Capitolo 1

AES ~ Matematica di base

Per introdurre tutte quelle dinamiche che stanno alla base del funzionamento dell'algoritmo crittografico noto con il nome di AES, si incomincerà con lo studio della sua matematica più elementare.
Innanzitutto, AES è un algoritmo standard per la cifratura dei dischi (es: hard disk), si tratta di un cifrario a blocchi conosciuto, all'inizio, come *Rijndael*. AES lavora con blocchi di dimensione fissa (128 bit) e la chiave può essere lunga, a seconda delle esigenze, 128/192/256 bit.
La sua matematica di base si fonda sui campi di Galois.

Operazione binaria. Dicesi operazione binaria su un insieme S non vuoto una funzione **f** che prende come suo *dominio* (quindi come elementi in ingresso a tale funzione) tutti gli elementi dell'insieme S e torna (quindi identifica come suo personale *codomino*) un qualsiasi altro elemento dello stesso insieme S.

Esempio.
Operatore Somma:
$+: S \rightarrow S$
Si prende un elemento dell'insieme S (che coinciderà quindi con l'identificativo S), lo si somma ad un altro elemento di S e si ottiene un terzo elemento dell'insieme S. L'elemento di S non dev'essere per forza un numero, può essere anche, per esempio, una qualsiasi matrice di dimensione a x b.
La funzione f potrebbe essere qualsiasi cosa, così è desiderabile darle un nome per specificarla:
$\Delta : S \rightarrow S$
allora si può riscrivere come:
$S_1 \ \Delta \ S_2 = S_1 + S_1^2 S_2$
Tutte le funzioni che andremo a vedere non saranno così semplici come l'operatore di somma ma saranno tutte del tipo Δ .

1.1 Anello

Un anello matematico è costituito nel suo complesso da una quantità di operazioni binarie, nel nostro caso 2: somma e moltiplicazione. Presi 3 valori esemplificativi come a, b, c $\in$ R si definisce un certo insieme di proprietà.
Sia allora
R = ring (anello) con
R, + (somma), **x** (moltiplicazione), **a, b, c** $\in$ **R**
Un anello vanta le seguenti proprietà:

→ *Commutatività dell'addizione*: `a+b = b+a`
→ *Associatività dell'addizione*: `(a+b)+c = a+(b+c)`
→ *Identità additiva*, esiste quindi un elemento neutro rispetto alla somma, ovvero lo zero (0)
→ *Inverso additivo*: `-a`
→ *Commutatività della moltiplicazione*: `axb = bxa`
→ *Associatività della moltiplicazione*: `(axb)xc = ax(bxc)`
→ *Identità moltiplicativa*, ovvero esiste un elemento neutro rispetto alla moltiplicazione che è l'uno (1)
→ *Proprietà distributiva*: `ax(b+c) = axb + axc`
Un insieme che vanta tutte queste proprietà è detto *anello*.
Quanto appena riportato è tutto ciò che un informatico definisce come anello (non è detto infatti che la stessa definizione la dia un matematico oppure un fisico, perchè si sono omessi alcuni dettagli, in questo contesto, di poco conto).

Come detto prima, le definizioni di un anello possono essere molteplici: alcuni preferiscono eliminare rispettivamente le proprietà di commutatività della moltiplicazione e di identità moltiplicativa: se alla definizione così ottenuta si vuole aggiungere la proprietà di commutatività della moltiplicazione si ottiene la definizione di un anello commutativo mentre se si aggiunge anche la proprietà dell'identità moltiplicativa si ottiene un anello commutativo con identità.

Si può definire quindi qual'è l'*operazione di sottrazione* sull'anello: basta sommare l'inverso additivo all'elemento in questione → `a+(-b)`.
Allo stesso modo si definisce l'*operazione di divisione sull'anello*, come la moltiplicazione per l'inverso moltiplicativo → $a \times b^{-1}$.

Abbiamo parlato di un inverso moltiplicativo, ma esiste sempre questo inverso su qualsiasi anello R? Ovviamente no, infatti a partire da un anello R non sempre esiste un R* di inversi moltiplicativi degli elementi di R.
Siano **a** e **b** due elementi qualunque appartenenti ad R, ma **b** ha anche l'inverso moltiplicativo (R* sarà un insieme di elementi con inverso moltiplicativo, sottoinsieme di R), quindi appartiene anche ad R* → $a,b \in R$ e contemporaneamente $b \in R^*$.

Chiusura rispetto alla moltiplicazione. Tipica proprietà di un insieme R* di inversi moltiplicativi, se $a,b \in R^*$ allora si ottiene che anche $a \times b \in R^*$ (quindi se due elementi appartengono a R* allora anche il loro prodotto appartiene all'insieme degli inversi).
Allora se $a \in R^*$ significa che $a^{-1} \in R^*$, allora qualsiasi identità moltiplicativa appartiene a $R^* \rightarrow 1 \in R^*$.
Queste sono tutte proprietà che appartengono all'anello.

Esempio. Si prende un ins. di numeri `mod 5 = {0,1,2,3,4}` → Z_5. Guardiamo se possiedono o meno un inverso moltiplicativo.

```
1 x y = 1 mod 5 → y = 1
2 x y = 1 mod 5 → y = 3
3 x y = 1 mod 5 → y = 2
4 x y = 1 mod 5 → y = 4
```

Si nota che tutti gli elementi in Z_5 possiedono in qualche modo un inverso moltiplicativo. L'identità moltiplicativa appartiene a Z_5 e, se si guarda, anche la proprietà dell'identità additiva è verificata.
In questo specifico caso R = R*.
Si ricorda che in generale tutti gli elementi/numeri hanno un inverso, salvo lo zero.

Esempio. Si ripete l'esempio precedente, ma con Z_{10}, quindi con un insieme di numeri in `mod 10 = {0,1,2,3,4,5,6,7,8,9}`.

```
1 x y = 1 mod 10 → y = 1
2 x y = 1 mod 10 → non esiste
3 x y = 1 mod 10 → y = 7 → MCD (7, 10) = 1
4 x y = 1 mod 10 → non esiste
5 x y = 1 mod 10 → non esiste
6 x y = 1 mod 10 → non esiste
7 x y = 1 mod 10 → y = 3
8 x y = 1 mod 10 → non esiste
9 x y = 1 mod 10 → y = 9
```

Il "trucco" per calcolare y secondo il modulo dato è quello di calcolare il MCD basandosi sul modulo, così da ottenere, come risultato, 1; in questa maniera è possibile calcolare con una certa immediatezza l'inverso moltiplicativo.
Si nota che Z_{10} è sì un anello ma non tutti gli elementi che lo compongono possiedono un inverso moltiplicativo, infatti lo hanno solamente {1,3,7,9} che quindi appartengono anche a $R^* = Z_{10}^*$.
È evidente che l'identità moltiplicativa è comunque presente.
Per effettuare la moltiplicazione si controlla dapprima che il risultato sia congruo nel modulo rispetto agli elementi appartenenti a R*: in questo caso l'*anello è chiuso rispetto alla moltiplicazione*.

Anello intelligente. Quando si lavora in $Z_p=Z_p^*$ tutti gli elementi risultano essere coprimi → MCD (a, b)=1 con a,b in R e primi, dove **p** è un numero primo, allora tutti i numeri primi sono ovviamente coprimi tra loro ed avranno sicuramente inverso moltiplicativo. A priori si può già intuire nel caso in cui R=R* e non nel caso in cui R sia diverso da R* dato che ogni suo elemento avrà sicuramente un inverso moltiplicativo.

Divisione per zero. Il computer normalmente lavora in aritmetica finita, quindi su un anello, allora è sempre possibile dividere un numero per qualsiasi altro. Lavorando in Z_n se **n** non risulta essere un numero primo bisogna fare attenzione che l'MCD sia sempre uguale a 1, cosa altrimenti trascurabile nel momento in cui n sia

assodato essere un numero primo. La macchina comunque non tratta numeri veri e propri ma multipli di byte (quindi multipli di 8 bit) che, però, non sono numeri primi → i registri da 64 bit (2^{64}) non sono numeri primi. In questo caso l'importanza del MCD uguale a 1 diventa fondamentale.

1.2 Campo

Adesso si può passare dagli anelli ai campi, ovvero F: R → F. Un campo F è definibile come un anello nel quale ogni elemento diverso da zero (quindi non nullo) è invertibile.

Esempio. Un *campo infinito* → R: l'inverso di $\sqrt{(2)}$ è 1/ $\sqrt{(2)}$.
Un altro esempio è dato dai numeri razionali Q.
Entrambi sono esempi pratici di campi infiniti dove ogni elemento è invertibile.

All'informatica interessano però i *campi finiti* → Z_p. Allora si può dire che tutti gli elementi sono invertibili (salvo lo zero, proprietà che appartiene a qualsiasi campo), quindi il *massimo comun divisore* sarà uguale a 1 → MCD (**a**,**p**)=1 con **a** ∈ F, **a** ∈ Z_p.
Ogni campo finito F = Z_p che si rispetti possiede un numero di elementi che lo caratterizzano. F ha almeno due elementi, quindi esiste un numero primo **p** ed un numero naturale **n** tali che il numero complessivo di elementi del campo è dato da $\mathbf{p^n}$ → |F| = p^n dove |F| è la cardinalità del campo F.
Allora si può dire che $5 = p^n = 5^1$.
E' vero anche che per ogni **p** ed **n** esiste un campo finito F che identifica quegli stessi elementi, detto campo di Galois → GF(p^n).
Tra le altre cose, si segnala che Galois è morto a 21 anni.
Il computer solitamente non lavora con numeri primi ,ma con i byte. Ha quindi a disposizione 256 elementi e non 251. La macchina lavora sempre con multipli di 2 *alla qualcosa*, così si sfruttano i campi di Galois: è questa la ragione per la quale il computer esegue dei conti nell'aritmetica finita, ed è per questo che ogni elemento ha anche il suo inverso.

Esempio. Si considera un insieme F formato da alcune stringhe di bit. Si definiscono allora, di conseguenza, due operazioni binarie: lo XOR (equivalente alla *somma*) ed X (equivalente al *prodotto*).
Si stabilisce che:

```
10 x 10 = 11
11 x 11 = 10
10 x 11 = 01
```

con `F = {00, 01, 10, 11}`. A questo punto si vuole generare la tabella delle operazioni per calcolare tutte le combinazioni degli

elementi appartenenti al campo.

XOR	00	01	10	11
00	00	01	10	11
01	01	00	11	10
10	10	11	00	01
11	11	10	01	00

X	00	01	10	11
00	00	00	00	00
01	00	01	10	11
10	00	10	*11*	*01*
11	00	11	*01*	*10*

Per completare la tabella relativa alla moltiplicazione (X) prima è opportuno inserire i risultati delle operazioni stabilite a priori, poi si può terminare facendo i conti secondo il senso comune.
Ora bisogna dimostrare che quello appena individuato sia davvero un campo. Si parte dal verificare la proprietà commutativa → la tabella dell'addizione è visibilmente simmetrica rispetto alla sua diagonale. Si può verificare la commutatività anche all'interno della tabella della moltiplicazione per lo stesso motivo di cui sopra.
Proprietà associativa rispetto al prodotto, perchè rispetto alla somma la proprietà è già visibile → provo tutte le possibili combinazioni: per esempio...

```
(10 x 10) x 11 = 10
10 x (10 x 11) = 10
```

Identità additiva → esiste e coincide con il valore 00.
Identità moltiplicativa → esiste e coincide con il valore 01 (si deduce benissimo da entrambe le tabelle), infatti nelle tabelle, dove si trova il valore 00 non ci sono particolari conti da effettuare, esattamente come quando si identifica il valore 01. Gli unici conti da fare sono rispettivamente sui valori 10, 11 e su tutte le loro combinazioni.
Le **tabelle di verità** aiutano in questi conti e si consiglia la loro realizzazione in merito ad ogni campo, per velocizzare le operazioni.

1.3 Polinomi irriducibili

AES lavora utilizzando i cosiddetti polinomi irriducibili, basandosi esso stesso su un certo campo di Galois. Si indica con il termine polinomio quell'espressione i cui coefficienti si trovano tutti su di un campo oppure su di un anello. In questo caso interessano tutti quei polinomi con coefficienti in $Z_p[x]$, ovvero si tratta di polinomi i cui coefficienti si individuano nel campo Z_p.
Dato un numero primo **p** ed un qualsiasi polinomio:
p → primo

$a_i \in Z_p \rightarrow f = a_n x^n + a_{n+1} x^{n+1} + \ldots + a_1 x^1 + a_0 = \sum_{i=0}^{n} a_i * x^i$

Il polinomio nullo ha coefficienti pari a zero.
Polinomi uguali possiedono stessi coefficienti.

La somma tra polinomi corrisponde alla somma tra i loro coefficienti (quindi non è necessario che siano entrambi della stessa lunghezza). Il coefficiente massimo corrisponde al massimo tra i vari coefficienti.

$$g=\sum_{i=0}^{n} b_i * x^i \qquad f=\sum_{i=0}^{n} a_i * x^i \qquad f+g=\sum_{i=0}^{(max(n,m))} c_i * x^i \quad c_i = a_i+b_i \bmod p$$

La moltiplicazione è resa da:

$$g*f=\sum_{i=0}^{n+m} d_i * x^i \quad \text{dove} \quad d_i=\sum_{j=0}^{i} a_j * b_{(i-j)} \bmod p$$

Esempio. $Z_5[x]$

$(x^5+4x^4+2)\ (3x^2+2x) =$
$= 3x^7+12x^6+6x^2+2x^6+8x^5+4x \pmod 5 =$
$= 3x^7+2x^6+x^2+2x^6+3x^5+4x$

Si ricorda che i coefficienti vanno normalizzati con il modulo!
Un qualsiasi polinomio può essere visto come un array/vettore le cui celle contengono i coefficienti in base all'indice/grado.
Nel nostro caso $Z_5[x]$ è un anello di polinomi con tutti i coefficienti nel campo Z_5.
Polinomio di grado 0 → corrisponde all'identità additiva;
Polinomio di grado 1 → corrisponde all'identità moltiplicativa.
Date queste indicazioni è possibile dimostrare se il polinomio preso in esame rispetta o meno le proprietà del campo.

Se p è un numero primo, $Z_p[x]$ è un anello di polinomi con coefficienti tutti nel campo Z_p. Si può dimostrare ogni proprietà dei campi e degli anelli anche sui polinomi.

Divisibilità tra polinomi di $Z_p[x]$. Si studia per quale **p** (se esiste) è vero che il polinomio dato $(x+1)$ divide il secondo polinomio (x^2-1) in $Z_p[x]$.
Risulta possibile solo se p < 10.
Si parte dalla scomposizione di $(x^2-1)=(x+1)(x-1)$, allora si guarda se a partire dalla nuova scomposizione il polinomio $(x+1)$ potrebbe o meno dividerla. Si capisce che $(x+1)$ lo divide per ogni **p** purché minore di 10.
Se $(x+1)$ divide (x^2-1) allora si ha che $(x^2+1) = f+(x+1)$, quindi si deve studiare com'è fatto il polinomio **f**. f potrebbe essere al massimo di grado 1 → $ax+b$.
$(ax+b)(x+1) = ax^2+bx+ax+b = ax^2+(a+b)x+b$
Si impone a=1 perchè $ax^2=1$ e $b=1$, così $(b+a)=0$ per rendere possibile l'annullamento. Sarà uguale a zero in Z_p solamente per $a+b=2 \bmod 2$.
In alcuni casi con i polinomi si lavora tranquillamente, altre volte vanno scomposti per forza.

Divisione tra polinomi. Siano **g(x)** e **f(x)** due polinomi, si divide allora f(x) per g(x), si otterrà quindi un quoziente **q(x)** ed un resto

r(x). Il tutto va eseguito in un anello di polinomi.

f(x)	g(x)
r(x)	q(x)

Esempio. $Z_2[x] \rightarrow$ mod 2

x^4+x^3+1	x^2+1
$-x^4-x^2$	x^2+x-1
x^3-x^2+1	
$-x^3-x$	
$-x^2-x+1$	
x^2+1	
$-x$	

r(x) sarebbe $-x+2 \mod 2$ allora va normalizzato a $-x$.
Non bisogna mai dimenticare di considerare il modulo.

Il grado di f(x)g(x) è dato dalla somma dei rispettivi gradi.
dig (f(x))+dig(g(x)) = dig (f(x)g(x))

Congruenza tra polinomi. Siamo partiti da un *anello infinito* per arrivare ad un *anello finito*, quindi siamo passati ad un *campo finito* → *Campo di Galois GF(p^n).*
Si passa poi direttamente ai campi di polinomi: con i coefficienti di un polinomio è possibile rappresentare un byte perchè tali anelli possiedono dei coefficienti che si trovano fisicamente su un campo.
La congruenza tra polinomi è data da:
f, g → polinomi
f, g, m su $Z_p[x]$
allora f(x) = g(x) mod m(x) se e solo se m(x)|(f(x)-g(x))

Esempio. $Z_2[x] \rightarrow x^5+x = x^2+1 \mod (x^3+x+1)$
Si effettua la divisione tra polinomi, tenendo conto che il dividendo è formato da g(x)+f(x) dove si tenta di provare la congruenza g(x)=f(x). Il divisore è dato invece dal modulo della congruenza m(x).

x^5+x^2+x+1	x^3+x+1
$-x^5-x^3-x^2$	x^2-1
$-x^3+x+1$	
$-x^3+x+1$	
$x+1$	

Si può dedurre quindi se i due polinomi sono o meno congruenti.

Stabilisco un polinomio $m(x)$ con un certo grado indicato $dig(m(x))$. Con la dicitura $Z_p[x]\ mod(m(x))$ indico la creazione di un *campo finito di polinomi*. Scelgo allora un polinomio irriducibile $m'(x)$ e costruisco $Z_p[x]\ mod\ (m'(x))$ e lo si può indicare semplicemente come Z_p. Individuo allora un campo di Galois $GF(p^{dig(m(x))})$.
$Z \rightarrow Z_p[x]$
$Z_m \rightarrow Z_p[x]$ mod (m(x)) con m → dig(m(x))
$Z_p \rightarrow Z_p[x]$ mod (m'(x)) con m'(x) irriducibile
$GF(p^m) \rightarrow GF(p^{dig(m(x))})$
Voglio sostituire un campo di polinomi contenente esattamente un certo numero di elementi, così basta scegliere un polinomio irriducibile $m(x)$ su $Z_p[x]$: $dig\ (m|x) = n$ tale che il suo grado sia uguale ad un certo n corretto, corrispondente al numero di elementi del campo.

Esercizio. Si vuole calcolare $GF(4)$. Si definisce un modo univoco con $p=2$ e $m=2 \rightarrow GF(p^m)=GF(2^2)=GF(4)$
Si calcola il polinomio m(x) irriducibile su Z_2, allora i polinomi di questo tipo di grado 2 sono: x^2, x^2+1, x^2+x e x^2+x+1. Quali tra tutti questi sono davvero irriducibili?

$m(x)$ {
- $x^2 \rightarrow (x)(x)$
- $x^2+1 \rightarrow (x+1)(x+1) = x^2+2x+1$ ma si elimina 2x perchè in Z_2
- $x^2+x \rightarrow x(x+1)$
- $x^2+x+1 \rightarrow$ irriducibile!

Ora si devono calcolare tutte le operazioni possibili sui campi del polinomio e le relative tabelle di verità.

SUM	**0**	**1**	**x**	**x+1**
0	0	1	x	x+1
1	1	0	x+1	x
x	x	x+1	0	1
x+1	x+1	x	1	0

MUL	**0**	**1**	**x**	**x+1**
0	0	0	0	0
1	0	1	x	x+1
x	0	x	x+1	1
x+1	0	x+1	1	x

Si ricorda che, poiché ci si trova in Z_2, bisogna ragionare in mod 2.
Tutti i polinomi scelti devono possedere un grado minore di quello di $m(x)$ per essere accettati.
$(x)(x) = x^2$ allora x^2: x^2+x+1 perchè il grado dev'essere minore a quello di $m(x)$. Ogni riga deve contenere tutto il set di valori, altrimenti il campo non può contenere inversi (e deve per forza contenerli).

Esercizio. $m(x) = x^4+x+1 \rightarrow GF(2^4) \rightarrow Z_2[x]$

Si controlla che $m(x)$ sia un polinomio irriducibile, allora si prova a dividerlo per tutti gli elementi più piccoli che compongono il polinomio, ma non proprio tutti: si scelgono solo i più "intelligenti".
Facendo i conti si nota che $m(x)$ non è affatto l'unico polinomio irriducibile del campo GF(2^4) di grado 4. Ce ne sono alcuni altri, allora quale scegliere come polinomio irriducibile definitivo? La scelta è assolutamente indifferente perchè a ciascun polinomio irriducibile si associa un campo di Galois adeguato, tuttavia è noto che ogni polinomio ha un solo campo di Galois associato → questo significa che i campi individuati sono isomorfi, ovvero che si tratta sempre degli stessi campi, solo con alcuni elementi spostati di posto.

1.4 Teoria dei polinomi in AES

AES possiede, alla sua base, un polinomio irriducibile m(x), tuttavia non è un polinomio unico dal momento che in GF(2^8) che è il campo di Galois di AES esistono più polinomi irriducibili (ce ne sono addirittura una ventina tant'è che è stato scelto per AES in maniera assolutamente casuale). m(x) è comodo sceglierlo in maniera tale da semplificare i conti, così da renderli il più immediati possibile.

Esempio. $m(x)=x^8+x^4+x^3+x+1$ → Si lavora in $Z_2[x]$
Si sceglie di fare il prodotto tra $(x^6+x+1)(x^4+x)$ per esemplificare la moltiplicazione che AES compie tra le parole (in questo caso ogni polinomio corrisponde ad una parola di AES).
Si ottiene $x^{10}+x^5+x^4+x^7+x^2+x$ ma x^{10} non va bene perchè il grado di questo polinomio risulta essere maggiore di quello di m(x), allora va ridotto in modulo x^8 (e per farlo basta dividere il polinomio ottenuto per il polinomio m(x) considerando solo il resto della divisione e non il risultato), ottenendo così:
$(x^6+x^5+x^3+x^2)+x^5+x^4+x^7+x^2+x = x^7+x^6+x^4+x^3+x$
La somma è effettuata tramite uno XOR bit a bit: equivale a sommare normalmente i polinomi ed a calcolare il modulo solo alla fine. Il binario è ottenuto tramite il polinomio dove ogni 1 corrisponde alla posizione indicata dagli esponenti delle x.

Calcolo dell'inverso di un polinomio. Si prende come esempio il polinomio $x^7+x^6+x^5+x^4+x^2=a(x)$ e si calcola un MCD (minimo comune multiplo) tra questo polinomio ed il polinomio irriducibile di AES. Approfittando dell'occasione, si calcola anche l'inverso del polinomio a(x). Per far ciò si sfruttano gli stessi meccanismi degli anelli: come avviene con qualsiasi altro numero che viene scomposto nei numeri primi che lo costituiscono, la stessa cosa vale per i polinomi che vengono anch'essi scomposti e si prenderà allora il massimo numero (o nel nostro caso polinomio) che scompone entrambi i polinomi come MCD. Si sfrutta a questo scopo l'*algoritmo di Euclide* perchè

potrebbe capitare di avere a che fare con dei valori piuttosto grandi, difficilmente gestibili in altra maniera.

$m(x)=a(x)q(x)+r(x) \rightarrow a(x)=r_1(x)q_2(x)+r_2(x)$

$x^8+x^4+x^3+x+1=(x^7+x^6+x^5+x^4+x^2)(x+1)+(x^2+x+1)$

E si va avanti nei conti e nei passaggi come nel passo generico dello algoritmo di Euclide fino agli ultimi due passaggi:

$x^7+x^6+x^5+x^4+x^2=(x^2+x+1)(x^5+x^2+x+1)+1$

$x^2+x+1=1(x^2+x+1)+0$

Quando $r_2(x)=0$ l'algoritmo si ferma. L'MCD è costituito dall'ultimo resto diverso da zero della funzione di terminazione, in questo caso corrispondente a 1.

Per trovare l'inverso, invece, si sfrutta l'*algoritmo di Euclide esteso*. Si parte da $r_2(x)=a(x)-r_1(x)q_2(x)$ allora $r_1(x)=m(x)-a(x)q_1(x)$ e non si fa altro che eseguire tutti i conti:

$1=a(x)+q_2(x)[m(x)+q_1(x)a(x)]$

$1=a(x)[1+q_2(x)q_1(1)]+m(x)q_2(x)$

ed è chiaro che $[1+q_2(x)q_1(1)]+m(x)q_2(x)$ essendo il polinomio che moltiplicato per a(x) da 1 è senz'altro il **polinomio inverso** di a(x).

Il computer utilizza diversi trucchetti per effettuare questi conti. Tuttavia lo fa in maniera intelligente.

Moltiplicazione di polinomi. Si prende sempre in considerazione il polinomio irriducibile di AES e si lavora ancora nel campo di Galois di AES → GF(2^8).

$m(x)=x^8+x^4+x^3+x+1$

I futuri conti che si andranno ad effettuare si baseranno su questa osservazione: $x^8 \bmod m(x) = x^4+x^3+x+1$

In generale:

$m(x)-x^8 \rightarrow m(x)-x^n$ solo se m ha grado $n \rightarrow dig(m(x))=n$

Allora la moltiplicazione tra polinomi avviene così:

$f(x)=b_7x^7+b_6x^6+b_5x^5+...+b_0x^0$

$(x)(f(x))= b_7x^8+b_6x^7+b_5x^6+...+b_0x^1$

Moltiplicare per x un polinomio corrisponde allo shiftare a sinistra di un passo. Se $b_7=0$ non ci sono problemi. Prima della moltiplicazione ci si trovava nel campo di Galois GF(2^8) e dopo l'operazione ci si trova ancora in quel campo. Se $b_7=1$ da b_6 in avanti il polinomio va riportato tale e quale. Bisogna, tuttavia, gestire x^8 perchè il resto del polinomio è fondamentalmente di grado inferiore a x^8 (che è il polinomio irriducibile di AES nel campo di Galois di AES nel quale si sta lavorando) e non desta alcuna preoccupazione. Per trattarlo è necessario dividerlo scomponendolo e guardarne il resto prodotto tramite l'algoritmo di Euclide, tuttavia basandosi sull'osservazione fatta all'inizio è sufficiente eliminare il coefficiente di grado massimo per risolvere il problema.

La moltiplicazione tra polinomi altro non è che uno shift combinato con una somma. Dopotutto la somma come operazione algebrica costa assai meno in risorse computazionali di una qualsiasi moltiplicazione.

Ora, invece di moltiplicare tutto il polinomio per un semplice x, si sceglie di farlo per un polinomio leggermente più complesso.
$f(x)=x^6+x^4+x^2+x+1$
$g(x)=x^7+x+1$
$m(x)=x^8+x^4+x^3+x+1$ → polinomio irriducibile di AES (perchè si lavora sempre nel campo di Galois di AES)
Allora inizio moltiplicando f(x) per g(x):
$f(x)g(x)$ → $f(x)x = 10101110$
$f(x)x^2 = 01011100 \text{ XOR } 00011011 = 0100111$
$f(x)x^3 = \ldots$
…
$f(x)x^7 = \ldots$
Eseguo quindi un passo di pre-computazione nel quale si rappresenta il polinomio tramite un vettore da b_7 fino a b_0. La prima volta è un semplice shift mentre la seconda risulta dalla moltiplicazione della prima ancora per x: in pratica si moltiplica f(x) per ogni x^n con n pari al grado del polinomio g(x). In seguito, per ogni moltiplicazione, si XORa il risultato con la parte restante del polinomio m(x); il tutto si ripete per ogni termine.
$f(x)g(x) \rightarrow f(x)[x^7+x+1]=f(x)x^7+f(x)x+f(x)+1$
Poichè la pre-computazione è già stata eseguita per ogni x da 0 a n, basta dedicarsi alle somme per completare il calcolo.

Generatore di un campo. Si definisce generatore di un campo un elemento che genera a sua volta tutti gli elementi del campo elevato per tutti gli elementi di quel campo. Un elemento è detto a sua volta radice se si tratta dell'elemento radice di m(x) dove il termine *radice* assume il significato di *soluzione*. Un elemento radice può essere un elemento generatore a tutti gli effetti.
Come sempre, si guarda un campo di Galois $GF(2^3)=GF(8)$; il suo polinomio irriducibile è $m(x)=x^3+x+1$. Si dovrà cercare una radice tale per cui $m(\ldots)=0$ dove il "qualche cosa" viene introdotto a sua volta da una radice (che coincide con quella che si sta cercando) detta elemento generatore → $m(g)=0$.
$g^3+g+1=0$ allora in Z_2 → $g^3 = g+1$
Si calcola $g^4 \rightarrow g^4 = g(g^3) = g(g+1) = g^2+g$
In questo caso g è una radice puramente inventata, detta *radice immaginaria,* come può accadere nel caso di numeri complessi, dopotutto non importa quanto effettivamente valga g.
$g^5 = g(g^4) = g(g^2+g) = g^3+g^2 = g^2+g+1$
Ed in questa maniera si costruiscono pian piano tutti i polinomi:
$g^6 = g^3+g^2+g = g^2+g+g+1 = g^2+1$
$g^7 = g(g^6) = g(g^2+1) = g^3+g = g+g+1 = 1$
In entrambe le ultime operazioni 2g è stato eliminato dal risultato finale poiché non bisogna dimenticare che si sta lavorando in Z_2 e che ad ogni operazione va sempre imposto il mod2!!
il generatore ha quindi generato tutte le potenze del campo e se si continuasse a calcolare si genererebbero nuovamente tutte le potenze

del campo, ancora ed ancora.
Ma è possibile velocizzare i conti? Si compia e si calcoli, per esempio, questa operazione:
$(x^2+x)(x^2+1)$
Da leggersi come:
$(g^2+g)(g^2+1)= g^4g^6= g^{10}$
ma se l'esponente oltrepassa la cardinalità del campo di Galois (e poiché il campo ha ordine 7 l'esponente lo oltrepassa eccome) si prende l'esponente e lo si mette in modulo rispetto al grado:
$g^{10 \bmod 7} = g^3 = g+1$
Il conto, se effettuato con i polinomi di partenza, porterà allo stesso identico risultato. In questa maniera la moltiplicazione è ridotta alla somma tra esponenti.
Si ricordi che g è scelta come una radice di un polinomio irriducibile.

Capitolo 2

Crittoanalisi Differenziale

Come già accennato in precedenza, il polinomio irriducibile di AES è stato scelto a caso tra una quantità di polinomi irriducibili tutti associati al campo di Galois GF(2^8), direttamente dalle pagine di un libro di algebra.
Keying material → generazione di una serie di chiavi a partire da una sola che viene divisa in 4 sottoparti (dette *parole*) sulle quali si basano tutte le altre chiavi. Le parole in questione sono le prime di ogni colonna della tabella nella quale è stata schematizzata la chiave primaria.

Attacchi a DES. DES è un cifrario a blocchi, antecedente a AES, da diverso tempo attaccabile con successo. Una volta non si conoscevano la natura ed il funzionamento della funzione di creazione degli S-Box, tant'è che si pensava che qualcuno di influente ci avesse messo le mani modellandole a proprio piacimento per certi scopi poco chiari. La **crittoanalisi differenziale** sfrutta le debolezze degli S-Box di DES.
Così avviene la creazione di un S-Box.

y\x	0	1	2	3	4	…	F
0	00	01	02	03	04	…	0F
1	10	11	12	13	14	…	1F
2	20	21	22	23	24	…	2F
3	30	31	32	33	34	…	3F
4	40	41	42	43	44	…	4F
…	…	…	…	…	…	…	…
F	F0	F1	F2	F3	F4	…	FF

Si calcola il vettore **b'** che corrisponde all'elemento dell'S-Box in una data posizione (per esempio, nella posizione **xy**).
Si prende una funzione non lineare → $b'_{xy}=xb+c$ questa sarà la funzione che definirà il valore di ogni elemento dell'S-Box.
Secondo quanto indicato nella funzione, si prende l'elemento in posizione xy e se ne calcola l'inverso moltiplicativo in GF(2^8) facendo riferimento al polinomio m(x) di AES, e lo si chiama **b**. Si moltiplica quindi b per una certa matrice **x** predefinita costituita da soli 0 e 1 e quindi per una costante **c**=63 (esadecimale).

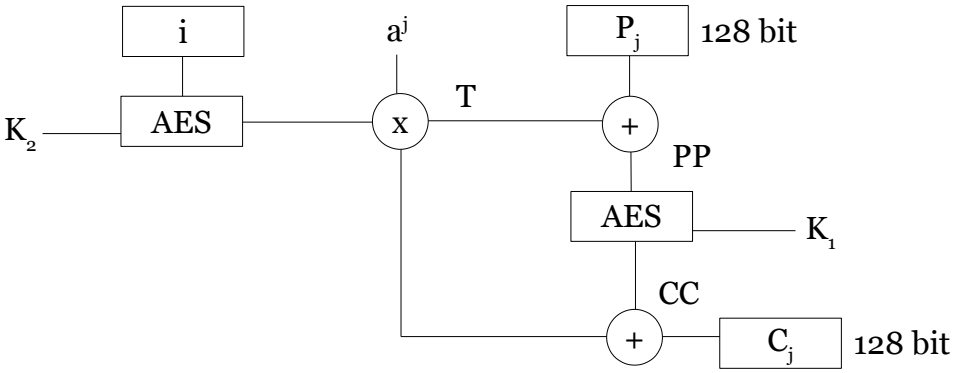
Esempio. `x=9, y=5 → xy=95`
Allora si scrivono 9 e 5 in binario poiché gli 8 bit che ne risulteranno indicheranno i vari coefficienti di un certo polinomio in GF(2^8).
`[9][5] → 4bit + 4bit → 1001 0101 →` $f(x)=x^7+x^4+x^2+1$
Si calcola allora l'inverso moltiplicativo di f(x) → $f^{-1}(x)$ in questa maniera $f(x)f^{-1}(x)=1 \mod g(x)$ sfruttando come sempre l'algoritmo di Euclide esteso.
Si ottiene allora un inverso moltiplicativo che corrisponde a b. La matrice x ha dimensione 8x8 ed il tutto va moltiplicato poi per c.
Sorge un problema nell'inversione del polinomio: bisogna trovare, in certi casi, un elemento che inverta anche lo zero e può capitare che questo elemento sia proprio lo zero. Tuttavia, poiché in questa maniera si rischia di creare un *punto fisso*, è necessario aggiungere una costante che è, per l'appunto, c.
b=8A (esadecimale)
Il risultato finale sarà b'=A2. Nell'S-Box che si andrà a creare, in posizione xy=95 si posizionerà il valore A2.

A cosa può servire saper calcolare dal nulla gli S-Box? La funzione di generazione degli S-Box è di fondamentale importanza per tutti coloro che dovranno implementare gli S-Box non tanto in software quanto in hardware (utile quando non si ha abbastanza RAM per effettuare i conti dovendosi appoggiare per forza su un circuito integrato). È utile anche a coloro che attaccano il sistema, ma in misura minore. Inoltre in AES i conti si fanno in parallelo: anche per questo può essere molto utile sapere come costruire uno o più circuiti integrati con lo stesso algoritmo.
Per decifrare a partire da un S-Box il ragionamento è esattamente lo stesso, tuttavia invertito; la matrice x, per esempio, cambia come cambia la costante c. La scelta ragionata di una certa costante c e di una specifica matrice x riducono la probabilità di un attacco efficacie ad AES (infatti per evitare attacchi la matrice x ha diverse proprietà specifiche).
Calcoli per l'inversa di un S-Box:
`| S-Box (S-Box(a))` è uguale ad `a`
`| S-Box(a)` è diverso da `S-Box(a)`
La fase **shift row** di AES è pensata apposta per distribuire in maniera più ampia possibile le parole della chiave.
AES è molto veloce nella cifratura ma lo è ovviamente meno a decifrare perchè, come tanti altri algoritmi suoi pari, invoca più volte la funzione di cifratura per poter effettuare una decifratura.

2.1 XTS

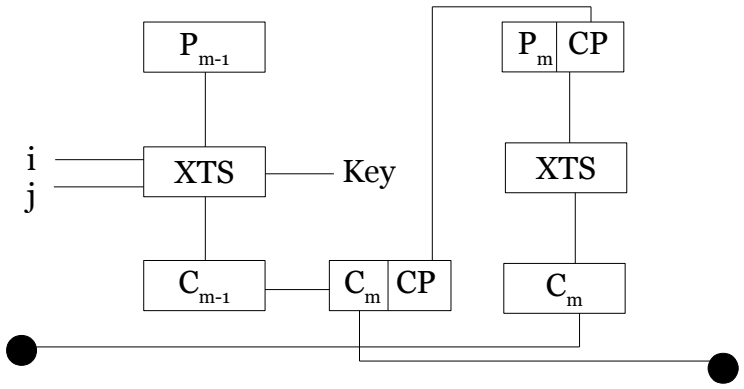
Modalità di utilizzo di AES sfruttata in direzione *block oriented* (avendo a che fare con blocchi di memoria contenenti informazioni),

pensata quindi studiata nel 2004 e sviluppata solo nel 2007.
Caratteristiche → l'eventuale attaccante ha a disposizione tutto il testo cifrato che desidera, mentre il testo da inserire nell'algoritmo è arbitrario. Le informazioni da cifrare si trovano su disco (es: hard disk) e si desidera che questa modalità generi una stessa quantità di testo cifrato indipendentemente dal testo in chiaro da cifrare. Due testi in chiaro identici si cifrano in due modi diversi a seconda del blocco di memoria in cui si vengono a trovare, quindi a seconda della loro presenza e sussistenza in una certa area di memoria piuttosto di un'altra, nonostante si utilizzi per cifrare sempre e comunque la stessa chiave. La cifratura si occupa di cifrare le informazioni in maniera indipendente dal contesto in cui si trovano, quindi senza farvi esplicito riferimento.



i indica il settore i-esimo e **j** costituisce il blocco j-esimo all'interno del settore i. Si vuole cifrare $\mathbf{P_j}$ che costituisce il testo in chiaro allo interno del blocco j-esimo che è lungo di norma 128 bit. Il testo cifrato sarà $\mathbf{C_j}$, anch'esso lungo 128 bit.
Sia, per esempio, che i=12 e j=3, allora faremo riferimento al settore 12 ed al suo blocco numero 3. Si prende il settore e lo si cifra con AES (ed una specifica chiave scelta), ottenendo così 128 bit i quali rappresentano un certo polinomio su un campo di Galois GF(2^{128}). Il polinomio irriducibile $m(x)=x^{128}+x^7+x^2+1$ sarà quello che genererà il suddetto campo di Galois. Tale valore verrà moltiplicato per una matrice primitiva $\mathbf{a^j}$ elevata per il numero del blocco, che è appunto j: si verificherà allora una moltiplicazione tra due vettori dalla quale risulterà un nuovo polinomio. Il risultato verrà XORato con il testo in chiaro da cifrare e di nuovo il tutto verrà inserito in AES con una nuova chiave, differente da quella utilizzata al primo utilizzo di AES. Infine, quanto ottenuto verrà nuovamente XORato per ottenere il testo cifrato finale.
Questa tecnica funziona la quasi totalità delle volte. A seconda del contenuto di P_j il risultato, ovviamente muterà. I file, infatti, non è detto che siano tutti multipli precisi di 128 bit e la cosa può far piacere ad un eventuale attaccante. Per questo motivo l'ultimo blocco di testo cifrato è indissolubilmente legato al penultimo i quali

vengono quindi cifrati assieme.



Se prima il testo in chiaro era semplicemente dato in pasto a XTS, ora si preferisce non esplicitare immediatamente il testo cifrato finale del penultimo blocco. Se capita che P_m sia lungo solo 3 bit anziché 128 se ne prendono altrettanti dal testo cifrato del blocco precedente e si riempie lo spazio che manca ad arrivare a 128 con del padding per il blocco successivo. Il tutto viene dato in input a XTS: si elabora il testo cifrato e lo si utilizza come testo cifrato dell'ultimo blocco mentre il testo cifrato dell'ultimo blocco è dato dai quei 3 bit tenuti da parte prima.
A livello matematico, il testo cifrato si ottiene come:

$C = CC \text{ XOR } T =$
$C = E\ (K_1;\ PP) \text{ XOR } T =$
$C = E\ (K_1;\ P \text{ XOR } T) \text{ XOR } T$

Mentre la fase di decifratura è data da:

$P = PP \text{ XOR } T =$
$P = D\ (K_1;\ CC) \text{ XOR } T =$
$P = D\ (K_1;\ C \text{ XOR } T) \text{ XOR } T$

Il testo in chiaro P è quindi così ottenibile, tuttavia C si può scrivere come $C = E\ (K_1;\ P \text{ XOR } T) \text{ XOR } T$ allora è possibile sostituirlo nella formula del testo in chiaro come segue:

$P = D\ (K_1;\ [E\ (K_1;\ P \text{ XOR } T) \text{ XOR } T\ ] \text{ XOR } T) \text{ XOR } T$
$P = (P \text{ XOR } T) \text{ XOR } T = P$

XORare con le T implica sommarle più e più volte, allora non è sempre necessario.

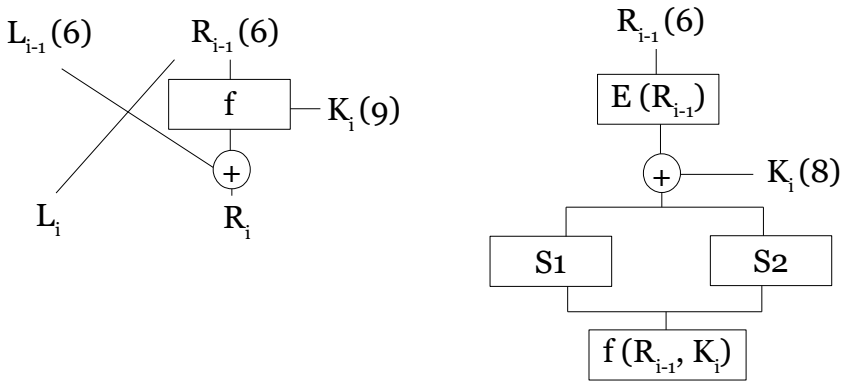
2.2 Attacchi ad AES

C'è da dire che attacchi interi che arrivino a scoprire la chiave di cifratura di AES non ne esistono; esistono solamente diversi studi ed attacchi teorici che non è detto che siano comunque fattibili. Anche gli attacchi ad XSL sono attacchi teorici.
AES è molto utilizzato a livello di mercati ed in ambiti governativi, per questo motivo si analizza la struttura del cifrario.
Nel 2009 si è tentato un attacco tramite l'uso di **chiavi correlate** sulla base di un lavoro sul calcolo dello scheduling della chiave (in

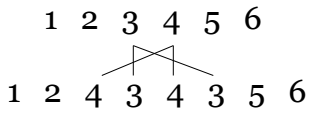
questo caso si tratta più che altro di uno studio che di un attacco teorico), dove per chiavi correlate s'intende quelle chiavi collegate tramite una certa funzione che le genera. Questo modello risulta non essere realistico perchè convincere all'utilizzo di chiavi di questo tipo da parte chi chi cifra è difficile anche se, nella pratica, chi implementa l'algoritmo ogni tanto lo fa.
Sempre nel 2009 si prova un attacco che fa uso di **permutazioni** simili a quelle utilizzate dall'algoritmo stesso di AES. Prevede la conoscenza della chiave mirando alla sequenza corretta di operazioni più fragili.

2.3 Crittoanalisi differenziale

L'attacco è eseguito sull'algoritmo di DES ridotto.
L'**algoritmo DES originale** e la sua **funzione Feistel** risultano essere così schematizzati:



L a **funzione di espansione E (R_{i-1})** che si trova nella funzione Feistel prende 6 bit da espandere in 8 e si comporta così:



Ciò che risulta dalla funzione di espansione va XORato con 8 dei 9 bit della chiave di cifratura, scelti a seconda del round in cui ci si trova: si prende come primo bit della chiave, a partire dai 9 dati, quello che nel conteggio corrisponde al numero del round in corso. In seguito allo XOR con la chiave ciò che risulta, ovvero una stringa di 8 bit, viene suddivisa in due parti per essere utilizzate dai due S-Box.
Ogni S-Box S1 e S2 è una matrice con 2 righe e 8 colonne: il primo bit della stringa in entrata indica quale riga selezionare mentre i restanti 3 riportano la colonna da scegliere: il valore che si trova in corrispondenza di tali valori verrà sostituito in output ai 4 bit in entrata agli S-Box.

Per poter fare i conti, esplicitiamo gli S-Box.

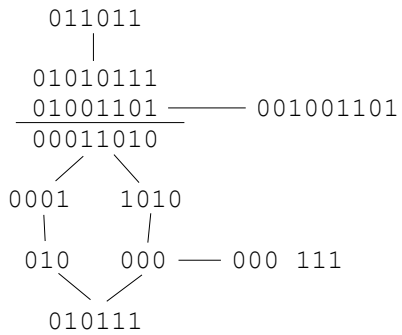
S1 →

101	010	001	110	011	100	111	000
001	100	110	010	000	111	101	011

S2 →

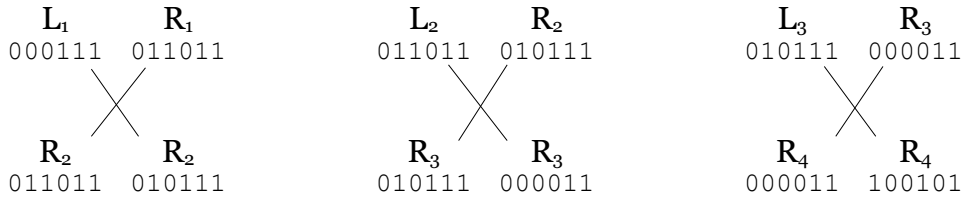
100	000	110	101	111	001	011	010
101	011	000	111	110	010	001	100

Ora che si sono ricapitolate le caratteristiche fondamentali di DES si può provare ad eseguire l'attacco della crittoanalisi differenziale. Tale attacco è di tipo choosen plaintext, questo significa che si guardano le differenze tra input tentando di carpire quelle tra i relativi output.
Si scelgono allora specifiche coppie di testi in chiaro da dare in pasto all'algoritmo in maniera tale che la scelta risulti il più intelligente possibile. La chiave di cifratura sarà sempre la stessa:
`K = 001001101`
Si scelgono allora le coppie di testi in chiaro:
L_1R_1 `= 000111 | 011011`
Si può studiare allora cosa succede, bit per bit, al testo scelto.
Nel nostro caso espliciteremo i calcoli solo una volta, che comunque si ripetono tali e quali anche per i round successivi intercambiando opportunamente i valori.



Si parte da una porzione del testo in chiaro `011011` quindi lo si espande grazie alla funzione di espansione nella stringa `01010111` per poi XORare quanto ottenuto con una sotto-parte della chiave originale che da 9 (`001001101`) bit passa ad 8 bit (`00011010`). Il risultato va a sua volta suddiviso in due sottoparti da 4 bit ciascuna, le quali indicheranno la riga e la colonna di ogni S-Box al fine di recuperare una nuova coppia di valori da sostituire a quelli in entrata (`010 000`).
Alla fine del procedimento, la stringa individuata entro gli S-Box andrà nuovamente XORata con la parte di testo in chiaro lasciata

immutata, ottenendo così il testo cifrato di questo primo round.



In questa maniera si ottiene la prima coppia di testo in chiaro e relativo testo cifrato:
L_1R_1 = 000111011011
L_2R_2 = 000011100101
Si sceglie allora una seconda coppia, del tipo:
$L^*_1R^*_1$ = 101110011011
$L^*_2R^*_2$ = 100100011000
Questo testo è stato scelto appositamente, infatti $R_1=R^*_1$ e devono esserlo per forza al fine di semplificare un po' l'attacco.
Rifacciamo i conti anche per il nuovo testo in chiaro.
$R_2 = L_1$ XOR $f(R_1, K_2)$
$L_3 = R_2 = L_1$ XOR $f(R_1, K_2)$
$R_4 = L_3$ XOR $f(R_3, K_4) = L_1$ XOR $f(R_1, K_2)$ XOR $f(R_3, K_4)$
ricordando sempre che $R_1=R^*_1$.
A questo punto si applica la crittoanalisi differenziale studiando la differenza tra gli input ed i loro relativi output attraverso una certa operazione di XOR.
$L'_i = L_i$ XOR L^*_i
$R'_i = R_i$ XOR R^*_i
Queste differenze hanno davvero effetto sul sistema? Le si studia rispetto ad R_4.
$R'_4 = R_4$ XOR $R^*_4 = L_1$ XOR $f(R_1, K_2)$ XOR $f(R_3, K_4)$ XOR L^*_1 XOR $f(R^*_1, K_2)$ XOR $f(R^*_3, K_4)$
e siccome $R_1=R^*_1$ allora $f(R_1, K_2)$ e $f(R^*_1, K_2)$ si semplificano a vicenda. Tuttavia L_1 XOR $L^*_1 = L'$ allora si riscrive tutto come:
$R'_4 = L'_1$ XOR $f(R_3, K_4)$ XOR $f(R^*_3, K4)$
ma non bisogna dimenticare che $L_4 = R_3$ e lo si può tranquillamente dedurre anche dalla funzione di output:
R'_4 XOR $L'_1 = f(L_4, K_4)$ XOR $f(R^*_3, K_4)$
Se si utilizzano i valori di input e di output opportunamente scelti manca solo la chiave K_4 a chiudere il quadro della situazione. Al 4° round è possibile calcolare la chiave a partire esclusivamente da quest'ultima funzione feistel.
Per calcolare K_4 si sa già a priori cosa c'è in input alla funzione feistel e cosa ne uscirà in output; in pratica la chiave viene calcolata un po' da sopra ed un po' da sotto.
Tuttavia non si ha propriamente a disposizione l'output delle funzioni feistel, anche se si sa che l'espansione della differenza risulta essere:
$(E(L_4)$ XOR $K_4)$ XOR $(E(L^*_4)$ XOR $K_4) =$
e d è possibile eliminare dall'espressione le chiavi perchè, al 4° round,

le differenze su di esse non hanno più alcun peso. Si ottiene così:
$= E(L_4)\ XOR\ E(L^*_4) = E(L_4\ XOR\ L^*_4) = E(L'_4)$
In questa maniera si avranno tutte le informazioni necessarie a disposizione, le quali entrano ed escono dagli S-Box. Sapendo a priori come sono fatti gli S-Box si può dedurre la parte di chiave che ancora manca da decifrare.
$R'_4\ XOR\ L'_1 = (R_4\ XOR\ R^*_4)\ XOR\ (L_1\ XOR\ L'_1) = 010100$
La somma quindi restituisce: $111101\ XOR\ 101001 = 010100$
Questa è la differenza tra le due coppie di testo in input, rispetto i loro vari output.
$E(L'_4) = E(L_4)\ XOR\ E(L^*_4) = 000011\ XOR\ 10101000 = 10101011$???
La *crittoanalisi differenziale* fa in modo di semplificare i conti utili per indovinare la chiave a partire dai dati sul sistema già in possesso.
Per trovare K_4 si gioca con le S-Box. Si parte con la parte sinistra del testo (1010) dopo l'espansione.
$E(L'_4) = 1010 = [K_4\ XOR\ E(L_4)]\ XOR\ [K_4\ XOR\ E(L^*_4)]$
Il che vuol dire che si trovano 2 coppie che sommate tra loro daranno come risultato 1010, allora le si prova un po' tutte fino a che non si sarà trovata la coppia adatta:
$1010\ XOR\ 0000 \rightarrow 1010$
$1000\ XOR\ 0010 \rightarrow 1010$
ecc.
Ma quali sono le coppie più adatte, quelle che vanno bene allo scopo? Sono adatte tutte quelle coppie il cui output restituisce un valore corretto. Per esempio:
0000 entra in S1 e si ottiene 101
1010 entra in S1 e si ottiene 110
e si XORano i due risultati come si XORano i due testi in input. Si otterrà allora come risultato 011 che però non è uguale a 010, allora si elimina la coppia e se ne sceglie una nuova per riprovare d'accapo.
Si faranno tutte le prove necessarie fino ad ottenere 2 coppie corrette che in questo caso risultano essere $1001/0011$ e $0011/1001$.
K_4 va calcolata su metà $E(L'_4)$ tramite un'equazione di primo grado:
$K_4\ XOR\ E(L'_4) = 1001$ oppure 0011
Ma $E(L'_4)$ già si conosce, ed è uguale a 0000, allora riscrivo come:
$K_4\ XOR\ 0000 = 1001$ oppure 0011
Così si ottengono i primi 4 bit della chiave. Anzi, si sono così trovati due possibili "pezzetti" della chiave. Ora è necessario trovare l'altra metà ripetendo d'accapo il procedimento, ma giocando con i bit di destra.
$E(L_4) = 1011$
quindi si calcolano anche qui le coppie idonee, che risultano essere $1100/0111$ e $0111/1100$. Si cercano quindi gli ultimi 4 bit della chiave:
$K_4\ XOR\ E(L_4) = K_4\ XOR\ 0011 = 1100$ oppure 0111
tutti i conti sono stati eseguiti portando a destra il valore 0011 ed ottenendo così 1111 e 0100.
In questo momento si hanno ben due candidati per entrambe le parti

della chiave: la prima coppia di testi in chiaro e cifrato va sempre lasciato invariato, prendendo altri testi per condurre altre prove accessorie grazie alle quai sarà possibile creare un insieme si probabili parti di chiave. La chiave corretta risulterà da un sottoinsieme di tutti i possibili risultati, tra quelli che compariranno, a prescindere dalla prova, il maggior numero di volte.
Si tratta di un attacco di tipo deterministico, soprattutto in questo caso con 3 soli round, tuttavia con 16 round diventa decisamente un attacco di tipo probabilistico.

Attacco a DES 4 round. In pratica si tratta di attaccare DES a 3 round, con la presenza di un round aggiuntivo. Il nuovo round non si trova tuttavia in coda, ma in testa al procedimento. Con un certo input dato in pasto all'S-Box si otterranno un certo numero di coppie che poi daranno il risultato corretto.
Se ci sono 16 coppie che restituiscono come output il valore 0011 allora 12 di queste 16 coppie daranno l'output corretto, per entrambi gli S-Box se correttamente stimolati. Nel caso di S2 è probabile che 8 coppie su 16 restituiscano il risultato corretto, mentre per S1 la probabilità di recuperare una coppia giusta sale a 12 su 16; la probabilità generale è comunque di 3 su 8, poco meno del 50% perchè coppie ed S-Box sono tra loro indipendenti.
Siccome si tratta di un attacco *choosen plaintext* si sceglie il testo in chiaro in maniera tale per cui all'interno degli S-Box al round 1 entri il valore 0011; in questa maniera la parte sinistra all'uscita dall'S-Box sarà del tipo 011 e la parte destra ABC. Nel 50% dei casi l'input del 2° round sarà formato dalla stringa `011ABC`. I restanti 3 round subiranno allora un attacco di tipo deterministico e si troverà la chiave K corretta anche se non è detto che lo sia.
Si ri-esegue tutto il gioco con $L^*_1R^*_1$ per cui si troverà K* che potrà essere uguale a K solo se la chiave è corretta. Si ripete nuovamente il procedimento calcolando K** ma non si può sapere quale chiave sia corretta, dipende tutto dal testo cifrato che porta a `011ABC`. Si hanno 3 casi su 8 nei quali sarà possibile trovare la chiave corretta. La chiave giusta sarà quella più frequente, quindi più saranno i tentativi ed i round più sarà necessario variare i testi in chiaro ed il tempo impiegato.

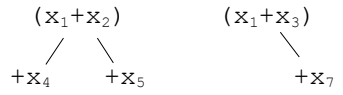
Capitolo 3

Crittografia in Hardware

Nel 1996, **Christoph Paar** si interessava all'implementazione di sistemi ottimizzati in hardware; di suo specifico interesse, nella fattispecie, risultava la realizzazione della parte lineare degli algoritmi Reed e Solomon.
Per esempio, si pensi ad AES: l'idea è sempre quella di compattare le S-Box in modo tale da velocizzare i conti, soprattutto in hardware dove le potenzialità sono finite e vanno sfruttate al meglio, il tutto attraverso una serie di AND e XOR per ottenere la massima ottimizzazione possibile per il sistema.
Sia f() una funzione che va ottimizzata per velocizzare il sistema ed al contempo per allontanare eventuali rischi di attacco allo stesso; f() sarà una funzione crittografica che farà largo uso di XOR (quindi di operazioni lineari) e di AND (operazioni non lineari) e potrà essere a sua volta una composizione di altre funzioni crittografiche. Paar quindi studiò il modo di prendere una funzione crittografica e di renderla minimizzabile al massimo consentito.
Sia f() una funzione la quale prende in input una certa quantità di valori x_i (costituenti tra loro l'unico input) e restituisce un'altra certa quantità di risultati y_j (costituenti tra loro un unico output), il cui numero di x e di y non è detto che per forza debba essere equivalente.

$$\begin{cases} y_1 = x_1+x_2+x_4 \\ y_2 = x_1+x_2+x_5 \\ y_3 = x_1+x_3+x_7 \end{cases}$$

L'output della funzione è costituto da 3 bit ($y_1y_2y_3$) a fronte di un input di 7 bit ($x_1x_2x_3x_4x_5x_6x_7$) in XOR (+) tra di loro. Ci sono 2 XOR per ogni funzione, per questo è possibile cancellare le operazioni che compaiono più di una volta. Per esempio:



che sta ad indicare che lo XOR tra x_1 e x_2, per esempio, è XORato a sua volta con x_4 ed x_5 in due casi differenti, quindi quella prima operazione andrebbe eseguita due volte, una per ogni caso, quando invece si potrebbe desiderare di eseguirla una volta per tutte: è così possibile ridurre il numero degli XOR. Si tratta di una tecnica banale che è possibile intuire anche ad occhio nudo, dando un semplice sguardo alle varie espressioni (tuttavia, solo in questo caso banale, perchè in situazioni più complesse diventa più difficile accorgersene).

In quale modo, allora, è possibile ricavare il circuito lineare con il minimo numero di XOR? Si sfrutta a riguardo un algoritmo greedy: in pratica si trascrive il crittogramma e lo si traduce in una o più funzioni, quindi si guarda l'input che compare più frequentemente in tali funzioni (in caso di conflitto o di un pari numero di presenze si sceglie tra i rivali quali ottimizzare in maniera totalmente casuale, perchè non si può sapere a priori quale sarà la scelta ottimale). È in questa maniera che si costruiscono i primi S-Box in hardware.
Si noti che non si prende mai in considerazione alcuna parte non lineare eventualmente presente nel crittogramma.

Alla fine del 1999 viene sviluppata una nuova tecnica. Quando, eseguendo l'algoritmo, ci si imbatte in casistiche ambigue si prosegue di qualche passo per poi studiare l'opzione scartata, avanzando anche in essa di qualche passo dell'algoritmo: in questa maniera si sceglie l'ottimizzazione locale vincente.

Nel 2009 tuttavia si introduce una variante a questa nuova tecnica perchè l'algoritmo, così pensato, non tiene mai conto della proprietà di cancellazione presente in Z_2:
$-(x_1+x_2)-(x_2+x_6) = x_1+x_6$
Infatti, se si introduce tale proprietà si nota un drastico calo nel numero di porte XOR da adoperare per ricostruire il circuito in hardware. Le cancellazioni, inoltre, sono possibili ad ogni passo dell'algoritmo, tuttavia il problema di applicare tale tecnica sta nell'eventuale *profondità del circuito* (quindi anche del diagramma ad albero della schematizzazione che segue alla riscrittura del crittogramma in funzioni logiche) che non viene tenuta mai in conto: la profondità risulta essere più importante del previsto poiché la sua presenza, a livello di codice, potrebbe risultare disagevole ai fini dell'efficenza di calcolo (si pensi alla quantità di if/else in cascata che si verrebbero a creare).
Si costruisce quindi un nuovo algoritmo che tiene conto anche della profondità, peggiorando leggermente il numero di XOR finale che non è più il numero ottimo che ci si aspetterebbe: risulta essere, tuttavia, un ottimo compromesso.
Per quanto riguarda la parte non lineare della funzione crittografica (quella relativa agli AND, per intenderci) a partire dagli S-Box di AES con un'inversione si possono estrarre le parti non lineari, applicando così la tecnica di ottimizzazione anche sulle porte AND. Si divide allora in 2 parti distinte l'algoritmo di ottimizzazione.
La funzione degli S-Box di AES risulta essere:
$y_1 = x_1+x_3+x_1x_2+x_4+x_5x_6x_7$
e si prova a distinguere le parti lineari da quelle non lineari (dove per le parti lineari il trattamento è il medesimo spiegato prima); le parti non lineari verranno trattate mediante l'uso di basi normali così da ottenere un nuovo sistema mediante una nuova rappresentazione (si

sceglie la base normale che minimizza i costi di elaborazione).

Nel 1988 si studia l'inversione di un polinomio in un campo di Galois $GF(2^8)$ che è possibile solo in un sottocampo riconducibile a $GF(2^8)$. Sfruttando questo teorema si ottiene $GF(2^8)=GF(2^4)^2$ così, calcolando la funzione di inversione su $GF(2^4)$ si prova a passare a $GF(2^8)$, ma non è detto che questa sia la soluzione migliore per effettuare il passaggio. Itoh ci prova a partire direttamente da $GF((2^2)^2)^2$.
Si prova allora via forza bruta.
La tecnica purtroppo risulta essere esponenziale rispetto al tempo impiegato, così solo nel 2012 si è riusciti a lavorare su circuiti molto più grandi (e lo hanno fatto in UNIMI sotto la supervisione del prof. Visconti).
Paar allora introduce un'ulteriore modifica all'algoritmo: calcolando l a *negata* si otterrà un risultato inverso a quello atteso, ribaltabile attraverso una porta NOT.

$$\begin{cases} y_1=x_1+x_2+x_4 \\ y_2=x_0+x_1+x_3+x_4 \end{cases} \qquad \begin{cases} NOT(y_1)=x_0+x_3 \\ NOT(y_2)=x_2 \end{cases}$$

Ovviamente la tecnica per il calcolo della negata ha senso solo nel momento in cui le funzioni sono molto ricche di termini (quando un'espressione possiede solo pochi termini, negandola si otterranno tutti gli altri, allungandola).
L'intero sistema è rappresentabile tramite una matrice, per valutare l'opportunità o meno di calcolare la negata si può studiare il numero di 1 presenti nella matrice stessa: più ce ne sono, più è utile sfruttare la negata.

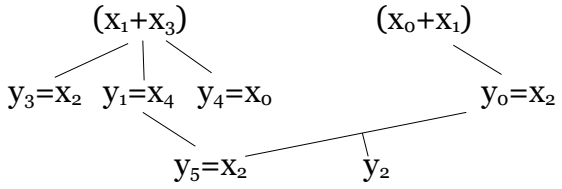
	x_0	x_1	x_2	x_3	x_4
y_1	0	1	1	0	1
y_2	1	1	0	1	1

Esempio.
Si applica la tecnica *Per Alta* sulla parte lineare (no Hito e no Campi di Galois).

	x_0	x_1	x_2	x_3	x_4
y_0	1	1	1	0	0
y_1	0	1	0	1	1
y_2	1	0	1	1	1
y_3	0	1	1	1	0
y_4	1	1	0	1	0
y_5	0	1	1	1	1

2	2	2	2	2	2	1	0
2	1	1	1	0	0	0	0
3	3	3	3	3	2	1	0
2	1	1	0	0	0	0	0
2	1	0	0	0	0	0	0
3	2	2	1	1	0	0	0

Si tratta di un sistema 6x5.
Per prima cosa si sceglie l'operatore più frequente, ovvero quello che minimizza il vettore delle distanze (ovvero quanti XOR si fanno per ogni riga). Si sceglie quindi la coppia che si presenta più spesso, la si somma e la si cancella così da eliminare uno XOR alla volta. Si provano tutte le somme possibili e si trovano quelle che minimizzano il vettore comparendo più e più volte in altre somme.
In questo caso si sceglie $x_1+x_3 = t_5$ ed è facile vedere che si tratta della somma migliore. La somma successiva sarà $x_0+t_5 = t_6$: se le due coppie risultano avere la stessa frequenza si sceglie tra le due quella che massimizza la norma del vettore. È dimostrabile che un vettore con più zeri, detto *super-concentrato* (quindi con una norma più alta) è più facile da calcolare con meno XOR a disposizione.
I conti restanti sono i seguenti:
$t_7 = x_2+t_5$
$t_8 = x_4+t_5$
$t_9 = x_2+t_8$
$t_{10} = x_0+x_1$
$t_{11} = x_2+t_{10}$ in questo caso si trova il target ma non si abbassa nulla
$t_{12} = t_8+t_{11}$ qui avviene una cancellazione
Ciò che si ottiene da questo conto è una rappresentazione binaria ottima. L'algoritmo termina quando si hanno tutti zeri o non è più possibile convergere.



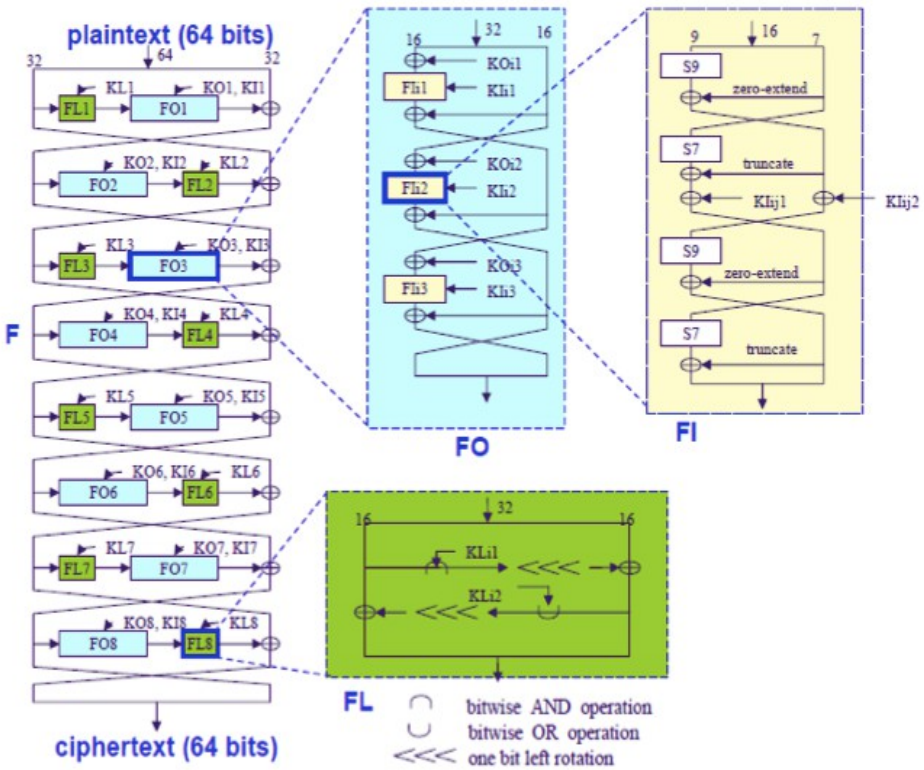
3.1 Cifrari “strani”

Kasumi. Si tratta di un cifrario “cellulare” per la protezione dei dati. Per collegarsi, il cellulare comunica con la base tramite una chiave. Il provider ascolta e contemporaneamente monitora le comunicazioni. La rete non è mai sicura, allora ci si appoggia spesso a **Kasumi**, non importa quanto sia sicuro di per sé il dispositivo. Voce e dati vengono protetti da una specifica cifratura mediante autenticazione.
Kasumi è stato progettato da SAGE nel 1999 a partire da Misty, del 1997. Utilizzato per il GSM, GPRS e UMTS. Si tratta di un cifrario a blocchi sullo stile di DES.
Si compone di alcune funzioni:

1. **funzione FL**: piccoli cambiamenti in input non valgono grandi cambiamenti in output. Si tratta di una funzione lineare; la crittoanalisi viene resa difficoltosa dai vari *shift rotate* inseriti qua e là nel calcolo;

2. **funzione FO**: si compone fondamentalmente di 3 feistel round, richiama un'ulteriore funzione chiamata *funzione FI*. Fissata una certa chiave permuta una certa quantità di blocchi.
3. **funzione FI**: effettua una suddivisione in due parti per poter utilizzare due S-Box differenti ed altre due funzioni aggiuntive. Gli S-Box sono implementabili in logica combinatoria ed, in pratica, costituiscono le fondamenta di tutto l'algoritmo.

L'algoritmo si compone di 8 round, quindi servono 8 chiavi: la chiave originale viene divisa in 8 parti ed ogni chiave originale viene allungata con alcune costanti per evitare che si ripetano dei bit nelle sottoparti.



Crittoanalisi differenziale. Studia, in Kasumi, la propagazione delle differenze (XOR) che se filtrano attraverso alle funzioni lineari tendono a conservarsi. Per recuperare la chiave si lavora con una grande quantità di testi in chiaro: la chiave corretta sarà quella che risulterà essere più frequente.

Boomerang attack. Si divide il cifrario in 2 parti, una con alto potenziale e l'altra con basso potenziale. Le differenze tra gli input vengono conservate dai relativi output. *Attacco distinguisher*: lo scopo è capire dove, all'interno di uno stream, si ottiene il testo cifrato piuttosto di una serie di bit di rumore, continuamente inviati senza sosta, così da nascondere il testo cifrato in essi; tale attacco

sfrutta una caratteristica qualsiasi del testo che però rimane invariata fino alla fine, anche nel testo cifrato.

Sandwitch attack. Rispetto al boomerang attack, aggiunge qualche cosa in più, ovvero applica il distinguisher per trovare la chiave forzando l'algoritmo.

Kasumi nei sistemi reali. Si tratta di cifrari stream che utilizzano la funzione KGCORE basata su Kasumi. Kasumi pulito è forzabile ma, se utilizzato incapsulato, la complessità sale.
Il sandwitch attack non attecchisce su Misty.

Capitolo 4
Crittografia in Rete

4.1 SEED

SEED non è un cifrario classico, ma si tratta dello standard nazionale coreano nato nel 1998. Sviluppato per offrire un'alternativa di matrice coreana ad AES, anch'esso si compone di blocchi e di chiavi da 128 bit. Si tratta di un cifrario a blocchi di tipo simmetrico e caratterizzato da funzioni feistel, suddiviso in 16 round: in pratica "ruba" le buone idee di altri cifrari e le compone a modo suo. Dai cifrari Misty/Kasumi, SEED prende in prestito il meccanismo della funzione feistel ricorsiva; da Saifer (??) eredita la struttura e la natura degli S-Box e dal cifrario Tea impara l'utilizzo esclusivo di alcune specifiche costanti.
Rispetto ad altri cifrari che utilizzano porte XOR, SEED compie somme e sottrazioni in modulo 32.

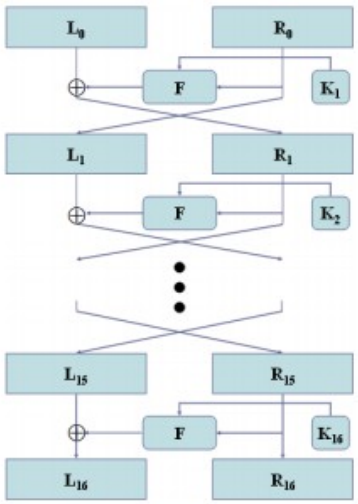


Figure 1. Structure of SEED

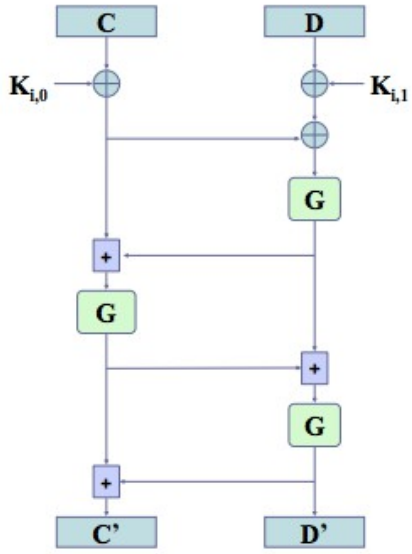


Figure 2. Round function F

Funzione feistel. SEED si compone di una funzione G utilizzata in maniera simmetrica in combinazione con operazioni modulo 32.

Funzione G. Si tratta di una funzione che spezzetta l'input e lo fa passare per gli S-Box, per poi sottoporlo ad operazioni con costanti predefinite.
Come accadeva già in AES, SEED lavora basando la sua aritmetica su

un polinomio irriducibile negli anelli tipici di AES, è in questa maniera che genera gli S-Box.

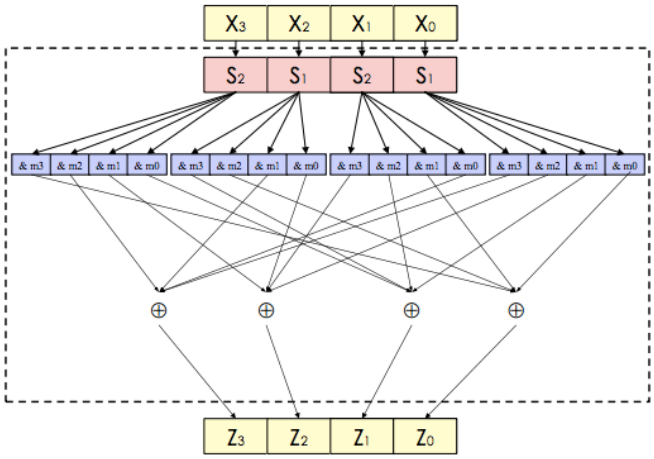


Figure 3. Function G

Key schedule. La chiave viene generata all'inizio del procedimento applicando somme e sottrazioni in modulo 32, oltre alla funzione G; è facile intuire che la schedulazione della chiave risulti più complicata e sporca rispetto a quella di AES.
SEED veniva utilizzato solo in Corea, tuttavia nel 2005 è diventato uno standard per il web (utilizzato spesso in TLS, IP-SEC). SEED ha più o meno l'età di AES, esattamente come altre varianti di altri cifrari, ad esempio Tea.

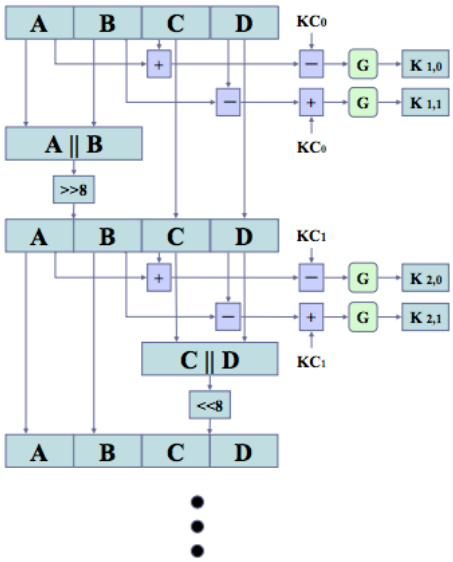
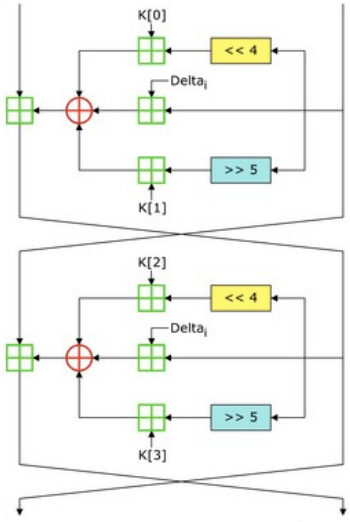


Figure 4. Key Schedule

4.2 Tea

Tea (*Tiny Encryption Algorithm*) è un semplicissimo cifrario inglese, di dimensioni ridotte e dalla struttura banale quanto basta per renderlo velocissimo nella computazione: consta solamente di 5 righe di codice nelle quali cifra ed in altrettante decifra qualsiasi input. Si compone di blocchi da 64 bit, con una chiave da 128 bit. È ovviamente un cifrario a blocchi.



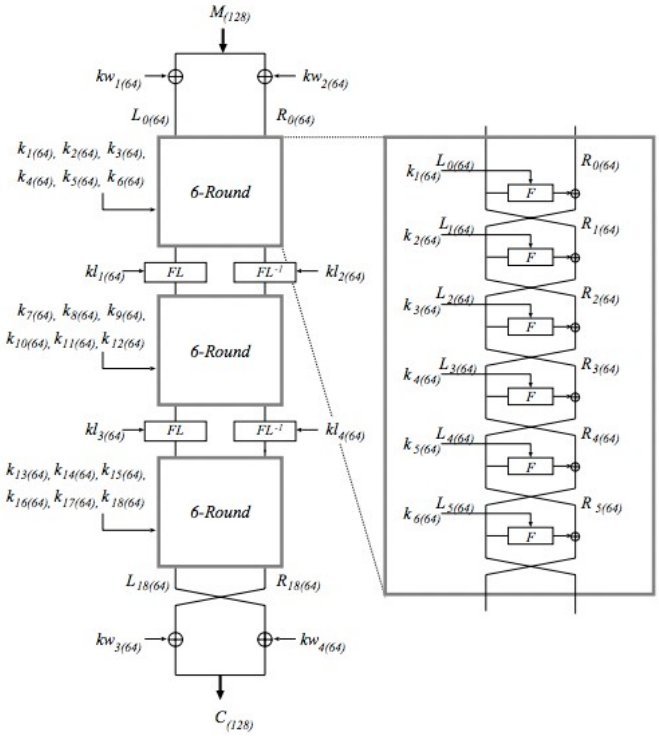

```c
#include <stdint.h>

void encrypt (uint32_t* v, uint32_t* k) {
    uint32_t v0=v[0], v1=v[1], sum=0, i;           /* set up */
    uint32_t delta=0x9e3779b9;                     /* a key schedule constant */
    uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];   /* cache key */
    for (i=0; i < 32; i++) {                       /* basic cycle start */
        sum += delta;
        v0 += ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        v1 += ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
    }                                              /* end cycle */
    v[0]=v0; v[1]=v1;
}

void decrypt (uint32_t* v, uint32_t* k) {
    uint32_t v0=v[0], v1=v[1], sum=0xC6EF3720, i;  /* set up */
    uint32_t delta=0x9e3779b9;                     /* a key schedule constant */
    uint32_t k0=k[0], k1=k[1], k2=k[2], k3=k[3];   /* cache key */
    for (i=0; i<32; i++) {                         /* basic cycle start */
        v1 -= ((v0<<4) + k2) ^ (v0 + sum) ^ ((v0>>5) + k3);
        v0 -= ((v1<<4) + k0) ^ (v1 + sum) ^ ((v1>>5) + k1);
        sum -= delta;
    }                                              /* end cycle */
    v[0]=v0; v[1]=v1;
}
```

Solitamente i cifrari a blocchi si possono impiegare come funzioni di hashing, tuttavia non tutti i cifrari a blocchi producono un buon hash. Su Tea lo sforzo computazionale è di 126 bit su 128 bit (2 non si prendono in considerazione perchè sono intercambiabili).
Tea è forse (tristemente) famoso per via del suo utilizzo da parte di

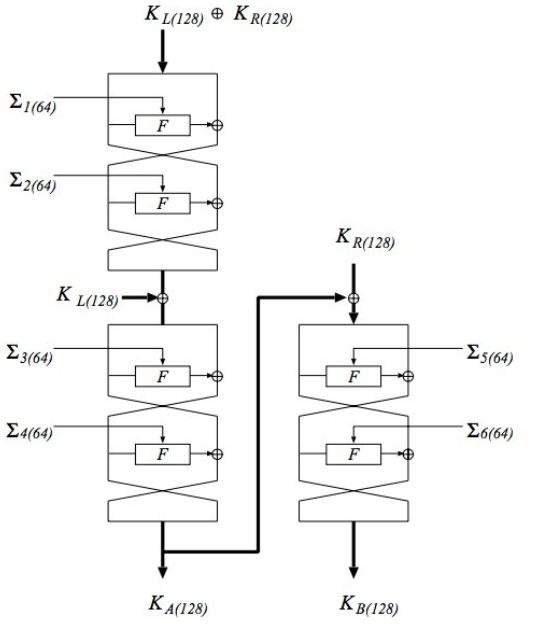
Microsoft in Xbox, sulla cui consolle non si voleva dar la possibilità di installare un sistema operativo non Microsoft e si voleva impedire la lettura di dischi e videogiochi non Microsoft. Il kernel del sistema operativo di Xbox va di norma installato sulla macchina, tuttavia Microsoft, proprio per evitare danni, sceglie di installarlo su una memoria flash, oppure su una memoria ROM (in questo caso l'aggiornamento del sistema diventa un vero incubo), tuttavia il pericolo della sostituzione è ancora in agguato. Microsoft allora sceglie di implementare un sistema operativo il cui kernel si trova su una memoria flash ma il cui hash verrà salvato all'interno di una ROM nascosta in maniera intelligente (la ROM in questione si trova nel *south bridge* che è collegato alla CPU da un BUS difficile da "sniffare", in questa maniera risulta assai difficile intervenire per dissaldare la ROM). Tuttavia, se si memorizzano certe informazioni sulla ROM sorge immediatamente un problema di spazio (con RC4, metodo utilizzato in precedenza, si faceva una semplice firma sull'hash del kernel che andava poi salvata sulla ROM). Microsoft allora si accorge che il codice contenuto nella ROM è comunque leggibile, quindi le serve una tecnica migliore di RC4 per mettere al sicuro certi dati: si implementa un doppio bootloader con il kernel. Il primo boot carica le informazioni della ROM per passare poi il testimone al secondo boot che conosce la reale ubicazione del kernel, così da poterlo caricare in maniera indipendente nella ROM; il secondo boot ed il kernel si trovano comunque memorizzati sulla memoria flash. Microsoft ha comunque bisogno di una funzione hash per la validazione, allora sceglie Tea per via del pochissimo spazio di memoria a disposizione. Peccato che Tea sia una pessima funzione hash e che sia altamente vulnerabile al *birthday attack*.
L'attacco a Tea sfrutta anche il *related key attack* (utilizzato spesso anche su WEP il quale sfrutta la password inserita ed in IV da 24 bit facendo così soccombere Tea all'attacco con soli 4000 pacchetti spediti). Il problema di WEP è stato risolto con WPA il quale utilizza una master key tra client e access point, creando così una working key da utilizzare come vettore di inizializzazione (IV).
Varianti. XTea è una variante a Tea ancora più veloce ma senza particolari differenze con Tea. Ha una debolezza che risiede nella crittoanalizi differenziale, assai peggiore rispetto a quella di Tea. XXTea si basa sulle operazioni classiche.

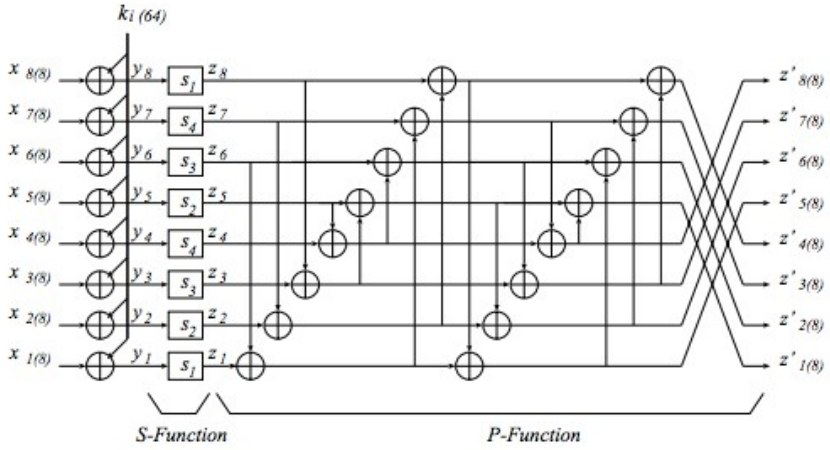
4.3 Camellia

Camellia è un cifrario giapponese alternativo ad AES. Si compone di blocchi da 128 bit e di chiavi di lunghezza variabile. Si tratta, come si è già intuito, di un cifrario a blocchi. Gli attacchi portati a questo cifrario si limitano alla vers. a 9 round tuttavia i round di Camellia sono molti di più. Camellia lavora sui campi di Galois, come AES.



Compatibile con AES, si compone di 18/24 feistel round a seconda della lunghezza della chiave scelta. Rispetto ad AES, Camellia può essere indicato come cifrario feistel. La chiave viene generata da un processo di key scheduling. Utilizza 8 S-Box che, in realtà, sono solo 4 ripetuti un paio di volte. Nella *main function* di Camellia sussiste una combinazione di funzioni feistel che lavora attivamente con tutti gli S-Box.





La chiave subisce trasformazioni tramite la funzione fesitel. Se la chiave è molto lunga, la trasformazione viene iterata tramite più funzioni feistel accodate. In soldoni, la funzione feistel prende l'input e lo XORa con gli S-Box che sono ripetuti più di una volta per semplificare e velocizzare l'hardware. Gli S-Box sono "taroccati": sembrano diversi, tuttavia il punto di partenza per la loro creazione è unico così da partire da un unico S-Box che poi subisce una serie di trasformazioni affini, inversioni e shift per diversificarne versioni leggermente differenti (il tutto potrebbe risultare una debolezza per il sistema).
Camellia è una variante di AES molto sfruttata; nel suo piccolo ha passato i test di sicurezza di NESSIE.

4.4 NESSIE

Il **NESSIE** è un concorso/progetto lanciato dalla comunità europea al fine di trovare una serie di cifrari da standardizzare, del tutto indipendente dalla volontà del NIST. In questo ambito sono stati sottomessi e testati molti algoritmi crittografici, i quali poi sono stati smistati in diverse categorie a seconda delle loro caratteristiche e della loro stessa efficenza.
Block cipher → *Misty*, *Camellia*, *AES*, *SHACalc2* (basato su SHA1).
Public Encryption → *RSA Encryption*, *Nippon Telegraph*.
Hash → Università Cattolica di Leuven (*TOTrakMac*) , *Whirlpool*, *SHA* (nelle sue prime 3 versioni).
Digital Signature → *DSA* (nella versione classica), *ECDSA*, *RSAPSS*, *SFLASH* (ceduto sotto attacco nel 2007), *GPSAutentication*.
Durante il concorso nessuno degli algoritmi sottoposti a verifica vince nella categoria **Stream**, tutti falliscono. Quando fu indetta una seconda gara l'adesione degli algoritmi "aspiranti" standard Stream fu più massiccia della precedente, alcuni di loro erano veloci in hardware ed altri più veloci in software. Oltre alla velocità, venne

messa alla prova la chiarezza espositiva della loro documentazione.
Stream Software → *HC128*, *Rabbit*, *Salsa20*.
Stream Hardware → *Trivium* (cifrario molto semplice composto da 9 porte XOR, un registro a scorrimento e qualche shift qua e là), *Grain* (cifrario banalissimo pensato per l'implementazione hardware), *Mickey*.
Passarono in secondo piano altri cifrari ideati da personaggi illustri. In Giappone si svolge una NESSIE giapponese, durante la quale vennero indetti standard algoritmi come *Creep Treck*; divennero standard giapponesi molti cifrari che erano già diventati standard anche in Europa.

4.5 Key Derivation Functions

Esistono diverse **KDF** (**Key Derivation Functions**) di diverso tipo, ma fondamentalmente tutte prendono in ingresso un testo ed a partire da questo fanno derivare una certa *chiave*. Le KDF generano una chiave a partire dal testo che si vuole cifrare, appunto, con quella stessa chiave, così da ottenere una chiave sicura. La chiave scelta dipende dal testo d'ingresso. Questi algoritmi non possiedono una funzione di hash ben definita, ma cambiano nome a seconda della funzione di hash a loro associata.
Le KDF sono molto sfruttate soprattutto da cifrari. Il NIST suggerisce l'utilizzo di una funzione di hash da 128 bit come minimo, alcune KDF utilizzano direttamente AES.
Di KDF, ne esistono alcune studiate direttamente per scopi precisi: per stabilire ed imparare come queste funzioni sono utilizzare nei vari cifrari basta leggere le loro relative RFC.
WPA2 è uno di quegli algoritmi che fa uso di KDF.
Nella generazione di chiavi sicure si utilizzano alcune maschere.
Di funzioni di derivazione della chiave ne esistono moltissime ed ogni algoritmo tende ad utilizzare la propria. L'output generato da queste funzioni può essere tagliato e suddiviso per essere poi utilizzato in seguito dove è più opportuno nella maniera più adatta.
Due KDF si possono “rincorrere” prendendo l'una in input l'output dell'altra, in questa maniera la generazione della chiave diventerà ancora più complicata.
L'entropia misura la casualità della generazione della chiave: la si può studiare generando chiavi in diverse maniere. Le prime 1000 chiavi ottenute da una KDF sono probabilisticamente identiche tra loro a prescindere dall'input utilizzato (i loro bit sono indistinguibili, così anche le chiavi che questi bit formano). L'attaccante, allora, non ricava alcun vantaggio nell'ascoltare e modificare i bit della chiave, dopotutto l'output è fortemente influenzato dai bit in input. Tutto questo però non dipende dalla funzione di hash scelta, tant'è che si può sfruttare la peggiore delle funzioni di hash senza turbare la

genuinità del risultato, sarà la KDF a fare tutta la differenza.
Riducendo il numero di chiavi generate, la distribuzione dei bit tende a cambiare: da 10 a 100 chiavi generate la varianza aumenta; la probabilità di indovinare un bit è del 50%, per questo l'utilizzo delle KDF è aumentato drasticamente nel tempo. Il tutto in maniera indipendente dalla bontà degli input, poiché l'output sarà sempre comunque ottimo.

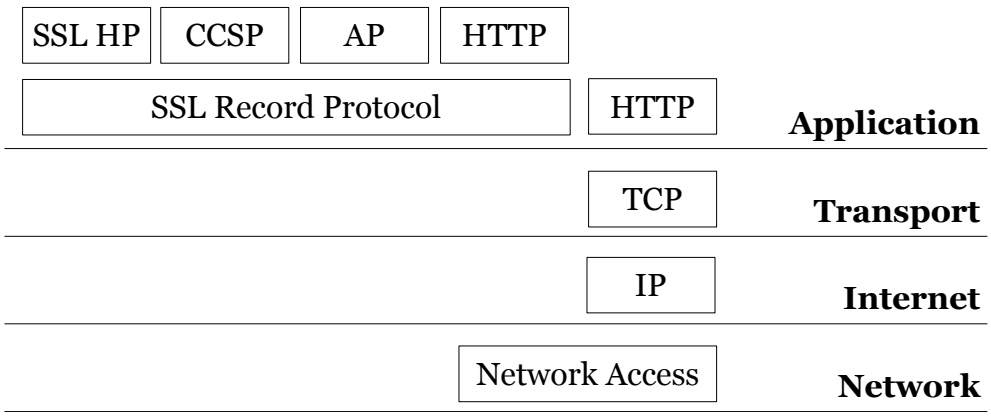
Capitolo 5
Sicurezza in Rete

5.1 Protocolli di sicurezza in Rete

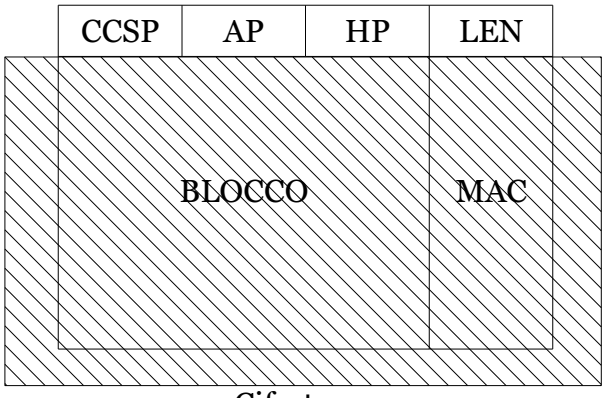
Gli obiettivi dei **protocolli di sicurezza** come **SSL** e **TLS** sono, tra le altre cose, la garanzia della sicurezza e dell'integrità dei dati, così come la loro estensibilità (ovvero la predisposizione all'inserimento di novità all'interno della vecchia struttura) e l'eventuale efficienza (utile nella generazione di una comunicazione client/server dove i parametri sono decisi *una tantum* all'inizio del contatto e poi utilizzati durante tutta la transizione senza mai essere rinegoziati visto gli alti costi computazionali della loro elaborazione).

Il **client** è colui che comunica al server quali protocolli di sicurezza supporta fornendone una lista completa di scelte tra cui prendere una decisione; il **server**, letta la lista, restituirà al client l'indicazione del protocollo scelto tra quelli della lista che anche lui supporta nella versione più bassa in comune.

Sessione. Associazione ad alto livello tra client e server durante la quale si condividono diversi parametri per ogni connessione attiva (istituita tramite protocollo di handshake, i cui parametri verranno condivisi durante tutta la durata della connessione).
Connessione. Si tratta di un "qualche cosa" di più piccolo rispetto alla sessione, ne possono esistere molteplici in una sola sessione e tutte diverse e distinte. Permettono la comunicazione, quindi l'invio e la ricezione dei dati. Una connessione solitamente è un collegamento punto a punto a livello di trasporto (vedi la classica pila protocollare ISO/OSI). Lo stato di una connessione è data da un certo insieme di parametri.



SSL Record Protocol. Lo scopo di questo protocollo è la garanzia di confidenzialità ed integrità delle informazioni.
Il **pacchetto dati** è diviso in **blocchi** di dimensione massima pari a 2^{14} byte (l'ultimo blocco nel quale il pacchetto è diviso sicuramente sarà di dimensioni minori a seconda della lunghezza del dato); in seguito alla suddivisione, avviene la **compressione** di ogni blocco di dati, anche se si tratta di un'operazione di compressione del tutto opzionale (non è detto, infatti, che avvenga e non è essenziale che si compia). Al risultato di quest'ultima operazione di compressione si aggiunge un'ulteriore informazione: il **MAC** (*Message Autentication Code*) il quale costituisce a tutti gli effetti un hash del blocco stesso. Questo digest viene chiamato MAC e non semplicemente hash perchè è l'hash stessa ad essere un certo tipo di MAC (*HMAC*: funzione hash che riassume il blocco). Se non si ha a disposizione una funzione di hash si può impiegare qualsiasi altro cifrario a blocchi (ed: CMAC-AES). Dopo l'aggiunta del MAC si **cifra** tutto il pacchetto/blocco tramite un certo algoritmo dato; al blocco cifrato, in ultima istanza, si aggiungerà un certo **header**. L'*header* possiede delle caratteristiche precise: contiene, per esempio, quattro campi tra cui la lunghezza dell'intero blocco, l'indicazione della versione del protocollo utilizzato all'interno della comunicazione nella versione supportata massima e minima, content type che riporta delle informazioni di stato del blocco/pacchetto servendosi di 4 diversi flag.
Il blocco/pacchetto può raggiungere una lunghezza massima di 2^{14} byte ai quali si sommano 2048 byte aggiuntivi (per ogni evenienza), e non può mai diventare più grande di così.



Cifratura

I 4 flag dell'header sono:

1. **CCSP (Change Cipher Spec Protocol).** Flag costituito da un solo byte che introduce a futuri cambi di specifiche riguardo al cifrario sfruttato nella connessione. Se il byte di questo flag è settato significa che il protocollo va aggiornato, accade spesso quando si fanno degli aggiornamenti periodici (es: portale della banca che giornalmente aggiorna i certificati

per le connessioni dei clienti). Questa richiesta può essere inoltrata sia dal client che dal server.

2. **AL (Alert Protocol).** Protocollo tipico di SSL/TLS utile nella gestione delle situazioni di errore che insorgono puntualmente quando va storto qualcosa. In questa maniera viene forzata una scelta/azione a seconda della situazione contingente:
   1. alert fatal/warning: il tipo *fatal* indica la chiusura di una connessione a seguito di un comportamento inatteso (magari di un attacco o di una intrusione); il tipo *warning* invece lascia la connessione aperta ma impedisce che se ne generino di nuove all'interno della stessa sessione.

   I messaggi che generano alert sono del tipo:
   2. *decompression failure*;
   3. *handshake failure*;
   4. *close notify*;
   5. *no/bad/insupported/revocated/ecc. certificate*;
   6. ecc.
3. **SSL HP (Handshake Protocol).** Protocollo pensato per la comunicazione tra client e server e può contenere, nella sua indicazione, 3 tipi distinti di informazione:
   1. tipologia della richiesta inviata di volta in volta durante la fase di handshake (1 byte);
   2. lunghezza del messaggio di richiesta (1 byte);
   3. content del messaggio.

   Il client ed il server possono scambiarsi diversi tipi di richieste a seconda delle operazioni che si vogliono intraprendere; il *content* contiene i parametri spediti ed utilizzati durante le varie richieste e le varie fasi dell'handshake, ovviamente questi parametri variano a seconda della fase in cui ci si trova e della richiesta inoltrata in quel momento.
   Il suddetto meccanismo è sfruttabile da parte di un attaccante che, in veste di client, comunica al server una richiesta mirata; spesso i server supportano le versioni più vecchie di qualsiasi protocollo di sicurezza proprio per essere sicuri di poter rispondere alle richieste di qualsiasi client, anche il meno aggiornato possibile, in questa maniera, però, supportano non solo le versioni più vecchie ma anche le più deboli.

   L'handshake tra client e server si compone di 4 fasi.
   Fase 1. Attivazione del servizio: client e server stabiliscono le condizioni di sicurezza sulle quali basare la connessione e si passano tali parametri all'interno del content (tra i quali si troverà anche la versione più nuova del protocollo supportato; un valore random formato da 4 + 28 byte, 4 di timestamp e 28 di un numero casuale per evitare di subire il replay attack; l'identificatore di sessione che sarà un numero pari a 0 se il client vuole aprire una nuova connessione sulla sessione

oppure diverso da 0 se il client vuole solo aggiornare i parametri della connessione in atto; una cipher suite, ovvero un elenco di cifrari supportati; un compression method, ovvero un elenco di cifrari utilizzati). Le richieste di server-hello e client-hello si assomigliano molto tra loro.Solitamente nella cipher suite sono elencati cifrari come RSA, Fortezza (anche se in TLS non esiste più), un set di 3 versioni di Diffie-Helman (anonimous, fixed, ephemeral con chiave one time pad), RC2, RC4, DES, Triplo DES, DES 40 bit, Idea. Durante questa fase si specifica anche l'algoritmo di hash e se si sceglie di cifrare a blocchi oppure stream, quindi si indica il key material e vari altri parametri crittografici.

Fase 2. La compie solo il server che invia al client il proprio certificato, scambia con lui le chiavi pubbliche, fa richiesta del certificato posseduto dal client e comunica al client che questa fase di "hello" è finita (questa è l'unica azione fissata che il server deve eseguire assolutamente in questa fase, le altre sono puramente opzionali). Il server può non possedere il certificato e può non richiederlo, oppure sì (a seconda della natura della comunicazione).

Fase 3. Di unica competenza del client, il quale può eseguire 3 azioni tra cui la verifica del certificato da parte del server, lo scambio delle chiavi del client e l'invio del suo certificato. Non tutte queste azioni sono da svolgere obbligatoriamente; la verifica del certificato serve ad evitare diversi attacchi come quello *men in the middle*.

Fase 4. Consolidamento: la compie inizialmente il client e poi la replica il server. Il messaggio di "ho finito" viene accostato a quello di cambiamento di specifica del cifrario, dove però non è richiesta nessuna modifica di tale specifica. D'ora in avanti tutti i messaggi scambiati tra client e server saranno cifrati secondo gli accordi presi tra entrambi.

Questo è ciò che accade in SSL durante la conversazione tra client e server.

5.2 SSL/TLS: differenze

TLS è un'estensione di SSL 3, quindi sostanzialmente non esistono particolari differenze tra i due protocolli (a parte la dismissione di MD5 come algoritmo di hash a favore di SHA256). Entrambe fanno largo utilizzo di HMAC e di CMAC-AES. In entrambi i casi avviene un'aggiunta di padding ed una cifratura opportuna del tutto.

5.3 SSL/TLS: attacchi

Scegliere esattamente quale protocollo utilizzare tra SSL e TLS in

tutte le loro versioni è facile e veloce, attraverso le impostazioni di qualsiasi browser; capita che alcuni browser però non supportino le versioni più nuove a favore delle più vecchie o che le supportino ma non siano abilitate di default dal browser.

Multi Prefix Attack. Attacco che si basa su come avviene la valutazione da parte dei vari browser e server rispetto al carattere di terminazione **0**. Ogni *certification authority* (CA) ed ogni browser si riserva di interpretare in maniera personale il significato di questo specifico carattere, creando spesso delle incomprensioni e dando modo di lasciare intravedere agli attaccanti dei pertugi nei sistemi di sicurezza.
L'attaccante può sottoporre alla CA un certificato falso inserendo il carattere \0 all'interno dell'url (componendo un url contenente il nome del certificato falso fatto precedere dal carattere \0). Di solito le CA ignorano l'esistenza di eventuali sottodomini, così leggono il certificato e lo validano senza problemi lasciando al proprietario del dominio il compito di far aderire il certificato anche agli eventuali sottodomini. Quando il certificato viene validato lo è unicamente per il dominio falso (seconda parte dell'url seguente il carattere \0), tuttavia qualsiasi browser legge il carattere \0 come carattere di terminazione così da ignorare la parte di url del dominio falso per visitare unicamente l'indirizzo antecedente il carattere \0: in questa maniera il browser pensa che il certificato appena validato sia per il sito web appena raggiunto e lo associa univocamente ad esso. In questa maniera il browser viene ingannato.

TLS Renegotiation Attack. In un sistema client/server, un client manda al server un messaggio del tipo *client-hello*, tuttavia un eventuale attaccante che si frappone tra client e server intercetta il messaggio e lo blocca per pochissimo tempo (o per una quantità di tempo che inganni il client e non lo convinca a mandare una nuova richiesta od ad accorgersi che è in atto un attacco). Nel frattempo l'attaccante manda una sua richiesta personalizzata, in veste del client, per instaurare una comunicazione col server e per decidere il protocollo da utilizzare nel mentre (il client aveva comunque intenzione di fare la stessa cosa). La richiesta dell'attaccante è inviata al server su HTTP semplice (e non su HTTPS) ed il messaggio mandato, riguardante una re-indirizzazione da HTTPS ad HTTP semplice, viene lasciato appositamente aperto, senza carattere di terminazione, così da lasciarlo pendente per costringere il server a rimanere in attesa della conclusione di quello specifico messaggio. Nel messaggio lasciato incompleto l'attaccante comunica al server di ignorare un “qualche cosa” che però non viene indicato.
L'attaccante manda poi il vero messaggio del client al server che così crede di dover iniziare una nuova negoziazione con lo stesso client di poco prima. Il server allora risponde ed il client rimane soddisfatto.

In questa maniera il server si trova per le mani la richiesta mandata dal client, correttamente chiusa, che contemporaneamente chiude anche la richiesta pendente mandata dall'attaccante. Ora che anche il messaggio dell'attaccante risulta essere chiuso, può essere preso in considerazione: quel “qualche cosa” da ignorare è appunto l'intero messaggio mandato dal client, la cui unica funzione agli occhi dell'attaccante è stata quella di porre il carattere di terminazione al messaggio lasciato prima pendente.
Ora il client può connettersi e tutti i dati che scambierà con il server in futuro verrano tranquillamente passati in chiaro su HTTP.

BEAST. Suggerito da tale Bard nel 2004 che sapeva come ingannare TLS notificandolo pubblicamente, nonostante alla fine in tantissimi non abbiano poi scelto di aggiornare TLS per porre rimedio al problema. Questo problema è esistito fino alla versione 1.0 di TLS.
Nel 2011, Duong e Rizzo hanno seriamente provato ad implementare il procedimento suggerito da Bard.
Partendo da un vettore di inizializzazione (IV) in modalità CDC in cifratura, l'attaccante può intercettare testi cifrati a volontà: la prima porzione di testo cifrato coinciderà con il vettore di inizializzazione. Ad ogni passo della cifratura, il vettore di inizializzazione diventa predicibile perchè liberamente leggibile sul canale.
L'attaccante allora si premura di installare sulla macchina attaccata (client) un programmino in Javascript il quale prende una stringa casualmente generata (tipicamente di tutte A) composta di un byte in meno rispetto al testo in chiaro da cifrare (e rispetto al modulo minimo di testo da cifrare che prende solitamente in input l'algoritmo di cifratura) e la antepone al testo in chiaro nel momento della cifratura. L'algoritmo di cifratura taglierà il testo in chiaro quanto basta per poterlo cifrare un po' alla volta: in questa maniera l'unica cifra/carattere che l'attaccante non conosce è l'ultima della stringa, si avranno allora 254 possibilità di indovinare di quale cifra o carattere si tratta.
Si continua così di volta in volta generando stringe casuali con sempre meno caratteri, così da avere come incognita solo ed unicamente il byte finale della stringa.
Poiché si tratta del codice che autorizza la richiesta del client, è sufficiente indovinare solo la prima parte del messaggio e non necessariamente l'intera conversazione.
Si tratta di un attacco deterministico che non funziona sulle versioni successive alla prima di TLS, grazie ad una modifica del vettore di inizializzazione.

CRIME. Attacco a TLS che funziona per tutte le altre versioni. Sfrutta una certa proprietà degli algoritmi di compressione (così che una parola corretta ed una errata si comportino in maniera diversa in seguito alla loro annessione al pacchetto).

L'unico modo per evitare la buona riuscita dell'attacco è di evitare di compiere una compressione o di cambiare radicalmente l'algoritmo di compressione di TLS.