protocolli crittografici: **s/MIME + OpenPGP (AES-CBC)**, vettore di attacco: **Man in the middle**. L'attaccante espose un servizio web che corrispondeva a

<http://efail.de/ltipart/signed> e riuscì a leggere la mail in chiaro all'interno della richiesta.

Capitolo 5

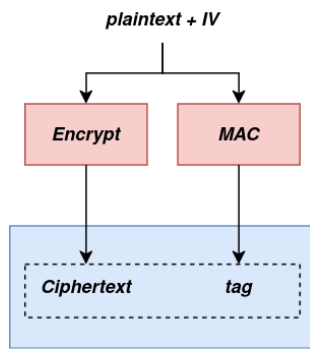
Encryption schemes with integrity guarantees

5.1 IND-CCA2 Authenticated Encryption

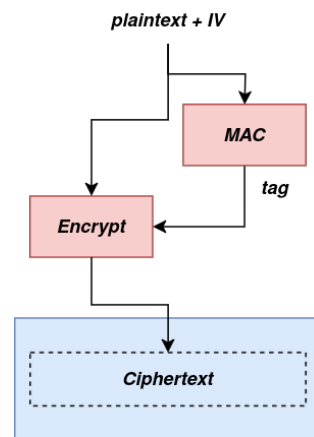
La crittografia “normale” deve rispettare il requisito di **confidenzialità** contro un modello di attaccante del tipo **CPA** e **CCA1**. Per quanto riguarda la modellazione **CCA2** per riuscire a garantirne confidenzialità è necessario aggiungerci anche garanzie di **integrità** (tramite le *hash function*) e **autenticità** (tramite *MAC*). Chiameremo ***Authenticated Encryption*** questi schemi crittografici che garantiscono **confidenzialità**, **integrità** e **autenticità**. In casi di utilizzo reale:

- utilizzare schemi standard che garantiscono tutti questi requisiti.
- se non fossero disponibili, bisogna costruirsi uno partendo da uno schema non autenticato e da un MAC.

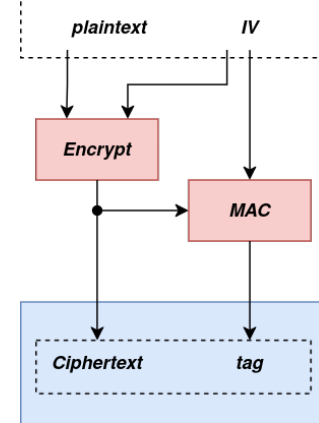
In maniera generale per riuscire ad ottenere *authenticated encryption* sono necessarie due primitive che verranno utilizzate a *black-box*: *block cipher* e *MAC scheme*. È possibile combinare questi tre elementi in tre maniere differenti.

Encrypt-and-MAC

In questo caso viene cifrato il testo in chiaro e composto il MAC in maniera parallela e poi vengono concatenati i risultati. Non ho alcuna garanzia che garantisca la confidenzialità del messaggio.

MAC-then-Encrypt

Prima calcola il *tag* e poi cifra la concatenazione tra *tag* e *plaintext*.

Encrypt-then-MAC

Prima si calcola il crittogramma che poi verrà utilizzato per generare il *tag*, concatenando in seguito il risultato. Il MAC è calcolato sul testo cifrato quindi non è possibile informazioni

aggiuntive

Considerando tutti e tre le combinazioni degli schema l'unica che garantisce il rispetto dei requisiti di sicurezza anche senza considerare le primitive sottostanti è l'***encrypt-then-MAC***.

1. la funzione **MAC** deve autenticare non solo il testo cifrato, ma anche l'IV.
2. lo schema di crittografia e la funzione di MAC devono utilizzare due tipologie di **chiavi** differenti.
3. nella fase di *decryption* il **tag** deve essere verificato **prima** di iniziare la decifratura in quanto un'attaccante potrebbe mettere in pratica un **timing attack**, ovvero ottenere informazioni dal tempo di esecuzione di una certa funzione.

Un'altra cosa fondamentale è che le funzioni crittografiche (*encryption* e *decryption*) devono essere **time-constant**

Authenticated Encryption with Associated Data - AEAD

```
1 key = keygen([size])
2 cipher = encrypt(key, nonce/iv, a, plain)
3 plain = decrypt(key, nonce/iv, a, cipher)
```

I **dati associati** non sono cifrati né inclusi nel crittogramma (principalmente per vincoli definiti dal contesto in cui sono), ma l'autenticità è invece vincolata anche a quell'informazione. In questo modo l'operazione di decifratura verificherà che gli *associated data* siano gli stessi dell'*encryption*, in caso contrario, **fallirà** riuscendo a garantire:

- confidenzialità, autenticità e integrità del *plaintext*.
- autenticità e integrità dei dati associati.

Aggiungono *overhead* sia a livello computazionale che di *storage*.

5.2 IND-CCA1 Encryption Schemes

length-preserving scheme for disk encryption

Non è possibile garantire forte autenticità (CCA2) senza adottare l'uso di MAC, ma in questo modo uno schema *CCA2 secure* introdurrà dell'*space overhead* rispetto a schemi non autenticati. In certi scenari è preferibile - se non richiesto - che il testo cifrato abbia la **stessa dimensione** del *plaintext*, molti di questi scenari sono i **disk (sector) encryption**. In questi scenari la migliore sicurezza che riusciamo a garantire è **IND-CCA1**.

Per come si è descritto lo scenario un *stream cipher* - che per definizione è completamente **malleabile** - è la peggior progettazione possibile in quanto non possiamo autenticare (in quanto introdurrebbe un *overhead*), è quindi guidata la scelta verso i *block cipher* - storicamente è sempre stato preferito il *mode of operations* **CBC**. I problemi di **CBC** legati ad attacchi attivi sono:

- molto vulnerabili contro la **manipolazione dell'IV**.
- molto vulnerabili contro attacchi del tipo **ciphertext substitutions** (es. *efail*).
- il *padding* può aiutare l'attaccante.
- “*functional issue*”: i blocchi sono **interdipendenti**.

Per la cifratura del disco ci sono alcuni requisiti che vanno rispettati, tra cui: la lunghezza del testo cifrato deve essere uguale al testo in chiaro (nessun *IV* o *nonce* e nessun *tag*), livello di sicurezza contro **attacchi CCA1** (è impossibile per l'attaccante avere un riscontro di cosa è stato modificato - scenario non adattivo) e ultimo, ma non per importanza è che devono essere veloci. Da questo possiamo andare a definire cosa alcune intuizioni di progettazione:

- la crittografia deve essere a **blocchi**.
- i blocchi devono essere **indipendenti** uno dall'altro, ma comunque non malleabili (**no CTR**) e random (**no ECB**).
- la randomicità non deve però introdurre problematiche di malleabilità e deve essere indipendente da ogni blocco (**no CBC-IV** con effetto a valanga).
- è possibile utilizzare i *sector number* - anche noti come *tweaks* - al posto del nonce.

Ricordiamoci che la garanzia di sicurezza richiesta da questo scenario è **CCA1 secure**, senza però essere autenticata il che comporta non riuscire a rilevare modifiche a livello crittografico. È possibile però incorporare una struttura aggiuntiva a livello di codifica per avere un'autenticazione forte - il *filesystem* può garantire l'integrità ad esempio con un *checksum*.

XTS operation mode: è stata standardizzata nel 2007 dalla *IEEE* e nel 2010 dal *NIST*, garantisce che una modifica nel *ciphertext* causi una modifica random nel *plaintext*, queste modifiche sono propagate unicamente in quel blocco

Adiantum encryption: molto veloce in *software*, consigliata nel momento in cui non è presente un acceleratore hardware per AES. È utilizzato come default per versioni di Android ≥ 10 e per *lower-end device* senza supporto AES hardware (negli altri casi viene preferito **AES-XTS**). Può utilizzare diversi *block cipher* come primitive (NH hash, ChaCha12, Poly1305, AES) ed è stata resa disponibile nel kernel main-line di linux dalla versione 5.0.

HCTR2: necessita accelerazione hardware per AES, è stato integrato nel kernel di linux nel Maggio 2022.

Adiantum e **HCTR2** sono **CCA1 secure wide-block encryption**.

Narrow Block Encrypt

XTS è **IND-CCA1 narrow block encryption** e lavora con blocchi di lunghezza pari alla *block size* (128bit per **AES**). Nel caso di compromissione o tentata manipolazione di un bit, questa verrà propagata unicamente all'interno del blocco corrente.

Wide Block Encrypt

Adiantum e **HCTR2** sono invece **IND-CCA1 wide block encryption** ovvero lo schema accetta un'altro parametro: la **block size**, riesce a ricostruire uno algoritmo partendo da *block cipher* con *block size* più piccola (esistono limitazioni legate alla sicurezza). Viene modellato come una **Super Pseudo Random Permutation - SPRP**. Nel caso di compromissione o tentata manipolazione di un bit, questa verrà propagata anche nei blocchi adiacenti aiutando ad identificarle in maniera più precisa.

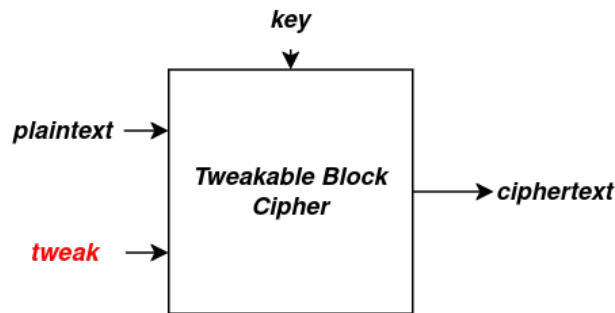
Un *disk sector* è di 4096 bytes, se noi andiamo ad impostare la dimensione del blocco in un algoritmo *wide block encryption* questo sarà molto più ottimizzato.

5.3 XTS Operation Mode

CCA1 narrow block encryption

XEX-based *Tweaked-codebook* mode with *ciphertext Stealing*

- **Tweakable Block Cipher**: può essere considerata una **primitiva crittografica** che estende un classico **block cipher** con un input aggiuntivo: il **tweak** che è simile al *nonce* in modello che è resistente al *nonce-reuse*: pubblico, univoco (la duplicazione non rompe la cifratura) e non ha requisiti di randomicità. È applicato ad un singolo blocco.



- **Xor Encrypt Xor - XEX** è una progettazione standard per costruire un *tweakable cipher* basandosi su un *block cipher* standard, garantendo **length-preserving** accetta in input dati che abbiano dimensione multipla della *block size*. XEX necessita di **due chiavi distinte**, che possono essere derivate o estese dalla stessa dal *layer* sopra.

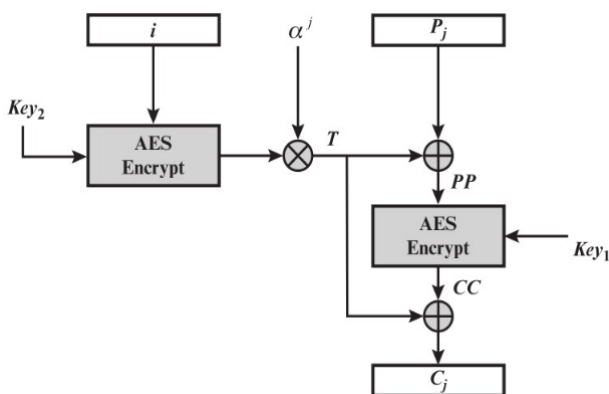


Figura 5.1: XEX Encryption

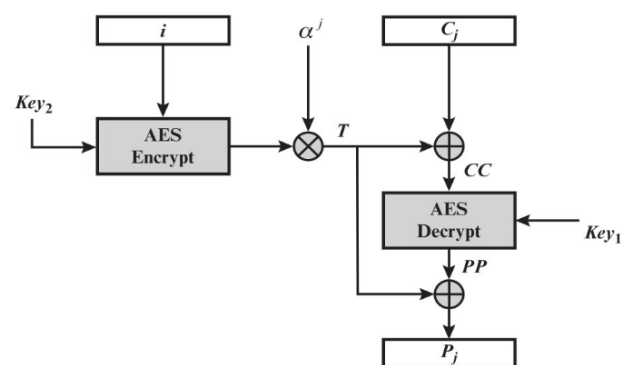
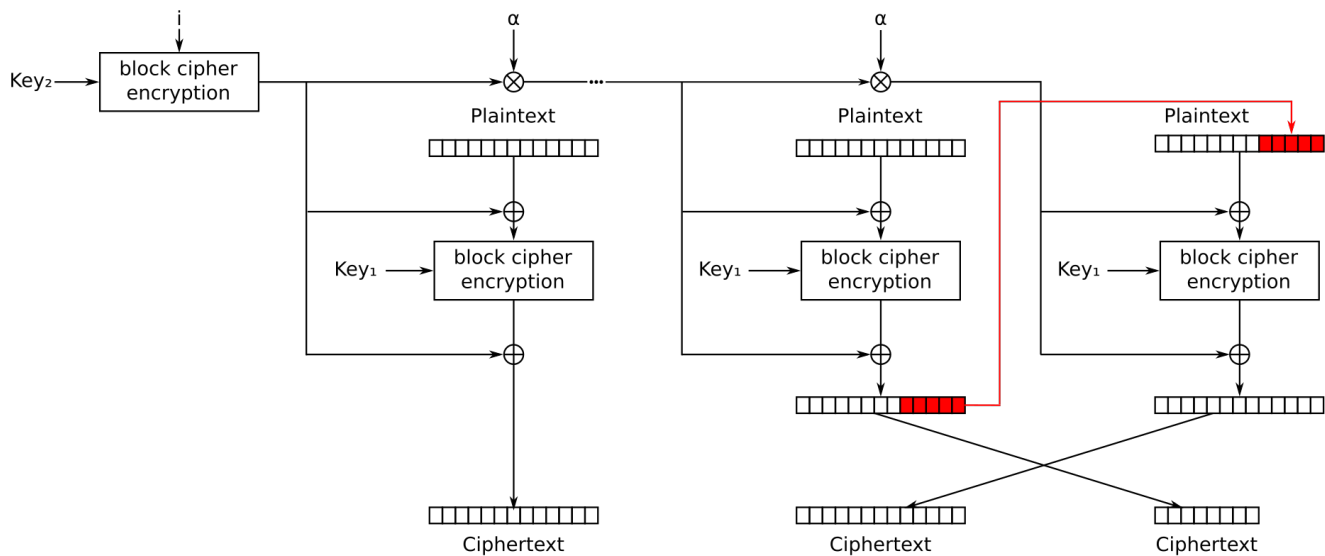


Figura 5.2: XEX Decryption

- **Ciphertext Stealing - CTS**: è una tecnica utilizzata per supportare dati di lunghezza variabile negli schemi di cifratura a blocchi, tuttavia introduce una stretta dipendenza tra gli ultimi due blocchi, a volte è stata anche utilizzata per mitigare gli attacchi legati al *padding oracle*, oggi si preferisce la cifratura autenticata.



XEX with tweak and ciphertext stealing (XTS) mode encryption

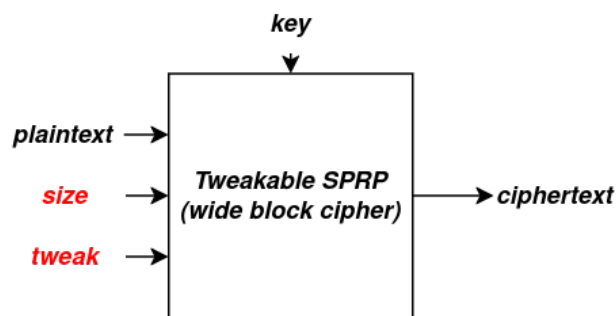
Figura 5.3: XTS Encryption

5.4 Wide Block Encryption

Un blocco *wide encryption* è modellato per la cifratura di *disk sector* ed è rappresentabile come un *Super Pseudo Random Permutation* che accetta come parametri:

- un *tweak* per supportare la randomizzazione basata sulla locazione dell'informazione.
- una *block size* che regola la quantità di dati che è contenuta nella permutazione.

È basato su un *block cipher* con una dimensione ridotta e fissa.



Capitolo 6

Randomness

I protocolli di crittografia sono spesso basati su **generatori di dati random**, ad esempio per la generazione delle **chiavi** o dell'**IV**. È importante differenziare un valore random con un valore **crittograficamente randomico** ovvero dati **impredicibili** che possono essere campionati da una distribuzione uniforme o coinvolgere altre tipologie di distribuzione - ad esempio quella Gaussiana. Molti eventi del mondo reale sono noti - nella fisica - come governati da:

- eventi probabilistici che sono “realmente” casuali o imprevedibili, ad esempio quelli studiati dalla fisica quantistica.
- comportamenti deterministici che sono molto difficili da prevedere in quanti gli output sono molto sensibili anche a variazioni molto piccole degli input, normalmente studiati nella teoria del caos.

Entropia: in ambito crittografico rappresenta l'imprevedibilità di un'informazione, maggiore è l'entropia, più è difficile per un attaccante indovinare o ricostruire quel dato. Ad esempio consideriamo una chiave a 128bit generata partendo da una scelta casuale di un solo bit attraverso un algoritmo deterministico, a quel punto avremo che la lunghezza della chiave sarà 128bit, ma i suoi bit di entropia sarà soltanto uno e questo fatto la rende facilmente attaccabile.

Normalmente in crittografia si richiede che la chiave abbia n bit di **entropia**, ovvero il nostro spazio delle chiavi sarà formato da 2^n tutti **equiprobabili**. Queste congettura, tuttavia, non valgono per **password** o **PIN**.

Consideriamo **chiavi crittografiche segrete uniformemente casuali** su stringhe binarie: in questo caso la lunghezza in bit è pari all'entropia della chiave. Nel caso di **PIN** avremo una sequenza di n numeri, possiamo calcolare la probabilità che esca un determinato numero: $\log_2(10^n)$ - ad esempio per $n = 4$ avremo che i bit di entropia saranno $\log_2(10^4) \simeq 13.28\text{bit}$. Invece, per una **password**, dipende dall'alfabeto scelto (n) - quindi la sua cardinalità - e dalla lunghezza della

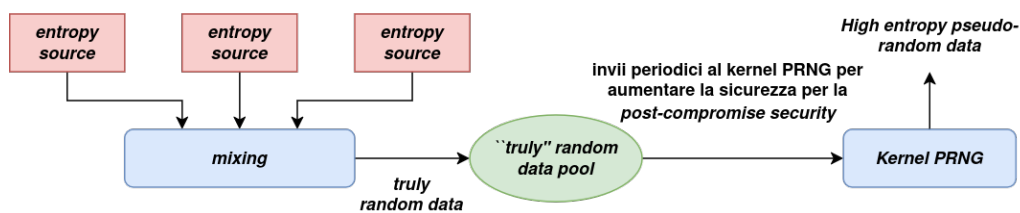
password (N), il calcolo dei bit di entropia sarà: $N \times \log_2(10^n)$. Nel caso di **password** il reale livello di sicurezza è molto più basso in quanto il campionamento dei simboli non è sempre uniforme all'interno del loro dominio o indipendentemente l'uno dall'altro. Infatti difficilmente la mente umana è capace di ricordarsi dati completamente random e quindi la sicurezza di una password di dimensione N è molto inferiore alla precedente stima.

Considerando applicativi su cui è necessario far girare algoritmi di crittografia, e considerando che tali applicativi sono software, per ottenere un dato random “buono” non possiamo affidarci ad un computer in quanto sono intrinsecamente **deterministici** e **predicibili** è quindi necessario fare affidamento su **fonti esterne** che sono associate a dati intrinsecamente randomici o eventi fisici imprevedibili (camere, input dell'utente, *interrupt hardware*). Esistono degli hardware dedicati, chiamati: **True Random Number Generator - TRNG** che vengono alimentati da eventi fisici (*thermal noise*) che vengono quindi utilizzati come fonti esterne di bit random (molto utili per server e sistemi embedded), queste sorgenti esterne vengono spesso identificate come **noise source**

Le **sorgenti di entropia** vengono prima processate dal sistema operativo prima di essere rese disponibili ad altri servizi. Questo permette di ottenere fonti di entropia miste:

- aumento di “fiducia” nella qualità dei dati casuali.
- mitigare potenziali rischi di attacchi alle fonti.
- ridurre i rischi relativi alle fonti di **backdoor**

Inoltre permette anche di espandere i “veri” dati casuali per aumentare la disponibilità di dati ad alta entropia.



Il lavoro del **Kernel PRNG** è simile a quello di uno *stream cipher* ma il suo schematico è leggermente differente in quanto vi è una modellazione dell'avversario diversa. Che permette di aumentare la sicurezza attraverso lo sviuppo di mitigazioni contro a **compromissioni non permanenti**:

- **forward secrecy**: se l'avversario compromette il sistema in un istante di tempo t_i non dovrà riuscire ad ottenere informazioni su istanti di tempo t_k con $k < i$, per ottenere questa garanzia si possono utilizzare delle funzioni **one-way** - ad esempio le *hash function*.

- **post-compromise security**: è duale alla *forward secrecy* ovvero che se l'avversario compromette il sistema in un istante t_i non dovrà riuscire ad ottenere stati futuri in maniera deterministica.

Queste due proprietà (**Security Guarantees**) vanno a descrivere un attaccante che non deve essere capace di creare persistenza nel sistema.

Kernel vs. User space, in passato molte librerie crittografiche non si fidavano della routine del sistema operativo per generare numeri casuali sicuri, infatti si credeva che il **kernel** avesse una **PRNG** debole venne quindi adottata una soluzione, ovvero le librerie crittografiche **implementavano** un priorio *PRNG custom* in **user space** - chiamato anche **userland PRNG**, ma veniva spesso sbagliata l'implementazione e questi dati avere un entropia molto bassa. Un esempio famoso fu nel 2008 che debian, dopo una modifica al codice sorgente rese prevedibile la generazione della chiave private ssh. Oggigiorno ci si affida maggiormente al sistema operativo e alle API per la generazione di valori randomici.

Un altro problema molto importante era il **reusing at fork** infatti se non specificato, nel momento in cui veniva eseguita una **fork** e successivamente *spawnato* un processo figlio se non veniva aggiornato il pool di entropia, sia il padre che il/i figli avrebbero generato la stessa sequenza di valore *pseudo-random*.

Virtual Machine Cloning: clonare una macchina virtuale poteva causare problemi al *kernel PRNG* simili all'utilizzo di una **fork** questo comportava:

- multi-VMs accese nello stesso istante generavano gli stessi pseudo-random data.
- la soluzione fu quella di re-inizializzare il pool durante l'avvio della VMs

Low entropy in early boot phases and embedded devices nelle fasi iniziali di boot, soprattutto nei dispositivi embedded, il pool potrebbe essere molto basso, se non inesistente questo comportava l'arresto o la generazione di pseudo-random deboli a seconda della strategia adottata - bloccante o attesa - la soluzione fu quella di adottare una **TRNG Hardware**

Capitolo 7

Derived Schema of Symmetric Crypto

7.1 SHA3 Derived Schemes

Abbiamo detto che una funzione hash ritorna come output una quantità fissa di bytes. Se quindi ci dovesse essere la necessità di avere **meno dati**, allora “basterebbe” **troncare** il *digest*, nel caso opposto invece possiamo ri-progettare un algoritmo simile a **CTR** e invocarlo più volte, il che, però, porterebbe molto più sforzo da parte degli sviluppatori e rischierebbe di introdurre delle vulnerabilità nel codice.

È stato quindi introdotto il *eXtensible Output Function - XOF* ovvero una *variable-length hash function*.

cSHAKE: è una variante di SHA3 e accetta i seguenti input addizionali:

1. *output length*: definisce la dimensione del digest.
2. *custom string*: specializza l'esecuzione della funzione *hash* per un certo **contesto**. È molto utile per *domain (contest) separation*.

cSHAKE128 ha 128bit di sicurezza ed è una variante di SHA3-256.

cSHAKE256 ha 256bit di sicurezza ed è una variante di SHA3-512.

TupleHash: è una funzione *hash* ha delle interfacce con un astrazione *high-level*, infatti permette di accettare una **lista di valori** e una *custom string* come input. È un *wrap* di **cSHAKE** che permette di supportare delle liste come valore di input, alle quali viene prima applicata una codifica in modo non ambiguo assegnando ad ogni elemento un'unica sequenza di bit - bitstring - poi verrà passata a cSHAKE come input. La *custom string* ha un valore di default al quale viene concatenato quello dell'utente: *TupleHash*. È utilizzata per evitare ambiguità di concatenazione. È deterministica e strutturate, utile in contesti crittografici dove serve un hash sicura su strutture dati complesse.

ParallelHash: è progettato per sfruttare ambienti paralleli (multi-core, GPU, ecc.) allo scopo di accelerare il calcolo dell'hash su grandi quantità di dati.

1. i dati vengono suddivisi in blocchi.
2. per ogni blocco viene calcolato il *digest* in parallelo.
3. i *digest* dei vari blocchi (***digest parziali***) vengono poi combinati per calcolare un *digest* finale sugli output precedenti, ottenendo il digest complessivo del messaggio.

Questo approccio ricorda un **Merkle Hash Tree - MHT** dove le foglie sono gli hash dei blocchi e il nodo radice è l'hash finale ottenuto dall'unione degli hash sottostanti.

N.B. esistono funzioni di hash - ad esempio **Blake2** - che hanno meccanismi interni che la rendono automaticamente parallelizzabili anche senza un architettura come quella di ParallelHash.

MAC for SHA3: **SHA3** è stato progettato e sviluppato per evitare *length extesion attack* è quindi possibile non utilizzare l'**HMAC** basato su SHA3, in quanto ci sarebbe dell'*overhead* inutile, ma sarebbe possibile avere un MAC che si basa su SHA3 come:

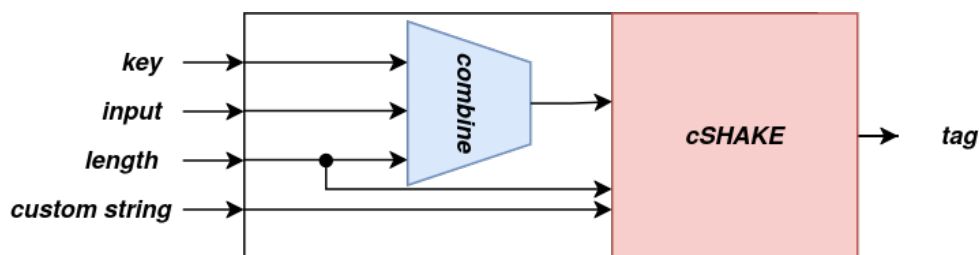
$$\text{tag} \leftarrow \text{SHA3}(k \parallel \text{message})$$

Bisogna solo fare attenzione a non utilizzare questo metodo con funzioni di hash che utilizzano come primitiva la costruzione **Merkle-Damgard** - per SHA2 utilizzare HMAC.

KMAC: è un **MAC** a dimensione variabile che si basa su **cSHAKE**. Come input **KMAC** richiede:

1. una **chiave** segreta k .
2. un **messaggio** m .
3. una **custom string** che verrà concatenato al valore di default **KMAC**.
4. una **lunghezza** desiderata dell'output.

Vengono combinati, in maniera sicura, la chiave, il messaggio e la *custom string*, il tutto viene passato in input a cSHAKE che ne calcola il **tag**.



7.2 Key Derivation Function

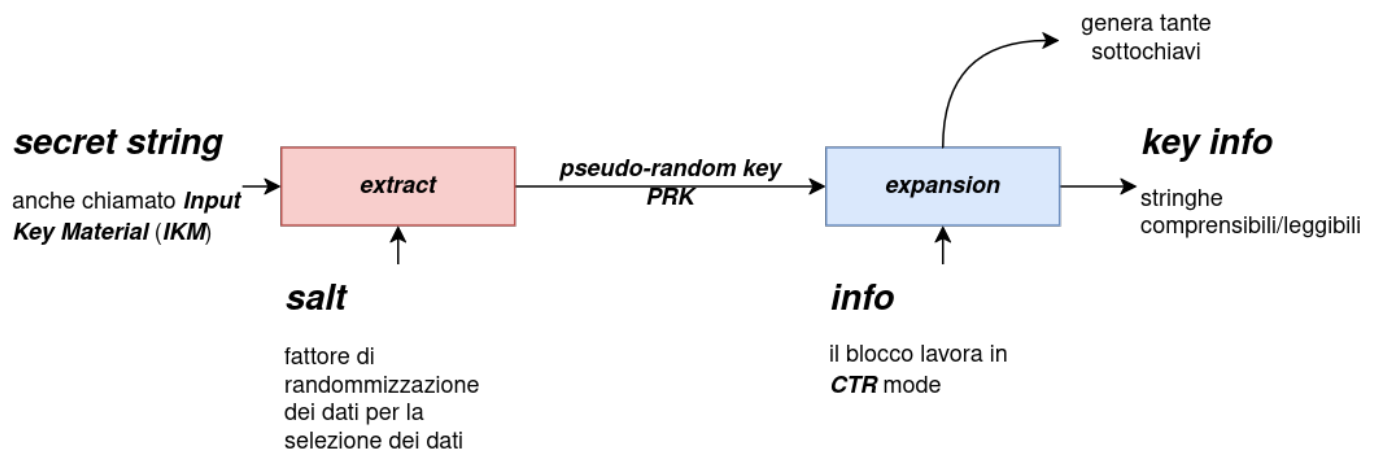
Molti schemi crittografici e protocolli spesso richiedono almeno due chiavi segrete non correlate, ma di contro si cerca di dover gestire un'unica chiave durante le procedure di *key management* e di *key distribution*.

Le **KDF** sono funzioni che permettono di generare più *pseudo-random keys* partendo da una singola chiave. Le **KDF** sono simili a delle **PRF**, infatti, prendono in input una singola stringa segreta e in maniera **deterministica** in output verrà generata una sequenza più lunga di dati pseudo-random che possono essere utilizzate come chiavi indipendenti e multiple. A differenza dei **PRF**, però, la **KDF** è progettata per generare una sequenza più corta e in più possono supportare altre funzionalità - ad esempio *input string*, *explicit salting* e lo *scoping*.

È possibile re-implementare una **KDF** utilizzando come primitive *block cipher*, *stream cipher*, MAC o *hash function*. Esistono, però, delle **KDF** stardandizzate dal NIST, la più popolare è chiamata **HKDF** e si basa su un **HMAC**.

Un'implementazione di alto livello di una **KDF** ha due funzioni interne:

1. **extraction**: che dato una *secret string* - possibilmente **non uniforme** e con un **alto livello di entropia** (comparabile con il *security level*) - genera **una** chiave segreta (pseudo-random) uniforme.
2. **expansion**: data una *secret key* uniforme genera una chiave **pseudorandom** più lunga - ad esempio generare una chiave da un'altra per *key scoping* o *key rotation*



Una **KDF** è un metodo sicuro per derivare una chiave segreta anche se l'input è una string non uniforme (uniforme = alta entropia), ma l'input deve avere alta entropia e, ad esempio, una password per definizione - segreto che può essere ricordato da un umano - raramente ha abbastanza entropia → **Password-Based Key Derivation** anche note come **Password Hashing**. Nel caso in cui si abbia già un segreto random binario possiamo *bypassare* il blocco di **extract** e passare

subito all'*expansion*. Una **KDF** può dare funzionalità aggiuntive tra cui: aggiungere **randomicità** e aggiungere *scoping information* (ad esempio per fare *domain separation*).

Gli input di una **HKDF** sono:

- **l'algoritmo**: la funzione di hash interna (SHA256).
- **string**: è la stringa segreta (con alta entropia).
- **length**: lunghezza desiderata per la chiave derivata - simile all'input della dimensione per XOF e KMAC.
- **info**: è una stringa che rappresenta il contesto per la separazione dei domini.
- **salt**: è una stringa, è opzionale, se viene utilizzata normalmente è una stringa randomica con lunghezza compatibile con la dimensione dell'algoritmo di hash utilizzato per l'**HMAC**

L'output è una **chiave pseudorandom** di dimensione *length*. L'**HKDF** include due sottofunzioni (che vengono spesso esposte dalle librerie crittografiche):

- **HKDF-Extract**: dato un **IKM** ritorna una chiave pseudo-random **PRK** di dimensione *alg.digest_size*
- **HKDF-Expand**: dato un **PRK** ritorna una chiave pseudo-random **key_{info}** di dimensione *length*

Esempio - PyCryptoDome

```

1  # from source code
2  def HKDF(master, key_len, salt, hashmod, num_keys=1, context):
3      output_len = key_len * num_keys
4      if output_len > (255 * hashmod.digest_size):
5          raise ValueError("Too much secret data to derive")
6      if not salt:
7          salt = b'\x00' * hashmod.digest_size
8      if context is None: context = b""
9      prk = _HKDF_extract(salt, master, hashmod)
10     okm = _HKDF_expand(prk, context, output_len, hashmod)
11     if num_keys == 1:
12         return okm[:key_len]
13     kol = [okm[idx:idx + key_len]
14             for idx in iter_range(0, output_len, key_len)]
15     return list(kol[:num_keys])

```

HKDF-Extract: trasforma una *key* ad alta entropia in un segreto con distribuzione di entropia uniforme di dimensione `hashmod.digest_size`, può non essere invocata nel caso in cui si abbia già una chiave segreta con distribuzione uniforme e possiamo soltanto fare il procedimento di espansione.

Chiave ad Alta Entropia: significa che la chiave è imprevedibile, contiene molta “casualità”. Per esempio, una stringa di 256 bit generata da un generatore casuale crittograficamente sicuro. Tuttavia, non è detto che sia uniforme: alcuni bit potrebbero avere distribuzioni sbilanciate (es. più 1 che 0).

Chiave Uniforme: ogni possibile valore è **equiprobabile**. Questo è importante per alcune operazioni crittografiche, come l’uso in HMAC, perché riduce il rischio di bias che può compromettere la sicurezza.

Normalmente il **salt** può essere utile per integrare l’**HKDF** in certi protocolli ed è necessario che sia una stringa uniforme, verrà utilizzata come chiave dell’**HMAC**.

Esempio - PyCryptoDome

```

1  # from source code
2  def _HKDF_extract(salt, ikm, hashmod):
3      # HMAC.new(key, msg=b"", digestmod=None):
4      prk = HMAC.new(salt, ikm, digestmod=hashmod).digest()
5      return prk

```

Listing 7.1: `_HKDF_extract` da PyCryptoDome

HKDF-Expend: dopo che si è ottenuta la *pseudorandom key* **PRK** bisogna calcolare tanti **HMAC** quanti ne servono per ottenere la lunghezza finale di `key_info`, che è pari **length** moltiplicata per il numero di chiavi che si vuole generare. Si può notare in 7.2 che l’**HMAC** viene richiamata in maniera iterativa similmente alla *counter mode*, dove nel caso dell’**HKDF** il counter è di dimensione 1 byte, è quindi necessario impostare una lunghezza massima per evitare ripetizioni della generazione della chiave che è pari alla **dimensione** della funzione di **hash** scelta moltiplicata per le possibili combinazioni di un byte: 255.

$$\begin{cases} t_0 = \text{HMAC}(\text{PRK}, "" \parallel \text{info} \parallel 0, \text{hashmod}) \\ t_1 = \text{HMAC}(\text{PRK}, t_0 \parallel \text{info} \parallel 1, \text{hashmod}) \\ \vdots \\ t_{i+1} = \text{HMAC}(\text{PRK}, t_i \parallel \text{info} \parallel i, \text{hashmod}) \end{cases}$$

Esempio - PyCryptoDome

```
1  # from source code
2  def _HKDF_expand(prk, info, L, hashmod):
3      t, n, tlen = [b""], 1, 0
4      while tlen < L:
5          hmac = HMAC.new(prk, t[-1] + info + struct.pack('B', n),
6                          digestmod=hashmod)
7          t.append(hmac.digest())
8          tlen, n = tlen + hashmod.digest_size, n + 1
9      okm = b"".join(t)
10     return okm[:L]
```

Listing 7.2: `_HKDF_expand` da PyCryptoDome

7.3 Hash Table Flooding & small tag MAC's

Hash Table (Hash Map)

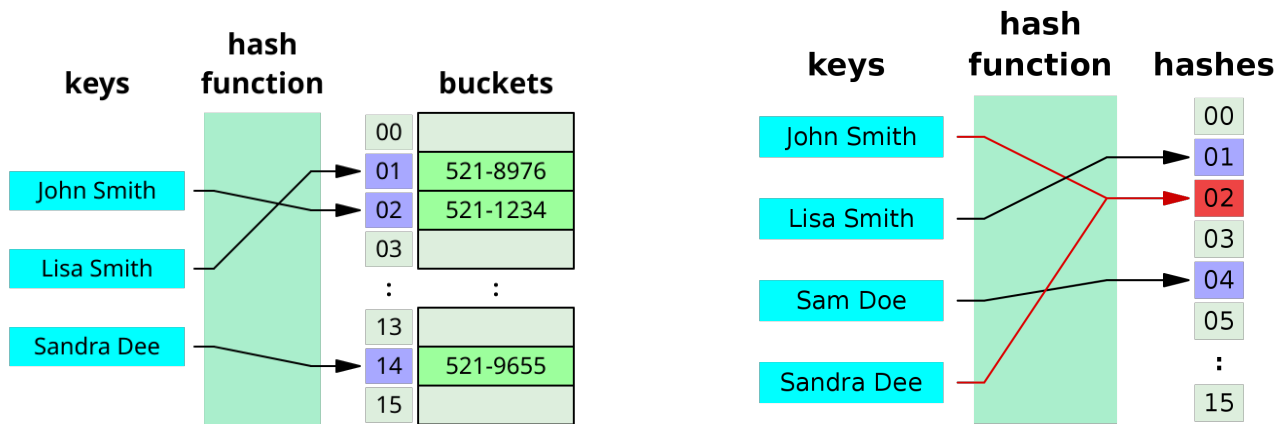
L'insieme delle possibili chiavi è rappresentato dall'**insieme universo** \mathcal{U} di dimensione u . Ad esempio considerando che una chiave è il “nome e cognome” di una persona, l'insieme universo sarà formato da tutte le combinazioni dell'insieme dei nomi \mathcal{N} con quello dei cognomi \mathcal{C} :

$$\mathcal{N} \times \mathcal{C} \mapsto \mathcal{U}$$

Ma noi avremo bisogno di memorizzare le chiavi in una struttura dati limitata ad esempio in un vettore $T[0, \dots, m-1]$ di dimensione m , detta **tabella di hash**. Definiamo la funzione *hash* è definita come:

$$h : \mathcal{U} \mapsto \{0, 1, \dots, m-1\}$$

La coppia chiave valore $\langle k, v \rangle$ viene memorizzata in un vettore nella posizione $h(k)$



L'insieme delle chiavi è potenzialmente infinito, mentre la dimensione della tabella hash non vogliamo che sia infinito - non ho abbastanza memoria. È quindi possibili che ci siano due o più chiavi del dizionario che abbiamo la stessa immagine all'interno della **tabella di hash**, in questo caso diremo che è avvenuta una collisione. Idealmente vorremo evitarle.

Andiamo ora ad analizzare la **funzione hash** - una funzione hash h si dice **perfetta** se è **iniettiva**, ovvero se non da origine a collisioni: $\forall k_1, k_2 \in \mathcal{U} \mid k_1 \neq k_2 \Rightarrow h(k_1) \neq h(k_2)$ -, differenziando in base alla cardinalità dell'insieme \mathcal{U} :

- **tabelle ad accesso diretto:** nel caso in cui $\mathcal{U} \subset \mathbb{Z}^+$ andremo ad utilizzare quella che viene definita **funzione hash identità** $h(k) = k$. Ad esempio se dobbiamo mappare l'insieme dei pokemon di kanto, numerati da 1 a 151.

Cerchiamo una funzione che **distribuiscono uniformemente** le chiavi negli indici $[0, \dots, m-1]$ della tabella hash.

Assunzione: le chiavi possono essere tradotte in valori numerici, anche interpretando la loro rappresentazione in memoria come un numero (python: ord, bin, int). Una delle possibili funzioni possibile è il **metodo della moltiplicazione di knuth**:

Teoria

m qualsiasi, meglio se potenza di 2

C costante reale, $0 < C < 1$

sia $i = \text{int}(k)$

$$H(k) = \lfloor m(C \cdot i - \lfloor C \cdot i \rfloor) \rfloor$$

Implementazione

scelto un valore $m = 2^p$

sia w - ad esempio 64bit - la dimensione in bit della parola di memoria: $i, m \leq 2^w$

sia $s = \lfloor C \cdot 2^w \rfloor$

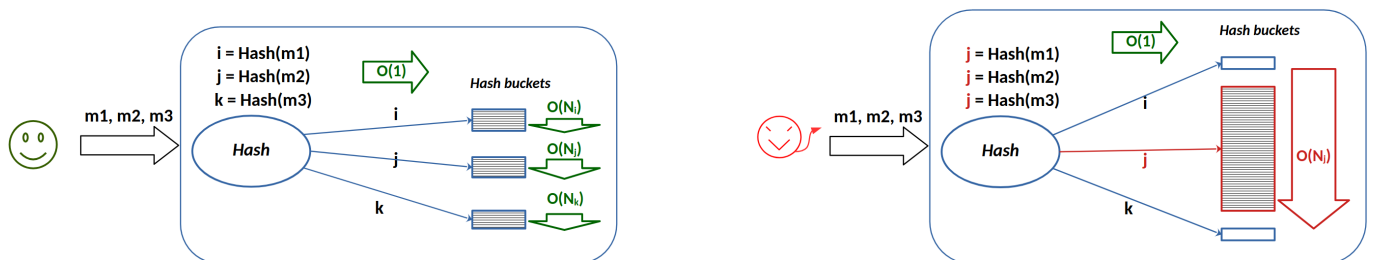
$i \cdot s$ può essere riscritto come $r_1 \cdot 2^w + r_0$, dove r_1 contiene la parte intera di iC e r_0 contiene la parte frazionaria di iC

si restituiscano i p -bit più significativi di r_0

Gestione collisione: siccome la funzione di **hash** non è totalmente iniettiva è possibile che avvengano delle collisioni, ovvero che due chiavi abbiano come immagine lo stesso valore, è possibile gestire queste casistiche in due modi differenti:

- **liste di trabocco** (memorizzazione esterna) - anche noto come **chaining**: le chiavi con lo stesso valore hash h vengono memorizzate in una lista monodirezionale o in vettori dinamici.
- **indirizzamento aperto** (memorizzazione interna)

Denial of Service via Hash Collision: consideriamo una tabella hash, che sfrutta una tabella hash per distribuire i dati su più *bucket*, avremo una ricerca molto efficiente (costo costante), ma solamente finché i dati sono distribuiti uniformemente. Si analizzi il caso in cui sia un **avversario** a scegliere il messaggio e supponiamo sia in grado di generare in maniera efficiente un numero elevato di messaggi m tale che $M = \{m \mid j = \text{Hash}(m)\}$



Anche nel caso in cui ci fosse un funzione hash crittografica, il costo dell'attacco sarebbe determinato dal **birthday attack**, in più normalmente i *bucket* hanno dimensione limitata, ad esempio 2^{16} , mentre se utilizzasse una funzione hash crittografica i possibili valori, ad esempio di SHA256, sarebbe 2^{256} il che comunque non garantirebbe *collision resistance* su un bucket. Se la struttura dati è esposta ad un attaccante potrebbe causare un **Denial of Service** - un esempio possono essere i dizionari che vengono utilizzati per il *routing* delle richieste web. Per

mitigare questa casistica è possibile utilizzare un **Hash-based MAC** - HMAC - dove il segreto viene scelto a *runtime*. **vantaggi**: per un avversario è praticamente impossibile trovare messaggi della tipologia definita in precedenza, **svantaggi**: le funzioni MACs sono molto più costose.

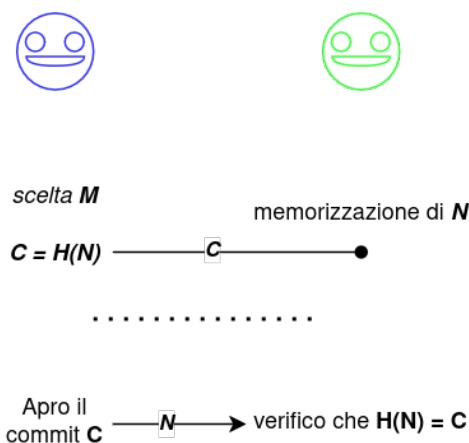
SipHash è una funzione MAC specializzata per *tag* più piccoli - 64 bit. Pensata appositamente per casi d'uso del genere, ma non sicuro per la maggior parte dei contesti di comunicazione. La chiave non deve essere memorizzata, può essere cancellata nel momento in cui la tabella viene cancellata.

7.4 Commitment Schemes

Uno **schema di commitment** viene utilizzato per impedire ad un'entità di negare, in seguito, una certa scelta; possono essere utilizzati in maniera indipendenti o all'interno di altri protocolli. Un *commitment scheme* deve avere due proprietà:

1. **(obbligatorio) Binding**: il *commitment* è associato al valore scelto, non deve essere possibile fornire un valore diverso da quello iniziale che soddisfa la verifica.
2. **(opzionale) Hiding**: il *commitment* deve nascondere il valore iniziale, deve essere garantita confidenzialità su ogni valore iniziale.

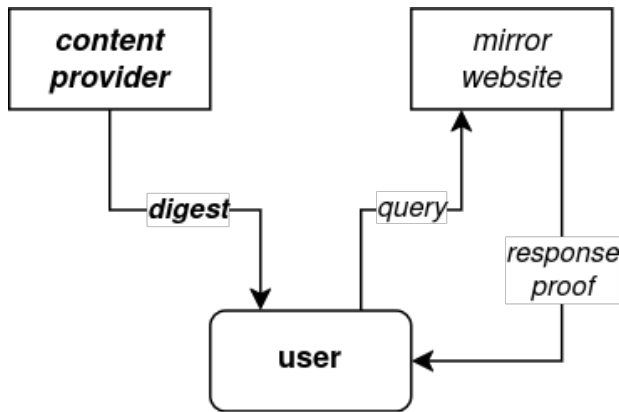
Gli schemi di *commitment* vengono utilizzati ad esempio: protocolli *challenge-response* per l'autenticazione, alcuni schemi di firma digitale o autenticazione (**OAuth PKCE**).



In questo caso viene utilizzata come funzione di commit una funzione **hash**, se questa rispetta la proprietà di *collision resistance*, è possibile utilizzare anche una funzione **MAC**.

7.5 Authenticated Data Structure

Un'altra applicazione popolare per le funzioni hash sono gli schenari di *data outsourcing* come precedentemente discusso per i *web mirrors*. Vengono chiamati *authenticated data structure* che è una versione estesa delle *strutture dati* - come ad esempio *linked list*, *binary tree* - alle quali vengono applicati i paradigmi di *integrità dei dati* mentre continuano a supportare *efficient queries*.



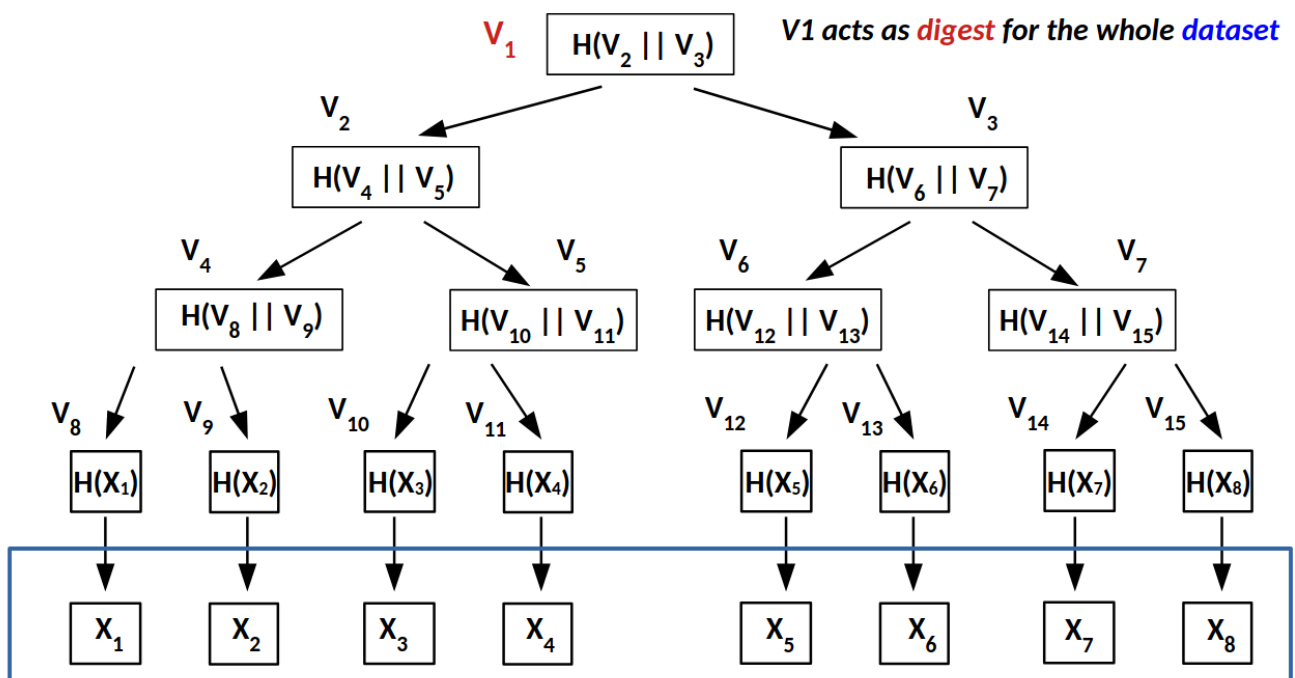
1. download del *digest*
2. richiesta del dato
3. risposta e prova
4. l'utente accetta il dato se passa il controllo `verify(response, proof, digest)`

Analizziamo ora l'esempio di autenticazione di *multiple data*, consideriamo un *dataset*

$\mathcal{X} = \{x_0, x_1, \dots, x_n\}$, una possibilità per autenticare sarebbe quella di calcolarsi l'*hash* di \mathcal{X} ma questo vorrebbe dire calcolarsi $t = \text{HASH}(x_0 || x_1 || \dots || x_n)$, in questo modo **verificare**

l'**integrità** di soltanto un dato avrebbe costo lineare in quanto bisognerebbe trasmettere tutti gli altri valori. **Soluzione: Merkle Hash Tree - MHT**, dove gli hash vengono concatenati tramite un **albero binario**, le foglie sono gli hash dei valori di tutti i dati appartenenti al *dataset*.

L'inserimento e la validazione di un nuovo elemento costa $\mathcal{O}(\log n)$.



Per validare - ad esempio - x_3 , invece che dover inviare tutto il *dataset* basterà inviare $x_3, v_{11}, v_4, v_3, v_1$ in questo modo si potrà calcolare $v_1 \stackrel{?}{=} h(h(h(x_3) \parallel v_{11}) \parallel v_4) \parallel v_3$, in questo modo si riesce a validare la appartenenza di un certo elemento al *dataset*.

A differenza della *membership proof* di un elemento che è direttamente supportata in un **MHT**, la *non-membership proof* non è supportata direttamente ma esistono modi per riuscirsela a ricavare:

- **MHT ordinato**: se gli elementi nei nodi della foglia sono ordinati si può provare che un elemento non è presente mostrando due **foglie adiacenti** x_3, x_4 tali che $x_3 < x < x_4$ e fornendo le *membership proof* delle due foglie.
- **Sparse Merkle Tree**: solitamente in un **SMT**, ogni possibile valore hash occupa una posizione predeterminata (tipicamente in base all'elemento), per dimostrare che x **non esiste** si fornisce il cammino nel **SMT** dove dovrebbe trovarsi x e si dimostra che è vuoto o contiene un elemento $x' \neq x$.

7.6 Synthetic Initialization Vectors

SIV denota una branchia di *mode of operation* per gli schemi di crittografia autenticata che ha come scopo mitigare il danno che potrebbe fare il riuso di *nonce* o dell'*IV*. Il riuso di *nonce* o *IV* renderebbe lo schema **deterministico**, ma non perderebbe di sicurezza grazie a proprietà matematiche - nessun *leak* di informazioni - a differenza di altri *stream cipher* polinomiali tipo **GHASH** o **Poly1305**. “*Synthetic*” in questo contesto significa che il *nonce* o *IV* forniti dall'utente non vengono utilizzati direttamente per cifrare, ma utilizzati per derivarne uno attraverso una funziona crittografica. In questo modo anche fornendo sempre lo stesso *nonce* o *IV* se il contenuto del messaggio cambia allora anche il **SIV** effettivo per cifrare cambierà.

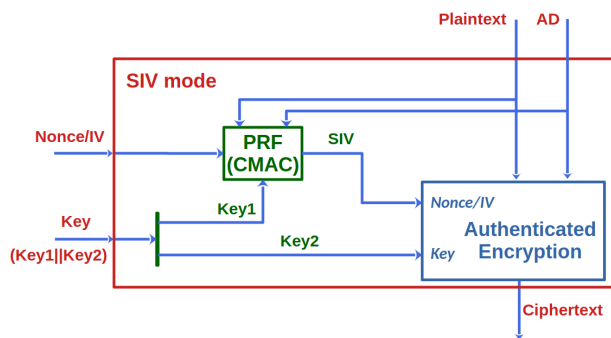


Figura 7.1: **AES-SIV**

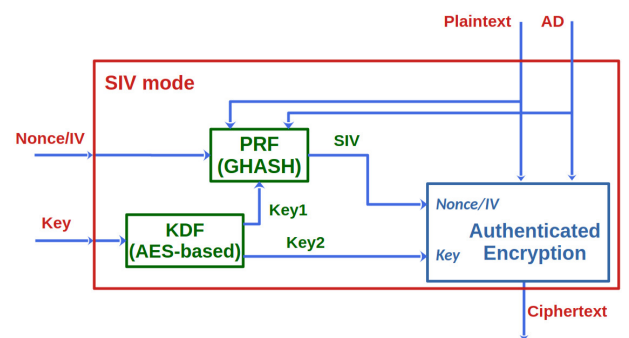


Figura 7.2: **AES-GCM-SIV**

Il maggior svantaggio di uno schema **SIV** è che la cifrazione del messaggio non può essere *streamable*, infatti è necessario leggere il *plaintext* deve essere letto due volte, la prima per generare

il **SIV** (che dipende da tutto il testo in chiaro) e la seconda per l'algoritmo di *encryption*, può essere un problema in caso di *plaintext* molto grandi. Gli schemi **SIV** possono essere utilizzati per implementare *Deterministic Encryption & DEAE*.

7.7 (extra) Format-Preserving Encryption