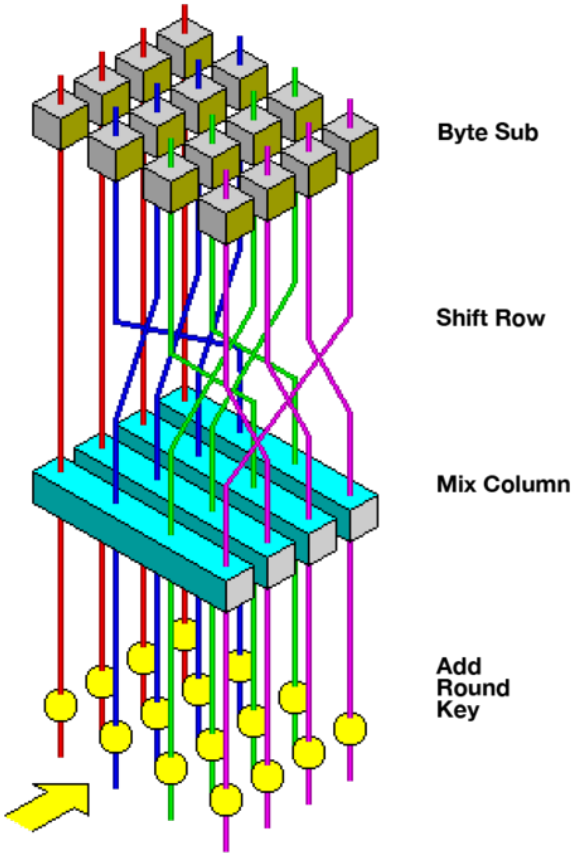
Advanced Encryption Standard
AES

Luca Rengo

v1.0.0
Maggio | Aprile
2022 | 2023



Indice

Capitolo 1

Storia di AES

Introduzione

AES (***A**dvanced **E**ncryption **S**tandard*) è un cifrario a blocchi simmetrico, inventato da due matematici belghi, Vincent Rijmen e Joan Daemen, da cui viene il nome *Rijndael*, nel 1998 per sostituire il precedente standard: DES (***D**ata **E**ncryption **S**tandard*).

Breve storia di AES

DES era divenuto lo standard dopo un bando dell'NBS (***N**ational **B**ureau of **S**tandards*), oggi NIST *(**N**ational **I**nstitute for **S**ecurity and **T**echnology)* per trovare un buon e sicuro algoritmo per proteggere le comunicazioni private dei cittadini americani.

Venne così proposto un algoritmo chiamato *Lucifer*, sviluppato dall'*IBM* che dopo esser stato modificato dall'NSA (***N**ational **S**ecurity **A**gency*), riducendone la grandezza della chiave da 128 a 56 bits e rettificandone le funzioni contenute nell'S-box, venne designato come *Data Encryption Standard* (**DES**).

DES regnò per 20 anni, venne studiato in lungo e in largo dagli accademici e criptoanalisti di tutto il mondo, grazie a ciò, ci fu finalmente per la prima volta un cifrario certificato che tutti potevano studiare: nacque così il moderno campo della crittografia.

Negli anni, molti sfidarono DES e dopo diverse battaglie fu finalmente sconfitto.

L'unico modo per ovviare a questi attacchi era quello di combinare des tre volte, formando il 3DES (*Triplo DES*). Il problema di questo però era la sua lentezza.

Per questo, nel 1997, il NIST indisse un nuovo bando per cercare un nuovo algoritmo di cifratura, forte come il triplo-DES, ma veloce e flessibile.

AES vs DES

	DES	**AES**
Date	1976	1999
Block size	64	128
Key length	56	128, 192, 256
Number of rounds	16	9,11,13
Encryption primitives	Substitution, permutation	Substitution, shift, bit mixing
Cryptographic primitives	Confusion, diffusion	Confusion, diffusion
Design	Open	Open
Design rationale	Closed	Open
Selection process	Secret	Secret, but accept open public comment
Source	IBM, enhanced by NSA	Independent cryptographers

Figura 1.1: AES vs DES

Vari algoritmi competerono: Serpent, Twofish, MARS, RC6, ma alla fine spuntò Rijndael per la sua semplicità e velocità.

AES vs Rijndeal

AES è un'implentazione di Rijndael, divenuto l'algoritmo di cifratura standard del governo degli Stati Uniti d'America. Una differenza tra i due è che AES utilizza blocchi di dati da 128 bits, mentre Rijndael permette oltre a blocchi da 128, anche blocchi da 192 e 256 bits.

Sia AES che Rijndael permettono una grandezza della chiave di 128, 192 o 256 bits, da cui ne ricaviamo il numero di rounds: 10, 12 o 14 rispettivamente.

Cifratura simmetrica vs asimmetrica

Nella cifratura simmetrica viene usata una chiave sia per la cifratura che per la decifratura di un messaggio.

La cifratura asimmetrica è basata sul concetto di chiave pubblica e chiave privata. Vengono, quindi usate due chiavi sia per la cifratura che per la decifratura. Usiamo la chiave pubblica per cifrare il messagio e la chiave privata per decifrarlo.

Ulteriori differenze:

Simmetrico	**Asimmetrico**
Richiede una sola chiave sia per la cifratura che la decifratura.	Richiede due chiavi, una pubblica e una privata, una per cifrare e una per decifrare.
Lo spazio del testo cifrato è lo stesso o più piccolo del messaggio originale.	Lo spazio del testo cifrato è lo stesso o più grande del messaggio originale.
Il processo di cifratura è molto veloce.	Il processo di cifratura è molto lento.
È usato quando un grosso ammontare di dati deve essere trasferito.	È usato per trasferire piccole quantità di dati.
Fornisce solamente la confidenzialità.	Fornisce confidenzialità, autenticità e non ripudio.
La chiave usata è di solito di lunghezza 128 o 256 bits.	La lunghezza della chiave è di 2048 o più bits.
L'utilizzo delle risorse è basso.	L'utilizzo di risorse è alto.
Esempi: DES, 3DES, AES, RC4	Esempi: DSA, RSA, Diffie-Hellman, ECC, El Gamal

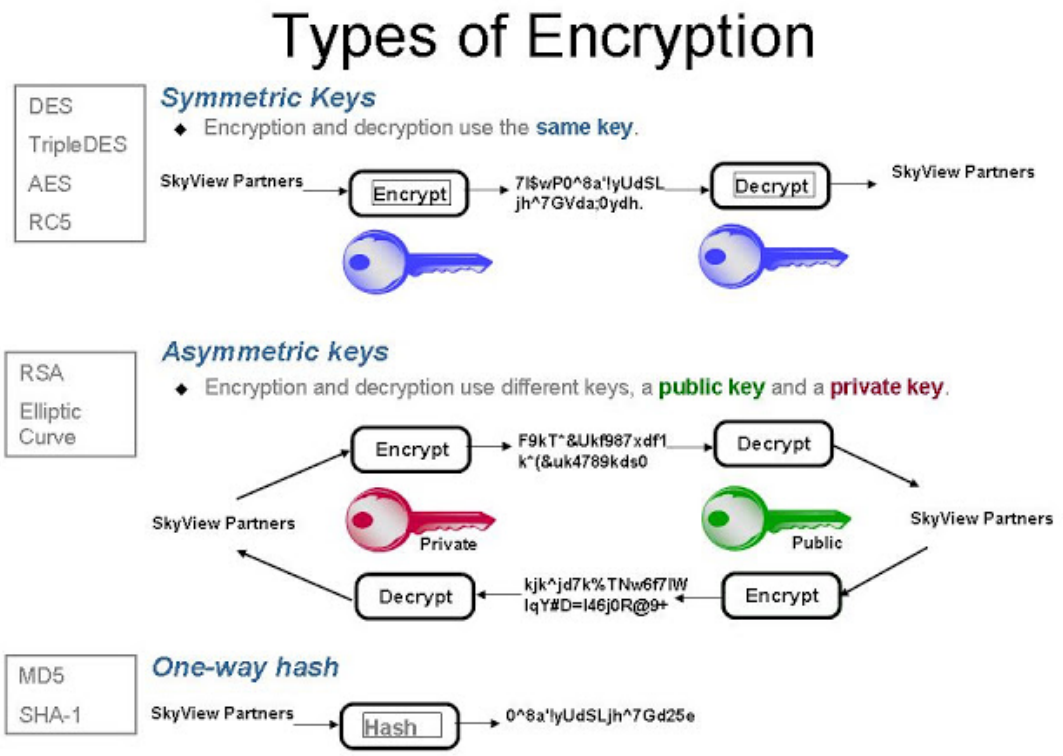


Figura 1.2: Tipi di cifratura

AES è di tipo simmetrico, quindi useremo la stessa chiave sia per cifrare il nostro messaggio sia per decifrarlo.

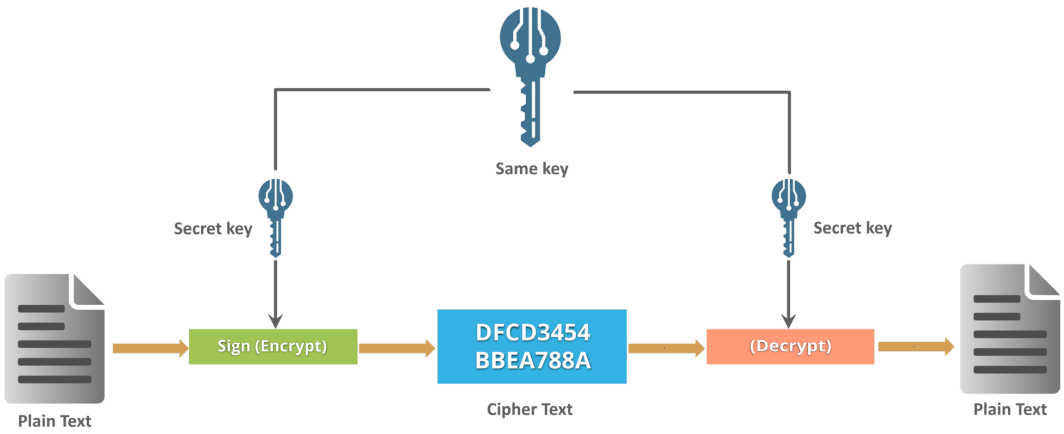


Figura 1.3: Cifratura a chiave simmetrica

Stream vs Block Ciphers

Cifrario a flusso: Il testo in chiaro viene separato in [singoli] bits che vengono poi singolarmente cifrati.

Cifrario a blocchi: Un cifrario a blocchi prende un messaggio in chiaro e lo suddivide in blocchi di bytes.

AES è un cifrario a blocchi che prende un blocco di grandezza fissa di 16 bytes (o 4 words).

Principio di Kerchoffs

L'olandese Auguste Kerckhoffs elaborò, nel 1883, due articoli riguardanti la crittografia. Nel primo enunciò sei principi, di cui il più famoso (il numero 2) sostiene che la sicurezza di un sistema crittografico non deve risiedere nella segretezza del suo algoritmo, ma soltanto nel mantenere la sua chiave segreta.

I sei principi di Kerckhoffs:

1. Il sistema deve essere praticamente, se non matematicamente, indecifrabile.
2. Non dovrebbe richiedere segretezza e non dovrebbe essere un problema se cade nelle mani nemiche.
3. Deve essere possibile comunicare e ricordare la chiave, senza utilizzare note scritte e i corrispondenti devono essere in grado di modificarla a piacimento.
4. ve essere applicabile a comunicazioni attraverso il telegrafo.
5. Deve essere portabile, e non dovrebbe richiedere diverse persone per poterlo utilizzare.
6. Il sistema dovrebbe essere facile da usare e non dovrebbe essere stressante o richiedere ai propri operatori di seguire una lunga lista di procedure.

Capitolo 2

L'Algoritmo

Introduzione

In questo capitolo, tratteremo il funzionamento dell'algoritmo di AES con una panoramica dall'alto, per poi affrontare nel prossimo capitolo, più in dettaglio, la sua matematica.

I tre concetti dietro la Crittografia

Alla base della crittografia, ci sono due importanti proprietà dei cifrari a chiave simmetrica, elaborati dal padre della teoria dell'informazione, Claude Elwood Shannon, ovvero: *diffusione* e *confusione*.

Il principio della *confusione* vela la connessione tra il messaggio originale e il testo cifrato.

La proprietà di *diffusione*, invece, riguarda lo scombussolamento della posizione dei caratteri del messaggio.

Un altro importante concetto è quello della *segretezza della chiave*, ovvero che l'algoritmo alla base del cifrario è conosciuto, è pubblico, ma la sola conoscenza di

questo non è sufficiente per poter conseguire l'accesso alle informazioni, perché per poter attingerle sarà necessario conoscere la chiave segreta.

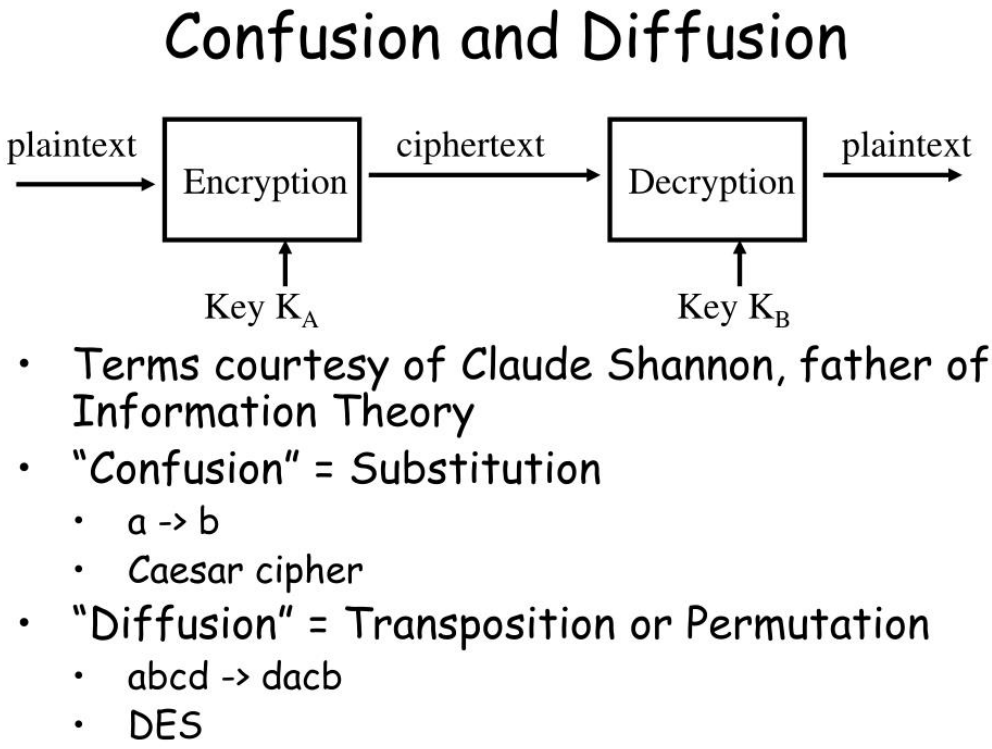


Figura 2.1: Confusione e Diffusione

Una panoramica sull'Algoritmo

I dati di input vengono caricati in una matrice 4x4, anche chiamata *state matrix* (matrice di stato), dove ogni cella rappresenta 1 byte di informazione e su queste compiamo diverse operazioni: *sub-bytes* (sostituzione dei bytes), *shift rows* (spostamento delle righe), *mix columns* (mescolamento delle colonne), *add round key* (aggiunta della chiave del round) per un numero di volte, di rounds pari alla grandezza della chiave.

	Key Length (*Nk words*)	Block Size (*Nb words*)	Number of Rounds (*Nr*)
AES-128	4	4	10
AES-192	6	4	12
AES-256	8	4	14

Key-Block-Round Combinations

Figura 2.2: Key Size e Numero di Rounds

Nel primo round svolgiamo uno XOR tra il messaggio d'input e la chiave segreta.

Lo *XOR* (*EXclusive-OR*) bit-a-bit è un'operazione di macsheratura dei bit, dove se i due bit di input sono diversi, allora produrrà un 1 in uscita, altrimenti se sono uguali, uno zero.

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Figura 2.3: Tabella della verità dello XOR

Perché lo XOR è usato in crittografia?

- Lo XOR non *leaka* informazioni sull'input originale.
- Lo XOR è una *involutory function* (funzione involutoria) tale che se la applichi due volte riottieni il testo originale.
- L'output dello XOR dipende da entrambi gli input. Non è così per le altre operazioni (AND, OR, NOT, ecc.).

Key Expansion | Key Schedule

Per poter elaborare i rounds, l'algoritmo ha bisogno di molte chiavi, una per round, queste vengono tutte derivate dalla chiave iniziale.

Il procedimento per ricavarle è questo:

1. Sposta la prima cella dell'ultima colonna della precedente chiave in fondo alla colonna.

2. Ogni byte viene posto in una substitution box che lo mapperà in qualcos'altro.

3. Viene effettuato uno XOR tra la colonna e una *round constant* (costante di round) che è diversa per ogni round.

4. Infine viene realizzato uno XOR con la prima colonna della precedente chiave.

Per le altre colonne, vengono semplicemente eseguiti degli XOR con la stessa colonna della precedente chiave (eccetto per le chiavi a 256 bit che hanno un procedimento un po' più complicato).

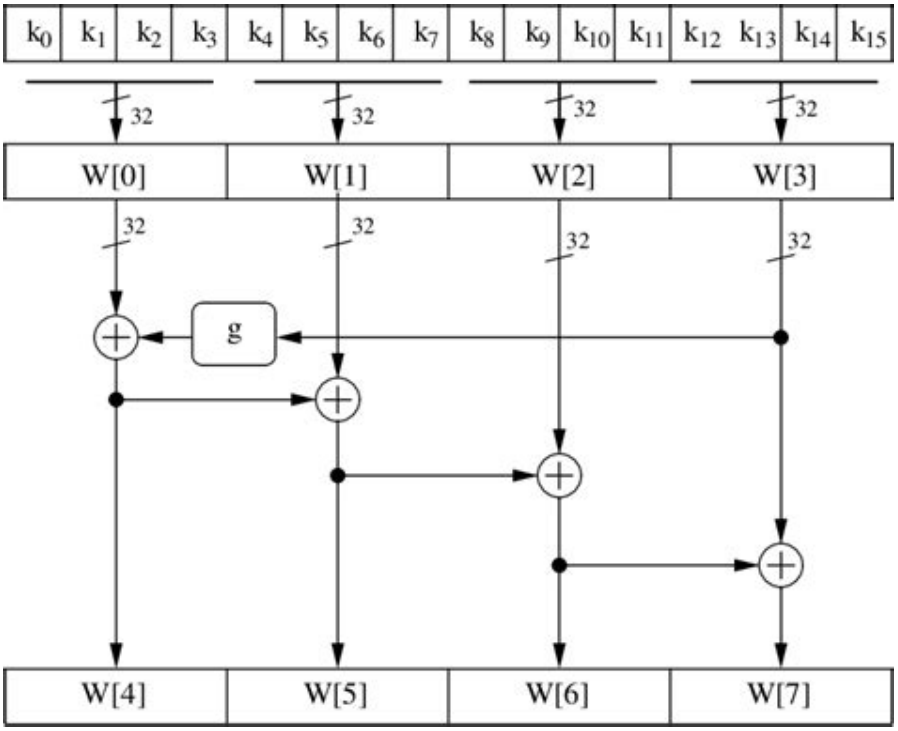


Figura 2.4: Key Expansion

I Rounds

Dopo aver ottenuto le chiavi, vengono compiuti i vari rounds.

Per ogni round, eseguiamo questi passaggi, tranne per l'ultimo dove non effettuiamo il passaggio delle *Mix Columns*, perché non aumenterebbe la sicurezza e semplicemente rallenterebbe:

Applichiamo il principio di *confusione* attraverso il passaggio *Sub-bytes*.

- *Sub-bytes*: Ogni byte viene mappato in un diverso byte attraverso una s-box. Questo step applica la proprietà di *confusione* di Shannon, perché oscura la relazione tra ogni byte.

Applichiamo la proprietà di *diffusione*:

- *Shift Rows*: La seconda riga della matrice viene spostata di 1 verso sinistra. La terza riga di 2 posizioni e la quarta di 3 (sempre verso sinistra).
- *Mix Columns*: Ogni bit delle colonne della matrice (di stato) vengono mischiate.

Applichiamo la proprietà di *segretezza della chiave*:

- *Add Round Key*: Viene applicata la chiave del prossimo round attraverso uno XOR.

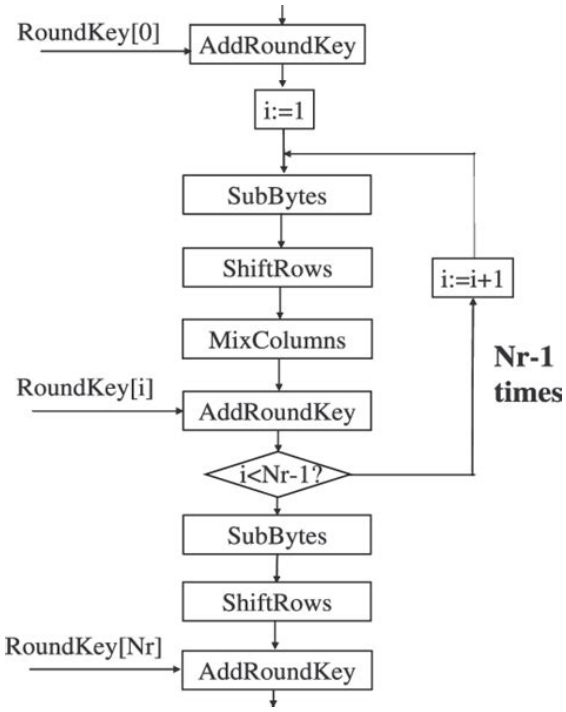


Figura 2.5: AES Rounds Flowchart

Più rounds aggiungiamo, più sicurezza, ma questo porterebbe ad un rallentamento dell'algoritmo e quindi delle performance. Per questo serve un compromesso tra sicurezza e prestazioni.

Quando AES era in sviluppo venne trovata una scorciatoia attraverso 6 rounds, per evitare ciò, sono stati aggiunti 4 rounds extra, come *margine di sicurezza*.

Add Round Key

Sub Bytes

Shift Rows

Mix Columns

Le modalità di AES

AES non può essere utilizzato così com'è, ma necessita di essere adoperato in combinazione a una modalità. Una modalità è un processo/sistema/procedimento per aumentare/incrementare/trasformare/avanzare l'efficacia di un algoritmo crittografico.

Di seguito, alcune delle modalità di AES:

- ECB (*Electronic Code Book*)
- CBC (*Cipher Block Chaining*)
- CFB (*Cipher FeedBack)*
- OFB (*Output FeedBack)*
- CTR (*Counter mode*)

Capitolo 3

La Matematica dietro AES

Introduzione

In questo capitolo tratteremo più a fondo la matematica che sta dietro questo algoritmo. Per poterlo fare avremo bisogno di introdurre alcune strutture matematiche come: i gruppi, gli anelli e i campi, dopodiché ci soffermeremo su uno specifico campo, quello di Galois, ne mostreremo le operazioni possibili e come ne usufruiremo nei passi trattati nel capitolo degli algoritmi.

Gruppi, Anelli e Campi

Prima di introdurre il campo di Galois, facciamo chiarezza tra i Gruppi, gli Anelli e i Campi.

Gruppo

Un gruppo è una struttura algebrica composta da un'insieme vuoto $\emptyset$ e un'operazione binaria (+ e $\cdot$) per l'elemento neutro e per l'elemento inverso.

Il gruppo deve soddisfare tre assiomi:

I. proprietà associativa: dati a, b, c; $(a \cdot b) \cdot c = a \cdot (b \cdot c)$

II. esistenza dell'elemento neutro: elemento neutro *e* tale che $a \cdot e = e \cdot a = a$

III. esistenza dell'inverso: esiste a', detto inverso, tale che a $\cdot a' = a' \cdot a = e$

Monoide

Il monoide è una struttura algebrica fornita dell'operazione binaria associativa e di un elemento neutro.

Gruppo abeliano

È un gruppo che gode della proprietà commutativa.

Anello

Un anello è una struttura algebrica, dove sono definite due operazioni: + e $\cdot$ (somma e prodotto) che rispetta le proprietà: commutativa, distributiva (della moltiplicazione rispetto all'addizione) e dove esiste un elemento neutro nell'addizione ($a + 0 = 0 + a = a$) e nella moltiplicazione ($a \cdot 1 = 1 \cdot a = a$).

Quindi, un anello è:

- due operazioni binarie + e $\cdot$ che rispettano i 3 assiomi degli anelli:
  - gruppo abeliano rispetto all'addizione (proprietà commutativa).

- monoide rispetto alla moltiplicazione (proprietà associativa ed elemento neutro)
- moltiplicazione distributiva rispetto all'addizione.

Anello commutativo

È un tipo di anello in cui l'operazione moltiplicativa gode della proprietà commutativa.

Anello finito

Un anello finito è un anello che possiede un numero finito di elementi.

Campo

Un campo $\mathbb{K}$ è una struttura algebrica formata da un insieme non vuoto e due operazioni binarie + e · in cui:

- l'insieme e l'operazione + sono un gruppo abeliano con elemento neutro.
- $\mathbb{K} \setminus \{0\}$ e l'operazione · sono un gruppo abeliano con elemento neutro 1.
- la moltiplicazione è distributiva rispetto all'addizione.

Un campo può essere definito anche come un anello commutativo dove ogni elemento non nullo possiede un inverso. O anche come un corpo commutativo alla moltiplicazione.

Campi vs Anelli

La differenza fra queste due strutture è questa, ovvero:

Un anello è un gruppo abeliano rispetto alla prima operazione (+ addizione), ma non rispetto alla seconda, mentre, il campo è un gruppo abeliano rispetto a entrambe le operazioni.

Di seguito, alcuni esempi:

Esempio: L'anello dei numeri interi $(\mathbb{Z}, +, \cdot)$ è un gruppo abeliano rispetto all'addizione: 5 + (-5) = 0. Ma non è un gruppo abeliano rispetto alla moltiplicazione, perché non esiste un inverso di un numero intero n tale che $n \cdot n^{-1} = 1, 5 \cdot \frac{1}{5} \notin \mathbb{Z} = 1$

Esempio: Anello numeri frazionari è un campo: $(\mathbb{Q}, +, \cdot)$ è un gruppo abeliano rispetto all'addizione e alla moltiplicazione: $5 + (-5) = 0. \quad 5 \cdot \frac{1}{5} = 1$

Estensione di campi

Essa è uno studio di coppie di campi, l'uno contenuto nell'altro. Una tale coppia prende il nome di **estensione di campi**.

Se L è un campo e K è un campo contenuto in L tale che le operazioni di campo in K sono le stesse di quelle in L, diciamo che K è un sottocampo di L, che L è un'estensione di K e che L/K è un'*estensione di campi*.

Estensione algebrica

L'estensione algebrica, in algebra astratta, una estensione di campi L/K è detta algebrica se ogni elemento L è ottenibile come radice di un qualche polinomio a coefficienti in K.

Sia K un campo, una estensione è il dato di un altro campo L e di un omomorfismo iniettivo di K in L. Tramite l'omomorfismo K può essere visto come un sottocampo di L. L'estensione è generalmente indicata con la notazione L/K.

Gruppo di Galois

Il gruppo di Galois è un gruppo associato a un'estensione di campi, in particolare i gruppi associati a estensioni che sono di Galois.

Estensione di Galois

È un'estensione algebrica E/F che soddisfa le condizioni descritte qui sotto. Il senso è che un'estensione di Galois ha un gruppo di Galois e obbedisce al teorema fondamentale della teoria di Galois.

Il campo di Galois

Il campo finito o anche denominato il campo di Galois, dal matematico francese Évariste Galois, è un campo che contiene un numero finito di elementi che ci permette di attuare operazioni su numeri a 8 bits.

Fornisce una connessione tra teoria di campi e teoria dei gruppi, attraverso il teorema fondamentale della teoria di galois.

Lo introdusse per studiare le radici dei polinomi e risolvere queste equazioni polinomiali in termini di gruppi di permutazione delle loro radici.

Un'equazione è risolvibile dai suoi radicali se le sue radici si possono esprimere da una formula che riguarda numeri interi, radici ennesime e le 4 operazioni aritmetiche basilari $(+, -, \cdot, /)$.

Ora arriviamo al motivo per cui ci interessa:

Il più comune campo di Galois è il campo GF(256) o anche designato GF(2^8) (GF = Galois Field) che ci permette di definire operazioni sono a livello di byte, dove i bits vengono interpretati come coefficienti dei polinomi del campo finito.

Le operazioni del campo finito

Nel campo finito sono disponibili le quattro operazioni basilari dell'aritmetica, $(+, -, \cdot, /)$ e di conseguenza l'esponenziazione e i logaritmi.

I numeri vengono rappresentati come polinomi di 8 bits, ovvero, per esempio il numero esadecimale 0x53, ovvero 83 in decimale o 01010011 in binario può essere rappresentato come $x^6 + x^4 + x + 1$ perché 01010011 $(1 \cdot 2^6, 1 \cdot 2^4, 1 \cdot 2^1, 1 \cdot 2^0$ <=> $x^6 + x^4 + x + 1)$.

Addizione e sottrazione

Somma e sottrazione vengono eseguite tramite uno XOR.

Per esempio: prendiamo due polinomi: $x + 1$ e $x^2 + 1$. $x + 1$ equivale a $2^1 + 2^0$, ovvero in binario il numero 11. $x^2 + 1$ equivale a $1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$, ovvero in binario al numero 101. dopodiché eseguiamo, semplicemente, uno XOR tra i due polinomi. $011 \oplus 101$ oppure $x + 1 \oplus x^2 + 1 = 110$ o $x^2 + x$

Moltiplicazione

Per la moltiplicazione è un pochino più complicato.

Avvenuta l'operazione, è necessario compiere un'operazione di modulo con il polinomio $x^8 + x^4 + x^3 + x + 1$.

per esempio $0x53 \cdot 0xCA = 0x01$, perché:

$0x53 = 83 = 1010011 = x^6 + x^4 + x + 1$

$0xCA = 202 = 11001010 = x^7 + x^6 + x^3 + x$

$(x^6 + x^4 + x + 1)(x^7 + x^6 + x^3 + x)$

$= (x^{13} + x^{12} + x^9 + x^7) + (x^{11} + x^{10} + x^7 + x^5) + (x^8 + x^7 + x^4 + x^2) + (x^7 + x^6 + x^3 + x)$

$= x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + x^2 + x$

$x^{13} + x^{12} + x^{11} + x^{10} + x^9 + x^8 + x^6 + x^5 + x^4 + x^3 + x^2 + x$ modulo $x^8 + x^4 + x^3 + x + 1 = 1$.

Esponenziazione

L'esponenziazione è semplicemente la ripetizione della moltiplicazione.

Alcuni numeri possiedono la proprietà di attraversare tutti i 255 numeri non-zero del campo quando vengono moltiplicati per se stessi. Questi numeri vengono chiamati *generatori*.

Possiamo raccogliere questi numeri in una tabella, la tabella dei generatori.

Logaritmi

Per eseguire i logaritmi possiamo avvalerci della tabella dei generatori.

Divisione

Si fa prendendo i logaritmi dei due numeri e eseguedo il modulo 255.

$g^{(x-y)mod 255}$ dove g è uno dei generatori.

Il teorema fondamentale della teoria di Galois

È un teorema che crea un legame tra gli intercampi di un'estensione di Galois e i sottogruppi del relativo gruppo di Galois.

Data un'estensione di campi M/K con gruppo di Galois $G =$ Gal(M/K) definiamo *i* e *j* (due funzioni):

- Per ogni intercampo L (cioè tale che $K \subseteq L \subseteq M$) poniamo *i*(L) = Gal(M/L), ossia il sottogruppo degli automorfismi di M che lasciano fissi gli elementi di L.
- Per ogni sottogruppo H di G, *j*(H) che classicamente è indicato con M^H, è l'intercampo costituito dagli elementi di M lasciati fissi da tutti gli automorfismi di H.

Capitolo 4

Le modalità di AES

Introduzione

In questa parte tratteremo le varie modalità di crittografia disponibili per AES. Queste ci permetteranno di cifrare messaggi di varia lunghezza.

IV, Nonce, Salt e Pepper

Prima di poter trattare le modalità è necessario introdurre alcuni componenti che ci serviranno nell'utilizzo di queste. Questi sono l'IV, il nonce, il sale e il pepe.

Nonce | Number Used Once

Il nonce, ***n**umber used **once*** (numero utilizzato una volta), sono dei bits (ovvero un numero) che vengono utilizzati una singola volta. Il riutilizzo non è proibito, ma deve essere limitato.

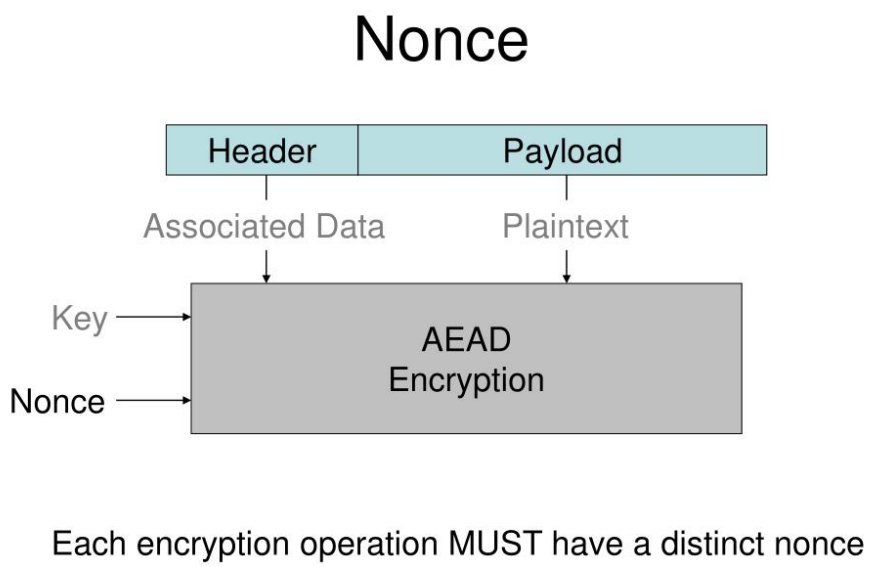


Figura 4.1: Nonce

Viene utilizzato nei protocolli di autenticazione per garantire la sicurezza nelle comunicazioni (private) (ed evitare replay attacks).

Nonce sequenziali

Garantiscono la non ripetizione, se molto grandi.

IV | Initialization Vector

L'initialization vector o anche definito starting variables è un input di dimensione fissa, ovvero un numero arbitrario che viene adoperato per impostare lo stato iniziale di un algoritmo crittografico. Viene utilizzato in varie modalità di cifratura per randomizzare la cifratura in modo da produrre diversi testi cifrati anche se lo stesso testo viene cifrato molteplici volte.

IV vs Nonce

Salt

Il sale viene utilizzato nelle password per incrementarne la sicurezza. Vengono aggiunti dei caratteri unici casuali alla password, in questo modo un attaccante non necessita solamente di un dizionario di password comuni, ma anche di uno per ogni tipo di sale possibile.

Figura 4.2: Salt

Questo tecnica è molto importante e viene utilizzata nei database affinché le password degli utenti siano sicure. (e non possano essere rubate).

Le password non vengono mai, possibilmente, memorizzate nei database in chiaro, ma gli viene aggiunto il salt e poi viene eseguito l'hashing su esse.

Pepper

Il pepper è simile al salt, ma questo a differenza del sale che è meramente unico più che segreto, è un (singolo) carattere segreto aggiunto alla fine della password.

Questo, (a differenza del sale) non viene memorizzato nello stesso database assieme alla password hash e al sale, ma a parte.

Il padding

Un altro elemento utilizzato nei cifrari a blocchi è il padding. Il padding serve per riempire i blocchi del cifrario con dei bytes.

È un modo per cifrare messaggi anche di grandezze che il cifrario non sarebbe in grado di decifrare. Non aumenta la sicurezza, anzi se mal implementato può portare ad attacchi di padding (*padding oracle attack*).

Grazie a questa tecnica, è possibile aggiungere, all'inizio, al centro o in fondo al messaggio, del nonsense per oscurare parti del messaggio che altrimenti sarebbero prevedibili, come: *Caro..., Gentile..., Cordiali Saluti..,* ecc.

I principali meccanismi di padding sono:

- **No padding**: Nessun padding viene aggiunto al messaggio.
- **0-Padding**: Vengono aggiunti dei bytes con degli zeri alla fine dell'ultimo blocco, finché non abbiamo il corretto numero di bytes (in AES è 16 bytes); se già abbiamo il numero necessario di bytes, allora non viene aggiunto nulla
- **1-0-Padding**: Viene aggiunto un byte a 1 e il restante viene riempito da bytes di 0 finché non abbiamo il numero necessario di bytes; se il numero di bytes necessario è già presente, allora aggiungiamo un altro blocco.
- **ANSI X9.23 Padding**: Aggiungiamo bytes di 0 alla fine dell'ultimo blocco, finché non abbiamo 16 - 1 bytes. Nell'ultimo blocco inseriamo il numero totale di bytes a 0 che abbiamo aggiunto; se l'ultimo blocco è già pieno (corrisponde a 16 bytes), allora aggiungiamo blocco addizionale.
- **ISO 10126 Padding**: Vengono aggiunti dei bytes casuali in fondo all'ultimo blocco, finché non abbiamo 16 - 1 bytes e nell'ultimo byte inseriamo il numero totale di bytes casuali che abbiamo immesso; Se l'ultimo blocco è già 16 bytes, allora aggiungiamo un altro blocco.
- **PKCS#7**: Aggiungiamo il numero totale di bytes che restano per riempire il blocco in ogni byte rimanente, finché non abbiamo 16 bytes.
-
-

A cosa servono le modalità?

Le modalità servono, perché AES è un cifrario a blocchi di 16 bytes, dopo aver preso un blocco e una chiave in input restituisce un altro blocco della medesima grandezza.

Quindi, se si desidera cifrare una sequenza di bytes di lunghezza maggiore di 16, è necessario utilizzare una *modalità di operazione* o *modalità di cifratura*.

Queste modalità sono indipendenti dal tipo di cifrario che sta alla base.

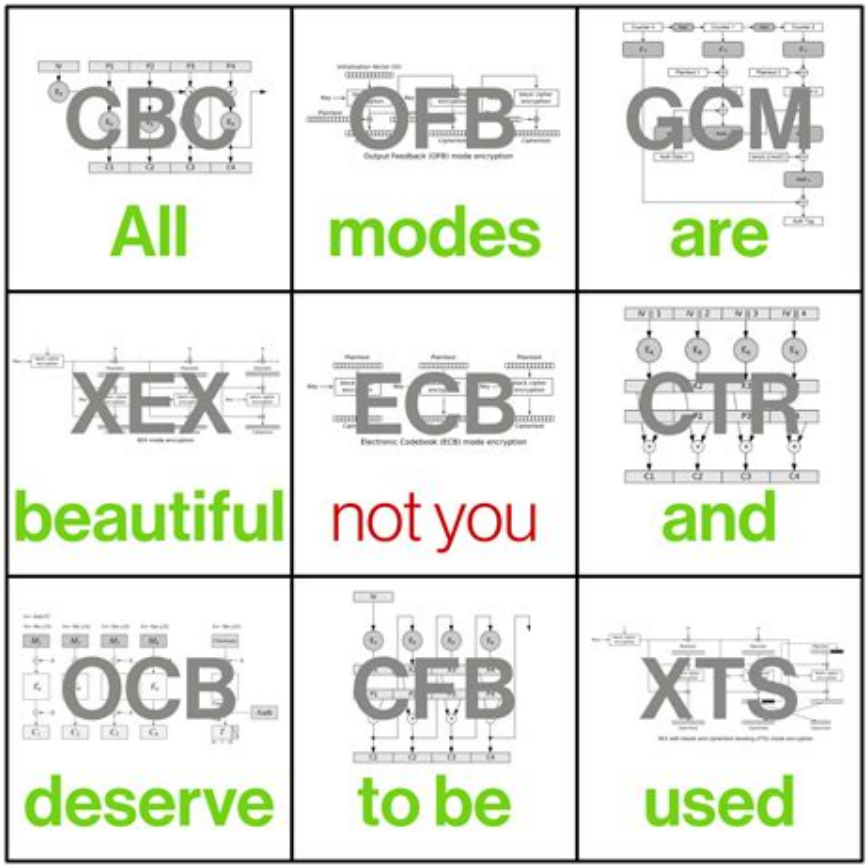


Figura 4.3: Modalità di AES

Le modalità

Qui, di seguito elencherò le caratteristiche e proprietà delle principali modalità di cifratura:

Modalità di cifratura senza integrità del messaggio

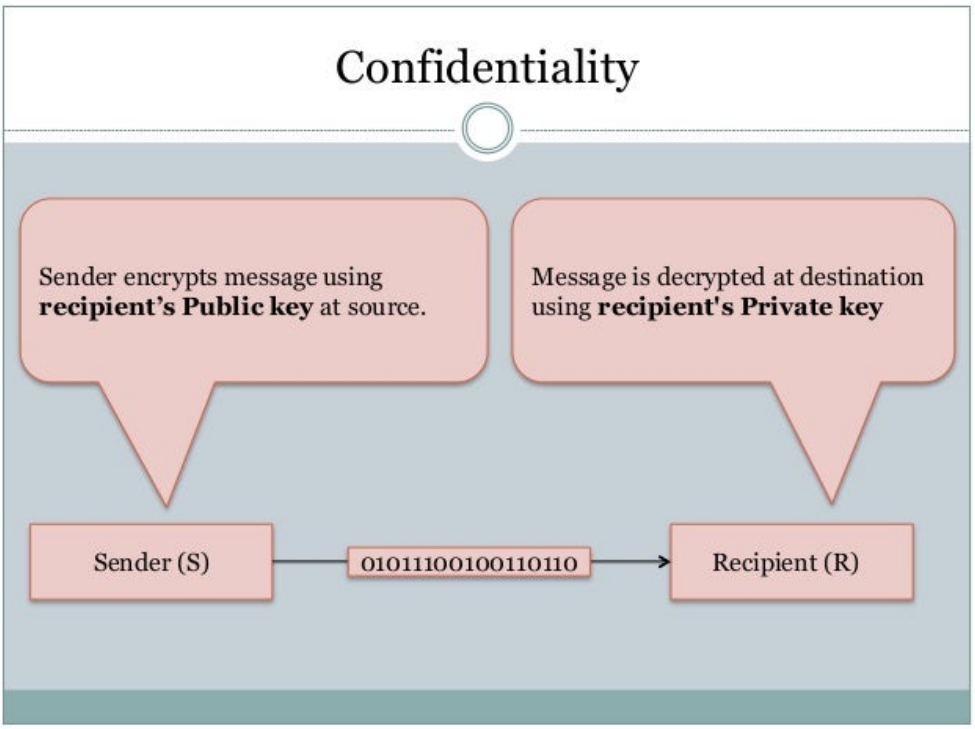


Figura 4.4: Confidenzialità

Queste modalità forniscono la cifratura del messaggio, ma non garantisco l'integrità del messaggio, ovvero non è assicurato che il messaggio non sia stato manomesso e quindi non è possibile accertare l'autenticazione del messaggio originale.

ECB | Electronic Code Book

Questa modalità di cifratura divide il nostro messaggio in diversi blocchi e ognuno di questi viene cifrato separatamente.

Questa modalità manca il principio di diffusione e quindi cifra lo stesso messaggio in chiaro nello stesso testo cifrato, quindi non vela il pattern dei dati.

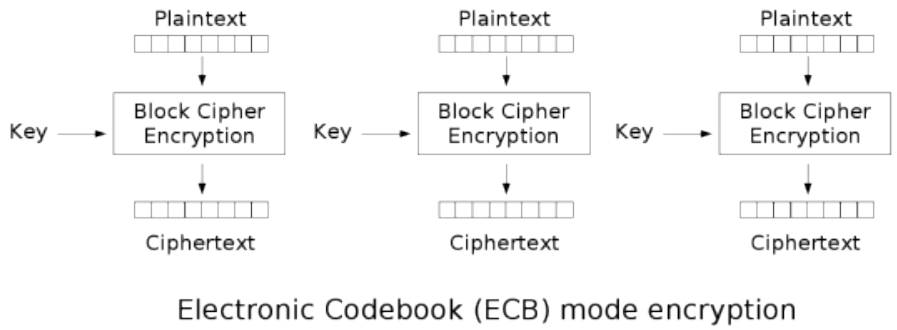


Figura 4.5: ECB

Come si deduce dall'immagine, ogni blocco viene cifrato indipendentemente. Viene inserito il messaggio in chiaro e la chiave all'interno del cifrario per ottenere il testo cifrato. Blocchi di testo in chiaro uguali vengono cifrato allo stesso modo.

Per questo, osservando il testo cifrato è possibile riconoscere, se presenti nel testo in chiaro, parti cifrate allo stesso modo.

Per decifrare, eseguiamo l'operazione inversa, immettendo un blocco cifrato assieme alla chiave, per ottenere il testo decifrato.

Ricapitolando i problemi di ECB:

- ECB è deprecato e non dovrebbe essere utilizzato.

- Gli stessi blocchi di messaggio vengono cifrati con gli stessi blocchi cifrati.

CBC | Cipher Block Chain

La modalità CBC cerca di ovviare ai problemi dell'ECB aggiungendo della casualità a ogni operazione di cifratura usufruendo di un initialization vector (IV) per il primo blocco e dopodiché applicando uno XOR a ogni blocco di testo in chiaro con il blocco cifrato precedente.

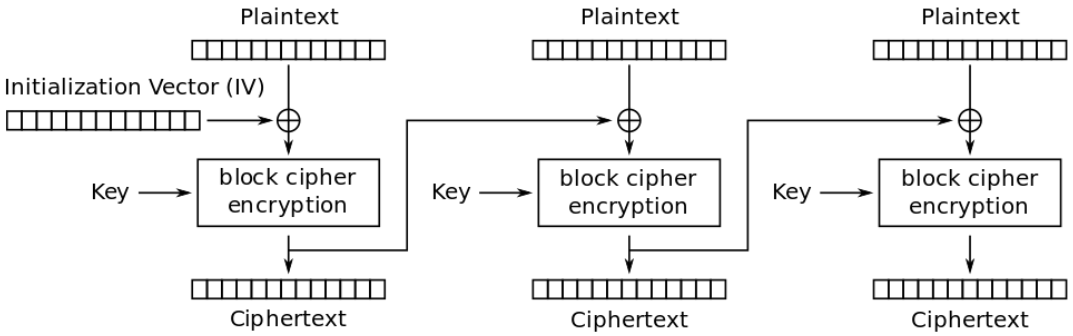


Figura 4.6: CBC

Ricapitolando le fasi che si evincono dallo shema, qui sopra:

- Collega ("Incatena" da *Chaining*) tutti i blocchi.
- L'IV (***I**nitialization **V**ector*) è usato per modificare il testo in chiaro.
- Viene eseguito lo XOR tra il blocco cifrato precedente e il blocco col testo in chiaro corrente.
- Risolve il problema dei blocchi di testo in chiaro uguali vengano cifrati allo stesso modo. (problema presente in ECB)
- Modificando un bit di un blocco di testo in chiaro, modifica di conseguenza tutti gli altri blocchi cifrati a seguire.

Sono possibili degli attacchi su CBC, se l'attaccante modifica ogni secondo blocco in chiaro se è a conoscenza dell'intero testo in chiaro.

Questa modalità è sicura come uno schema probabilistico e la confidenzialità non viene raggiunta se l'IV è un nonce semplice.

CFB | Cipher Feedback

CFB è molto simile a CBC, ma al posto di essere un cifrario a blocchi, è un cifrario a stream, *stream cipher*, ovvero un cifrario che prende una chiave e un algoritmo e lo applica allo stream (un bit alla volta), a ogni singolo bit (bit a bit) e non a blocchi di bits.

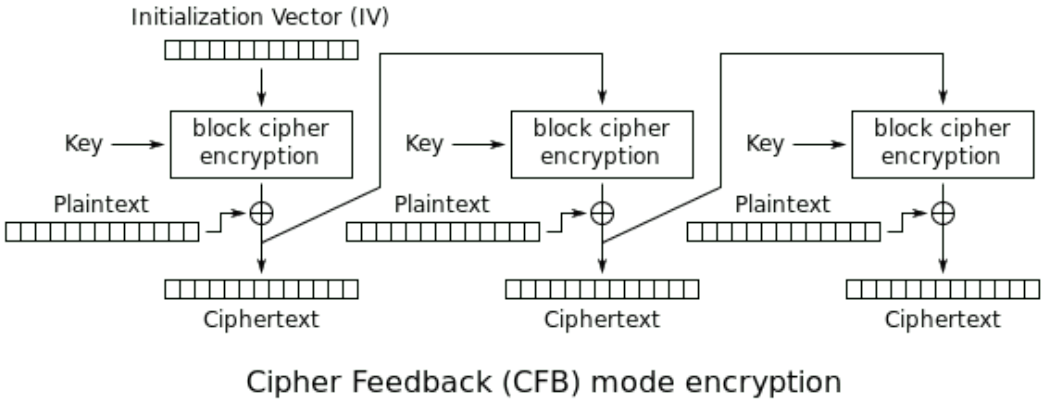


Figura 4.7: CFB

Ricapitolando lo schema presente qui sopra:

- Lega tutti i blocchi (proprio come CBC).
- Genera dei bytes casuali, un flusso usando il cifrario che viene poi successivamente XOR col blocco di testo in chiaro.
- Trasforma il cifrario a blocchi in uno stream cipher (cifrario a flusso).

OFB | Output Feedback

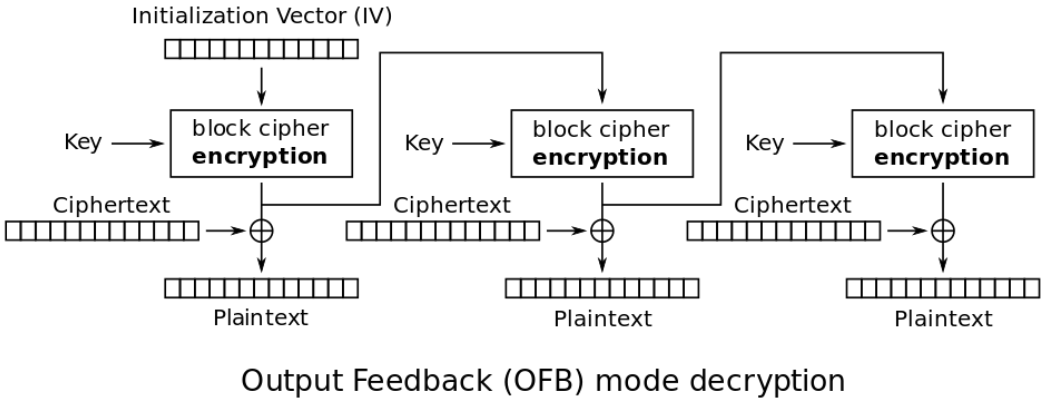


Figura 4.8: OFB

È anch'esso un *stream cipher*, cifra l'iv (genera una serie di caratteri casuali) e poi esegue uno XOR per ottenere il testo cifrato.

Ogni operazione dipende da quelle precedenti.

Ricapitolando le operazioni dell'OFB, proprio come il CFB:

- Lega tutti i blocchi (proprio come CBC e CFB).
- Genera dei bytes casuali, un flusso usando il cifrario che viene poi successivamente XOR col blocco di testo in chiaro.
- Trasforma il cifrario a blocchi in uno stream cipher (cifrario a flusso).

L'unica differenza con CFB è che al posto di utilizzare il testo cifrato nei blocchi successivi, utilizza l'output dei bytes casuali generati dal cifrario attraverso la chiave e l'IV per il blocco successivo.

CTR | Counter Mode

La Counter Mode o CM (*integer counter mode*) o SIC (*segmented integer counter*) usufruisce di un counter (contatore), ovvero di una funzione che genera sequenze irripetibili (nel tempo), al posto di utilizzare un IV.

Il più semplice contatore è il (banale/banalissimo) incrementa di 1. Questa modalità può essere parallelizzata.

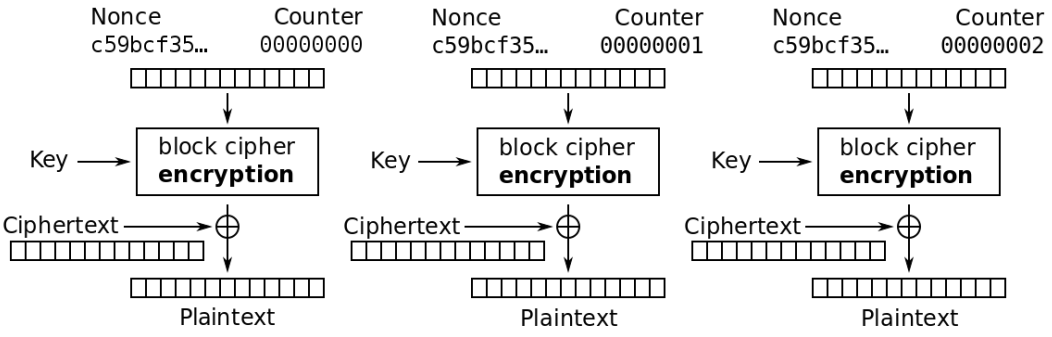


Figura 4.9: CTR

XTS | AES-XTS (XEX) Tweakable Block Cipher

Questa modalità è utilizzata per cifrare/decifrare i dischi rigidi (hard-disk), quindi la cifratura verrà operata sui segmenti da 512 bytes (ovvero 32 blocchi di AES-128) e i rispettivi settori (del disco).

Questa modalità compie le seguenti operazioni:

- Definite due chiavi: K1, K2 (utilizzando AES-128), Blocco di input, p, numero settore, *i* e un numero del blocco, *j*, definiamo il "tweak", *X*, nel seguente modo:
  - $X = E_{K_2}(i) \otimes 2^j$
- Di conseguenza, il testo cifrato C è definito così:
  - $C = P \oplus X$
  - ovvero testo cifrato = blocco di input "XORato" con il tweak.
  - $C = E_{K_1}(C) \oplus X$
- Per poter decriptare, invece, eseguiamo:
  - $X = E_{K_2}(i) \otimes 2^j$
- il testo in chiaro, per ogni blocco, è:
  - $P = C \oplus X$
  - $P = E^{-1}_{K_1}(P) \oplus X$
- In questo modo, il tweak fa da IV.

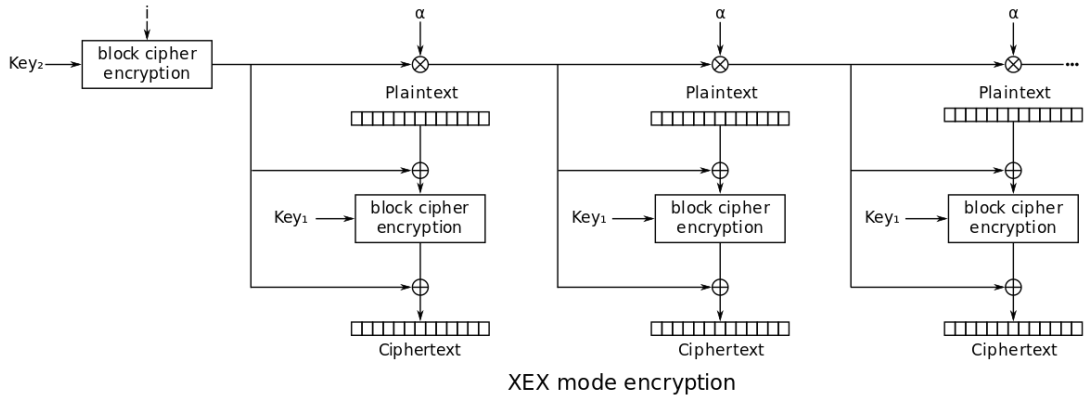


Figura 4.10: XEX

MACS | Message Authentication Codes

I MACS permettono di verificare l'integrità del messaggio per potersi assicurare che non sia stato alterato da parti esterne e che il messaggio giunga integro ai destinatari originali.

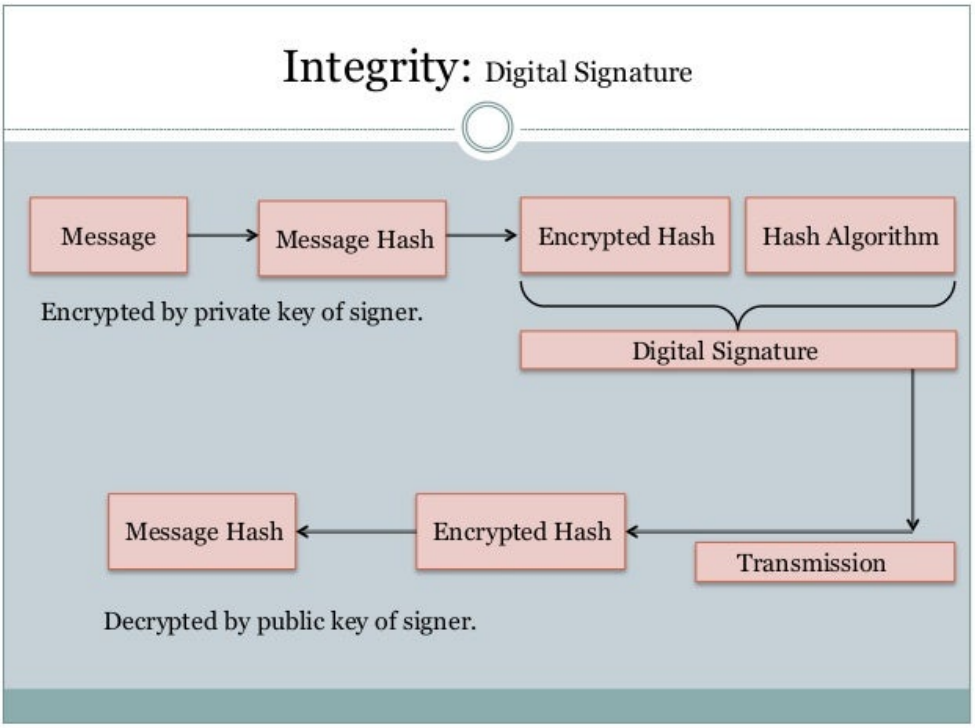


Figura 4.11: Integrità e non ripudio

ALG1-6

È una collezione di MACs basati su CBC-MAC, di cui alcuni sicuri come il VIL PRF, FIL PRF altri non ne abbiamo certezza.

CMAC | Cipher-based Message Authentication Code

CMAC è una modalità basata su CBC-MAC. Questa crea un messaggio di autenticazione (MAC) utilizzando un cifrario a blocchi e una chiave segreta.

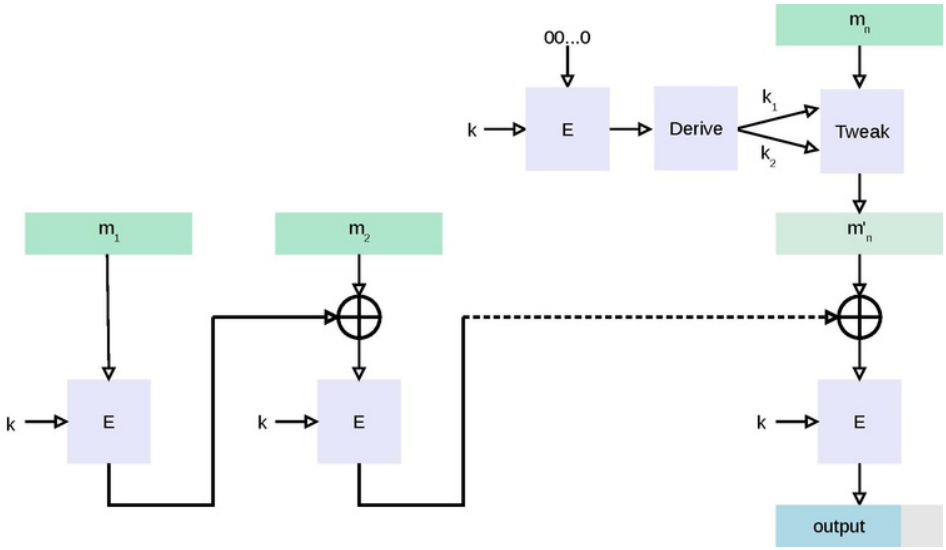


Figura 4.12: CMAC

HMAC | Keyed-hash Message Authentication Code

HMAC è una modalità che permette di autenticare gli emittenti del messaggio attraverso autenticazione basata su algoritmi di hashing, come: MD5, SHA-1, SHA-256 (SHA-2) e SHA-3.

La sicurezza di questa modalità è stata provata.

GMAC | Galois Message Authentication Code

Questa modalità è una specializzazione della modalità **GCM** (***G**alois/**C**ounter **M**ode*). Viene utilizzata per l'autenticazione.

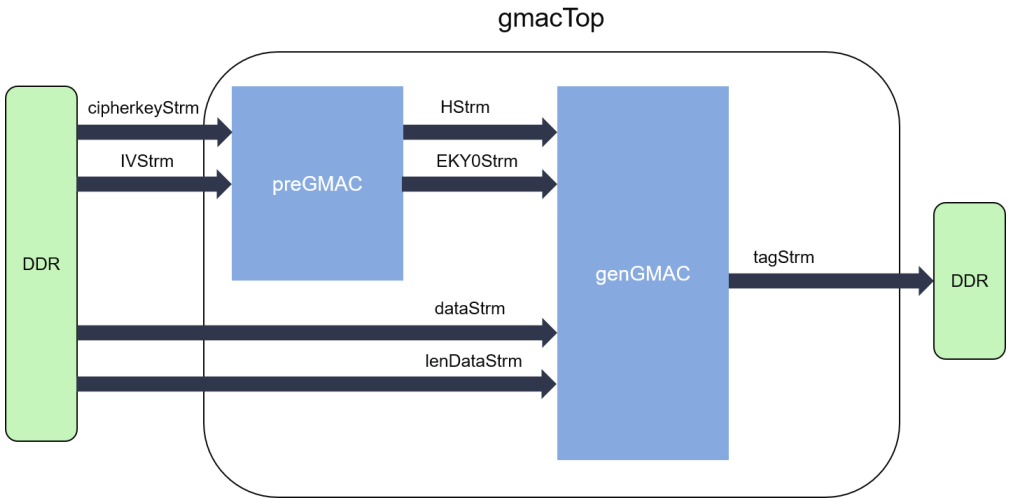


Figura 4.13: GMAC

CBC-MAC

CBC-MAC combina la modalità CBC (Cipher Block Chaning) assieme al MAC per l'autenticazione dei mittenti del messaggio.

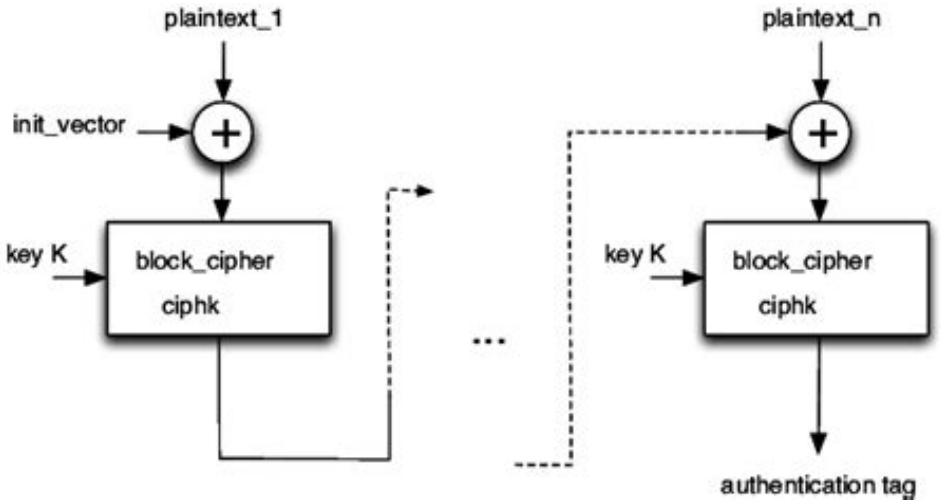


Figura 4.14: CBC-MAC

AEAD | Authenticated Encryption with Associated Data

AEAD è un modello di cifratura che assicura sia la riservatezza del messaggio sia la sua autenticazione.

OCB | Offset Codebook

Questa modalità è stata ideata da Phillip Rogaway con l'assistenza di Mihir Bellare, John Black e Ted Krovetz, basata sull'IAPM (***I**ntegrity-**A**ware **P**arallelizeable **M**ode*) che permette di parallelizzare la modalità.

Sono presenti tre versioni di OCB: OCB1, OCB2 e OCB3.

OCB, rispetto alle altre modalità, è molto più veloce.

Erano presenti due brevetti su questa modalità che ne impedivano l'esteso utilizzo, ma dal 2021 questi sono stati abbandonati.

CCM | Counter con CBC-MAC

CCM è un AEAD basato su Nonce che combina le modalità CTR e CBC-MAC i cui blocchi devono essere composti da 128 bits.

GCM | Galois Counter Mode

È un AEAD basato su Nonce che combina le modalità CTR per l'operazione di cifratura e il campo di Galois (Galois Field) GF(2^{128}) per l'autenticazione.

Può essere usato come Nonce-MAC, in quel caso viene denominato **GMAC**.

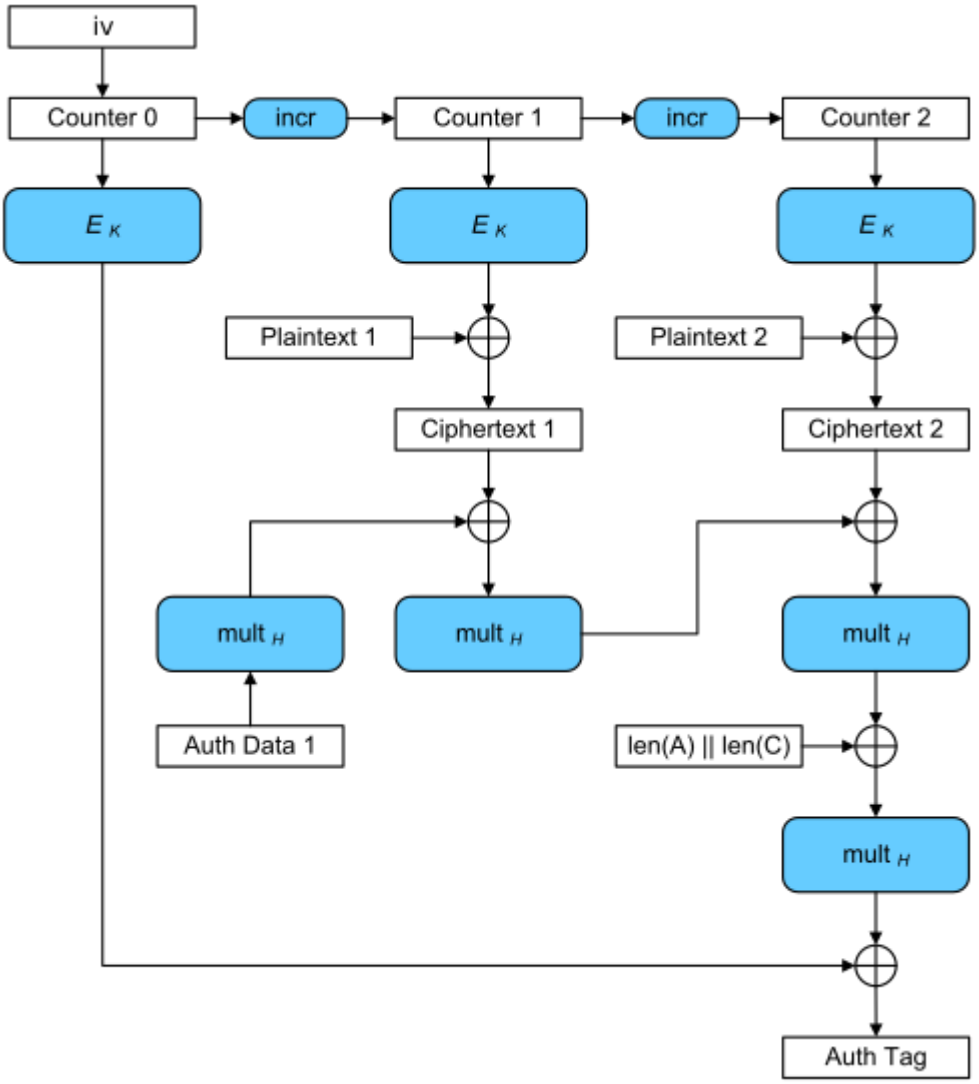


Figura 4.15: GCM

Capitolo 5

L'implementazione

Introduzione

In questo capitolo, presenterò due implementazioni, una elaborata esclusivamente per questo progetto in C++ e un'altra in Java che avevo creato per altri progetti universitari. Quella in C++ risulterà più completa rispetto a quella in Java.

Implementazione in C++

Ho adottato C++23 per questo progetto. In esso sono presenti una interfaccia grafica e una applicazione da linea di comando, entrambe hanno le stesse operazioni.

Inanzittutto, esibirò, le funzioni riguardanti la matematica di Galois di cui mi sono avvalso.

Matematica di Galois

```
/**
 * @brief corresponds to 27 = 00011011 = x^4 + x^3 + x + 1.
 */
static constexpr uint8_t IRREDUCIBLE_POLYNOMIAL = 0x1B;
```

Figura 5.1: Polinomio irriducibile

Riguardo alla matematica nel campo di Galois, ho adoperato tre funzioni: una che implementa l'addizione e la sottrazione, una per la moltiplicazione e una per generare le costanti di round.

Nel campo di Galois, sia l'addizione che la sottrazione sono semplicemente un'operazione di XOR. Questa funzione prende due parametri x ed y di tipo *uint8_t* (che corrisponde a un *unsigned char*) e restituisce lo XOR tra essi.

```
/**
 * @brief This function returns the galois addition or subtraction between two numbers. It xors the two numbers.
 * @param x: a uint8_t aka unsigned char.
 * @param y: a uint8_t aka unsigned char.
 * @return an xor between x and y.
 */
[[nodiscard("Pure function")]] static constexpr inline uint8_t galois_addition_subtraction(const uint8_t& x, const uint8_t& y) noexcept
{
    return x ^ y;
}
```

Figura 5.2: Addizione e sottrazione nel campo di Galois

galois_multiplication() prende due *uint8_t* come parametri e restituisce la moltiplicazione tra questi nel campo di Galois.

static constexpr indicano che la funzione può essere eseguita a compile time. *noexcept* indica che la funziona non lancia eccezioni. *[[nodiscard]]* indica che il risultato che viene restituito non può essere ignorato, ma deve essere utilizzato.

Facciamo un loop su ogni bit del byte e verifichiamo se il secondo valore *y* ha il bit meno significativo attivo (y & 0x01) allora aggiungiamo *x* (ovvero eseguiamo uno XOR) al risultato. Dopodiché verifichiamo se il bit più significativo *high_bit* è attivo in *x*. Poi ruotiamo x di 1. Se l'*high_bit* è *true* eseguiamo uno XOR tra x e il polinomio irriducibile *IRREDUCIBLE_POLYNOMIAL*, ovvero $0x1$B. Infine ruotiamo il secondo valore di 1 per ruotarlo a destra. Poi restituiamo il risultato finale delle operazioni.

```cpp
/**
 * @brief This function returns the multiplication in the galois field of x and y.
 * @param x: a uint8_t aka unsigned char.
 * @param y: a uint8_t aka unsigned char.
 * @return the galois multiplication between x and y.
 */
[[nodiscard]] static constexpr uint8_t galois_multiplication(uint8_t x, uint8_t y) noexcept
{
    uint8_t result = 0;

    for(unsigned short i = 0; i < 8; i++) {
        if(y & 0x01) {
            result ^= x; // ^= è l'addizione in GF(2^8); potrei anche fare result = galois_addition_subtraction(result, x);
        }

        const bool high_bit = x & 0x80; // x >= 0x80 = 128
        x <<= 1; // ruotiamo x di 1 (moltiplicazione in GF(2^8)
        if(high_bit) {
            x ^= IRREDUCIBLE_POLYNOMIAL; // x -= 0x1B, ovvero mod(x^8 + x^4 + x^3 + x + 1)
        }

        y >>= 1; // ruotiamo y a destra (divisione in GF(2^8)
    }

    return result;
}
```

Figura 5.3: Moltiplicazione nel campo di Galois

round_constant_generator(const uint8_t& x) prende un numero in input e restituisce la round constant.

Questa funzione è utilizzata per generare la round constant. L'algoritmo di questa funzione è il seguente:

- il round_constant(1) = 1.
- round_constant(i) = $2 \cdot$ round_constant($i-1$)se round_constant(i - 1) < 0x80
- round_constant($2 \cdot$ round_constant(i - 1)) $\oplus$0x11B $\geq 0x80$

```cpp
/**
 *
 * @param x: a uint8_t aka unsigned char.
 * @return the round constant.
 */
[[nodiscard]] static constexpr inline uint8_t round_constant_generator(const uint8_t& x) noexcept
{
    return (x << 0x01) ^ (((x >> 0x07) & 0x01) * gal::IRREDUCIBLE_POLYNOMIAL);
}
```

Figura 5.4: Generatrice delle costanti di round

Add Round Key

Nella fase di *add_round_key()* la chiave di round viene aggiunta alla matrice di stato.

La funzione prende la matrice di stato 4x4 (formata da due std::array di grandezza BLOCK_WORDS che indica il numero di words in un blocco, ovvero 4) e la chiave del round come puntatore e le aggiunge.

```
void add_round_key(std::array<std::array<uint8_t, aes::BLOCK_WORDS>, aes::BLOCK_WORDS>& state, const uint8_t* key)
{
    for(uint8_t i = 0; i < aes::BLOCK_WORDS; i++) {
        for(uint8_t j = 0; j < aes::BLOCK_WORDS; j++) {
            state[i][j] = state[i][j] ^ key[i + aes::BLOCK_WORDS * j];
        }
    }
}
```

Figura 5.5: Add Round Key

Sub Bytes

Nella funzione *sub_bytes* ogni byte della matrice di stato viene sostituito con quelli presenti nella S-BOX.

Quindi, nella funzione viene passata la matrice di stato come reference, quindi tutti le modifiche verranno applicate anche all'esterno della funzione e poi viene eseguito un loop e ogni elemento viene sostituito con il corrispettivo della S-BOX.

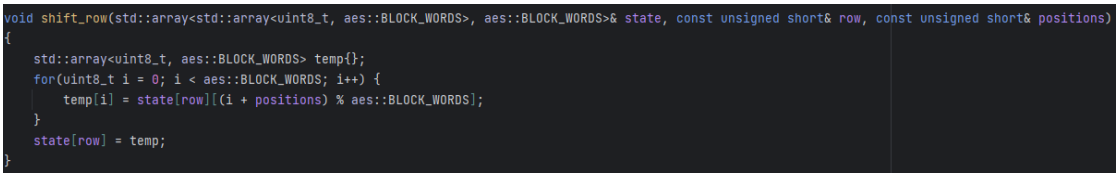
```
void sub_bytes(std::array<std::array<uint8_t, aes::BLOCK_WORDS>, aes::BLOCK_WORDS>& state)
{
    for(auto& s : array<unsigned char, 4> & : state) {
        for(uint8_t& i : s) {
            i = S_BOX[i];
        }
    }
}
```

Figura 5.6: Sub Bytes

Shift Rows

Nel passaggio di *shift rows* le righe della matrice di stato verranno *shiftate* di una posizione la seconda riga, di due posizione la terza e di tre la quarta.

Per fare questo ci avvaliamo di due funzioni, una *shift_row* per shiftare effettivamente le righe e nell'altra *shift_rows* per chiamare la precedente funzione per shiftare delle posizioni stabilite.



```
void shift_row(std::array<std::array<uint8_t, aes::BLOCK_WORDS>, aes::BLOCK_WORDS>& state, const unsigned short& row, const unsigned short& positions)
{
    std::array<uint8_t, aes::BLOCK_WORDS> temp{};
    for(uint8_t i = 0; i < aes::BLOCK_WORDS; i++) {
        temp[i] = state[row][(i + positions) % aes::BLOCK_WORDS];
    }
    state[row] = temp;
}
```

Figura 5.7: Shift Row

```
void shift_rows(std::array<std::array<uint8_t, aes::BLOCK_WORDS>, aes::BLOCK_WORDS>& state)
{
    shift_row( &: state,  row: aes::FIRST_SHIFT_ROW,  positions: 1);
    shift_row( &: state,  row: aes::SECOND_SHIFT_ROW,  positions: 2);
    shift_row( &: state,  row: aes::THIRD_SHIFT_ROW,  positions: 3);
}
```

Figura 5.8: Shift Rows

Mix Columns

La procedura *mix_columns* prende in input la matrice di stato, mescola i suoi bytes.

```
void mix_columns(std::array<std::array<uint8_t, aes::BLOCK_WORDS>, aes::BLOCK_WORDS>& state)
{
    std::array<std::array<uint8_t, aes::BLOCK_WORDS>, aes::BLOCK_WORDS> temp{};

    for(uint8_t i = 0; i < aes::BLOCK_WORDS; ++i) {
        for(uint8_t j = 0; j < aes::BLOCK_WORDS; ++j) {
            for(uint8_t k = 0; k < aes::BLOCK_WORDS; ++k) {
                if(CIRCULANT_MDS[i][j] == 1) {
                    temp[i][k] ^= state[j][k];
                } else {
                    temp[i][k] ^= gal::galois_multiplication( x: CIRCULANT_MDS[i][j],  y: state[j][k]);
                }
            }
        }
    }

    state = temp;
}
```

Figura 5.9: Mix Columns

Key Expansion

In questa funzione vengono generate le altre chiavi dei rounds a partire dalla prima chiave. Gli viene passata un array con la chiave, una word e la tipologia di AES: 128, 192 o 256.

Dopodiché eseguiamo le operazioni di: *rot_word*, *sub_word*, e *rcon*. Dopodiché viene eseguito uno XOR tra la chiave e il rcon. Dopodiché si continua a eseguire uno XOR con le chiavi precedenti.

```
void key_expansion(const uint8_t key[], unsigned char word[], const AES& aes)
{
    std::array<uint8_t, aes::AES_128_NUMBER_OF_KEYS> temp{};
    std::array<uint8_t, aes::AES_128_NUMBER_OF_KEYS> rcon{};

    const unsigned short& number_of_keys = aes::get_number_of_keys(aes);
    const unsigned short& number_of_rounds = aes::get_number_of_rounds(aes);

    unsigned int i = 0;
    while(i < aes::AES_128_NUMBER_OF_KEYS * number_of_keys) {
        word[i] = key[i];
        i++;
    }

    i = aes::BLOCK_WORDS * number_of_keys;
    while(i < aes::AES_128_NUMBER_OF_KEYS * aes::BLOCK_WORDS * (number_of_rounds + 1)) {
        temp[0] = word[i - aes::AES_128_NUMBER_OF_KEYS + 0];
        temp[1] = word[i - aes::AES_128_NUMBER_OF_KEYS + 1];
        temp[2] = word[i - aes::AES_128_NUMBER_OF_KEYS + 2];
        temp[3] = word[i - aes::AES_128_NUMBER_OF_KEYS + 3];

        if (i / aes::AES_128_NUMBER_OF_KEYS % number_of_keys == 0) {
            rot_word( &: temp);
            sub_word( &: temp);
            aes::rcon( &: rcon,  number_of_keys: i / (number_of_keys * aes::AES_128_NUMBER_OF_KEYS));
            for (unsigned short k = 0; k < aes::AES_128_NUMBER_OF_KEYS; k++) {
                temp[k] = gal::galois_addition_subtraction( x: temp[k],  y: rcon[k]);
            }
        } else if (number_of_keys > aes::AES_192_NUMBER_OF_KEYS &&
                   i / aes::AES_128_NUMBER_OF_KEYS % number_of_keys == aes::AES_128_NUMBER_OF_KEYS) {
            sub_word( &: temp);
        }

        word[i + 0] = gal::galois_addition_subtraction( x: word[i - aes::AES_128_NUMBER_OF_KEYS * number_of_keys],  y: temp[0]);
        word[i + 1] = gal::galois_addition_subtraction( x: word[i + 1 - aes::AES_128_NUMBER_OF_KEYS * number_of_keys],  y: temp[1]);
        word[i + 2] = gal::galois_addition_subtraction( x: word[i + 2 - aes::AES_128_NUMBER_OF_KEYS * number_of_keys],  y: temp[2]);
        word[i + 3] = gal::galois_addition_subtraction( x: word[i + 3 - aes::AES_128_NUMBER_OF_KEYS * number_of_keys],  y: temp[3]);

        i += aes::BLOCK_WORDS;
    }
```

Figura 5.10: Key Expansion

Rot Word

```
void rot_word(std::array<uint8_t, aes::AES_128_NUMBER_OF_KEYS>& keys)
{
    const uint8_t temp = keys[0];
    keys[0] = keys[1];
    keys[1] = keys[2];
    keys[2] = keys[3];
    keys[3] = temp;
}
```

Figura 5.11: Rot Word

In questa operazione ogni byte (word di 32 bits, ovvero 4 bytes) viene ruotato di 1 posizione.

Sub Word

```
void sub_word(std::array<uint8_t, aes::AES_128_NUMBER_OF_KEYS>& keys)
{
    for(uint8_t i = 0; i < aes::AES_128_NUMBER_OF_KEYS; i++) {
        keys[i] = S_BOX[keys[i]];
    }
}
```

Figura 5.12: Sub Word

Nella procedura *sub_word()* ogni byte della chiave viene sostituita con quella della S-BOX.

Rcon

```
void rcon(std::array<uint8_t, aes::AES_128_NUMBER_OF_KEYS>& keys, const uint8_t& number_of_keys)
{
    uint8_t temp = 1;
    for(uint8_t i = 0; i < number_of_keys - 1; i++) {
        temp = gal::round_constant_generator( x: temp);
    }

    keys[0] = temp;
    keys[1] = keys[2] = keys[3] = 0;
}
```

Figura 5.13: Rcon

Nella funzione *rcon,* le *round constants* vengono generate attraverso una funzione ricorsiva.

Xor Blocks

Con questa funzione eseguiamo uno XOR per ogni bit *i* tra i blocchi x e y e assegniamo il risultato a ogni bit del blocco z.

Il loop viene eseguito in base alla grandezza del blocco.

```
void xor_blocks(const uint8_t* x, const uint8_t* y, uint8_t* z, const unsigned int& block_length)
{
    for(unsigned int i = 0; i < block_length; i++) {
        z[i] = gal::galois_addition_subtraction(x[i], y[i]);
    }
}
```

Figura 5.14: Xor Blocks

Modes

ECB

ECB è la modalità più semplice e anche quella che non dovrebbe mai essere usata.

In questa modalità, semplicemente, ogni blocco viene cifrato com'è. Nessun vettore di inizializzazione viene utilizzato. Lo stesso input genererà lo stesso identico output.

Questa modalità inoltre accetta solo blocchi divisibili per 16, questo viene garantito attraverso la funzione *verify_length()* che verifica e lancia un'eccezione altrimenti.

```
uint8_t* encrypt_ECB(const uint8_t input[], const unsigned int& input_length, const uint8_t key[], const aes::AES& aes)
{
    verify_length( size: input_length);

    const unsigned short& number_of_rounds = aes::get_number_of_rounds(aes);
    uint8_t* output = new unsigned char[input_length];
    uint8_t* round_keys = new unsigned char[aes::BLOCK_WORDS * aes::BLOCK_WORDS * (number_of_rounds + 1)];

    aes::key_expansion(key,  word: round_keys, aes);
    for(unsigned int i = 0; i < input_length; i += aes::BLOCK_WORDS * aes::BLOCK_WORDS * sizeof(unsigned char)) {
        aes::encrypt_block( input: input + i,  output: output + i,  keys: round_keys, aes);
    }

    delete[] round_keys;

    return output;
}
```

Figura 5.15: Cifratura ECB

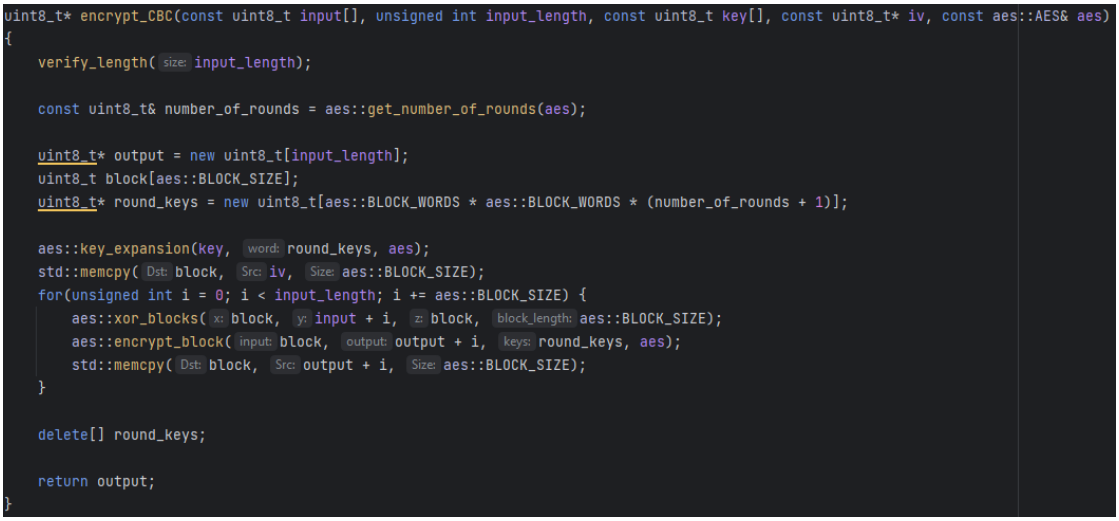
Per la decifrazione è lo stesso procedimento, ma inverso.

```
uint8_t* decrypt_ECB(const uint8_t input[], const unsigned int& input_length, const uint8_t key[], const aes::AES& aes)
{
    verify_length( size: input_length);

    const unsigned short& number_of_rounds = aes::get_number_of_rounds(aes);
    uint8_t* output = new unsigned char[input_length];
    uint8_t* round_keys = new unsigned char[aes::BLOCK_WORDS * aes::BLOCK_WORDS * (number_of_rounds + 1)];

    aes::key_expansion(key,  word: round_keys, aes);
    for(unsigned int i = 0; i < input_length; i += aes::BLOCK_WORDS * aes::BLOCK_WORDS * sizeof(unsigned char)) {
        aes::decrypt_block( input: input + i,  output: output + i,  keys: round_keys, aes);
    }

    delete[] round_keys;

    return output;
}
```

Figura 5.16: Decifrazione ECB

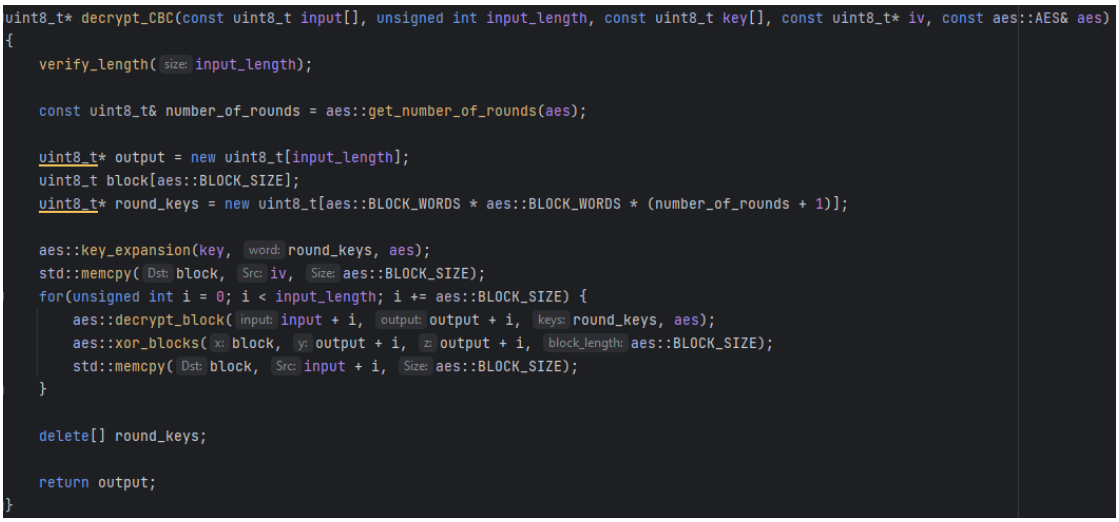
CBC

CBC è la modalità di chaining, in cui viene utilizzato un IV (initialization vector) per aggiungere casualità e viene eseguito uno XOR tra il messaggio in chiaro e il testo cifrato.



```
uint8_t* encrypt_CBC(const uint8_t input[], unsigned int input_length, const uint8_t key[], const uint8_t* iv, const aes::AES& aes)
{
    verify_length( size: input_length);

    const uint8_t& number_of_rounds = aes::get_number_of_rounds(aes);

    uint8_t* output = new uint8_t[input_length];
    uint8_t block[aes::BLOCK_SIZE];
    uint8_t* round_keys = new uint8_t[aes::BLOCK_WORDS * aes::BLOCK_WORDS * (number_of_rounds + 1)];

    aes::key_expansion(key,  word: round_keys, aes);
    std::memcpy( Dst: block,  Src: iv,  Size: aes::BLOCK_SIZE);
    for(unsigned int i = 0; i < input_length; i += aes::BLOCK_SIZE) {
        aes::xor_blocks( x: block,  y: input + i,  z: block,  block_length: aes::BLOCK_SIZE);
        aes::encrypt_block( input: block,  output: output + i,  keys: round_keys, aes);
        std::memcpy( Dst: block,  Src: output + i,  Size: aes::BLOCK_SIZE);
    }

    delete[] round_keys;

    return output;
}
```

Figura 5.17: Cifratura CBC



```
uint8_t* decrypt_CBC(const uint8_t input[], unsigned int input_length, const uint8_t key[], const uint8_t* iv, const aes::AES& aes)
{
    verify_length( size: input_length);

    const uint8_t& number_of_rounds = aes::get_number_of_rounds(aes);

    uint8_t* output = new uint8_t[input_length];
    uint8_t block[aes::BLOCK_SIZE];
    uint8_t* round_keys = new uint8_t[aes::BLOCK_WORDS * aes::BLOCK_WORDS * (number_of_rounds + 1)];

    aes::key_expansion(key,  word: round_keys, aes);
    std::memcpy( Dst: block,  Src: iv,  Size: aes::BLOCK_SIZE);
    for(unsigned int i = 0; i < input_length; i += aes::BLOCK_SIZE) {
        aes::decrypt_block( input: input + i,  output: output + i,  keys: round_keys, aes);
        aes::xor_blocks( x: block,  y: output + i,  z: output + i,  block_length: aes::BLOCK_SIZE);
        std::memcpy( Dst: block,  Src: input + i,  Size: aes::BLOCK_SIZE);
    }

    delete[] round_keys;

    return output;
}
```

Figura 5.18: Decifrazione CBC

CFB

In queste funzioni, innanzitutto, verifico che la lunghezza dell'input sia divisibile per 16, poi ottengo il numero di rounds in base a quale AES stiamo utilizzando. Imposto l'output, il blocco, il blocco cifrato e le chiavi di rounds che ottengo dalla key_expansion. Dopodiché cifro il blocco, eseguo uno XOR tra il plaintext e il blocco cifrato. Infine restituisco l'output.

```
uint8_t* encrypt_CFB(const uint8_t input[], unsigned int input_length, const uint8_t key[], const uint8_t* iv, const aes::AES& aes)
{
    verify_length( size: input_length);

    const uint8_t& number_of_rounds = aes::get_number_of_rounds(aes);

    uint8_t* output = new uint8_t[input_length];
    uint8_t block[aes::BLOCK_SIZE];
    uint8_t encrypted_block[aes::BLOCK_SIZE];
    uint8_t* round_keys = new uint8_t[aes::BLOCK_WORDS * aes::BLOCK_WORDS * (number_of_rounds + 1)];

    aes::key_expansion(key,  word: round_keys, aes);
    std::memcpy( Dst: block,  Src: iv,  Size: aes::BLOCK_SIZE);
    for(unsigned int i = 0; i < input_length; i += aes::BLOCK_SIZE) {
        aes::encrypt_block( input: block,  output: encrypted_block,  keys: round_keys, aes);
        aes::xor_blocks( x: input + i,  y: encrypted_block,  z: output + i,  block_length: aes::BLOCK_SIZE);
        std::memcpy( Dst: block,  Src: output + i,  Size: aes::BLOCK_SIZE);
    }

    delete[] round_keys;

    return output;
}
```

Figura 5.19: Cifratura CFB

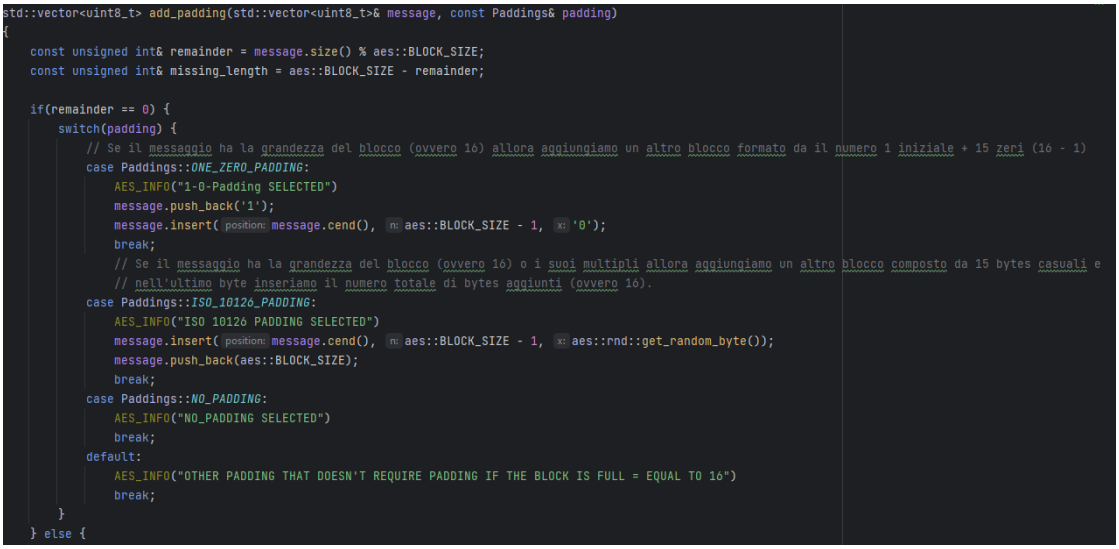
```
uint8_t* decrypt_CFB(const uint8_t input[], unsigned int input_length, const uint8_t key[], const uint8_t* iv, const aes::AES& aes)
{
    verify_length( size: input_length);

    const uint8_t& number_of_rounds = aes::get_number_of_rounds(aes);

    uint8_t* output = new uint8_t[input_length];
    uint8_t block[aes::BLOCK_SIZE];
    uint8_t encrypted_block[aes::BLOCK_SIZE];
    uint8_t* round_keys = new uint8_t[aes::BLOCK_WORDS * aes::BLOCK_WORDS * (number_of_rounds + 1)];

    aes::key_expansion(key,  word: round_keys, aes);
    std::memcpy( Dst: block,  Src: iv,  Size: aes::BLOCK_SIZE);
    for(unsigned int i = 0; i < input_length; i += aes::BLOCK_SIZE) {
        aes::encrypt_block( input: block,  output: encrypted_block,  keys: round_keys, aes); //TODO:
        aes::xor_blocks( x: input + i,  y: encrypted_block,  z: output + i,  block_length: aes::BLOCK_SIZE);
        std::memcpy( Dst: block,  Src: input + i,  Size: aes::BLOCK_SIZE);
    }

    delete[] round_keys;

    return output;
}
```

Figura 5.20: Decifrazione CFB

Paddings

Il padding viene utilizzato per aggiungere dei caratteri prestabiliti alla fine di un messaggio. Abbiamo due funzioni, una per aggiungere il padding, add_padding e una per rimuoverlo, remove_padding. Entrambe queste funzioni richiedono il messaggio da cui vogliamo aggiungere/rimuovere il padding e quale tipologia di padding applicare.

Otteniamo, innanzitutto, il resto per vedere se il messaggio è perfettamente divisibile del BLOCK_SIZE, ovvero 16 oppure no. Dopodiché calcoliamo la lunghezza mancante, sottraendo il BLOCK_SIZE al resto.



```
std::vector<uint8_t> add_padding(std::vector<uint8_t>& message, const Paddings& padding)
{
    const unsigned int& remainder = message.size() % aes::BLOCK_SIZE;
    const unsigned int& missing_length = aes::BLOCK_SIZE - remainder;

    if(remainder == 0) {
        switch(padding) {
            // Se il messaggio ha la grandezza del blocco (ovvero 16) allora aggiungiamo un altro blocco formato da il numero 1 iniziale + 15 zeri (16 - 1)
            case Paddings::ONE_ZERO_PADDING:
                AES_INFO("1-0-Padding SELECTED")
                message.push_back('1');
                message.insert( position: message.cend(), n: aes::BLOCK_SIZE - 1, x: '0');
                break;
                // Se il messaggio ha la grandezza del blocco (ovvero 16) o i suoi multipli allora aggiungiamo un altro blocco composto da 15 bytes casuali e
                // nell'ultimo byte inseriamo il numero totale di bytes aggiunti (ovvero 16).
            case Paddings::ISO_10126_PADDING:
                AES_INFO("ISO 10126 PADDING SELECTED")
                message.insert( position: message.cend(), n: aes::BLOCK_SIZE - 1, x: aes::rnd::get_random_byte());
                message.push_back(aes::BLOCK_SIZE);
                break;
            case Paddings::NO_PADDING:
                AES_INFO("NO_PADDING SELECTED")
                break;
            default:
                AES_INFO("OTHER PADDING THAT DOESN'T REQUIRE PADDING IF THE BLOCK IS FULL = EQUAL TO 16")
                break;
        }
    } else {
```

Figura 5.21: Aggiunta del padding (1/2)

Se il resto è 0, ovvero il blocco è formato da esattamente 16 caratteri, allora eseguiamo questo:

- **1-0-Padding**: Aggiungiamo un ulteriore blocco di lunghezza 16, ove il primo elemento è il carattere ′1′ e il rimanente è composto da zeri.

- **ISO_10126_PADDING**: Aggiungiamo 15 caratteri casuali e nell′ultimo carattere inseriamo il numero di caratteri casuali aggiunti.

- **NO_PADDING** o qualsiasi altro padding: non facciamo nulla.

```
    switch(padding) {
        // Aggiungiamo 0 fino a riempire la stringa, se la stringa non è un divisore di 16 (che è la grandezza del blocco)
        // Ma se il blocco è già 16, allora non aggiungiamo gli 0.
        case Paddings::ZERO_PADDING:
            AES_INFO("0-Padding SELECTED")
            message.insert( position: message.cend(), n: missing_length, x: '0');
            break;
            // Aggiungiamo un 1 all'inizio del padding e poi tanti zeri quanto missing_length - 1 (quel -1 perché inseriamo quel 1 iniziale)
        case Paddings::ONE_ZERO_PADDING:
            AES_INFO("1-0-Padding SELECTED")
            message.push_back('1');
            message.insert( position: message.cend(), n: missing_length - 1, x: '0');
            break;
            // Aggiungiamo tanti zeri e nell'ultimo byte mettiamo il numero totale di bytes aggiunti come singolo valore.
        case Paddings::ANSI_X9_23_PADDING:
            AES_INFO("ANSI X9.23 PADDING SELECTED")
            message.insert( position: message.cend(), n: missing_length - 1, x: '0');
            message.push_back(missing_length);
            break;
            // Aggiungiamo dei bytes casuali, tranne nell'ultimo byte in cui inseriamo la somma totale di bytes aggiunti.
        case Paddings::ISO_10126_PADDING:
            AES_INFO("ISO 10126 PADDING SELECTED")
            message.insert( position: message.cend(), n: missing_length - 1, x: aes::rnd::get_random_byte());
            message.push_back(missing_length);
            break;
            // Aggiungiamo come padding il numero totale di bytes
        case Paddings::PKCS7:
            AES_INFO("PKCS7 SELECTED")
            message.insert( position: message.cend(), n: missing_length, x: missing_length);
            break;
        default:
        case Paddings::NO_PADDING:
            AES_INFO("NO_PADDING SELECTED")
            break;
    }
}

AES_DEBUG("message dopo l'aggiunta del padding: {}", cvt::get_string_from_vector<std::string, uint8_t>( vector: message))

return message;
```

Figura 5.22: Aggiunta del padding (2/2)

Se il resto non è pari a 0, quindi il blocco non è pieno, non corrisponde a 16, allora eseguiamo:

- **0-Padding**: Aggiungiamo tanti zeri quanti necessari per riempire il blocco.
- **1-0-Padding**: Aggiungiamo un 1 e poi tanti zeri quanti necessari per riempire il blocco.
- **ANSI_X9_23_PADDING**: Aggiungiamo tanti zeri quanti necessari per riempire il blocco - 1, perché nell'ultimo byte inseriamo il numero degli zeri che abbiamo aggiunto.
- **ISO_10126_PADDING**: Aggiungiamo dei numeri casuali fino a riempire il blocco - 1, tranne che nell'ultimo byte in cui inseriamo il numero di bytes casuali aggiunti.
- **PKCS7**: Aggiungiamo il numero di bytes che mancano al riempire il blocco in tutti i blocchi rimanenti.

- **NO_PADDING**: Non eseguiamo alcuna operazione al testo in chiaro.

```
std::vector<uint8_t> remove_padding(std::vector<uint8_t>& decrypted_message, const Paddings& padding)
{
    const unsigned int& remainder = decrypted_message.size() % aes::BLOCK_SIZE;

    if(remainder == 0) {
        switch(padding) {
            case Paddings::ONE_ZERO_PADDING:
                AES_INFO("1-0-Padding SELECTED")
                // abbiamo il messaggio che potrebbe essere anche più di 16 e partiamo da 0 con .cbegin() + tutto tranne il blocco aggiunto
                // c.begin() = posizione 0 + (tutto il messaggio - il blocco) = posizione inziale da cui vogliamo cancellare.
                decrypted_message.erase( first: decrypted_message.cbegin() + static_cast<long long>(decrypted_message.size() - aes::BLOCK_SIZE), last: decrypted_message
                break;
            case Paddings::ISO_10126_PADDING:
                AES_INFO("ISO 10126 PADDING SELECTED")
                decrypted_message.erase( first: decrypted_message.cbegin() + static_cast<long long>(decrypted_message.size() - aes::BLOCK_SIZE), last: decrypted_message
                break;
            case Paddings::NO_PADDING:
                AES_INFO("NO_PADDING SELECTED")
                break;
            default:
                AES_INFO("OTHER PADDING THAT DOESN'T REQUIRE PADDING IF THE BLOCK IS FULL = EQUAL TO 16")
                break;
        }
    } else {
```

Figura 5.23: Rimozione del padding (1/2)

```
        switch(padding) {
            case Paddings::ZERO_PADDING:
                AES_INFO("0-Padding SELECTED")
                // dall'inizio del vettore + numero posizioni del remainder - fino alla fine del vettore -> cancello padding.
                decrypted_message.erase( first: decrypted_message.cbegin() + remainder, last: decrypted_message.cend());
                break;
            case Paddings::ONE_ZERO_PADDING:
                AES_INFO("1-0-Padding SELECTED")
                decrypted_message.erase( first: decrypted_message.cbegin() + remainder, last: decrypted_message.cend());
                break;
            case Paddings::ANSI_X9_23_PADDING:
                AES_INFO("ANSI X9.23 PADDING SELECTED")
                decrypted_message.erase( first: decrypted_message.cbegin() + remainder, last: decrypted_message.cend());
                break;
            case Paddings::ISO_10126_PADDING:
                AES_INFO("ISO 10126 PADDING SELECTED")
                decrypted_message.erase( first: decrypted_message.cbegin() + remainder, last: decrypted_message.cend());
                break;
            case Paddings::PKCS7:
                AES_INFO("PKCS7 SELECTED")
                decrypted_message.erase( first: decrypted_message.cbegin() + remainder, last: decrypted_message.cend());
                break;
            default:
            case Paddings::NO_PADDING:
                AES_INFO("NO_PADDING SELECTED")
                break;
        }
    }

    return decrypted_message;
}
```

Figura 5.24: Rimozione del padding (2/2)

Per quanto riguarda la rimozione del padding, dal testo decifrato, è la stessa identica operazione dell'aggiunta, ma all'inverso, rimuovendo i bytes che avevamo aggiunto.

API

Per poter interfacciarsi con tutte queste funzioni, ne sono presenti due: *encrypt()* e *decrypt()* che richiedono entrambe il messaggio (cifrato o in chiaro), la chiave, l'eventuale IV, opzionale, perché potrebbe non essere presente se si utilizza la modalità ECB, la tipologia di padding, la modalità e infine quale tipologia di AES.

Cifratura

Nella cifratura, viene aggiunto il padding chiamando *add_padding()* al messaggio in chiaro. Dopodiché, in base alla modalità scelta, viene restituito il messaggio cifrato.

```
std::vector<uint8_t> encrypt(std::vector<uint8_t>& message, std::vector<uint8_t>& key, const std::optional<std::vector<uint8_t>>& iv,
                             const aes::pad::Paddings& padding, const aes::mod::Modes& mode, const aes::AES& aes)
{
    const std::vector<uint8_t>& message_with_padding = aes::pad::add_padding( &: message, padding);

    AES_DEBUG("unencrypted message_with_padding: {}", aes::cvt::get_string_from_vector<std::string, uint8_t>( vector: message_with_padding))

    switch(mode) {
        case mod::Modes::ECB:
            return aes::mod::encrypt_ECB( input: message_with_padding, key, aes);
        case mod::Modes::CBC:
            return aes::mod::encrypt_CBC( input: message_with_padding, key,  iv: iv.value(), aes);
        case mod::Modes::CFB:
            return aes::mod::encrypt_CFB( input: message_with_padding, key,  iv: iv.value(), aes);
        default:
            AES_CRITICAL("Error! Should not be here!")
            std::exit( Code: EXIT_FAILURE);
    }
}
```

Figura 5.25: Cifratura

Decifratura

Nella decifratura, viene decifrato il messaggio in base alla modalità selezionata e dopodiché viene restituito il messaggio dopo la rimozione del padding, se presente.

```cpp
std::vector<uint8_t> decrypt(std::vector<uint8_t>& encrypted_message, std::vector<uint8_t>& key, const std::optional<std::vector<uint8_t>>& iv,
                             const aes::pad::Paddings& padding, const aes::mod::Modes& mode, const aes::AES& aes)
{
    std::vector<uint8_t> plaintext_with_padding{};

    switch(mode) {
        case mod::Modes::ECB:
            plaintext_with_padding = aes::mod::decrypt_ECB( input: encrypted_message, key, aes);
            break;
        case mod::Modes::CBC:
            plaintext_with_padding =  aes::mod::decrypt_CBC( input: encrypted_message, key,  iv: iv.value(), aes);
            break;
        case mod::Modes::CFB:
            plaintext_with_padding = aes::mod::decrypt_CFB( input: encrypted_message, key,  iv: iv.value(), aes);
            break;
        default:
            AES_CRITICAL("Error! Should not be here!")
            std::exit( Code: EXIT_FAILURE);
    }

    AES_DEBUG("deciphered plaintext_with_padding: {}", aes::cvt::get_string_from_vector<std::string, uint8_t>( vector: plaintext_with_padding))

    return aes::pad::remove_padding( &: plaintext_with_padding, padding);
}
```

Figura 5.26: Decifratura

Implementazione in Java

Per l'implementazione in Java, mi sono avvalso di Java 11 e ho usufruito delle librerie standard.

Ho definito, innanzitutto, delle variabili membro statiche, per definire la grandezza della chiave, quale algoritmo utilizzare, quale tipo di padding usufruire e altre inerenti l'iv e la salatura della password.

```java
public final class AES {

    1 usage
    private final static String ENCRYPTION_STANDARD = "AES";
    1 usage
    private final static int KEY_SIZE = 256;
    1 usage
    private final static String TRANSFORMATION_ALGORITHM = "PBKDF2WithHmacSHA256";
    2 usages
    private final static String TRANSFORMATION_ALGORITHM_NO_PADDING = "AES/GCM/NoPadding";
    1 usage
    private final static int ITERATION_COUNTER = 65_536;
    2 usages
    private final static int IV_SIZE = 12;
    2 usages
    private final static int SALT_SIZE = 16;
    2 usages
    private final static int TAG_SIZE = 128;

    4 usages
    public final static String PASSWORD = "SECRET-CODE";

    /**
     * private constructor because the class doesn't have to be instantiated.
     */
    no usages  Luca
    private AES(){}
```

Figura 5.27: Costruttore e variabili membre

Attraverso il metodo *getRandomBytes()* otteniamo un Nonce.

```java
/**
 *
 * @param bytesNumber : number of bytes
 * We use SecureRandom to create a pseudo-random number
 * which we will use to generate a random IV and Salt.
 * IV: stands for initialization vector, it adds randomness to the start of the encryption process.
 * It may also be called nonce since it's used only once.
 * Salt: it is some bytes that gets added to the password before it goes through the hashing algorithm.
 * @return : random bytes.
 */
2 usages  Luca
public static byte[] getRandomBytes(final int bytesNumber){
    final SecureRandom secureRandom = new SecureRandom();

    final byte[] bytes = new byte[bytesNumber];
    secureRandom.nextBytes(bytes);

    return bytes;
}
```

Figura 5.28: Nonce

Col metodo *getKeyFromPassword* otteniamo la chiave dalla password assieme alla salatura.

```java
/**
 *
 * @param password: it's a user defined password.
 * @param salt: it is some bytes that gets added to the password before it goes through the hashing algorithm.
 * @return : a SecretKey
 * @throws NoSuchAlgorithmException: algorithm doesn't exist.
 * @throws InvalidKeySpecException: invalid key specifications.
 */
2 usages  Luca
public static SecretKey getKeyFromPassword(final String password, final byte[] salt) throws NoSuchAlgorithmException, InvalidKeySpecException {
    final SecretKeyFactory secretKeyFactory = SecretKeyFactory.getInstance(AES.TRANSFORMATION_ALGORITHM);

    final KeySpec keySpec = new PBEKeySpec(password.toCharArray(), salt, AES.ITERATION_COUNTER, AES.KEY_SIZE);

    return new SecretKeySpec(secretKeyFactory.generateSecret(keySpec).getEncoded(), AES.ENCRYPTION_STANDARD);
}
```

Figura 5.29: Password

Nel metodo *encrypt()* otteniamo la salatura e l'IV attraverso il metodo *getRandomBytes()* e la chiave attraverso *getKeyFromPassword()*. Dopodiché, inizializziamo il cifrario, cifriamo il messaggio e restituiamo il messaggio cifrato, assieme alla salatura e all'IV.

```java
/**
 *
 * @param message: the message, in plain text, that has to be encrypted.
 * @param password: it's a user defined password.
 * @return : the encrypted text.
 * @throws NoSuchAlgorithmException: algorithm doesn't exist.
 * @throws InvalidKeySpecException: invalid key specifications.
 * @throws NoSuchPaddingException: padding not available.
 * @throws InvalidAlgorithmParameterException: invalid or inappropriate algorithm parameters.
 * @throws InvalidKeyException: invalid key.
 * @throws IllegalBlockSizeException: provided wrong length of data to the block cipher.
 * @throws BadPaddingException: input data not properly padded.
 */
3 usages  Luca
public static byte[] encrypt(final byte[] message, final String password) throws NoSuchAlgorithmException, InvalidKeySpecException, NoSuchPaddingException, Inva

    final byte[] salt = AES.getRandomBytes(AES.SALT_SIZE);

    final byte[] iv = AES.getRandomBytes(AES.IV_SIZE);
    //TODO: byte[] iv = params.getParameterSpec(IvParameterSpec.class).getIV(); , meglio quella sotto.
    //TODO: IVParameterSpec ivSpec = new IvParameterSpec(iv);

    final SecretKey secretKey = AES.getKeyFromPassword(password, salt);

    final Cipher cipher = Cipher.getInstance(AES.TRANSFORMATION_ALGORITHM_NO_PADDING);

    cipher.init(Cipher.ENCRYPT_MODE, secretKey, new GCMParameterSpec(AES.TAG_SIZE, iv));

    final byte[] encryptMessage = cipher.doFinal(message);

    //TODO: Base64.getEncoder().encodeToString(encryptMessage);

    //TODO: ivSpec.getIv().length
    return ByteBuffer.allocate( capacity: iv.length + salt.length + encryptMessage.length).put(iv).put(salt).put(encryptMessage).array();
}
```

Figura 5.30: Cifratura

Nel metodo *decrypt()* otteniamo l'IV e il sale dal messaggio e recuperiamo la password, dopodiché inizializziamo il cifrario e decifriamo il messaggio.

```java
/**
 *
 * @param message: the encrypted text that has to be decrypted.
 * @param password the user defined password. (it has to match with the one used in the encryption)
 * @return : the decrypted text.
 * @throws NoSuchAlgorithmException: algorithm doesn't exist
 * @throws InvalidKeySpecException: invalid key specifications.
 * @throws NoSuchPaddingException: padding not available.
 * @throws InvalidAlgorithmParameterException: invalid or inappropriate algorithm parameters.
 * @throws InvalidKeyException: invalid key.
 * @throws IllegalBlockSizeException: provided wrong length of data to the block cipher.
 * @throws BadPaddingException: input data not properly padded.
 */
3 usages  Luca
public static byte[] decrypt(final byte[] message, final String password) throws NoSuchAlgorithmException, InvalidKeySpecException, NoSuchPaddingException, Inva
    final ByteBuffer byteBuffer = ByteBuffer.wrap(message);

    final byte[] iv = new byte[AES.IV_SIZE];
    byteBuffer.get(iv);

    final byte[] salt = new byte[AES.SALT_SIZE];
    byteBuffer.get(salt);

    //TODO: byte[] bytes = message.digest(password.getBytes(StandardCharsets.UTF_8));

    final byte[] encryptedMessage = new byte[byteBuffer.remaining()];
    byteBuffer.get(encryptedMessage);

    final SecretKey secretKey = AES.getKeyFromPassword(password, salt);

    final Cipher cipher = Cipher.getInstance(AES.TRANSFORMATION_ALGORITHM_NO_PADDING);

    cipher.init(Cipher.DECRYPT_MODE, secretKey, new GCMParameterSpec(AES.TAG_SIZE, iv));

    return cipher.doFinal(encryptedMessage);
}
```

Figura 5.31: Decifratura

In *encryptFile* ci avvaliamo del metodo *encrypt* per cifrare. Leggiamo i dati presenti nel file *inputFilePath* e scriviamo il testo cifrato nel file *outputFilePath*.

```java
/**
 *
 * @param inputFilePath: the path of the file with the plain text.
 * @param outputFilePath: the file with the encrypted text that will be created.
 * @param password: the user-defined password.
 * @throws IOException: fail or interrupted I/O operations.
 * @throws NoSuchAlgorithmException: algorithm doesn't exist
 * @throws InvalidKeySpecException: invalid key specifications.
 * @throws NoSuchPaddingException: padding not available.
 * @throws InvalidAlgorithmParameterException: invalid or inappropriate algorithm parameters.
 * @throws InvalidKeyException: invalid key.
 * @throws IllegalBlockSizeException: provided wrong length of data to the block cipher.
 * @throws BadPaddingException: input data not properly padded.
 */
1 usage  Luca
public static void encryptFile(final String inputFilePath, final String outputFilePath, final String password) throws NoSuchAlgorithmException, InvalidKeySpecEx

    final byte[] message = Files.readAllBytes(Paths.get(inputFilePath));

    final byte[] encryptedMessage = AES.encrypt(message, password);

    final Path filePath = Paths.get(outputFilePath);

    Files.write(filePath, encryptedMessage);
}
```

Figura 5.32: Funziona per la cifratura di un File

Per *decryptFile* adoperiamo il metodo *decrypt* per decifrare il messaggio cifrato presente nella path indicata dalla variabile *encryptedFilePath*.

```java
/**
 *
 * @param encryptedFilePath: the path of the file with the encrypted text.
 * @param password: the user-defined password. (it has to match with the password used in the encryption)
 * @return : the decrypted text in plain text.
 * @throws IOException: fail or interrupted I/O operations.
 * @throws NoSuchAlgorithmException: algorithm doesn't exist
 * @throws InvalidKeySpecException: invalid key specifications.
 * @throws NoSuchPaddingException: padding not available.
 * @throws InvalidAlgorithmParameterException: invalid or inappropriate algorithm parameters.
 * @throws InvalidKeyException: invalid key.
 * @throws IllegalBlockSizeException: provided wrong length of data to the block cipher.
 * @throws BadPaddingException: input data not properly padded.
 */
1 usage  Luca
public static byte[] decryptFile(final String encryptedFilePath, final String password) throws NoSuchAlgorithmException, InvalidKeySpecException, NoSuchPadding

    final byte[] encryptedMessage = Files.readAllBytes(Paths.get(encryptedFilePath));

    return AES.decrypt(encryptedMessage, password);
}
```

Figura 5.33: Funziona per la decifrazione di un File