



# COMPUTER AND NETWORK SECURITY

## Computer and Network Security

Ingegneria dell'Automazione

**Autore:** Andrea Efficace

**Istituto:** Università di Roma Tor Vergata

**Data:** 31 gennaio 2022



# Indice

<b>1</b>	<b>Introduzione sulla Crittografia</b>	<b>1</b>
1.1	Cifratura . . . . .	1
1.1.1	Tecniche di Cifratura . . . . .	1
1.2	Confidentiality, Integrity, Availability . . . . .	3
1.3	Indistinguishability under Chosen Plaintext Attack . . . . .	5
<b>2</b>	<b>Stream Ciphers</b>	<b>6</b>
2.1	Gli Initialization Vectors . . . . .	6
2.2	WEP: errori di design . . . . .	7
2.2.1	Errori di Confidentiality . . . . .	8
2.2.2	Errori di Authentication . . . . .	8
2.2.3	Errori di Integrity . . . . .	10
<b>3</b>	<b>MSG Authentication come MSG Integrity</b>	<b>11</b>
3.1	Message Authentication con Chiave Simmetrica . . . . .	12
3.1.1	Message Authentication Code . . . . .	12
3.1.2	Vulnerabilità ai Man In The Middle Attacks . . . . .	13
3.1.3	Vulnerabilità ai Reply Attacks . . . . .	13
3.2	Funzioni Hash Crittografiche . . . . .	14
3.2.1	Caratteristiche di una Cryptographic Hash Function . . . . .	14
3.2.2	Inserire un Segreto in un Hash . . . . .	16
3.3	Hash Based Message Authentication Code . . . . .	17
<b>4</b>	<b>User Authentication</b>	<b>20</b>
4.1	Password vs Segreto . . . . .	21
4.2	Authentication Protocols . . . . .	22
4.2.1	Password Authentication Protocol (PAP) . . . . .	22
4.2.2	Challenge Handshake Authentication Protocol (CHAP) . . . . .	23
4.2.3	PAP vs CHAP - chi scegliere? . . . . .	24
4.3	One Time Password (OTP) . . . . .	26
4.4	Two-Factor Authentication . . . . .	27
4.5	Mutual Authentication . . . . .	28
4.5.1	Come prevenire Reflection Attacks? . . . . .	29
<b>5</b>	<b>Cellular Authentication</b>	<b>31</b>
5.1	Rete 2G (GSM) . . . . .	31
5.1.1	Vulnerabilità delle reti GSM . . . . .	32
5.2	Rete 3G (UMTS) . . . . .	34

---

5.2.1	(Pochi) Dettagli su IMSI-Catcher Attack . . . . .	35
5.2.2	(Pochi) Dettagli su Rete 4G/5G (LTE) . . . . .	35
<b>6</b>	<b>RADIUS</b>	<b>36</b>
6.1	Funzionamento e Autenticazione . . . . .	36
6.1.1	Servizio di Proxy . . . . .	37
6.2	Pacchetti RADIUS . . . . .	38
6.3	Autenticazione in dettaglio . . . . .	39
6.3.1	Challenge in PPP-CHAP . . . . .	40
6.4	Servizi di Sicurezza . . . . .	40
6.4.1	Vulnerabilità nella Sicurezza . . . . .	41
6.4.2	Confidenzialità della password in PPP-PAP . . . . .	42
6.4.3	Cattiva gestione dei PRNG . . . . .	44
6.5	DIAMETER . . . . .	45
6.5.1	Feature Principali in DIAMETER . . . . .	45
6.5.2	Pacchetti DIAMETER . . . . .	46
6.5.3	Agenti Intermedi . . . . .	48
<b>7</b>	<b>Block Ciphers</b>	<b>50</b>
7.1	Funzionamento di Base . . . . .	50
7.2	Modes of Operations . . . . .	51
7.2.1	Cipher Block Chaining: CBC . . . . .	52
7.2.2	Cipher Feedback Mode e Output Feedback Mode . . . . .	53
7.3	Short Cycle Problem . . . . .	55
7.4	Counter Mode . . . . .	55
<b>8</b>	<b>Authenticated Encryption with Associated Data</b>	<b>57</b>
8.1	Meccanismi di AE . . . . .	58
8.1.1	Associated Data . . . . .	60
8.2	AES-GCM: Galois Counter Mode . . . . .	60
8.2.1	Misuse Resistance per GCM . . . . .	62
<b>9</b>	<b>Asymmetric Cryptography</b>	<b>63</b>
9.1	Public Key Encryption & Digital Signature . . . . .	63
9.2	Key Agreement Algorithms . . . . .	65
9.3	Diffie Hellman . . . . .	66
9.3.1	Implementazioni di DH . . . . .	67
9.4	RSA . . . . .	68
9.4.1	RSA Key Transport . . . . .	70
9.4.2	Bleichenbacher's Oracle . . . . .	71
9.5	Pub-Key Infrastructure . . . . .	72

---

9.5.1	Digital Certificate . . . . .	74
9.5.2	Certificate Revocation List . . . . .	76
9.5.3	Public Key Cryptography Standards . . . . .	77
9.5.4	Certificate Chains . . . . .	78
9.5.5	Merkel's Tree . . . . .	79
9.5.6	Certificate Transparency . . . . .	81
9.6	Authentication with Asymmetric Crypto . . . . .	84
<b>10</b>	<b>TLS - Transport Layer Security</b>	<b>85</b>
10.1	Il Supporto Applicativo . . . . .	86
10.2	TLS Protocol Stack . . . . .	86
10.2.1	TLS Record Protocol . . . . .	87
10.3	Handshake Protocol . . . . .	89
10.3.1	Fase 1: Hello . . . . .	91
10.3.2	Fase 2: Server Authentication . . . . .	92
10.3.3	Fase 3: Client Authentication . . . . .	93
10.3.4	Fase 4: Finish Message . . . . .	94
10.3.5	Handshake Abbreviato . . . . .	95
10.4	Key Computation . . . . .	95
10.5	Other Minor Protocols . . . . .	98
10.5.1	Change Chiper Spec Protocol . . . . .	98
10.5.2	Alert Protocol . . . . .	99
10.5.3	Renegotiation . . . . .	100
10.6	DTLS . . . . .	101
10.7	Original Sin of TLS (up to v1.2) . . . . .	101
10.7.1	Attacks to Encryption - Padding Oracle . . . . .	102
10.7.2	Solutions of next versions and Final Considerations . . . . .	104
10.8	Major Vulnerabilities . . . . .	105
10.8.1	BEAST Attack: Chosen Boundary Attack . . . . .	106
10.8.2	CRIME Attack: Compression after Encryption leakage . . . . .	108
10.8.3	Downgrade Attack . . . . .	109
10.8.4	Truncation Attack . . . . .	111
10.8.5	Renegotiation Attack . . . . .	112
10.9	TLS v1.3 . . . . .	113
10.10	Perfect Forward Secrecy . . . . .	113
10.10.1	Handshake . . . . .	115
10.10.2	Perfect Forward Secrecy con Pre-Shared Key . . . . .	116
10.10.3	0-RTT Data . . . . .	116

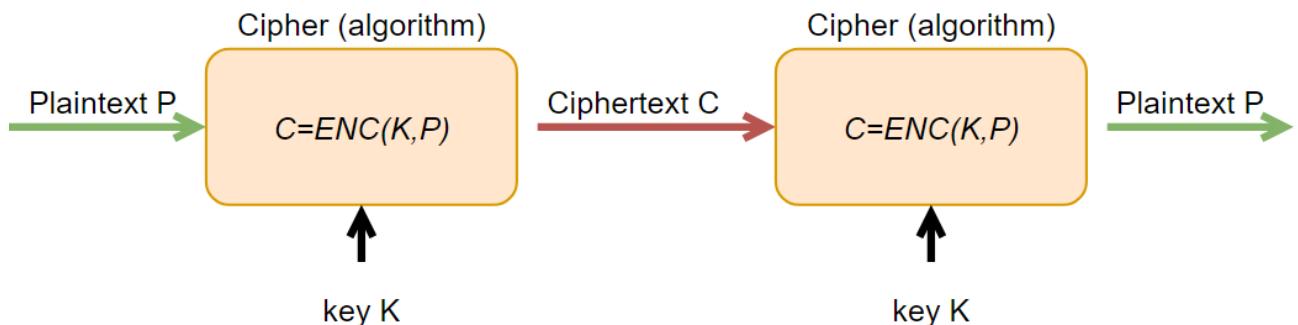
<b>11 IPSec: IP Security</b>	<b>117</b>
11.1 Protocol Structure . . . . .	117
11.1.1 Security Association . . . . .	118
11.1.2 Security Protocol . . . . .	119
11.2 IKEv2 - Internet Key Exchange . . . . .	122
11.2.1 Funzionamento . . . . .	123
11.2.2 Key Computation . . . . .	124
11.3 Vulnerabilità . . . . .	124
11.3.1 Against Downgrade Attack . . . . .	124
11.3.2 Against DoS Attacks . . . . .	126
11.4 VPN . . . . .	128
<b>12 Modern Secret Sharing</b>	<b>129</b>
12.1 Trivial Secret Sharing . . . . .	129
12.2 Shamir Secret Sharing . . . . .	130
12.2.1 Secret Sharing for Multipart Computation . . . . .	135
12.3 Verifiable Secret Sharing . . . . .	138
12.3.1 Pedersen Commitment . . . . .	140
12.3.2 Prime for Dummies . . . . .	141
12.4 Distributed Secret Sharing . . . . .	144
<b>13 Threshold and Policy-Based Cryptography</b>	<b>145</b>
13.1 Asymmetric Ciphers and Hybrid Encryption . . . . .	146
13.1.1 Threshold ElGamal . . . . .	146
13.2 Threshold Signature . . . . .	147
13.2.1 RSA Common Modulus Attack . . . . .	150
<b>14 Mobile Devices Resilient to Capture</b>	<b>152</b>
14.1 Capture-Protection Server . . . . .	152
<b>15 Linear Secret Sharing</b>	<b>156</b>
15.1 Monotone Span Programs . . . . .	157
15.2 Access Structure . . . . .	159
<b>16 Elliptic Curve Cryptography</b>	<b>162</b>
16.1 EC Minor Details . . . . .	162
16.2 EC over $\mathbb{Z}(p)$ . . . . .	164
16.3 EC Crypto . . . . .	165
<b>17 Pairing-Based Cryptography</b>	<b>170</b>
17.1 Bilinear Maps . . . . .	170
17.1.1 MOV Reduction . . . . .	171

17.2 Identity Based Encryption . . . . .	172
--	-----

# Capitolo Introduzione sulla Crittografia

## 1.1 Cifratura

Quando vogliamo cifrare il contenuto di un messaggio in chiaro (*plaintext*) abbiamo bisogno di un cifratore che sia in grado di trasformare il messaggio in un messaggio cifrato (*ciphertext*) e un secondo cifratore in grado di decriptare il messaggio in modo tale da ottenere nuovamente un testo leggibile dal ricevitore.



**Figura 1.1:** Schema di Crittografia Standard

La cifratura è una tecnica che serve **proteggere la confidenzialità dei dati** trasformandoli in qualcosa di incomprensibile ad un interprete generico. Come indicato nella fig. 1.1, la **trasformazione** messa in atto da un cifratore **deve essere reversibile**, in modo tale che il destinatario del messaggio cifrato sappia ritrasformare il testo cifrato per poterlo leggere. Al fine di fare ciò, entrambe le parti devono conoscere due fattori:

1. **Chiave:** la chiave è l'informazione che **deve** essere segreta e condivisa solo ed esclusivamente tra il mittente e il destinatario. Può essere di tipo **simmetrico** se viene usata la stessa chiave per cifrare e decifrare il messaggio, **asimmetrica** altrimenti.
2. **Algoritmo di Cifratura:** il meccanismo che applica la trasformazione al plaintext sulla base della chiave che viene segreta. Tipicamente gli algoritmi sono pubblici e standardizzati.

### 1.1.1 Tecniche di Cifratura

#### Proposizione 1.1 (Substitution Cypher)

Tecnica di cifratura che opera per sostituzione di lettere secondo un preciso schema (es:  $A \rightarrow B$ ). □

Il problema di sostituire secondo uno schema, detto dizionario, è che con un'analisi più o meno attenta (*ad esempio analizzando la frequenza di ripetizione delle lettere*) è possibile dedurre lo schema di sostituzione e ricostruire il messaggio originale senza la chiave che era stata usata per cifrarlo.

**Proposizione 1.2 (Vernam Cypher)**

Tecnica di cifratura che, dato un messaggio di lunghezza  $L$ , usa una chiave **randomica**<sup>1</sup> di lunghezza  $\hat{L} = L$  che vengono unite tramite  $\oplus$  (xor) per generare il messaggio cifrato. Infine, sempre tramite  $\oplus$ , il messaggio viene decifrato usando la stessa chiave.

La chiave deve essere generata in modo **truly random** ossia veramente casuale. Impossibile da fare tramite funzioni semplici, che generano sequenze di numeri pseudo-random. □

**Proprietà [XOR]** Ricordiamo che:  $A \oplus 0 = A$ ,  $A \oplus 1 = A'$ ,  $A \oplus A' = 1$ .

Il vernam cypher è considerato il meccanismo di cifratura *perfetto* in quanto se le ipotesi su cui si basa sono soddisfatto, il messaggio ottenuto in uscita è indecifrabile senza l'opportuna chiave. Tuttavia non è realizzabile, per i seguenti motivi:

1. **Una chiave per ogni messaggio:** La robustezza del cifratore si basa sull'utilizzo di una chiave, completamente randomica, per ogni messaggio che viene inviato. Supponiamo di usare la stessa chiave  $K$  per due messaggi cifrati  $C_1$  e  $C_2$  e facciamone lo  $\oplus$ :

$$C_1 \oplus C_2 = (M_1 \oplus K) \oplus (M_2 \oplus K) = M_1 \oplus M_2$$

In questo modo abbiamo dedito 1 contenuto del messaggio cifrato senza neanche sapere la chiave. Noto questo, tramite un *Chosen Plaintext Attack* (attacco tramite invio di messaggi scelti), potremmo dedurre anche la chiave.

2. **Ogni chiave deve essere lunga quanto il messaggio:** Non un reale problema computazionale. Per rendere la chiave lunga quanto il messaggio, possiamo ripetere i caratteri della chiave se sono troppo pochi o usare metodi più sofisticati. Tuttavia scomodo dal punto di vista della scalabilità. Per 1GB di messaggio dovremmo disporre una chiave di 1GB.
3. **La chiave deve essere truly random:** impossibile da generare con funzioni di libreria o tecniche particolari, in quanto il numero casuale è definito da un algoritmo e, pertanto, generabile da terzi.

**Corollario 1.1 (Lo scopo della crittografia)**

Quando criptiamo un messaggio dobbiamo ragionare in ottica di **mettersi in sicurezza da una minaccia specifica**, piuttosto che da un pericolo in generale. □

## 1.2 Confidentiality, Integrity, Availability

I principi alla base della cifratura e della crittografia sono i seguenti:

### Definizione 1.1 (Confidentiality)

*Misura adottata per evitare che le informazioni sensibili arrivino alle persone sbagliate, assicurando che solo chi sia autorizzato possa accedervi.*

### Definizione 1.2 (Integrity)

*Con il concetto di integrità si vuole specificare che, durante la trasmissione di un messaggio, questo arrivi a destinazione integro e senza che qualcuno possa alterarne il valore*

### Definizione 1.3 (Authenticated Encryption with Associated Data)

*Un sistema in grado di garantire sia integrità che confidenzialità è detto di tipo AEAD.*

### Definizione 1.4 (Availability)

*Mantiene aggiornato e bug-free il software che veicola lo scambio del messaggio.*

Difatti, il modello di cifratura utilizzato fino ad ora non copre i casi in cui una minaccia intenta ad effettuare un attacco di tipo *Man in the Middle (MITM)* riesca a modificare un bit o più del messaggio, modificandone il contenuto e passando inosservato. Possiamo dire che:

### Proposizione 1.3

*Una cifratura mantiene un'elevata confidenzialità ma una scarsa integrità.*

**Esempio 1.1 RFID:** E' un sistema composto, solitamente, da un *TAG* e un *READER* (per esempio la carta e il pos). Per il funzionamento è necessario che tra i due oggetti ci sia una mutua autenticazione in quanto sia il *TAG* che il *READER* potrebbero non essere autentici.

Tipicamente un *TAG* è composto da:

- Un segreto, statico,  $S$
- Una chiave, temporanea,  $K$

Il messaggio contenente il segreto viene cifrato tramite  $\oplus$  con la chiave associata all'utente.

Il *READER* invece detiene un database con le chiavi degli utenti a cui fare riferimento.

Quando il *TAG* viene a contatto con il *READER*, quest'ultimo sarà in grado di decifrare il messaggio qualora fosse genuino, trovando un riscontro con qualche entry del database.

In questo sistema, ogni chiave è generata **pseudo-randomicamente** a partire dalla chiave precedente. Una volta ottenuto il segreto, il *READER* associa una nuova chiave temporanea al *TAG* in questione, aggiornando il suo database.

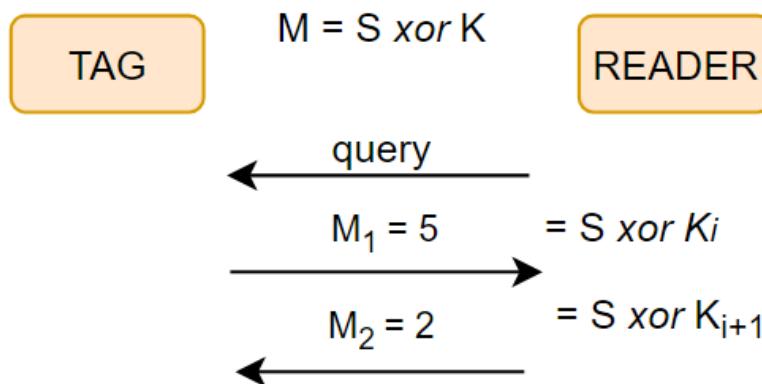
**Osservazione:** In un *RFID* il segreto viene criptato usando una *pseudo-random key*.

Tale sistema presenta due problemi. Supponiamo che  $S$  sia truly random e  $K$  pseudo-random, allora:

- Se  $S$  è il plain-text, andando in  $xor$  con  $K$  **viola la proprietà 3** dei cifratori, in quanto la chiave non è truly random.
- Se  $K$  è il plain-text,  $S$  fa da chiave, ma **viene violata la proprietà 1**, in quanto la stessa chiave è associata a messaggi differenti.

Poiché l'algoritmo pseudo-random per generare la chiave è noto, è possibile dedurre tutte le possibili combinazioni di bit che costituiscono  $K$ . Pertanto, facendo uno  $xor$  tra due messaggi consecutivi, è possibile ottenere un testo con cui fare  $xor$  con una delle due chiavi, ottenendo delle combinazioni possibili per il plain-text originale.

Dato che  $M_1 \oplus M_2 = 7$ , supponendo che  $K_i = 3$ ,  $k_{i+1} = 4$ , confrontando con la mappa delle chiavi pseudo-random è possibile dedurre sia chiave, che messaggi che lo schema con cui vengono generate le chiavi.



**Figura 1.2:** Esempio di scambio messaggi in RFID



## 1.3 Indistinguishability under Chosen Plaintext Attack

Definire un "*cifratore sicuro*" è generalmente difficile, in quanto una buona definizione deve tenere in considerazione la minaccia rispetto alla quale è considerato sicuro. Tipicamente viene adottata infatti la convenzione ***Goal-Model***.

Poiché un tipo di attacco è il cosiddetto ***Chosen Plaintext Attack (CPA)*** una definizione che possiamo dare è che sia di tipo **IND-CPA**, ovvero, non deve rilasciare informazioni ad un'entità malevola quando questa possa attaccare il sistema sfruttando un plaintext scelto. Facciamo un esempio:

1. Supponiamo che un attaccante  $A$  abbia a disposizione due messaggi  $M_0, M_1$  della stessa dimensione, che invia **in chiaro** ad un utente.
2. L'utente sceglie randomicamente uno dei due, rispondendo ad  $A$  con un messaggio cifrato:  
 $C = ENC(K, M_0/M_1)$ 
  - L'attaccante ha il 50% di probabilità di ricevere uno dei due messaggi.
3. Supponiamo di avere a disposizione un **Oracolo**, in grado di conoscere l'*associazione chiavetesto cifrato*. L'attaccante invia  $M_0, M_1$  all'oracolo, che conoscendo  $K$  può rispondere con  $C_0 = ENC(K, M_0), C_1 = ENC(K, M_1)$ .
4. A questo punto l'attaccante confrontando le risposte ricevute, può individuare la chiave con cui viene eseguita la cifratura, con la stessa probabilità dei messaggi spediti all'utente.

**Osservazione:** La chiave rappresenta un fattore fondamentale per garantire la confidenzialità, per questo sarebbe ideale avere a disposizione una chiave diversa ogni volta. In questo modo un attaccante non avrebbe possibilità di capire quale messaggio cifrato corrisponde ai suoi plaintext perché ogni volta riceverebbe un messaggio diverso. ■

Le proprietà di un sistema dotato di "*Indistiguability*" sono:

- Un avversario non deve essere in grado di ricavare alcuna informazione su un plain text a partire dal cipher text, anche se egli può avere accesso ad un oracolo.
- L'encryption deve essere randomizzata, poiché uno stesso messaggio deve sempre essere criptato in diversi cipher text, che deve essere indistinguibile da uno randomico.
- Se la sequenza si ripete essa deve essere criptata con una chiave differente

### Corollario 1.2 (XOR e Segretezza)

Dato il funzionamento dell'operatore  $\oplus$ , se cifriamo una sequenza di bit informativi con una sequenza di bit perfettamente randomica, possiamo dire con certezza che la sequenza generata in uscita sarà ancora casuale al 100%. La probabilità di indovinare il plaintext a partire da quello cifrato è, a priori, quella di indovinare un numero casuale. Che sarebbe la stessa senza aver visto alcun cipher-text. □

# Capitolo Stream Ciphers

Sono una categoria di cifratori, tipicamente veloci nell'elaborazione del testo. Un esempio di stream cipher è il *Vernam Cipher*. L'idea di base per uno stream cipher è quella di utilizzare una **chiave statica** che, unita ad un vettore di bit, permette di ottenere un messaggio cifrato.

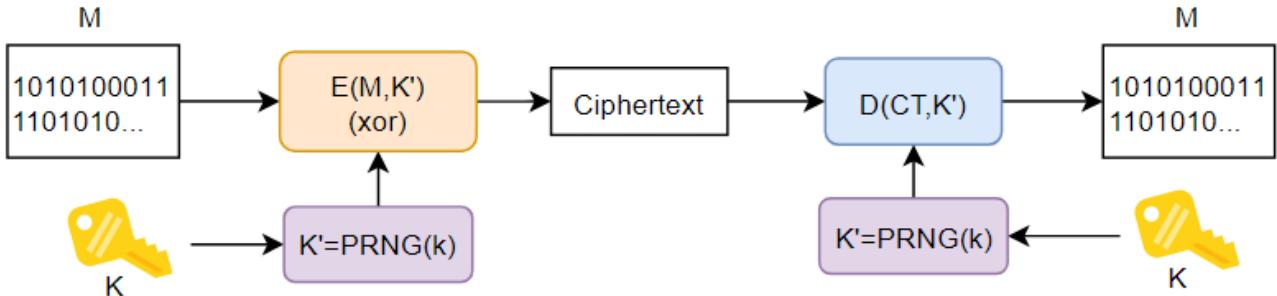


Figura 2.1: Stream Ciphers Scheme

In modo tale da rendere uno stream cipher sicuro da un *CPA*, introduciamo il concetto di **Keystream**. Poiché una caratteristica richiesta per i cifratori era, la necessità di avere una chiave della stessa lunghezza del messaggio che fosse *sempre* nuova, possiamo usare la chiave segreta precedentemente scambiata in qualche modo, come seed per un keystream generato tramite un algoritmo *pseudorandom* che va in *xor* con il messaggio.

**Osservazione:** E' importante non confondere la chiave random del One Time Pad con il keystream generato tramite algoritmo pseudorandom. ■

**Osservazione:** L'algoritmo che genera il keystream è ciò che differenzia un cifratore dall'altro. Alcuni esempi sono **RC4**, **Salsa20**, **ChaCha20**. ■

## Proposizione 2.1 (Pro e Contro)

**PRO:** Lo schema in fig. 2.1 permette di ottenere un testo cifrato in cui se una sotto-sequenza di bit si ripete, ognuna verrà cifrata *sempre* in modo diverso.

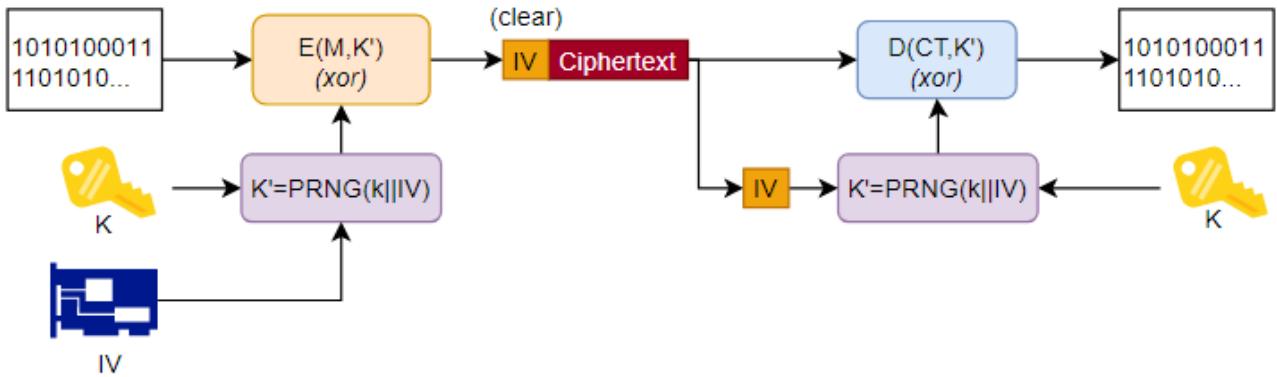
**CONS:** Se il messaggio viene cifrato una seconda volta, il testo cifrato sarà comunque *sempre* uguale a se stesso. ■

## 2.1 Gli Initialization Vectors

Poiché uno stream cipher implica l'utilizzo di una chiave statica e l'algoritmo di pseudo-random è deterministico per definizione, un messaggio verrà cifrato sempre nello stesso modo. Questo significa che perdiamo la proprietà di *IND-CPA*.

Gli Initialization Vectors sono dei vettori di numeri generati randomicamente ad ogni nuovo messaggio inviato che permettono di mantenere una *sicurezza semantica*<sup>1</sup>a patto che *non si ripetano MAI*.

<sup>1</sup>Un cifratore è **semantically secure** se è impossibile estrarre informazioni sensibili del plain-text dal cipher-text.

**Figura 2.2:** Stream Ciphers with IV

Abbiamo diversi modi per generare degli IV che siano sempre nuovi, al fine di applicare lo schema in fig. 2.2:

- **(Truly) Random:** ovvio. Di positivo ha che non può essere predicibile, ma non realizzabile se non affidandosi ad effettive fluttuazioni naturali.
- **Sequence Number:** un numero di sequenza può essere sempre nuovo, ma può essere predetto, ad esempio identificando dopo quanti cicli la sequenza inizia a ripetersi; basta catturare un numero sufficiente di pacchetti o fare in modo di individuare i momenti in cui la rete si riavvia, per sincronizzarsi con la scheda in trasmissione.

## 2.2 WEP: errori di design

Il sistema WEP (Wired Equivalent Privacy) era il sistema di sicurezza adottato all'inizio dei sistemi WiFi, prima di WPA/WPA2; Consisteva in uno stream cipher basato su RC4 (State-of-Art per il tempo, oggi obsoleto) per generare il keystream.

$$ENC(K, MSG) = MSG \oplus RC4(IV, K)$$

Tuttavia, in WiFi c'è un grosso problema legato alla perdita dei pacchetti durante una trasmissione, pertanto, suddividendo il messaggio in diversi pacchetti, per ognuno di questi pacchetti è necessario generare un nuovo keystream (ovvero un nuovo IV) e inoltre è necessario riconoscere il punto di perdita per ricominciare a trasmettere.

**Osservazione:** Tanti pacchetti significano tanti inizialization vector: anche usando numeri randomici, con pochi bit abbiamo tanta ripetizione. Usare numeri di sequenza non è praticabile in quanto sarebbe facilmente sincronizzabile.

Se collezionando i pacchetti un attaccante dovesse cominciare a fare uno *xor* di due messaggi potrebbe risalire alla chiave, risultando sensibile ad attacchi di tipo CPA o KPA (*known plaintext attack*) ■

## 2.2.1 Errori di Confidentiality

Riguardo alla confidentiality, i problemi furono principalmente due:

1. *Lunghezza IV troppo corta*: 24bit. La probabilità che l'IV si ripetesse in pochi frame era alta e, difatti:  $24 \rightarrow 2^{24} = 16.777.216$  frames.  
Assumendo di inviare 1500byte/frame  $\sim 7$ Mbps, in circa 8 ore si può trovare la chiave. Inoltre, è stato dimostrato che aumentare la dimensione della chiave non risolveva il problema, poiché l'attacco scala linearmente con la lunghezza dell'IV.
2. *Generatore di IV lasciato all'implementazione*: Lo standard non definiva il modo con cui gli IV dovessero essere generati, esponendo il sistema ad un utilizzo inesperto o improprio. Ad esempio, mettere tutti 0 o tutti 1 come keystream annienta la sicurezza in quanto lo xor di quei due stream fa sì che non venga cifrato lo stream key+IV. Ciò non toglie il fatto poi che basta un semplice reboot del sistema a re-inizializzare a tutti 0 l'IV (ripetizione).

Un attaccante, banalmente collezionando i pacchetti, poteva costruire un dizionario di coppie IV-keystream ed individuare lo schema di generazione, deducendo la chiave.

## 2.2.2 Errori di Authentication

Diamo prima una definizione di autenticazione:

### Definizione 2.1 (Authentication)

*L'autenticazione è un servizio essenziale nella sicurezza che consiste provare la propria identità digitale. Essa consiste nel provare che le credenziali di un individuo siano autentiche. Un'autenticazione andata a buon fine è tale se riesco ad essere sicuro che il soggetto che accede al servizio in un dato momento è lo stesso che ha acceduto in passato.*

### Proposizione 2.2 (Mezzi di Autenticazione)

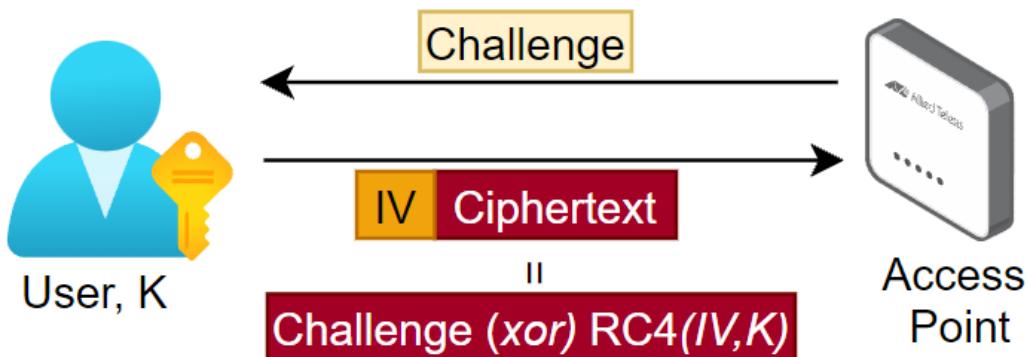
Alcuni mezzi di autenticazione sono:

- **Conoscenza di un segreto**: Password, PIN, chiave; Dimostro di conoscere un segreto.
- **Possesso esclusivo di un requisito fisico**: Dimostro di avere qualcosa (smart card o dispositivi fisici), autenticazione basata su unicità di un hardware (*Physically Unclonable Function*).
- **Autenticazione biometrica**: Dimostrare di essere qualcuno sulla base del proprio DNA (impronta digitale e simili).
- **Biometrika dinamica**: Azioni di un soggetto, riconoscimento della voce, calligrafia o movenze.

L'idea di *WEP* non era quella di autenticare realmente gli utenti, ma di costruire un database di chi poteva avere accesso alla rete. Tutti gli autorizzati nella rete avrebbero potuto comunicare senza ausilio crittografico, mentre chi si trovava fuori la rete avrebbe osservato invece tutto cifrato. Da qui la definizione di “**“Wired equivalent privacy”**”, ovvero la conoscenza di un segreto rende un utente fidato.

Il processo di autenticazione era di tipo *challenge*, con la stessa chiave pubblica. Ovvero:

1. L'access point invia un valore numerico in chiaro all'utente. Quel numero consiste nella **challenge** da soddisfare.
2. L'utente che accede invia all'access point un pacchetto contenente l'IV (deciso dall'utente), in chiaro, e la challenge cifrata tramite *xor* con  $RC4(IV, K)$ .



**Figura 2.3:** Authentication in WEP

Dato che la chiave è pubblica, l'access point può verificare la conoscenza della chiave da parte dell'utente semplicemente eseguendo a sua volta lo *xor*, applicando  $RC4$  all'IV spedito in chiaro.

Le debolezze di questo sistema di auth sono le seguenti:

1. Inviando la challenge in chiaro, se si dispone della relativa risposta, eseguendo

$$\text{Ciphertext} \oplus \text{Challenge} = RC4(IV, K)$$

si può derivare il keystream e da li è possibile produrre pacchetti firmati senza conoscere la chiave.

2. Poiché l'initialization vector è deciso dall'implementatore, l'access point non ha modo di capire se la persona che invia la richiesta di auth è fidata. Difatti, eseguendo:

$$\text{Challenge} \oplus \text{Challenge} \oplus RC4(IV, K) = RC4(IV, K)$$

Esponiamo il keystream e permettendo diversi attacchi.

Dal punto 1, capiamo che WEP finisce per implementare un KPA in quanto chiunque conosce la challenge può, costruendo un dizionario di coppie  $(IV, RC4(IV, K))$ , attendere che un IV si ripeta e violare il sistema.

Dal punto 2, si può vedere che un MITM in ascolto sul canale, può introdursi nella rete al primo IV che trova riscontro nel dizionario e da quel momento in poi restare autenticato per tutto il tempo che vuole, in quanto sarà sempre lui a decidere quale sarà l'IV da usare.

**Osservazione:** Una possibile soluzione, sarebbe potuta essere quella di **inviare l'IV insieme alla challenge**, in modo tale da non lasciare spazio agli utenti. ■

### 2.2.3 Errori di Integrity

Esistono diversi algoritmi per garantire l'integrità di un pacchetto. WEP al tempo implementava  $CRC - 32$ , un un algoritmo di error detection che può tener conto dei bit di partenza tramite bit di parità (posti alla fine del messaggio).

#### Teorema 2.1

$CRC - 32$  è lineare nelle operazioni di  $\oplus$ . E' possibile modificare precisi bit del plaintext per produrre lo stesso checksum del messaggio non modificato.



**Dimostrazione** La struttura del pacchetto WEP era costituita dal messaggio  $M$  seguito dal codice di integrità  $CRC32(M)$ . Quando il messaggio viene inviato, tutti i bit vengono criptati nel seguente modo:

$$C = M | CRC32(M) \oplus RC4(IV, K)$$

Consideriamo un vettore  $\delta$  di bit, lungo quanto  $M$  e contenente 1 nelle posizioni dei bit che si intende modificare nel messaggio originale.

Calcoliamo  $CRC32(\delta) = c(\delta)$  e calcoliamo infine:

$$\begin{aligned} C' &= C \oplus \{\delta, c(\delta)\} \\ &= [RC4(IV, K) \oplus \{M, c(M)\}] \oplus \{\delta, c(\delta)\} \\ &= RC4(IV, K) \oplus \{M \oplus \delta, c(M) \oplus c(\delta)\} \\ &= RC4(IV, K) \oplus \{M', c(M \oplus \delta)\} \\ &= RC4(IV, K) \oplus \{M', c(M')\} \end{aligned}$$

Vediamo quindi che unendo al messaggio cifrato la concatenazione  $\{\delta, c(\delta)\}$  la parità misurata dall'algoritmo di controllo non risulta alterata grazie alla parità.



**Osservazione:** In sintesi, il fatto che  $CRC(32)$  sia lineare, risulta una debolezza per il sistema.



#### Proposizione 2.3

Un buon sistema di integrità dovrebbe garantire che:

1. L'integrity checker deve essere non lineare.
2. l'integrity checker deve esplicitamente includere una chiave, in quanto cifrature esterne non forniscono alcuna garanzia.



**Osservazione:** Se si volesse tentare un attacco di tipo **Message Injection** di un messaggio  $M$  sarebbe ancora più facile. Una volta ottenuta una coppia valida di  $(IV, RC4(IV, K))$  basta calcolare  $c(M)$  e inviare  $[M, c(M)]$  in xor con la la firma.



# Capitolo MSG Authentication come MSG Integrity

Abbiamo visto che garantire la confidenzialità dell'informazione di un pacchetto non significa garantire anche la sua integrità. Questo significa che non abbiamo garanzie che un sistema che fornisce una corretta cifratura sia anche inattaccabile al suo interno tramite l'inserimento di un messaggio malevolo. In sintesi:

- Confidenzialità significa nascondere il contenuto di un messaggio per un esterno, con la sicurezza che soltanto il destinatario può leggerlo.
- Integrità significa autenticità del messaggio, ovvero la sicurezza che nessuno possa modificare il messaggio. Né estendendolo, né disturbandolo (questo potrebbe anche essere a problemi di telecomunicazione, che tralasciamo)

Queste osservazioni sono vere sempre, a meno di parlare di AEAD (1.3). In generale, quello che vogliamo è la sola integrità.

**Esempio 3.1 Il Curriculum** chiunque dovrebbe essere in grado di leggerlo, solo il proprietario dovrebbe essere in grado di modificarlo. ■

**Esempio 3.2 Integrità del One Time Pad** Sappiamo che il One Time Pad (vernam cipher) è il miglior cifratore possibile, se le sue ipotesi sono rispettate. Eppure, se un'attaccante modificasse un cipher-text, l'uscita sarebbe diversa. Infatti: ■

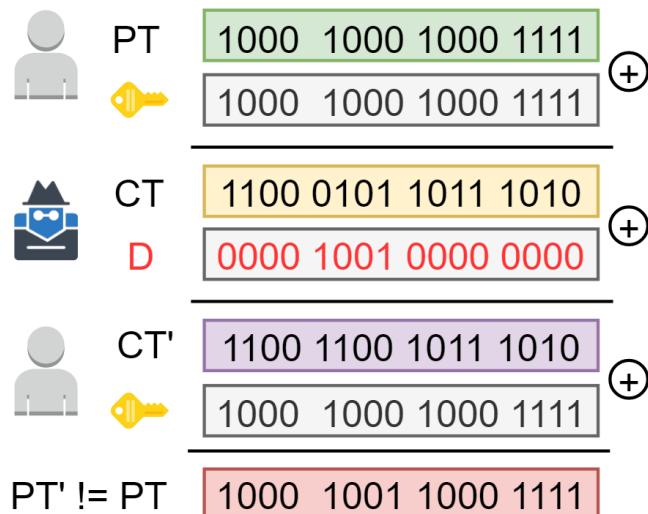
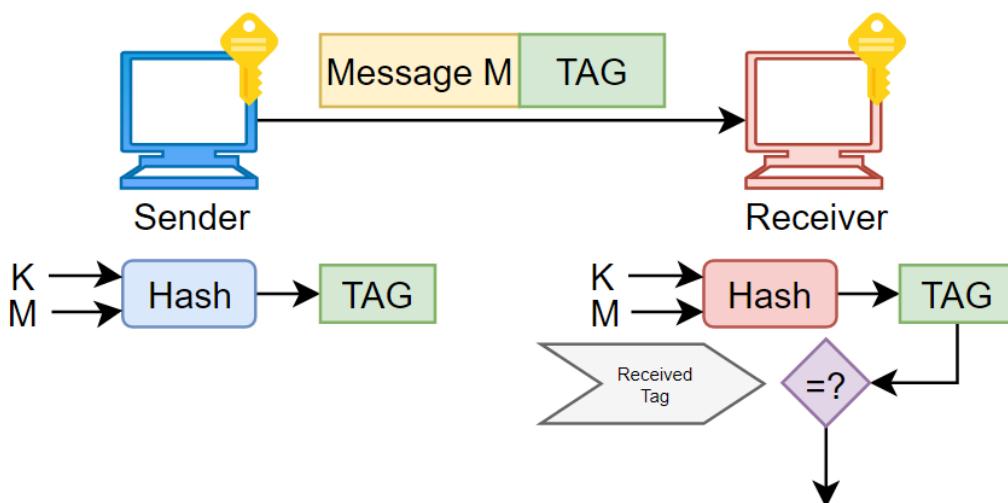


Figura 3.1: Fallimento del Vernam Cipher

## 3.1 Message Authentication con Chiave Simmetrica

Supponiamo che Sender e Receiver conoscano una chiave  $K$  usata *esclusivamente* per il servizio di integrità. Vogliamo aggiungere al messaggio un pacchetto che serva da **integrity check**. Chiamiamo **TAG** questa aggiunta.

Un modo per generare questo TAG è quello di usare una funzione hash che leghi il messaggio da inviare con la chiave, producendo un **digest** che farà appunto da TAG. Poiché la chiave è simmetrica, il ricevitore sarà in grado di generare un suo tag che potrà confrontare con quello inviato dal Sender. Se c'è un matching tra i due tag, allora il messaggio è necessariamente autentico.



**Figura 3.2:** Integrity Check Scheme

### 3.1.1 Message Authentication Code

I message authentication code sono una forma più debole di firma digitale (*Digital Signature*). Nella DS nessuno può modificare un messaggio a parte chi l'ha firmato. Con il MAC solo il sender ed il receiver possono farlo; Questo perché entrambi conoscono la chiave.  
Una definizione possibile di sicurezza per un sistema di autenticazione è:

#### Definizione 3.1 (Unforgeability)

*Un messaggio sicuro ed autenticato è inforgiabile, nel senso che un attaccante non deve essere né capace di creare né di modificare un messaggio.*

*Quindi, dati all'attaccante un numero di messaggi passati, non può generare un tag valido o almeno la probabilità di riuscire a forgiare una coppia messaggio/tag autonomamente deve essere praticamente nulla (negligible).*

**Osservazione:** Data la definizione 3.1, escludiamo l'idea di utilizzare un tag corto, in quanto l'unica possibilità dell'attaccante è provare ad indovinare la coppia. Avere un tag lungo implica più meno probabilità di indovinare un numero casuale.

Tipicamente, la lunghezza considerata minima per avere un tag sicuro è di 96bit.

### 3.1.2 Vulnerabilità ai Man In The Middle Attacks

Supponiamo che un attaccante possa intercettare la coppia  $(MSG, TAG)$  di un Sender e volesse modificare il messaggio  $M$  in  $M'$ . Se la funzione hash è considerata **crittograficamente forte**, allora:

- L'attaccante non è capace di calcolare la chiave in quanto  $H(\cdot)$  non è invertibile.
- l'attaccante non deve essere in grado di alterare il tag, in modo tale che  $tag' = H(K, M')$  senza conoscere  $K$
- L'attaccante non deve essere capace di cambiare  $M$  in  $M'$  in modo tale che

$$H(K, M) == H(K, M')$$

Poiché non è possibile prevenire un MITM, il tag fornisce un valido aiuto nell'intercettare un attacco in corso o avvenuto, in quanto il tag risulterà alterato.

### 3.1.3 Vulnerabilità ai Reply Attacks

I reply attacks consistono nel seguente schema: se vengono mandati due messaggi identici, i tag risultanti saranno identici.

#### Proposizione 3.1

*I MAC non proteggono dai reply attack, a meno che i messaggi non si ripetano mai.*

Un esempio tipico di reply attack sono i message spoofing:

#### Definizione 3.2 (Message Spoofing)

*Attacco nel quale il messaggio viene mandato dall'attaccante, ma spacciato per un altro mittente, senza che il destinatario se ne accorga.*

Consideriamo i due piani su cui si svolge la comunicazione tra due utenti, l'applicativo e quello protocollore (il vero responsabile della comunicazione). Supponendo che lo strato applicativo sia ben implementato (i messaggi non si ripetono), è il protocollo che deve garantire la sicurezza.

Il modo con cui possiamo garantire la non ripetibilità a livello protocollore è specificare il modo con cui generiamo il numero randomico che, unito alla chiave, genera il tag. Diamo una rapida revisione:

Tipo	Conseguenza
Sequence Number	Riavviando il sistema si azzera il contatore e potrebbe ripetersi.
Random Number	Ottimo, specialmente se sono true random. Potrebbe esserci un problema di collisione a seconda del numero di bit usati. Inoltre, se c'è collisione, non possiamo capire se un pacchetto si è ripetuto senza memorizzarli pertanto non è una soluzione scalabile.
Time stamp	Il problema di questo è che il tempo deve essere garantito come dato certo. Il NTP (network time protocol) non garantisce integrità. Il riferimento usato dall'operatore telefonico potrebbe essere disturbato con un segnale radio e cambiato.

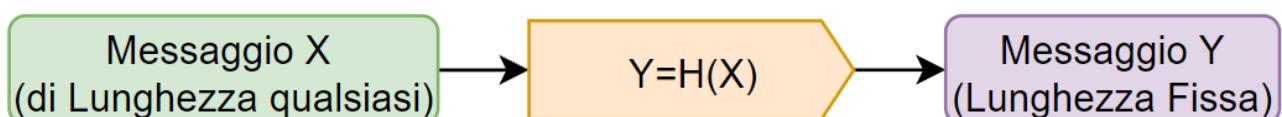
## 3.2 Funzioni Hash Crittografiche

Per costruire un MAC abbiamo bisogno di 2 ingredienti:

1. Una buona funzione hash (es: SHA256)
2. Includere il segreto nella funzione hash

### 3.2.1 Caratteristiche di una Cryptographic Hash Function

A meno di non possedere una funzione hash perfetta, ovvero un random oracle, è importante dove posizionare il segreto rispetto al messaggio all'interno della funzione. Tipicamente, lo schema di una funzione hash è questa:



**Figura 3.3:** Schema di una Funzione Hash

Una funzione hash è crittografica se soddisfa tre diverse proprietà, che rendono impossibile per un attaccante creare o modificare/estendere/rimpiazzare il messaggio di partenza per ottenere un digest valido.

#### Teorema 3.1 (Preimage Resistance)

Dato il risultato di un hash  $Y = H(X)$  è difficile trovare un valore qualsiasi di  $X$  :  $H(X) = Y$

Questo significa che una funzione hash crittografica deve essere resistente ad attacchi brute-force, pertanto, **la dimensione del digest è importante**.

**Esempio 3.3** Supponendo di avere un digest di 16bit, abbiamo  $2^{16}$  possibili combinazioni di 0 ed 1, ovvero la probabilità di indovinare è  $\frac{1}{2^{16}}$ . Supponendo di disporre di una macchina in grado di elaborare 66 Mhash/sec in meno di un millisecondo avremo trovato una soluzione. ■

**Osservazione:** Se dovessero esserci dei bit correlati nella sequenza di bit del digest, per ogni relazione ridurremmo la complessità del calcolo di un fattore 2, ovvero la complessità ha una decrescita esponenziale. Ad esempio, una chiave di  $2^{10}$ bit ha 1024bit di complessità computazionale. Quindi, per ogni 10bit, aggiungiamo un fattore 1000 di complessità. ■

#### Teorema 3.2 (Second Preimage Resistance<sup>1</sup>)

Dato un numero  $X$ , è difficile trovare un numero  $X'$  :  $H(X) = H(X')$

<sup>1</sup>Detta anche Weak Collision Resistance

**Esempio 3.4** Un esempio di funzione che soddisfa la proprietà 1 ma non la 2 è  $y(x) = g^x \text{ mod } p$ , dove  $g$  è un numero noto, non necessariamente grande, e  $p$  è un numero primo grande. Per trovare  $x$  esistono algoritmi che in tempo esponenziale risolvono il problema<sup>2</sup>.

Si può dimostrare che

$$g^x + (p - 1)k \text{ mod } p = g^x \text{ mod } p \forall k$$

e la proprietà 2 è violata.

### Teorema 3.3 (Collision Resistance<sup>3</sup>)

*E' difficile trovare due numeri generici  $X_1, X_2$  tale che  $H(X_1) = H(X_2)$*

<sup>3</sup>Detta anche Strong Collision Resistance

Vediamo un esempio importante per comprendere la terza proprietà:

**Esempio 3.5 Il Paradosso del Compleanno** Dato un numero  $N$  di persone, e presa una data di nascita, la probabilità che qualcuno **NON** sia nato nello stesso giorno è

$$\left(1 - \frac{1}{365}\right)^K = \left(\frac{364}{365}\right)^K; \quad K = N - 1$$

In una classe di  $N = 23$  persone, selezionata una data, restano 22 possibili date, quindi  $K = 22$  e abbiamo il  $\left(\frac{364}{365}\right)^{22} = 94.1\%$ . Ovvero, abbiamo il 6% di possibilità che ci siano due persone nate lo stesso giorno.

Mantenendo lo stesso numero  $N$ , consideriamo ora la probabilità che **due persone** siano nate nello stesso giorno.

$$1 \cdot \left(1 - \frac{1}{365}\right) \cdot \left(1 - \frac{2}{365}\right) \cdots \left(1 - \frac{22}{365}\right) = 49.3\%$$

**Abbiamo circa il 50% di possibilità che ci siano due persone nate lo stesso giorno.**

In un contesto di crittografia, possiamo sostituire lo stesso giorno di nascita con una collisione di una funzione hash, che può dare lo stesso output per due input diversi.

### Teorema 3.4 (Matematica del Paradosso del Compleanno)

*Consideriamo un digest di  $n$ bit. Il numero di messaggi che possiamo produrre è  $N = 2^n$ .*

*Supponendo di avere a disposizione  $K$  messaggi e che  $p$  sia la probabilità che ci sia una collisione, la probabilità che non ci siano collisioni è:*

$$P(\text{no coll.}) = 1 - p = \frac{N!}{N^K(N - K)!} \quad (3.1)$$

**Dimostrazione** Espandendo la formula eq. (3.1), risulta:

$$\begin{aligned} P &= \frac{N}{N} \cdot \frac{N-1}{N} \cdot \frac{N-2}{N} \cdots \frac{N-(K-1)}{N} \\ &= 1 \cdot \left(1 - \frac{1}{N}\right) \cdots \left(1 - \frac{K-1}{N}\right) \\ &= \prod_{i=1}^{K-1} \left(1 - \frac{i}{N}\right) \approx \prod_{i=1}^{K-1} e^{\frac{-i}{N}} = e^{-\frac{\sum_{i=1}^{K-1} i}{N}} \\ 1 - p &= e^{-\frac{K(K-1)/2}{N}} \approx e^{-\frac{K^2}{2N}} \end{aligned}$$

Risolvendo in  $K$  per calcolare il numero di messaggi affinché la probabilità di collisione sia del 50%,

<sup>2</sup>Discrete Logarithm Problem

ovvero  $p = \frac{1}{2}$ , e ricordando che  $N = 2^n$  :

$$\log(1 - p) \approx -\frac{K^2}{2N} \implies K = \sqrt{2N} \cdot \sqrt{\log\left(\frac{1}{1-p}\right)}$$

$$K \approx \sqrt{2} \cdot \sqrt{2\sqrt{N}} \approx 1.177\sqrt{N} \approx 1.777\sqrt{2^N} \approx 2^{\frac{n}{2}}$$

### Proposizione 3.2 (Collision Resistance Level)

Dato un digest di  $n$  bit, la probabilità di avere una collisione del 50% avviene dopo  $2^{\frac{n}{2}}$  messaggi.. □

Dalla proposizione 3.2 capiamo che la dimensione del digest deve essere impostata per ovviare al *birthday paradox*. Alcuni esempi di quanto incide la dimensione su un attacco brute-force sono i seguenti:

- Random 32 bits: 4.3 miliardi di output. 50% **di collisione dopo**  $2^{16} \approx 60.000$  **messaggi**.
- MD5, 128 bits: 50% **di collisione dopo**  $2^{64} = 1.8 \times 10^{19}$  (debole oggi). Inoltre, l'algoritmo è stato rotto nel 2005.
- SHA256, 256 bits: 50% **di collisione dopo**  $2^{128} = 3.4 \times 10^{38}$ . Ok al giorno d'oggi, richiederebbe almeno un miliardo di secoli per l'intera rete di bitcoin mondiale.

### 3.2.2 Inserire un Segreto in un Hash

Poiché il segreto è solitamente la stessa chiave di autenticazione, è computazionalmente difficile per un attaccante costruire un MAC valido. Per come abbiamo definito le funzioni hash crittografiche, un cambiamento nel messaggio in ingresso equivale ad un cambiamento del messaggio in uscita.

**Osservazione:** Il punto di inserimento del segreto è fondamentale, in quanto cambia il risultato del digest. Inoltre, la sua posizione rende più o meno sensibili a diversi tipi di attacchi. □

Supponiamo di usare una buona funzione hash, come SHA256.

### Definizione 3.3 (SHA256)

L'algoritmo SHA256 si basa sull'*Iterative Merkle-Damgard Construction*, un teorema che garantisce che se la funzione di compressione è sicura, tutta la costruzione è sicura.

Prende in input  $K$  bit arbitrari, aggiungendo un padding formato da 1 seguito da tanti 0 per formare un messaggio di  $N \times 512$  bits. Tra gli ultimi 64 bit, viene inserito un numero corrispondente alla lunghezza del messaggio. Vedi fig. 3.4

Il digest viene prodotto con un rapporto di compressione di 3:1, fornendo un IV da 256 bits in ingresso, che viene elaborato con uno dei blocchi da 512 bits. Il risultato della funzione è un blocco di 256 bits che viene fornito come IV al prossimo anello della catena. □



**Nota** Gli IV disponibili per SHA256 sono delle costanti ben definite da standard.

Vediamo come cambia il livello di sicurezza in funzione della posizione del segreto:

- **Segreto alla FINE del messaggio:** l'attaccante potrebbe calcolare una sola volta la prima parte del messaggio e provare tutte le combinazioni possibili per la compressione dell'ultimo blocco. Questo riduce la complessità del brute-force attack e, quindi, della sicurezza. Vedi fig. 3.5
- **Segreto all'INIZIO del messaggio:** inserire il segreto all'inizio permette di eseguire un message-extension attack. All'attaccante è sufficiente calcolare il padding e la lunghezza (ultime parti del messaggio) ed inserire alla fine un plaintext arbitrario, creando un nuovo chunk tale che riapplicando la funzione di compressione si ottiene un messaggio valido **non scritto dall'utente**. Vedi fig. 3.6

 **Nota** In questo caso l'attaccante non serve che conosca il segreto per sfruttare la vulnerabilità.

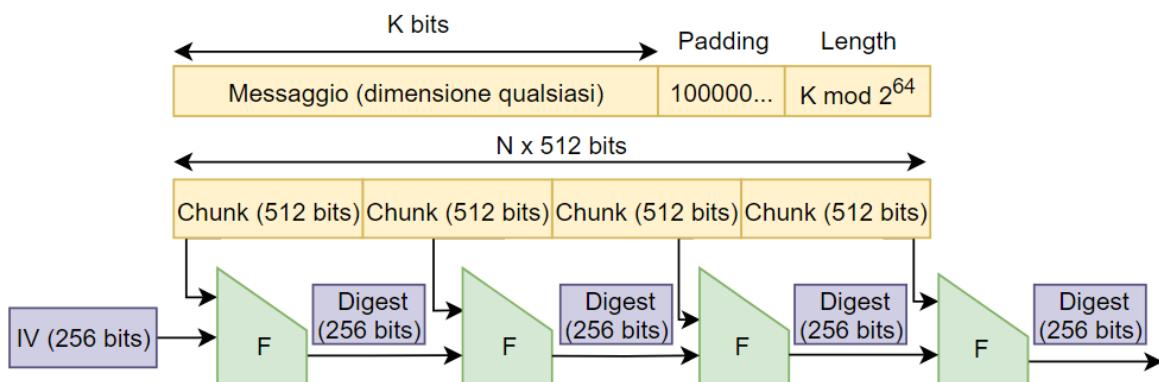


Figura 3.4: SHA256 hash scheme

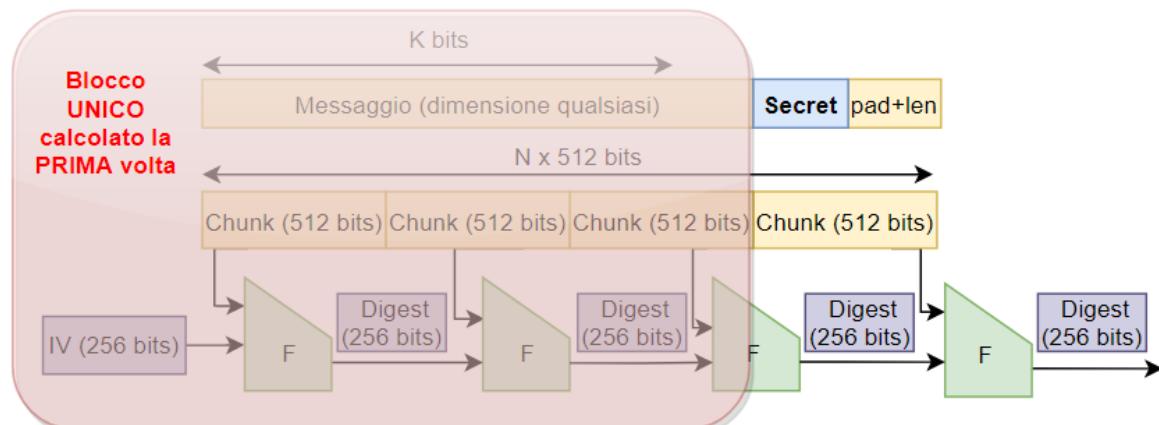
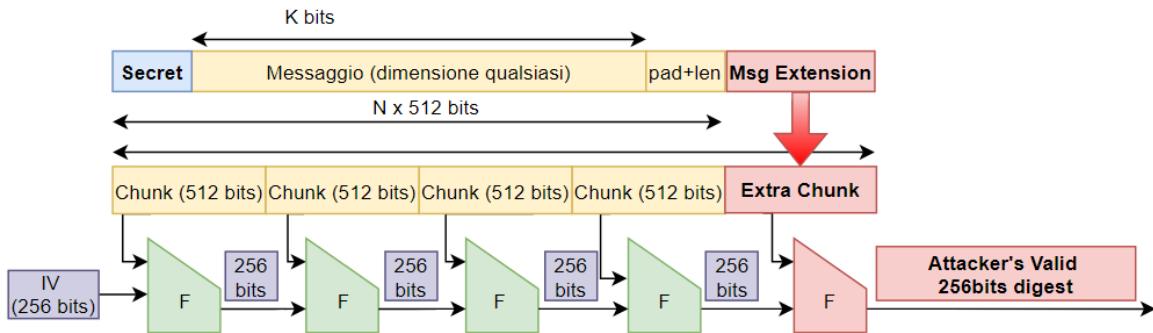


Figura 3.5: Precomputation con Segreto alla fine

## 3.3 Hash Based Message Authentication Code

Abbiamo visto che una funzione hash che risulta sicura dal punto di vista crittografico (cioè valgono: thms da 3.1 a 3.3) non è sufficiente a garantire l'integrità di un messaggio. Esiste però una costruzione che può rendere il tag perfettamente sicuro.



**Figura 3.6:** Message Extension Attack example

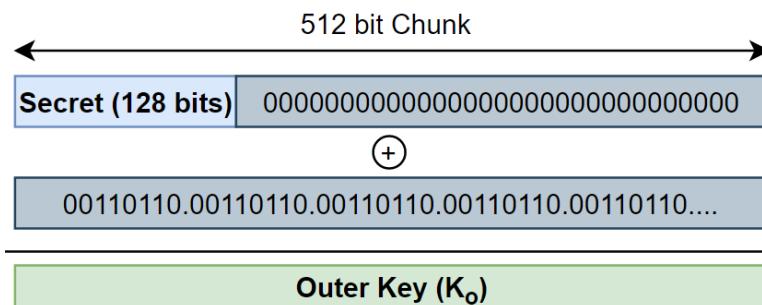
#### Definizione 3.4 (HMAC Construction)

Dato un segreto  $K$  pre-condiviso, e scelta una funzione hash  $H$  che rispetta le ipotesi descritte da thms da 3.1 a 3.3, si crea un codice di autenticazione per un messaggio  $M$  con la seguente costruzione:

$$HMAC_K(M) = H[K^+ \oplus opad || H(K^+ \oplus ipad || M)] \quad (3.2)$$

Dove con  $K^+$  indichiamo l'estensione della chiave  $K$  alla dimensione di block-size necessaria alla funzione hash  $H$  aggiungendo un padding di zeri. I numeri  $opad$  e  $ipad$  sono due costanti, ripetute quanto serve, definite da standard rispettivamente come:

- $opad = 0 \times 36 = 00110110$
- $ipad = 0 \times 5C = 01011100$



**Figura 3.7:** SHA256 Key Extension with Outer Pad

La costruzione di HMAC si basa quindi sul possesso di un singolo segreto che viene messo in xor in due modi diversi, per garantire che le due chiavi usate nell'hashing siano diverse. Sebbene l'inner hash prodotto da  $H(K^+ \oplus ipad || M)$  sia vulnerabile ad extension attack (fig. 3.6), la costruzione rende impossibile produrre un tag valido perché il digest finale prodotto dai 2 blocchi aggiuntivi non può essere esteso. Inoltre, la complessità per individuare la chiave tramite brute-force è pari a  $N$  blocchi più i due aggiuntivi dall'hash esterno.



**Nota** HMAC è più sicuro di una generica hash function. Inoltre, anche se la funzione hash presenta dei problemi di collisione, la costruzione risulta ancora sicura, anche se la collisione è calcolabile.

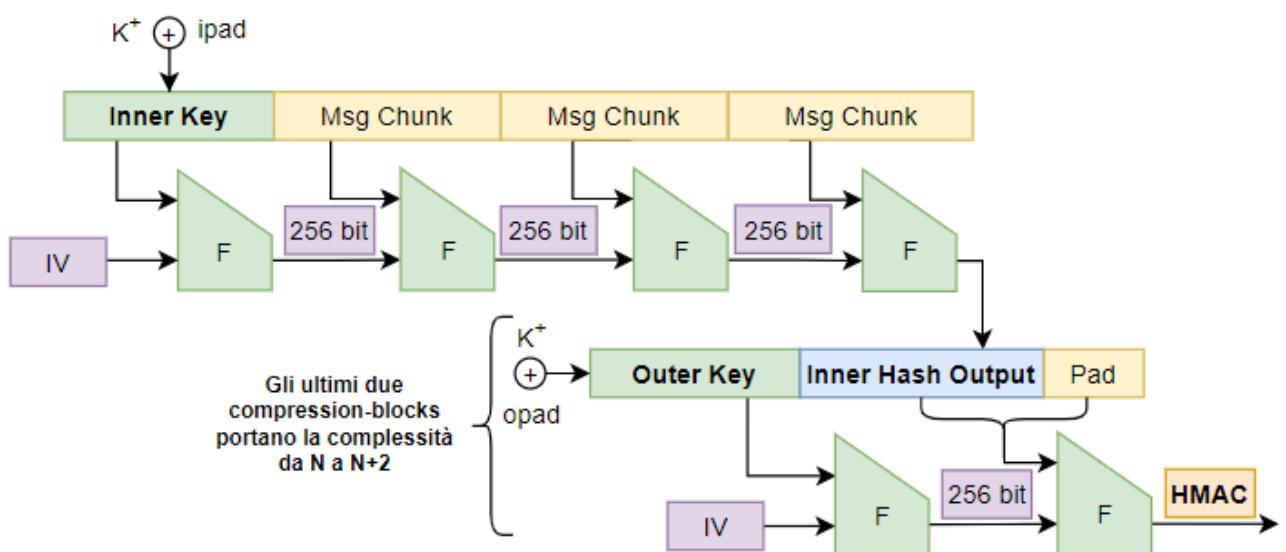


Figura 3.8: HMAC-SHA256 construction

# Capitolo User Authentication

Consideriamo uno scenario in cui è necessario autenticare un utente all'interno del sistema. Al giorno d'oggi confondiamo il concetto di password con quello di segreto, in quanto una password rappresenta la base usata dalla maggior parte dei sistemi di autenticazione moderni. In un contesto di sicurezza, tuttavia, è bene separarli:

- **Segreto:** Sequenza di bit costituita da un numero qualsiasi di caratteri. Tipicamente ad *alta entropia*.

- **Password:** Stringa di caratteri, solitamente alfanumerici, caratterizzata da una *bassa entropia*.

Un modo possibile di vedere l'entropia di un oggetto è come una quantità che rappresenta una stima di quanta casualità esso contiene. Per il nostro contesto, ci viene in aiuto il seguente teorema:

## Teorema 4.1 (Shannon Entropy)

Sia  $X$  una variabile aleatoria discreta, costituita da un numero  $n$  di caratteri ognuno dotato di probabilità di apparire pari a  $p_i$ ,  $i = 1, \dots, n$ . Definiamo il valore, **in bit**, dell'entropia di  $X$  come:

$$H(X) = - \sum_{i=1}^n p_i \log_2 (p_i) \quad (4.1)$$

## Proposizione 4.1 (Information Content)

Il valore informativo dell'evento  $x_i$  dipende da quanto lo stesso  $x_i$  è inatteso, ovvero, il contenuto informativo è funzione di  $\frac{1}{p_i}$ .

Considerando gli eventi come possibili valori di un bit, se un evento ha probabilità di apparire con  $p = \frac{1}{2^b}$  l'information content vale:

$$IC_i = - \log_2 \left( \frac{1}{2^b} \right) = b \text{ bits} \quad (4.2)$$

## Corollario 4.1

Possiamo vedere l'entropia come il valor medio dell'information content di una variabile aleatoria:

$$H(X) = E[IC(X)] = \sum_i^n p_i IC_i = - \sum_{i=1}^n p_i \log_2 (p_i) \quad (4.3)$$



**Nota** Per un insieme di  $N$  eventi, l'entropia può andare da 0 ad  $N$ , a seconda della probabilità degli eventi stessi.

- Se l'entropia è nulla abbiamo un caso di "**Minima Entropia**", ovvero l'evento è deterministico.
- Se l'entropia è  $N$  allora si ha "**Massima Entropia**" e ogni evento è equiprobabile.
- In tutti i casi intermedi, vuol dire che ogni evento non ha la stessa probabilità di accadere.

## 4.1 Password vs Segreto

Tipicamente le password presentano 4 tipi di problemi diversi, difficilmente risolvibili.

### Definizione 4.1 (Overload)

*Riuso della stessa password per diversi siti.*

Infatti, da una statistica americana si evince che:

- ⇒ il 38% degli utenti globali riusano stessa password su diversi siti, il che mette in pericolo la sicurezza.
- ⇒ Risulta anche che il 21% degli utenti che volendosi sentire più sicuri modificano la propria password, lo fanno in modo prevedibile.

### Definizione 4.2 (Restricted Charset)

*Le password sono tipicamente generate da tastiera e non tutti i caratteri sono usati: sui 256 caratteri totali vengono usati solo i 102 fisicamente presenti.*

- ⇒ I caratteri ASCII sono codificati con 1 byte, ovvero una stringa di 8bit corrispondenti a  $2^8 = 256$  possibilità. La probabilità di indovinare 1 bit è quindi di  $\frac{1}{256}$ .
- ⇒ Supponendo di avere un segreto di 8 byte, la probabilità è  $\frac{1}{256^8}$ .
- ⇒ Usando solo caratteri alfanumerici *lower-case*, con 1 byte copriamo al massimo 36 possibili valori. Pertanto la probabilità di indovinare 8 caratteri diventa  $\frac{1}{36^8}$ , che è **decisamente** inferiore.

### Definizione 4.3 (Low Entropy)

*Poiché tipicamente create con l'intento di essere ricordate, le password sono sequenze specifiche non propriamente random nella stragrande maggioranza dei casi.*

**Esempio 4.1 Bit Flip:** Consideriamo  $X_k = \{0, 1\}$  con probabilità  $p_k = 1/2$ . L'entropia vale:

$$H(X) = -\frac{1}{2} \log_2 \left( \frac{1}{2} \right) - \frac{1}{2} \log_2 \left( \frac{1}{2} \right) = \log_2(2) = 1 \text{ bit}$$

**Esempio 4.2 Sequenza Indipendente di 3 bit:** Consideriamo  $X_1, X_2, X_3$  con probabilità  $p_k = 1/2$ . Abbiamo quindi in totale 8 sequenze diverse equiprobabili. L'entropia vale:

$$H(X) = - \sum_{i=1}^8 \frac{1}{2^3} \log_2 \left( \frac{1}{2^3} \right) = 8 \cdot \frac{1}{8} \cdot 3 = 3 \text{ bit}$$

**Esempio 4.3 Coppia di bit Biased:** Consideriamo un bit  $X_k = \{0, 1\}$  i cui valori non sono a frequenza equa, ovvero capitano rispettivamente con probabilità  $p_1 = \frac{1}{4}$  e  $p_2 = \frac{3}{4}$

$$H(X) = -\frac{1}{4} \log_2 \left( \frac{1}{4} \right) - \frac{3}{4} \log_2 \left( \frac{3}{4} \right) = 0.81 \text{ bit}$$

Questo significa che ogni bit trasmesso trasporta 0.81bit di informazione e un'ipotetica sorgente di questo flusso di bit non potrebbe comprimere il suo messaggio più del 19%

**Esempio 4.4 Sequenza di 3 bit Dipendenti:** Consideriamo  $X_1, X_2, X_3$  con probabilità  $p_k = 1/2$  e supponiamo che  $X_2, X_3$  assumano lo stesso valore di  $X_1$ . Questo significa che abbiamo solo due casi possibili e l'entropia sarà:

$$H(X) = -\frac{1}{2} \log_2 \left( \frac{1}{2} \right) - \frac{1}{2} \log_2 \left( \frac{1}{2} \right) = \log_2(2) = 1 \text{ bit}$$

Pertanto, solo 1/3 del messaggio trasmesso sarebbe il reale portatore di informazioni. ■

#### Definizione 4.4 (Predictability)

*Solitamente una password viene scelta sulla base delle proprie esperienze o sulla base delle parole utilizzate da un individuo. Per attacchi mirati (**social engineering**) o tramite collezione di password trovate per il web si possono indovinare facilmente.*

- ⇒ Le password sono soggette ai **Dictionary Attack**: L'idea è molto semplice, si prende un dizionario di parole comuni, sulla base di quelle che si dicono di solito o sui gusti comuni. Ci sono dei database di pubblico dominio di password ottenute da brecce eseguite in sistemi informatici, dove sono riportate anche le password più usate.
- ⇒ La tecnica di **Password spraying attack** è proprio l'azione di provare tutte le differenti password più utilizzate.

**Osservazione:** Abbiamo quindi capito che se i bit di una sequenza sono indipendenti, l'entropia aumenta, e viceversa. Poiché le password sono strutturalmente dotate di minore entropia rispetto ad un segreto, sono necessariamente meno sicure. ■

## 4.2 Authentication Protocols

Un processo di autenticazione non è altro che una prova di conoscenza, nella quale si vuole dimostrare di conoscere una password o un segreto.



**Nota** *Provare la propria conoscenza NON implica, necessariamente, rivelarne il soggetto.*

### 4.2.1 Password Authentication Protocol (PAP)

È l'approccio di autenticazione più semplice possibile e consiste nella trasmissione della password in chiaro sul mezzo di trasmissione, **all'inizio della sessione** (non viene mai ripetuta finché viene chiusa).

#### Proposizione 4.2 (PAP - Password Authentication Protocol)

1. *L'utente in autenticazione invia un messaggio con (ID, PASSWORD) all'authenticator, dotato di un database con scritte tutte le associazioni utente/password.*
2. *L'authenticator controlla che i valori inviati di id e password corrispondano; In caso affermativo garantirà l'accesso.*



**Nota** *In PAP le password sono pre-condivise e trasmesse in chiaro ad ogni nuovo tentativo di autenticazione.*

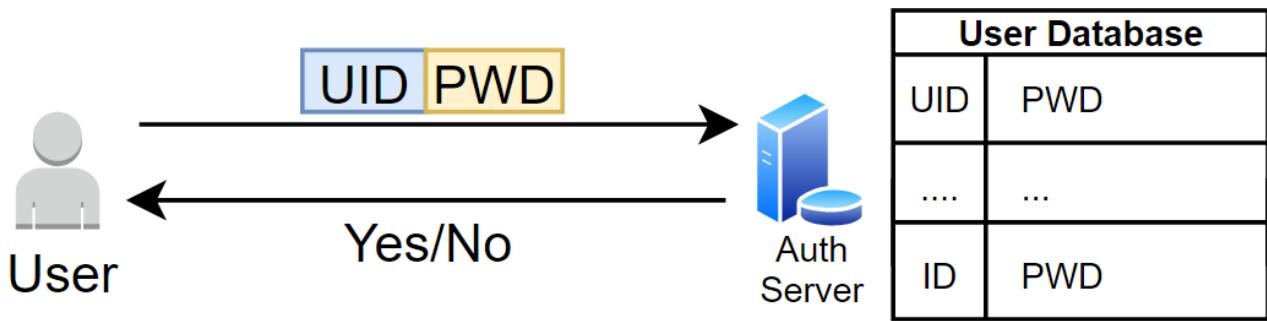


Figura 4.1: PAP scheme

E' facile notare che PAP ha delle limitazioni. In particolare:

- E' soggetto a **eaves-dropping**: la password e l'ID sono trasmessi in clear, se qualcuno ascolta mette a repentaglio la sicurezza.
- Non c'è protezione da Replay Attack (ad un eventuale attacker basta ascoltare il messaggio che viene inviato all'authenticator per acquisire credenziali valide per entrare).
- Il protocollo non specifica nulla sul numero dei tentativi per entrare (più tentativi a disposizione di un attacker gli garantiscono più possibilità di indovinare).

#### 4.2.2 Challenge Handshake Authentication Protocol (CHAP)

Consiste in un approccio più conservativo dal punto di vista della sicurezza, in quanto la password non viene condivisa in chiaro come in PAP (prop 4.2) ma nascosta tramite una funzione hash.

**Osservazione:** E' necessario porre delle premesse sulla funzione hash usata da CHAP: ■

- Deve soddisfare le proprietà dei thms da 3.1 a 3.3, in particolare la non invertibilità, altrimenti intercettando un messaggio basterebbe applicare la funzione inversa e decifrare la password.
- Seguendo le regole degli stream-ciphers,  $H(\cdot)$  **NON** deve essere funzione della sola password  $P$ . Deve **SEMPRE** includere qualcosa di "*fresco*"<sup>1</sup>

##### Proposizione 4.3 (CHAP - Challenge Handshake Authentication Protocol)

1. L'authenticator invia una **Challenge** (una **nonce**) all'utente, con il compito di **NON RIPETERE MAI** la challenge inviata.
2. L'utente risponde con un messaggio contenente lo **User ID** e una **Response**:

$$[UID, H(Challenge, Password, \text{altri oggetti utili}^2)]$$

3. L'authenticator cerca nel database l'entry relativa all'UID. Poiché conosce la challenge, è in grado di calcolare l'hash della password e può controllare il valore della response rispetto a quello da lui calcolato. In caso affermativo garantirà l'accesso.

<sup>1</sup>Spesso vengono aggiunti altri oggetti utili all'autenticazione, poiché non danneggiano la robustezza della funzione Hash. ■

<sup>2</sup>In genere una *nonce*, ovvero qualcosa di unico, che può essere usato solo una volta.

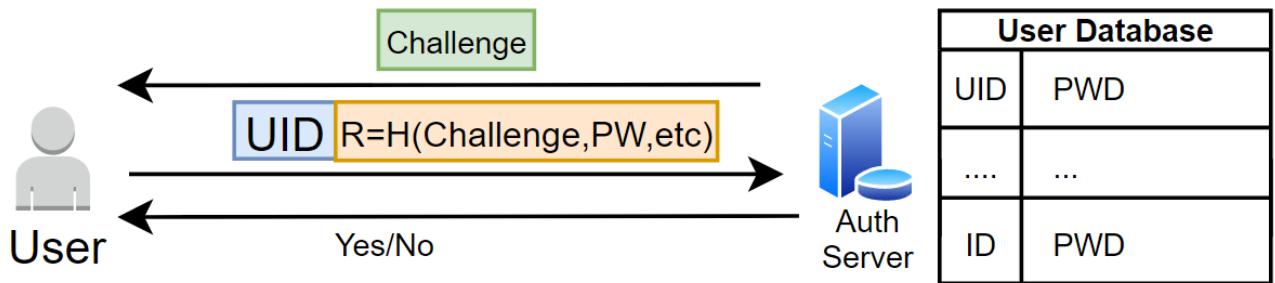


Figura 4.2: CHAP Scheme

**Nota** Poiché abbiamo richiesto che  $H(\cdot)$  non sia invertibile, CHAP sfrutta la conoscenza della challenge per consentire l'accesso. Se nella specifica applicazione vengono inviati ulteriori oggetti, l'authenticator dovrà usare anche quelli per calcolare l'hash.

Questo protocollo presenta diversi pro, ma anche un'importante controindicazione:

- ✓ CHAP protegge dai **replay attacks**, a patto che la challenge non si ripeta mai.
- ✓ Poiché la challenge è decisa dall'ente preposto alla verifica, questo può:
  - Controllare la frequenza e la tempistica con cui avvengono le richieste di autenticazione (utile per servizi di banca, ad esempio).
  - Limitare il tempo di esposizione del sistema ad ogni singolo attacco, ad esempio imponendo un timer tra una richiesta di autenticazione e l'altra.
- ✗ Il segreto deve essere disponibile nel database in formato plaintext, rendendo **impossibile** per un sistema che implementa CHAP salvare in DB delle password crittografate in quanto l'hash deve essere irreversibile e soprattutto la nonce deve essere sempre diversa.

### 4.2.3 PAP vs CHAP - chi scegliere?

Nonostante le applicazioni moderne usino PAP, per rispondere correttamente alla domanda su quale sia il miglior protocollo di autenticazione è importante considerare bene i modelli di attacco a cui possono essere sottoposti.

#### Teorema 4.2 (PAP vs CHAP)

Non esiste un'unica soluzione, dipende da quale attacco ci si vuole difendere.

Gli scenari principali sono due:

1. **Attacco al Canale di Comunicazione** (eaves-dropping): Con PAP, è necessaria proteggere il canale di comunicazione<sup>3</sup> perché chiunque in ascolto sul canale o chiunque in grado di catturare i pacchetti può conoscere *ID* e *PWD*.
2. **Attacco al Database Back-End** (Dictionary-Attack to DB): Tipicamente il bersaglio di un sistema di auth sono i sistemi di back-end delle applicazioni, come i database. Se un hacker è in grado di penetrare all'interno del sistema e rubare il DB di password, queste sono esposte

<sup>3</sup>HTTPS, EAP-TTLS, etc

nel poiché in chiaro. La soluzione è proteggere le password con un hash, rendendo PAP l'unica soluzione possibile.

Riassumendo:

#### Proposizione 4.4

*CHAP, usando una nonce può fare trasmissione in chiaro poiché la password è cifrata.*



#### Proposizione 4.5

*PAP risulta la scelta migliore per difendere i database, perché è possibile nascondere le password con una funzione hash forte e salvare direttamente  $H(P)$  piuttosto che la password stessa. Se il DB viene rubato, l'unico modo per bucarlo è facendo brute-force.*



**Nota** Usando password hashate la sicurezza dipende dalla scelta della password, che deve essere forte.

C'è comunque un modo per proteggere CHAP da attacchi back-end:

#### Proposizione 4.6 (CHAP with Explicit Salt)

*Supponiamo che il database abbia a disposizione una sorta di chiave numerica, detta **Salt**. Nel suo DB, è salvato l'hash del salto con la password precedentemente scambiata:  $H(Salt, Password)$ .*

1. L'authenticator invia una **Challenge** (nonce) e il **Salt** in chiaro all'utente.
2. L'utente risponde con un messaggio contenente lo **User ID** e una **Response**:

$[UID, H(H(Salt, Password), Challenge)]$

3. L'authenticator cerca nel database l'entry relativa all'UID. Calcolando l'hash di password e salt insieme alla challenge corrente viene verificato il risultato. In caso affermativo garantirà l'accesso.

*Periodicamente e/o se dovesse esserci una falla nella sicurezza, si rigenera il DB con un nuovo Salt.*



**Osservazione:** Se un hacker ruba un DB, non può utilizzare la password in quanto è nascosta, ma i dictionary-attack restano comunque validi.



**Nota** Se volessimo usare un database hashato, quale sarebbe la funzione hash da utilizzare? Sappiamo che SHA256 è la migliore al momento, ma presenta 2 problemi dal punto di vista di nascondere delle password:

1. E' estremamente veloce: posso crackare l'intero DB più velocemente.
2. Esiste hardware molto dedicato e specializzato: al 2021 abbiamo 67 TeraHash/s.

*Poiché gli attacchi ai DB sono i più comuni, è importante far fronte agli eventuali e successivi brute-force su di essi.*

## 4.3 One Time Password (OTP)

Abbiamo visto che PAP è, generalmente, la scelta più inflazionata per autenticare un utente. OTP si basa sull'idea di migliorare PAP per avere una password diversa ad ogni corretto tentativo di autenticazione.

 **Nota** Per prevenire replay attacks, per ogni utente viene memorizzata una password che corrisponde ad un anello di una catena di hash di password. In questo modo viene usata una password sempre diversa ma il database non viene riempito di password per uno stesso utente.

### Definizione 4.5 (Hash Chains Password)

Le password vengono generate applicando più volte una funzione hash a partire dalla password di partenza, garantendo che la password sarà sempre diversa da quella usata in un precedente tentativo di autenticazione terminato con successo.



**Osservazione:** Non stiamo risolvendo il problema dell'eavesdropping, in quanto una volta intercettata una delle password, un attaccante sarà in grado di generare anche tutte le successive. Inoltre, bisogna gestire dei meccanismi di perdita di pacchetti, inserendo delle finestre di tolleranza nei protocolli, ma come scegliamo la dimensione della finestra?



### Proposizione 4.7 (OTP - One Time Password)

1. L'authenticator scambia  $P[0]$  con il client e genera  $N + 1$  password a partire da  $P[0]$  calcolando:

$$P[i] = H(P[i - 1])$$

Nel database salverà soltanto la password  $P[N + 1]$

2. L'utente genera  $N$  password a partire da  $P[0]$ , seguendo la stessa logica.
3. L'utente procede all'autenticazione secondo PAP (prop 4.2), inviando le password a partire dall'ultima generata:  $(UID, P[N])$
4. L'authenticator controlla se  $H(P[N]) == P[N + 1]$ . Se vero, sovrascrive nel DB la password con  $P[N]$  e invia un messaggio di avvenuta autenticazione. Altrimenti invia un messaggio di rifiuto.



 **Nota** Durante l'autenticazione  $i$ -esima, la password salvata nel DB è  $P[i]$ , pertanto il controllo sarà:  $H(P[i]) == P[i - 1]$

**Osservazione:** Capiamo che in OTP, se un tentativo di autenticazione va a buon fine, viene salvata sempre la password **PRECEDENTE** a quella attualmente presente nel database.



Riassumendo, vantaggi e svantaggi sono:

- ✓ Ad ogni autenticazione viene eseguita una sola hash.
- ✓ Il DB memorizza un solo valore per utente.
- ✓ Robustezza contro attacchi server-side (è impossibile prevedere pwd precedenti).
- ✓ Le password possono essere trasmesse in chiaro sul canale.

- ✗ Serve un grande valore di  $n$  per evitare di finire le password a disposizione (servirà una nuova registrazione dopo).
- ✗ Vulnerabilità lato client (tiene memorizzato il seed della password).
- ✗ Possibilità di desincronizzazione (se per qualche motivo fallisce un accesso, si perde la catena, in quanto l'utente la volta dopo proverà ad entrare con  $p[n-1]$  in giù, mentre l'authenticator si aspetta ancora  $p[n]$ ).

## 4.4 Two-Factor Authentication

Assumiamo che sia client che server siano entità sicure e non penetrabili e che l'autenticazione avvenga attraverso l'ausilio di una password e di un oggetto aggiuntivo come un *One Time Authorization Token* (generato su un device differente o ricevuto su un canale differente, come e-mail, SMS etc), di lunghezza pari a 6/8 cifre (quindi serve una hash in grado di troncare la lunghezza).

Ci sono due protocolli per un sistema 2FA:

### Proposizione 4.8 (HOTP - Hash based OTP)

**idea:** Basato sull'utilizzo di un contatore  $N$ .

Sul server e sul client è memorizzata una secret-key, la password è calcolata come:

$$P[N] = H[K, N]$$

La password generata non è più una catena in quanto ogni volta il contatore cambia e il bitstream da hashare è sempre diverso.



**Nota** Anche se un attaccante dovesse conoscere  $P[N]$ , non potrebbe calcolare  $P[K, N + 1]$ .



**Nota** La password  $n$ -esima può essere calcolata in modo asincrono, purché sia nota la chiave di partenza.



**Nota** La differenza con il OTP (prop 4.7) è che la password non è più pre-calcolata in ordine inverso.

### Proposizione 4.9 (TOTP - Time based OTP)

Simile a HOTP (prop 4.8) ma il contatore è il tempo.



**Nota** Un possibile attacco potrebbe essere effettuato, sfruttando la possibilità che un attaccante possa manipolare il tempo registrato nella macchina, controllando di fatto il token.

### Corollario 4.2

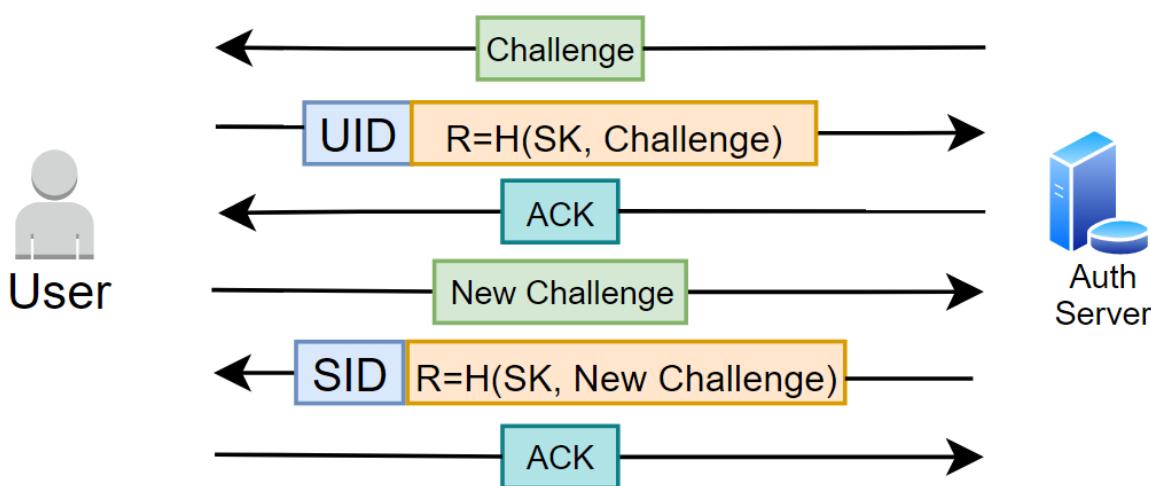
Come "esperti di sicurezza" vorremmo che un sistema 2FA sia sempre implementato e sempre utilizzato, in quanto rappresenta una protezione aggiuntiva difficilmente aggirabile dato che sfrutta un canale di comunicazione diverso.



## 4.5 Mutual Authentication

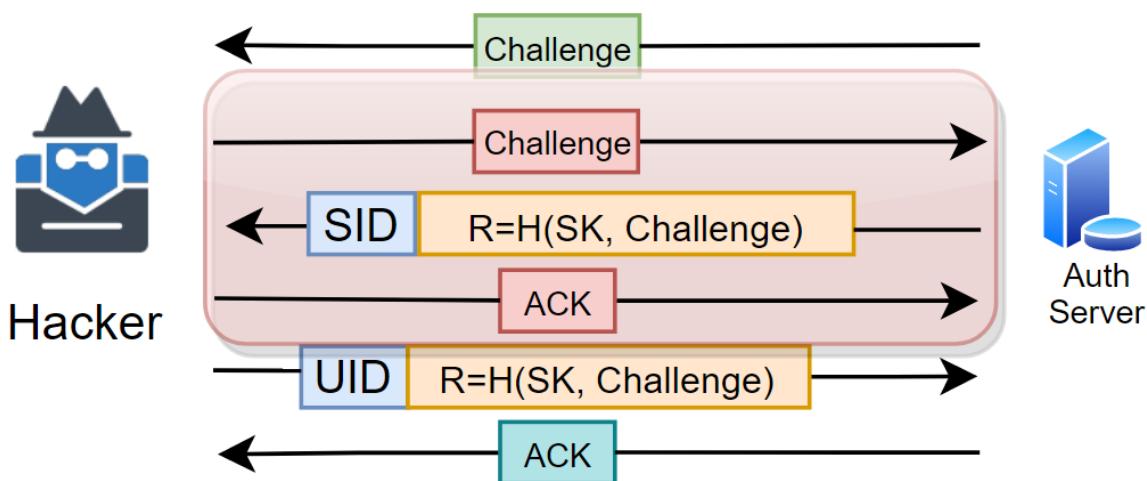
Nei protocolli descritti sopra abbiamo trattato come "entità poco sicura" esclusivamente l'utente. Tuttavia, in un contesto reale è molto utile **verificare la legittimità dell'entità a cui ci connettiamo**. Un possibile modo di fare mutua autenticazione è quello di usare CHAP (prop 4.3). L'idea consiste in:

1. Il server invia la challenge e l'utente risponde con  $(ID, H(shared-key, challenge))$ . Dopo i controlli, il server invia un *ACK* per indicare il successo dell'auth.
2. L'utente invia una nuova challenge e il server risponde con  $(ServerName, H(shared-key, new-challenge))$ . Dopo i controlli, l'utente invia un *ACK* al server per indicare il successo dell'auth.



**Figura 4.3:** Mutual Auth with Chap

**Osservazione:** L'idea di base non pone restrizioni sull'effettiva sequenza di messaggi che devono essere scambiati, questo pone un attaccante in posizione di vantaggio (fig. 4.4). Difatti, l'hacker potrebbe inoltrare al server la stessa challenge e lui dovrebbe rispondere con il suo nome e l'hash della challenge. A questo punto l'hacker si è mostrato legittimo per il server e può autenticare la sua identità digitale inoltrando la response precedente. ■



**Figura 4.4:** Mutual Auth attack scheme

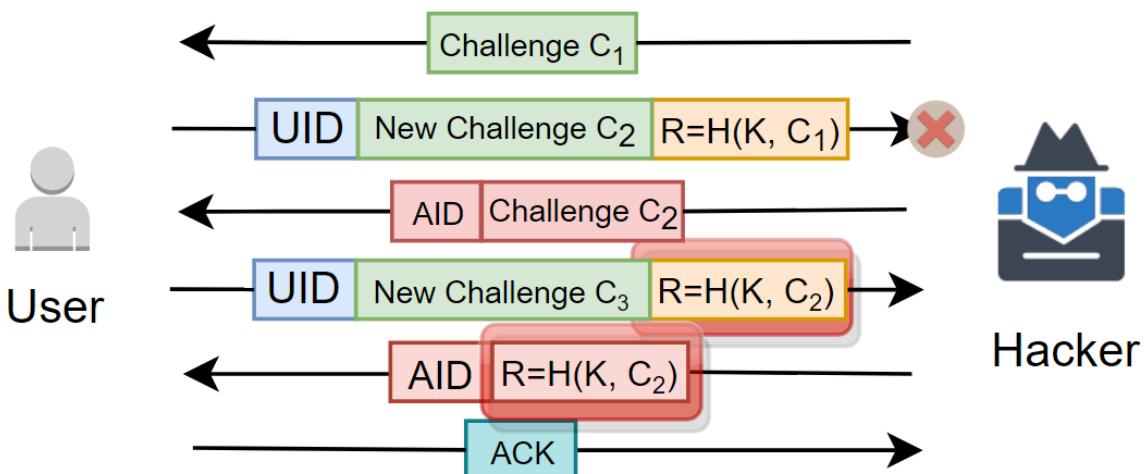
Lo schema precedente non è il solo attacco che diventa disponibile adesso, perché ora il sistema è soggetto al *reflection attack*

#### Definizione 4.6 (Reflection Attack)

L'attaccante è in grado di fingersi come access point. A questo punto può farsi inviare da un utente un messaggio con il quale può verificare la sua identità digitale. Lo schema è il seguente (fig. 4.5):

1. A si finge access point ed invia una challenge  $C_1$  ad U.
2. U invia la sua challenge  $C_2$  (per autenticare l'altro end-point) e la sua response.  
 $(UID, H(K, C_1))$
3. A ignora la response e inoltre ad U la sua stessa challenge:  $(AID, C_2)$ .
4. U risponde di nuovo come al punto 2. con  $C_2$  al posto di  $C_1$
5. A può autenticarsi, rispondendo con  $(AID, H(K, C_2))$

□



**Figura 4.5:** Reflection Attack in Mutual Auth

#### 4.5.1 Come prevenire Reflection Attacks?

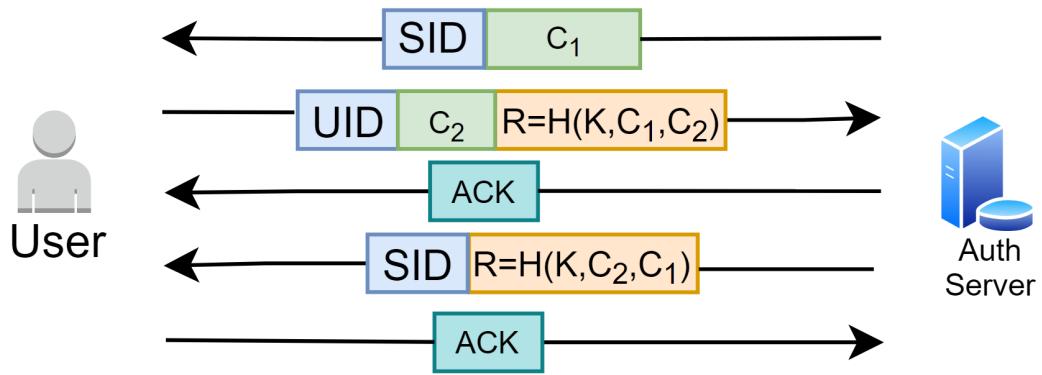
Un modo per impedire che le challenge siano riflesse, è tenere traccia della precedente challenge e dell'ordine con cui vengono inviate e calcolare l'hash di quella coppia come response. In questo modo anche osservando le challenge, un attaccante non potrà autenticarsi in quanto senza shared secret non potrà calcolare delle response valide. Lo schema, riportato in fig. 4.6, è:

1. L'authenticator invia  $(SID, C_1)$ .
2. U genera  $C_2$  ed invia  $(UID, C_2, H(K, C_1))$ . L'authenticator conosce l'ordine e può autenticare U.
3. L'authenticator risponde con  $(SID, H(K, C_1, C_2))$ . U conosce l'ordine e può autenticare il server.

#### Corollario 4.3 (Lo scopo di un protocollo)

*Mai usare un algoritmo per un obiettivo che non è quello per cui è stato pensato*

□



**Figura 4.6:** Avoid Reflection Attack Scheme

La mutua autenticazione con CHAP è sbagliata di fondo, in quanto è un algoritmo pensato per essere "*Single-Side authentication*", mentre noi lo stiamo forzando a lavorare in un contesto "*Dual-Side*". Infatti, i reflection attacks ci sono ogni qualvolta usiamo uno stesso protocollo per autenticare due end-point.



**Nota** *Un protocollo di mutua autenticazione non può autenticare indipendentemente le due parti, in quanto espone vulnerabilità per il sistema. L'unico modo per eseguire il processo correttamente è quello di legare insieme le due challenge crittografandole in una unica response.*

# Capitolo Cellular Authentication

Nelle reti cellulari passate<sup>1</sup>(2G, 3G) gli aspetti di sicurezza vennero considerati in modo crescente ma non senza commettere errori e/o tralasciare aspetti fondamentali. Consideriamo qui aspetti e meccanismi di come un utente può autenticarsi in una rete cellulare e di come una stazione radio può confermare di essere valida.

## 5.1 Rete 2G (GSM)

In una rete GSM, le principali entità coinvolte in un processo di autenticazione sono le seguenti:

- **MS:** Mobile Sim (User). La sim contiene al suo interno una chiave segreta detta  $K_i$ <sup>2</sup>, sconosciuta persino al dispositivo in cui la sim è inserita, così da poter non essere estratta.
- **SN:** Serving Network (Visitor Location Register - VLR).
- **HN:** Home Network. E' l'unica a conoscere le chiavi  $K_i$  degli utenti.

Quando un utente entra in rete, si avvia il seguente protocollo di autenticazione (fig. 5.1):

### Proposizione 5.1 (2G Authentication)

Assumiamo che l'utente abbia già notificato alla SN chi è e dove si trovi.

1. La SN invia l'**IMSI**<sup>3</sup>(identificatore dell'utente) e un **RAND** (sostanzialmente una challenge) alla HN.
2. L'HN risponde con **SRES**<sup>4</sup>(una response di tipo:  $H(RAND, K_i)$ ) tramite un algoritmo A3 (una funzione hash) e un **encryption key**  $K_C$  generata **dinamicamente** con l'algoritmo A8 a partire sempre da  $K_i$  e **RAND**.
3. La SN invia un'**authentication request** al MS inviando una nuova challenge **RAND** di 128bit.
4. La MS crea una **authentication response SRES** da 32bit a partire dalla challenge di 128 e da  $K_i$ , tramite A3.
5. La SN controlla se gli **SRES** corrispondono. In caso affermativo, consente l'accesso.

<sup>3</sup>International Mobile Subscriber Identity

<sup>4</sup>Signed Result

Il punto di forza di questo protocollo di autenticazione è il concetto di **Key Derivation**

### Definizione 5.1 (Key Derivation)

Concetto **fondamentale** con cui, a partire da una singola chiave che resta segreta per quasi<sup>5</sup>ogni entità, è possibile generare delle nuove chiavi sempre diverse.

<sup>5</sup>Generalmente la chiave è codificata nell'hardware della scheda e salvata nel database dell'operatore.

<sup>1</sup>Nella rete 1G la sicurezza non venne considerata affatto.

<sup>2</sup>Identity Key

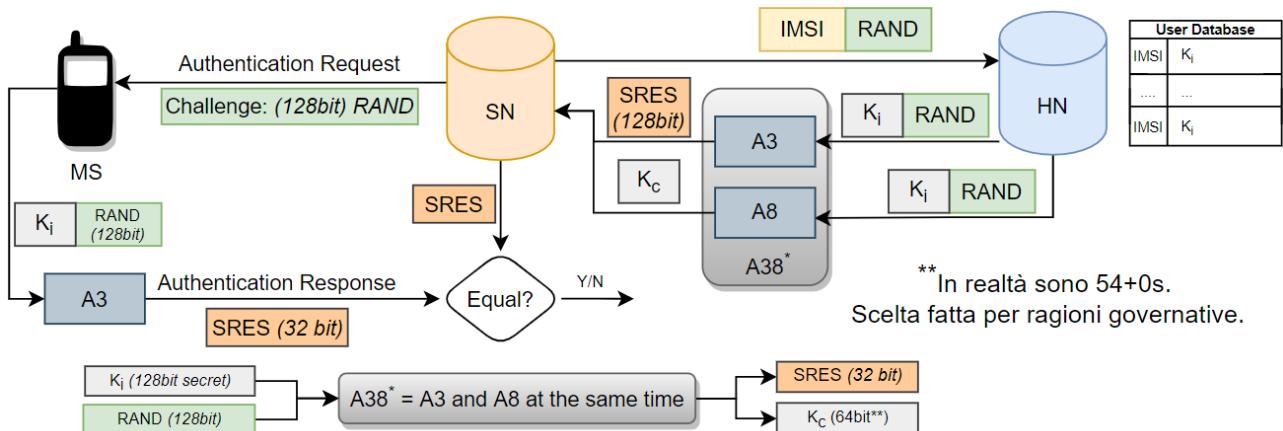


Figura 5.1: 2G Authentication Scheme

**Corollario 5.1 (Authentication Vector)**

*Tripletta di valori generata dalla Home Network per consentire l'autenticazione di un utente nella rete.*

$$\text{AuthVect} = \{\text{RAND}, \text{SRES}, K_c\}$$

Poiché in uno scenario reale un utente può autenticarsi molte volte in un lasso di tempo abbastanza breve, tipicamente il processo di auth viene ottimizzato **inviando alla SN una sequenza di authentication vector**. In questo modo, per  $N$  richieste di auth la SN sarà da subito in grado di autenticare un utente, senza dover avviare lo scambio di messaggi con la HN. Inoltre, viene ridotta anche la frequenza con cui una SN e la HN comunicano tra loro, riducendo l'impatto del processo sul traffico generale.

**5.1.1 Vulnerabilità delle reti GSM**

Nelle reti GSM la vulnerabilità principale è data dal fatto che il protocollo è **unilaterale** verso l'utente:



**Nota** non abbiamo garanzie sull'autenticità della service network/stazione radio base<sup>6</sup>.

**Proposizione 5.2 (Rogue BTS Attack)**

*I sistemi GSM non supportano mutua autenticazione, rendendo possibile la creazione di una fake BTS con la quale generare dizionari di utenti che provano a connettersi e, volendo, autenticarsi al posto loro.*

Una possibile soluzione è quella di spostare la responsabilità di generare la challenge alla home network, così che il fornitore stesso del servizio cellulare possa controllare l'autenticazione. In questo modo vengono inibiti attacchi basati sulla non affidabilità della SN e/o del canale tra SN e HN. Una seconda vulnerabilità è dovuta al fatto che gli algoritmi di cifratura e derivazione (rispettivamente A3 e A8) erano algoritmi proprietari e, teoricamente, ogni operatore avrebbe dovuto implementare il proprio. Questo approccio è definito in gergo come:

<sup>6</sup>In inglese BTS: base transceiver station

### Definizione 5.2 (Security by Obscurity)

*Filosofia di sicurezza per la quale un sistema è protetto semplicemente perché mantiene dei segreti sugli algoritmi che implementa.*



Ai tempi di GSM, gli operatori si misero d'accordo per implementare l'algoritmo *COMP128*, considerato sicuro proprio perché non reso pubblico. Tuttavia, negli anni successivi l'algoritmo trapelò e violato in breve tempo.

**Osservazione:** Mai affidarsi per la sicurezza di algoritmo al fatto che questo possa essere privato perché prima o poi questo verrà bucato. E' fondamentale, invece, affidarsi agli esperti di crittografia e mai implementare soluzioni "fatte in casa".



**Nota** *Riguardo alla dimensione effettiva del digest della chiave  $K_c$ , vennero standardizzati 64bit, suddivisi in 54 bit veramente sotto specifiche e una serie di zeri, in quanto a quei tempi nessuno era in grado di realizzare un attacco brute-force su una chiave di 54 bit (tantomeno su 64) se non i governi stessi, dotati di mezzi di calcolo più performanti.*

## 5.2 Rete 3G (UMTS)

Nella rete UMTS l'autenticazione è bilaterale (*Mutual Auth sezione 4.5*) e garantisce che i parametri usati nel processo siano sempre nuovi, evitando il riutilizzo di chiavi. Lo schema è il seguente:

### Proposizione 5.3 (3G Authentication - AKA: Authentication and Key Agreement)

Assumiamo che l'utente abbia già notificato alla SN chi è e dove si trovi.

1. La SN invia l'IMSI del MS all'HN.
2. L'HN invia  $N$  authentication-vectors (cor 5.1) fatti da quintuple, contenenti:

$$\{RAND, XRES, CK, IK, AUTN\}$$

- **RAND**: Authentication Challenge.
  - **XRES** =  $f_2(K, RAND)$ : Risultato atteso dalla CHAP-like authentication.
  - **CK** =  $f_3(K, RAND)$ : Cipher Key.
  - **IK** =  $f_4(K, RAND)$ : Integrity Key.
  - **AK** =  $f_5(K, RAND)$ : Authentication Key. Usato per mutual-authentication.
3. La SN inoltra a MS un **RAND** ed un **Autentication Number** definito come:

$$AUTN = \{SQN \oplus AK, AMF, MAC-A\}$$

- **SQN**: Sequence Number (48bit).
  - **AMF**: Auth&Key Management Field (16bit). Specifica quale algoritmo o chiave da usare se ci sono scelte disponibili, finestra di sync e altri parametri. Utile per il cosiddetto **In-Band Signaling**.
  - **MAC-A** =  $f_1(K, SQN, AMF, RAND)$ : Message Auth Code permette all'MS di autenticare la rete (64bit).
4. Per autenticare la rete, MS esegue i seguenti passi:
    - (a). genera la SUA **AK** a partire dall'AUTN ricevuto.
    - (b). verifica la "freschezza" dell'informazione calcolando  $SQN \oplus AK = SQN'$  tale che:

$$SQN' \in \{SQN-MS+1, SQN-MS+tollerance\}$$

Dove **SQN-MS** è il contatore interno del MS mentre la tolleranza è utile a fini di resync in caso di messaggi persi.

- (c). Se MS ha ricevuto un nonce valido, procede a rigenerare il **MAC-A**, verificando che sia uguale a quello ricevuto in **AUTN**. Se l'esito è positivo, invia la **RES** alla SN e aggiorna il valore del proprio sequence number come  $SQN-MS = SQN$ .
5. La SN confronta **RES** ed **XRES**, autenticando MS in caso affermativo.



**Nota** Gli algoritmi implementati all'interno della SIM sono  $f_1, f_2, f_3, f_4, f_5$ . Questi algoritmi sono pubblici e servono per derivare le chiavi necessarie all'autenticazione, tranne il primo che costituisce una funzione di criptaggio<sup>7</sup> e il secondo che è sostanzialmente una funzione hash.

<sup>7</sup>  $f_1$  è detta: *Actual Proof of Knowledge* e certifica la conoscenza della chiave.

**Osservazione:** E' interessante osservare come il protocollo di **mutua autenticazione** usato per **validare la rete** è sostanzialmente basato su **un singolo messaggio**, quando ci aspetteremmo un protocollo di tipo *3-way handshake* (3 messaggi). Analogamente, il meccanismo di **autenticazione del MS rispetto alla SN** avviene come nella rete **2G** (prop 5.1). ■

**Nota** E' bene specificare che l'autenticazione della HN nei confronti della MS avviene tramite **sequence number (SQN)**, mantenuto in sincronizzazione tra le diverse parti della rete<sup>8</sup> come **nonce implicita**. Questo evita l'invio di un'ulteriore challenge da parte del MS alla HN.

#### Corollario 5.2 (Il ruolo dell'Anonymity Key)

L'AK viene utilizzata per **oscurare/mascherare**<sup>9</sup> il sequence number in una fase in cui il **servizio di confidenzialità** non è ancora attivo. Ciò permette di risolvere il **Location Privacy Problem**, per il quale un attaccante può essere in grado di **determinare gli spostamenti di un utente tracciando incrementi del sequence number nella rete**.

<sup>9</sup>Tramite  $\oplus$ . □

### 5.2.1 (Pochi) Dettagli su IMSI-Catcher Attack

Nel processo di auth (sia 3G che 2G) abbiamo supposto che l'utente si autentifichi una singola volta e, pertanto, che l'IMSI venga inviato **una sola volta nella rete**. In condizioni normali è vero che viene inviato solo una volta l'IMSI dell'utente. Ma in generale, se ci dovessero essere errori di connessione o di autenticazione, la rete non può escludere l'utente e deve esserci un modo di recuperare la connessione. In quel caso, l'IMSI può essere inviato nuovamente, e potrebbe essere individuato. Per evitare che l'IMSI venga catturato, forzandone un invio ripetuto magari tramite una rogue-BSS che disconnette l'MS, il protocollo impone che dopo la prima connessione venga usato sempre un IMSI *temporaneo*: **TMSI**. Tale valore varia ogni volta, rendendo di fatto impossibile tracciare l'utente.

### 5.2.2 (Pochi) Dettagli su Rete 4G/5G (LTE)

La rete 4G (LTE) sfrutta lo stesso principio di autenticazione della rete 3G (UMTS), mentre la rete 5G sfrutta meccanismi molto più simili alle reti wi-fi, in quanto si pone l'obiettivo di fondere le tecnologie ed unificare le reti.

**Nota** [Riguardo 5G]

*Il meccanismo di key-derivation è migliorato. I moduli che dovrebbero garantire integrità in realtà non sono sempre implementati in quanto aggiungono molto carico di lavoro ed è stato fatto solo dal 4G in poi. Vi è inoltre una netta separazione per garantire protezione, detta **Security Edge Protection** per separare la Serving Network dalla Home Network*

<sup>8</sup>La Home Network mantiene il suo contatore nella variabile **SQN-HE**.

# Capitolo RADIUS

In contesti reali dove le reti hanno una dimensione molto elevata, usare un sistema di autenticazione CHAP-like come visto fino ad ora è impensabile. Questo perché esistono diversi modi di connettersi alla rete e quando i numeri sono grandi possiamo riscontrare tre problemi principali:

- Se mantengo DB distribuiti ci sono tanti punti che possono essere attaccati.
- Quando viene cambiato il valore di una variabile (es. una password) bisogna propagare l'informazione a tutti i DB.
- C'è necessità di avere *qualcuno* dotato di permessi di gestione (admin).

Per questo è stato sviluppato RADIUS, che sfrutta dei **server (logicamente) centralizzati**, mantenendo **1 server primario** più altri server secondari replicati.

## 6.1 Funzionamento e Autenticazione

### Definizione 6.1 (RADIUS - Remote Authentication Dial In User Service)

Protocollo di tipo AAA UDP-based<sup>1</sup>, che fornisce:

- **(Remote) Authentication:** certifica l'identità digitale di un utente, anche se questo non può connettersi alla stessa rete del server autenticante.
- **Authorization:** gestisce i permessi di accesso ai servizi degli utenti autenticati.<sup>2</sup>
- **Accounting:** tiene traccia delle azioni eseguite dagli utenti<sup>3</sup>

<sup>1</sup>Triple A

<sup>2</sup>ES: un set di canali di una pay-per-view che rientrano nell'abbonamento dell'utente.

<sup>3</sup>ES: quando tempo passano su un servizio X o la spesa fatta per un bene Y.

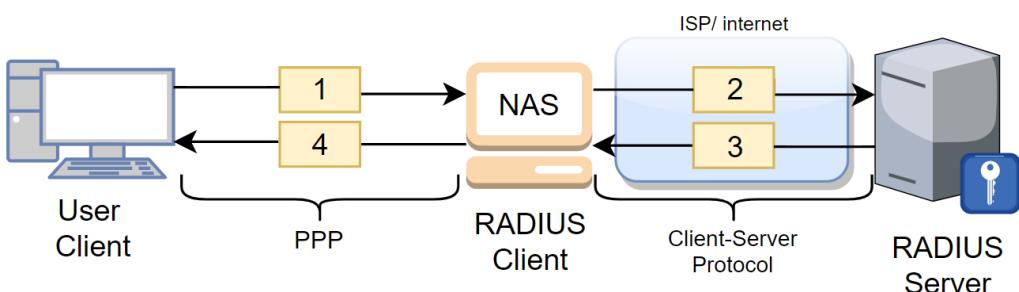


Figura 6.1: Main RADIUS components

Le principali componenti del protocollo sono le seguenti:

### Definizione 6.2 (RADIUS Entities)

- **PPP (Point to Point Protocol):** Utilizzato per l'autenticazione tra il sistema dell'utente ed il **NAS/RAD client**. Generalmente si usano PAP o CHAP.
- **RADIUS Client/NAS (Network Access Server):** punto di accesso che permette agli utenti di autenticarsi nella rete per sfruttarne i servizi.
- **RADIUS Server:** mantengono le credenziali degli utenti in DB centralizzati ed erogano i servizi offerti dalla rete. I database sono di tre tipi:
  - **Registered User Database:** per ogni utente si mantengono le credenziali e le modalità di autenticazione e gli attributi relativi al servizio di autorizzazione.
  - **Client Database:** Mantiene le informazioni relative ai NAS ammessi come radius client.
  - **Accounting Database:** Per ogni utente vengono mantenute informazioni relative al servizio di accounting.

Il processo di autenticazione avviene (nei minimi dettagli) nel seguente modo:

### Proposizione 6.1 (RADIUS Authentication fig. 6.1)

1. **PPP Request:** l'utente invia le proprie credenziali al **RAD client**.
2. **RADIUS Request:** il NAS converte/include le credenziali ricevute in un **Messaggio di Richiesta**, che invia al **RAD Server**
3. **RADIUS Response:** il **RAD Server** elabora la richiesta e produce un **Messaggio di Risposta**, che invia al **RAD client** che l'aveva contattato.
4. **PPP Response:** il NAS notifica all'utente l'esito dell'autenticazione ricevuto dal server.

## 6.1.1 Servizio di Proxy

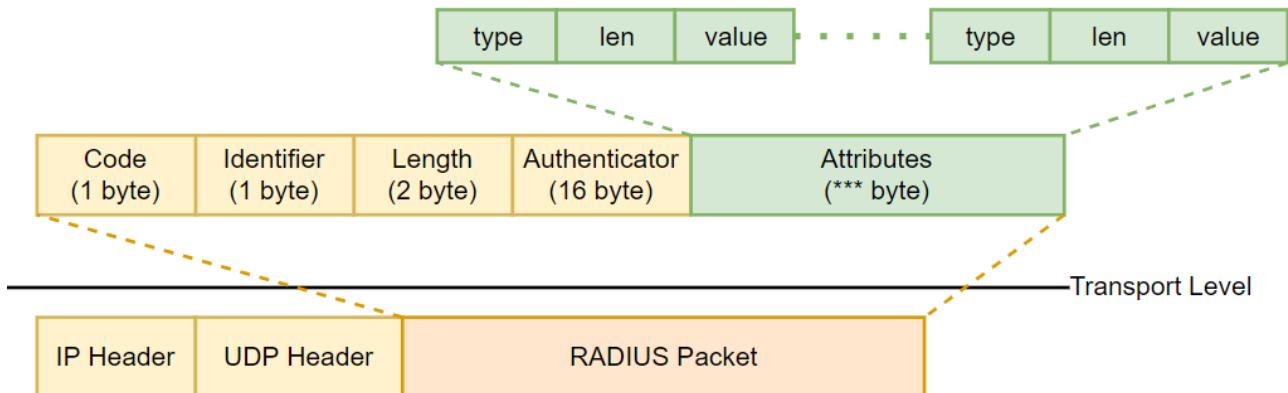
Il protocollo RADIUS supporta connessioni remote, anche quando non possiamo connetterci direttamente alla rete desiderata. Difatti i RADIUS server svolgono la funzione di **Proxy Server** per altri RADIUS server **remoti**.

### Teorema 6.1 (Servizio di Proxy)

Eseguono i servizi per conto di altri RADIUS server (es. facendo roaming tra diversi ISP). Le funzionalità di proxy possono essere svolte sia in maniera trasparente che non, specificandolo nel contenuto del messaggio.

## 6.2 Pacchetti RADIUS

Come ogni protocollo, RADIUS impone una precisa specifica per il formato dei suoi messaggi:



**Figura 6.2:** RADIUS Packet Structure

- **Code:** indica il **tipo** di Pacchetto RADIUS. I principali sono:

### Definizione 6.3 (Main Code Values)

- **Access-Request (1):** richiesta di autenticazione.
- **Access-Accept (2):** autenticazione avvenuta con successo.
- **Access-Reject (3):** autenticazione fallita/ rifiutata.
- **Accounting-Request (4):** richiesta di memorizzazione informazioni.
- **Accounting-Response (5):** risposta alla memorizzazione di informazioni.
- **Access-Challenge (11):** l'autenticazione richiede ulteriori messaggi.

- **ID:** specifica l'identificatore per l'autenticazione e serve a **legare** richiesta e risposta.
- **Length:** specifica la lunghezza del pacchetto, che è **sempre** compresa tra 20 e 4096 bytes.
- **Authenticator:** dipende dal tipo di pacchetto. Può essere:
  - **Request:** **nonce** generata casualmente per il protocollo di autenticazione.
  - **Response:** **digest MD5** per l'autenticazione del messaggio di risposta.
- **Attributes:** lista di **campi** in formato **AVP** (Attribute Value Pair). Ogni campo è costituito da:
  - **Type:** (1 byte) indica il tipo/significato del campo.
  - **len:** lunghezza in byte del campo.
  - **Value:** payload del campo.



**Nota** Il campo *attributes* è un *extensible information field*.

### Definizione 6.4 (Extensible Information Field)

Un campo è di tipo *extensible information* se il numero, il tipo e l'ordine delle informazioni contenute al suo interno può variare in base alle esigenze.

**Osservazione:** Per costruzione non possiamo avere più di  $2^8 = 256$  attributi differenti univocamente identificati a causa della dimensione del sottocampo **type**. In genere, un pacchetto di **Access-Request** contiene sempre i seguenti attributi:

- **User-Name:** identificatore dell'utente.
- **User-Password:** password dell'utente, oscurata nel protocollo PPP-PAP.
- **Chap-Password:** challenge del protocollo PPP-CHAP, se utilizzato.
- **NAS-Identifier:** identificatore univoco del RADIUS client.
- **NAS-IP-Address:** indirizzo IP del RADIUS client.
- **NAS-Port:** porta UDP del RADIUS client.



## 6.3 Autenticazione in dettaglio

Consideriamo il processo di autenticazione messo in atto dal RAD client e dal RAD server:

### Proposizione 6.2 (Authentication Details)

1. Il RAD client genera un **ID** per identificare la richiesta e una **nonce ReqAuth** per l'autenticazione. Mantiene la coppia  $\{ID, ReqAuth\}$  generata in un database locale ed invia una **RADIUS-Request**, contenente anche le credenziali di accesso ricevute tramite protocollo PPP dall'utente.
2. Il RAD server genera una **RADIUS-Response**, composta da un digest prodotto tramite **MD5** secondo il seguente schema:

$$\text{digest} = \text{MD5}(\text{Code}||\text{ID}||\text{Length}||\text{Auth}||\text{Attributes}||\text{S})$$

- **ID:** uguale per request e response.
  - **Code, Length, Attributes:** specifiche del messaggio di risposta.
  - **Auth:** nonce contenuta nel messaggio di richiesta e generata dal RAD Client.
  - **S:** secret key condivisa tra RAD-client e RAD-server.
3. Il RAD-Client rigenera il digest utilizzando il suo DB per recuperare l'ReqAuth relativo all'ID ricevuto e verifica che sia uguale a quello ricevuto nella **RADIUS-Response**.
    - In caso negativo, il RAD-Client comunica all'utente tramite protocollo PPP il fallimento dell'autenticazione.
    - In caso positivo, possiamo avere tre outcome diversi:
      - (a). **response = Access-Accept:** viene comunicato all'utente tramite protocollo PPP l'avvenuta autenticazione e tutti i parametri di configurazione necessari.
      - (b). **response = Access-Reject:** le credenziali di accesso e/o degli attributi nella richiesta non sono validi. Viene comunicato all'utente il fallimento tramite protocollo PPP.
      - (c). **response = Access-Challenge:** Il RAD-Server ha bisogno di ulteriori informazioni per poter autenticare l'utente. Il NAS si occupa perciò di recuperare tali informazioni dall'utente ed invia una nuova **Access-Request**.



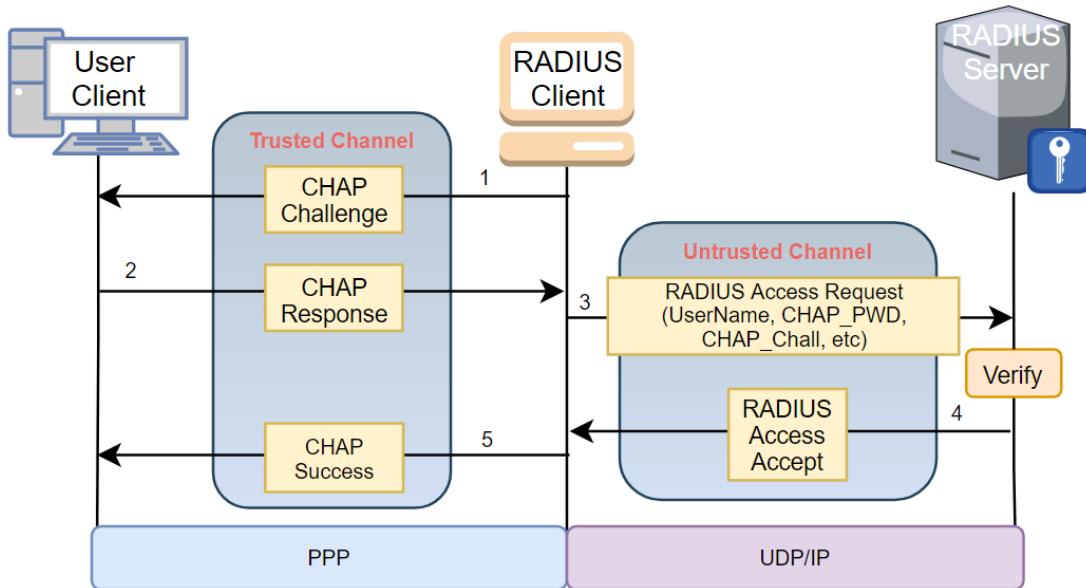


Figura 6.3: CHAP authentication in RADIUS

### 6.3.1 Challenge in PPP-CHAP

Nel caso in cui PPP sia di tipo CHAP, la challenge per l'autenticazione CHAP tra utente e NAS viene gestita nel seguente modo:

#### Definizione 6.5 (PPP-CHAP)

- Il NAS genera la challenge per l'utente che vuole essere autenticato.
- L'utente genera ed invia ed invia la CHAP-response<sup>4</sup> al RAD-client che la inoltrerà al RAD-Server insieme alla challenge.
- Il RAD-Server verifica la validità della coppia {CHAP-challenge, CHAP-Response} e di altri parametri tra cui l'ID, rigenerando la response tramite le credenziali in suo possesso e la challenge appena ricevuta.

<sup>4</sup>CHAP-Response = H(user-PWD, CHAP-Challenge)

## 6.4 Servizi di Sicurezza

I servizi di sicurezza offerti dal protocollo sono:

#### Teorema 6.2 (Servizio di Sicurezza)

- Autenticazione delle RAD Response:** i messaggi di risposta sono autenticati con un meccanismo **CHAP-Like**<sup>5</sup>, basato sulla conoscenza di un segreto condiviso e sull'uso di una crypto-hash function.
- Confidenzialità sulle PWD degli Utenti:** le password degli utenti sono criptate prima dell'autenticazione.

<sup>5</sup>Viene evitato l'invio in chiaro delle password. In altre parole non c'è trasmissione di segreti.

### 6.4.1 Vulnerabilità nella Sicurezza

I servizi di sicurezza offerti da RADIUS non sono perfetti, né lato autenticazione né su lato confidenzialità. Vediamo perché:

#### Corollario 6.1 (Errori di Autenticazione)

- Soltanto la **RADIUS Response** viene autenticata, lasciando spazio ad attacchi.
- Uso **forzato** della funzione hash **MD5** (non HMAC def 3.4)
- uso di chiavi simmetriche a bassa entropia.

#### Corollario 6.2 (Errori di Confidenzialità)

- Meccanismi **personalizzati** per la criptazione delle password.
- **Riutilizzo della chiave di auth per fare encryption.**
- Trasmissione in chiaro dei messaggi protocollari.

Osserviamo preliminarmente che la trasmissione in chiaro dei messaggi di request e response lascia spazio all'intercettazione di tali messaggi e, soprattutto, ad un eventuale modifica nel caso della request in quanto non autenticata.

Vediamo nel dettaglio cosa implicano queste debolezze:

#### Proposizione 6.3 (RADIUS Request Non Autenticata)

- **Vulnerabilità a MITM:** data la mancanza di cifratura sul canale, i messaggi di request/response possono essere creati.
- **Payload Replay Attack:** Attacco su due fronti che permette di autenticarsi con l'identità dell'utente vittima.
  1. Supponiamo di posizionarci tra il RAD-client e il RAD-server e di poter osservare un'autenticazione valida dell'utente, salvando **CHAP-Challenge** e **CHAP-Response**.
  2. L'attaccante avvia una nuova autenticazione a nome della vittima, utilizzando una password falsa. Poiché il NAS non fa controllo ed inoltra la request verso il RAD-server, l'attaccante può intercettarla e sostituendo **CHAP-challenge** e **CHAP-response** precedentemente memorizzate (**coppia valida**) viene visto come utente legittimo.

#### Teorema 6.3 (Soluzione con Message Authentication Extension Protocol)

Più che un protocollo vero e proprio è un container che specifica come un pacchetto di auth dovrebbe essere fatto. Tecnica utile per rendere il protocollo resistente ad attacchi MITM. Il funzionamento prevede l'**aggiunta** di un **nuovo attributo** nella Access-Request per il controllo dell'integrità.

Si hanno così due campi di autenticazione:

- Uno nell'**header RADIUS**, costituito dalla nonce a 16byte per la richiesta di autenticazione.
- Uno negli **attributi**: MAC per il controllo di integrità della request calcolato come

$$\text{MAC} = \text{HMAC-MD5}(\text{Code} \parallel \text{ID} \parallel \text{Length} \parallel \text{Auth} \parallel \text{Attributes} \parallel \text{S})$$

Dove il valore dell'attributo **Authenticator** è zero in quanto lo si sta calcolando e verrà poi rimpiazzato

### 6.4.2 Confidenzialità della password in PPP-PAP

Nel caso in cui il protocollo PPP sia di tipo PAP, la password dell'utente nella RAD-Request viene criptata utilizzando una nonce data dall'**ReqAuth** e la chiave segreta **S**, così da non rivelarla ad eventuali attaccanti in ascolto. Il processo di cifratura delle password è il seguente:

#### Definizione 6.6 (PWD Encryption in RADIUS)

1. Data la password si effettua un padding fino a 16 byte, questa è la dimensione del blocco.
2. Si calcola la nonce di ReqAuth e si esegue  $\text{MD5}(S \parallel \text{ReqAuth})$ .
3. La password prima di essere spedita viene cifrata calcolando

$$\text{PWD} \oplus \text{MD5}(S \parallel \text{ReqAuth})$$

Se la password è più lunga di 16 byte, viene suddivisa in blocchi da 16 e l'ultimo blocco riempito con un padding. In quel caso, eseguiamo:

*for all  $\text{PWD}_{blk}$  do*

$$\text{PWD}_{enc} = \text{PWD}_{blk} \oplus \text{MD5}(S \parallel \text{ReqAuth})$$

*end for*



### Proposizione 6.4 (Bassa entropia delle chiavi)

La bassa entropia è dovuta al basso numero di caratteri coinvolti nelle chiavi (viene usato il set ASCII). Inoltre, in molti casi, **una sola password è utilizzata per tutta la rete dell'ISP**. Gli attacchi possibili sono due:

- **Dictionary Attack alla chiave segreta S in PPP-PAP:** E' possibile creare un dizionario di coppie  $\{\text{ReqAuth}, \text{MD5}(S, \text{ReqAuth})\}$  ascoltando i messaggi di Access-Request. Poiché MD5 posiziona il segreto alla fine (sezione 3.2.2), è possibile precalcolare come blocco singolo la prima parte del messaggio e fare bruteforce diventa facile.

**Osservazione:** In realtà non c'è neanche bisogno della coppia: Inviando una coppia  $\{\text{user-ID}, \text{user-PWD}\}$  arbitraria, il NAS può agire come fosse un oracolo in un attacco CPA. Infatti intercettando l'Access-Request inviata, possiamo fare ottenere un digest MD5 valido calcolando:

$$\text{MD5}(S, \text{ReqAuth}) = \text{user-PWD attribute} \oplus \text{password}$$

E adesso possiamo cifrare una password qualsiasi. ■

- **Attacco alle pwd degli utenti in PPP-PAP:** L'attaccante invia un user ID questa volta valido e una password arbitraria da 16 byte. Intercettando i messaggi di Access-Request possiamo produrre un digest MD5 valido calcolando:

$$\text{MD5}(S, \text{ReqAuth}) = \text{user-PWD attribute} \oplus \text{password}$$

Adesso l'attaccante può cifrare una password qualsiasi. □



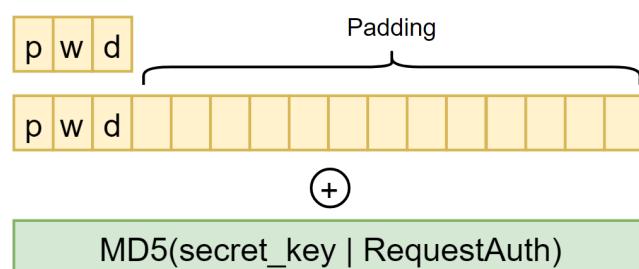
#### Nota [Dictionary Attack to Shared Secret Details]

L'attacco bruteforce è possibile solo con pwd inferiori a 16byte. In particolare, il NAS stesso può essere usato come un oracle in un attacco CPA. Intercettando i messaggi di Access-Request otteniamo la user-password attribute (16 bytes) e il Request Authenticator (16 bytes). Facendo uno xor tra la password cifrata e quella in chiaro otteniamo il digest dell'MD5.



#### Nota [User-PWD Attack Details]

Anche questo attacco funziona con password di 16 bytes. Una volta ottenuto il digest valido possiamo bypassare tutte le limitazioni imposte lato client, come il numero di tentativi valido di autenticazione al RADIUS server.



**Figura 6.4:** Password Encryption in RADIUS with PPP-PAP

### 6.4.3 Cattiva gestione dei PRNG

Un’ulteriore errore commesso da RADIUS è una scarsa implementazione degli algoritmi di PRNG per la generazione delle nonce di ReqAuth, tipicamente non crypto-functions con **ciclo breve** (alta probabilità di collisione) e **alta predicitività**. Questo significa che l’ipotesi di unicità della nonce viene meno e può permettere ad un attaccante di autenticarsi calcolando il valore della prossima nonce.

Ci sono 3 attacchi possibili che possono essere fatti:

- **Birthday Attack:** Consideriamo un PRNG con ciclo di  $2^n$  e al più  $2^n$  valori differenti. Per il **paradosso del compleanno** eq. (3.1) un valore si ripete con **probabilità del 50%** con  $2^{\frac{n}{2}}$  autenticazioni. Inoltre, alcune implementazioni potrebbero **usare più volte uno stesso PRNG** per generare una sola nonce, riducendo il ciclo di generazione.
- **Replay Attack:** L’attaccante può generare un dizionario **{ID, AUTH, MD5}** osservando coppie valide di **Access-Request/Access-Response** per effettuare un replay attack nel momento in cui il NAS genera una ReqAuth ripetuto. A quel punto è possibile autenticare/autorizzare un qualsiasi utente senza un pwd valida.  
L’ID può essere ricalcolato con brute-force in quanto costituito da un solo byte.
- **Attacco alle PWD degli utenti (se PPP-PAP):** Osservando le **Access-Request** l’attaccante genera un dizionario di **{ReqAuth, PW-enc}**. Se una nonce si ripete, è possibile ottenere informazioni sulle password degli utenti calcolando:

$$PW1_{enc} \oplus PW2_{enc} = (PW1 \oplus MD5(S, ReqAuth)) \oplus (PW2 \oplus MD5(S, ReqAuth)) = PW1 \oplus PW2$$

Una volta ottenuto lo xor delle due password in chiaro di due utenti bersaglio, poiché le dimensioni sono note e con buona probabilità di lunghezza differente, otteniamo in chiaro la parte finale della password più lunga e analizzando il risultato è possibile calcolare il risultato. Inoltre, è possibile velocizzare la generazione del dizionario effettuando delle Access-Request con password casuali (magari prese da un db online), consumando generazioni del PRNG.

## 6.5 DIAMETER

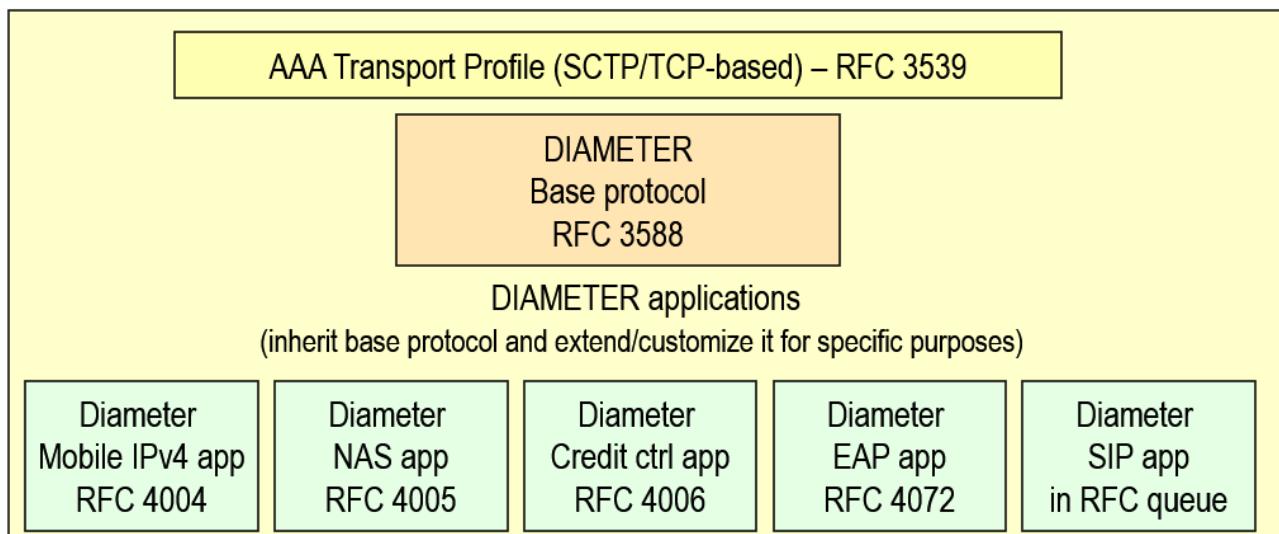
RADIUS al giorno d'oggi è meno utilizzato di una volta in quanto presenta delle limitazioni importanti. In particolare:

- **Scalabilità/Efficienza:** RADIUS supporta in maniera efficiente solo un numero ridotto di utenti. Questo è dovuto all'uso di UDP come protocollo di trasferimento, che implica un forte problema di packet loss nei casi di picchi di utenza.
- **Estensibilità:** il modo con cui gli attributi erano codificati imponeva limiti sulle nuove tecnologie e non era sufficiente a supportare possibili sviluppi futuri.
- **Interoperabilità:** non standardizzata e realizzata tramite soluzioni proprietarie (ogni venditore ha la sua soluzione).

DIAMETER è un'evoluzione di RADIUS, retrocompatibile e che risolve i le limitazioni analizzate prima. L'ottica con cui è sviluppato è simile alla logica "object-oriented": Viene definita una classe di **AAA Transport Profile** che specifica il tipo di livello di trasporto da usare (STCP/TCP-based) e una **base-class** DIAMETER più una serie di classi derivate per svariati tipi di servizi, tutte figlie della base-class.



**Nota** La DIAMETER Base Protocol è, più che un protocollo AAA, un protocollo di scambio di messaggi.



**Figura 6.5:** DIAMETER Structure

### 6.5.1 Feature Principali in DIAMETER

#### Proposizione 6.5 (Trasporto Affidabile dei pacchetti)

Utilizzo di STCP/TCP per il trasferimento affidabile dei pacchetti, instaurando connessioni persistenti tra client e server.



**Nota** L'affidabilità è fondamentale per avere un accounting efficiente.

**Proposizione 6.6 (Gestione degli Errori/Guasti standardizzata)**

*L'error-control è demandato al livello applicativo. Inoltre, vengono implementate le seguenti funzionalità:*

- **Rilevazione Duplicati:** vengono rilevati e scartati messaggi doppi.
- **Watchdog:** periodicamente vengono inviati dei pacchetti per conoscere lo stato dei vicini.



**Nota** In RADIUS la gestione degli errori dipende dall'implementazione

**Proposizione 6.7 (Miglior Supporto all'Estensibilità)**

*il formato dei pacchetti è **modificato** e **ampliato**, per supportare più funzionalità. Tra i servizi implementati troviamo:*

- **Rilevazione Duplicati:** per ogni pacchetto vi è un ID.
- **Negoziazione:** è previsto un flag per specificare il comportamento che il destinatario deve attuare alla ricezione di parametri non supportati.

**Osservazione:** RADIUS non supporta la rilevazione di duplicati né capacità di negoziazione dei parametri. Questo significa che il sistema non sa come rispondere ad un pacchetto con parametro non supportato.

**Proposizione 6.8 (Scoperta dei peers, configurazione e rilevamento capacità)**

*I sistemi DIAMETER vicini (peers) vengono **configurati** tramite **meccanismi automatici** accordandosi sulla versione del protocollo, sui meccanismi di sicurezza da usare e le applicazioni supportate.*

**Proposizione 6.9 (Supporto di Messaggi "Spontanei" da Server a Client)**

*Il server può inviare autonomamente dei messaggi al client al fine di supportare:*

- **Disconnessioni volontarie** da parte del server.
- **Comunicazione di Errori/Fallimenti** al client.
- **Supportare la ri-autenticazione/ri-autorizzazione**,



**Nota** RADIUS supporta solo lo schema client-server, dove solo il primo avvia la comunicazione.

**Proposizione 6.10 (Gestione delle Entità Intermedie)**

*Viene specificata esplicitamente l'esistenza di diversi tipi di agenti, come **proxies**, **redirects**, **relay**. Inoltre, viene fatta distinzione tra:*

- **Hop-By-Hop:** comunicazioni tra nodi intermedi.
- **End-To-End:** comunicazione tra nodi estremi, ovvero tra DIAMETER client e DIAMETER server.



**Nota** In RADIUS esiste solo hop-by-hop e non vengono specificate entità intermedie.

## 6.5.2 Pacchetti DIAMETER

I pacchetti DIAMETER possiedono il seguente formato:

- **Header** [20 byte]: intestazione del messaggio

#### Definizione 6.7 (Header Structure)

- **Version[1 byte]**: versione del protocollo.
- **Message Length[3 byte]**: lunghezza del messaggio.
- **Flags[1 byte]**: flag per specificare la funzionalità
  - **P (Request)**: indica se il messaggio è una **richiesta**(1) o una **risposta** (0).
  - **P (proxyable)**: Indica se è possibile effettuare **proxying** del messaggio.
  - **E (Error)**: indica se è un messaggio di **errore**.
  - **T**: indica se il messaggio è stato ritrasmesso.
- **Command Code[3 byte]**: indica il **tipo** di messaggio.
- **Application ID[4byte]**: indica l'applicazione di appartenenza.
- **Hop-By-Hop ID[4 byte]**: identifica il canale tra due nodi intermedi della comunicazione.
- **End-To-End ID[4 byte]**: identifica gli estremi della comunicazione.

Version Number	Message Length (3B)
R P E T res-flags	Command-Code (3B)
	Application-ID (4B)
	Hop-By-Hop Identifier (4B)
	End-To-End Identifier (4B)

**Figura 6.6:** DIAMETER Header

- Attributes: Lista dei campi in formato **AVP**<sup>6</sup>.

#### Definizione 6.8 (AVP format)

- **AVP Code[4 byte]**: indica il tipo di campo.
- **Flags[1 byte]**: flag per indicare funzionalità
  - **V (Vendor)**: indica se è presente il Vendor ID nell'AVP.
  - **M (Mandatory)**: indica se l'AVP deve essere **obbligatoriamente supportato**.
  - **P (Protected)**: Indica se è necessaria la criptazione end-to-end.
- **AVP Length[3 byte]**: lunghezza del campo.
- **Vendor ID[4 byte]**: presente se **V=1**. Sostituisce il ruolo dell'AVP Code.
- **Data**: payload del campo.

<sup>6</sup>Attribute Value Pair

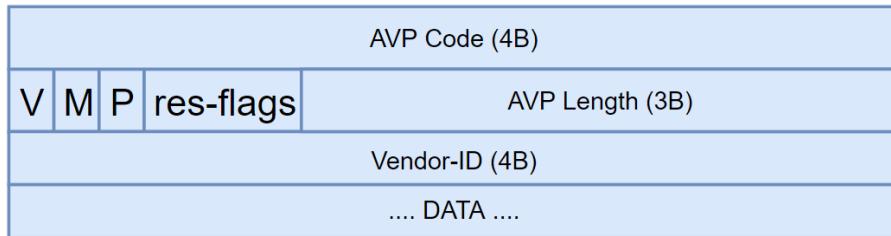


Figura 6.7: AVP Header

**Nota [Mandatory Flag-Bit]**

Questo flag negli AVP permette di realizzare il cosiddetto **Incremental Deployment**, ovvero la possibilità di implementare aggiornamenti nel sistema continuando a supportare meccanismi preesistenti. In particolare, per ogni AVP, ovvero per ogni attributo, viene applicata la seguente logica:

**M=1** il campo AVP è **necessario** alla funzionalità trasportata nel messaggio. Nel caso in cui **non venga compreso/supportato dal ricevente**, quest'ultimo avvia una **negoziazione** con il **mittente** per determinare una configurazione supportata da entrambi.

**M=0** il campo AVP è **non necessario** alla funzionalità trasportata dal messaggio. Nel caso in cui **non venga compreso/supportato dal ricevente**, viene silenziosamente **scartato**.

### 6.5.3 Agenti Intermedi

DIAMETER prevede e supporta i seguenti tipi di **agenti intermedi**:

**Definizione 6.9 (Relay Agent)**

Accetta richieste e le **instrada** al **DIAMETER server** appropriato basandosi sul contenuto della richiesta stessa e usando una specifica tabella di instradamento.

**Definizione 6.10 (Proxy Agent)**

Effettua le stesse operazioni del relay agent ma può modificare i messaggi che instrada.

**Definizione 6.11 (Redirect Agent)**

Permette di effettuare la **separazione del control-plane dal data-plane** cooperando con un relay agent. In particolare, si occupa di **prendere le decisioni di instradamento** al posto del relay agent.

Tali decisioni verranno comunicate al relay tramite **notifiche di direzione**, il quale applicherà l'instradamento deciso.

Ciò è utile nel caso in cui si vogliano **centralizzare** tutte le scelte di **intradamento** in un'unica entità, così da semplificare la gestione della rete.

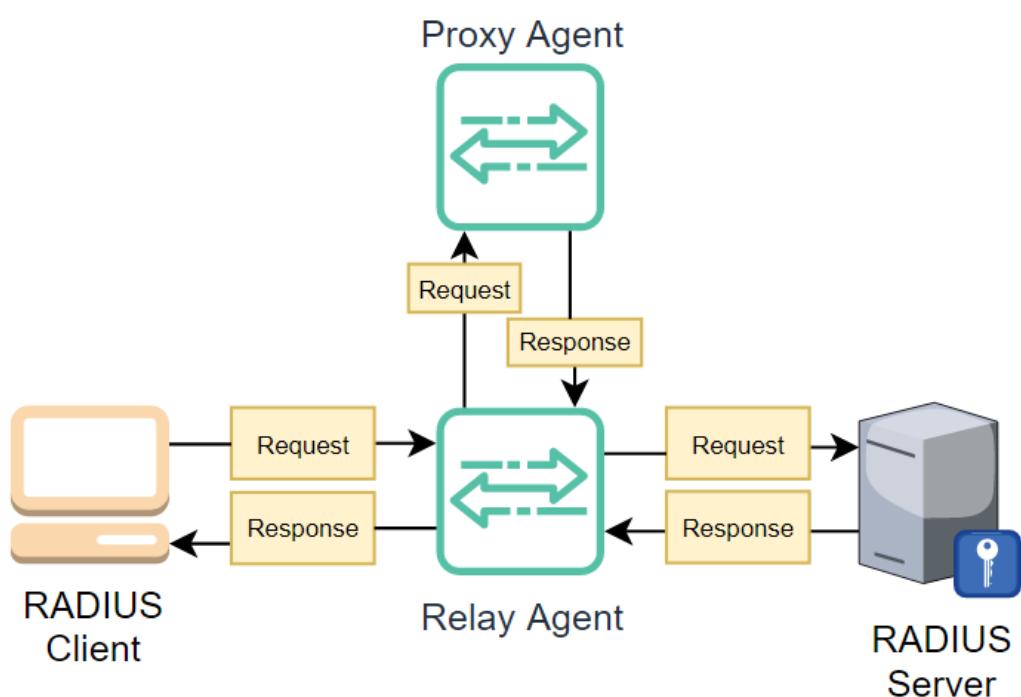
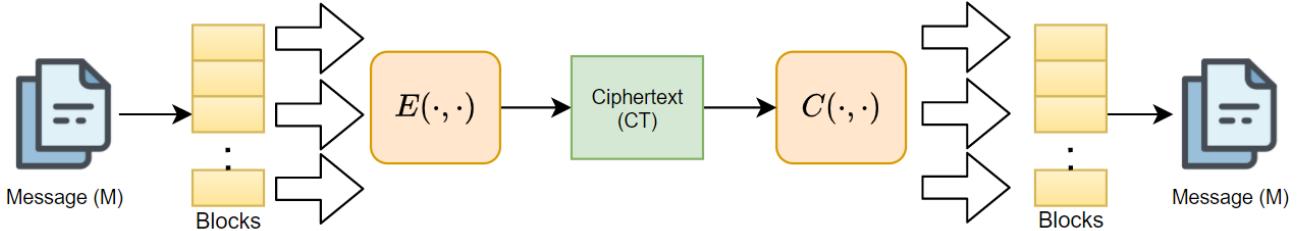


Figura 6.8: Relay Agent

# Capitolo Block Ciphers

L'obiettivo dei Block Ciphers è quello di fornire dei meccanismi più "generali" di cifratura rispetto ai substitution ciphers. Lo schema base di un cifratore a blocchi è il seguente:



**Figura 7.1:** Block Cipher Scheme

## 7.1 Funzionamento di Base

L'idea alla base dei block cipher è quella di dividere il messaggio in blocchi di dimensione specifica che vengono poi cifrati con una chiave  $K$ . Inoltre, ogni algoritmo di questo tipo **dove** implementare una logica di **pseudo-random permutation**, basata sulla dimensione della chiave usata. Tale permutazione deve avere le seguenti proprietà:

### Proprietà [Pseudo Random Permutation]

- **Biettiva e Reversibile:** dato un input abbiamo un solo output e dato un output dobbiamo poter risalire all'input.
- **Associazione Pseudo-Random:** l'associazione fra i caratteri di partenza con quelli di arrivo deve essere una permutazione casuale.

Il problema di avere una funzione biettiva, è che se un messaggio una volta diviso in blocchi presentasse due blocchi uguali, avremmo uno stesso testo cifrato per quei due blocchi. Stesso discorso se cifriamo due volte lo stesso messaggio.

Questo è il funzionamento della modalità **ECB** (*Electronic Code Book*) , che non va mai usato perché **perdiamo semantic security**.

**Osservazione:** Supponiamo di avere una chiave di dimensione  $n = 3$  bit. L'insieme di tutti i plaintext possibili è di dimensione 8, quindi le permutazioni possibili sono  $2^3! = 8! = 40320$ .

Se consideriamo AES la dimensione della chiave è di 128 bit, quindi il numero di permutazioni è incredibilmente elevato. ■



**Nota** In realtà in AES le permutazioni sono di meno, perché vengono richieste delle proprietà aggiuntive.

Una possibile soluzione per ovviare al problema descritto sopra è ovviamente quello di inserire degli *initialization vectors* per nascondere il plaintext al momento dell'applicazione della funzione di **PRP** (*pseudorandom permutation*) secondo il seguente schema:

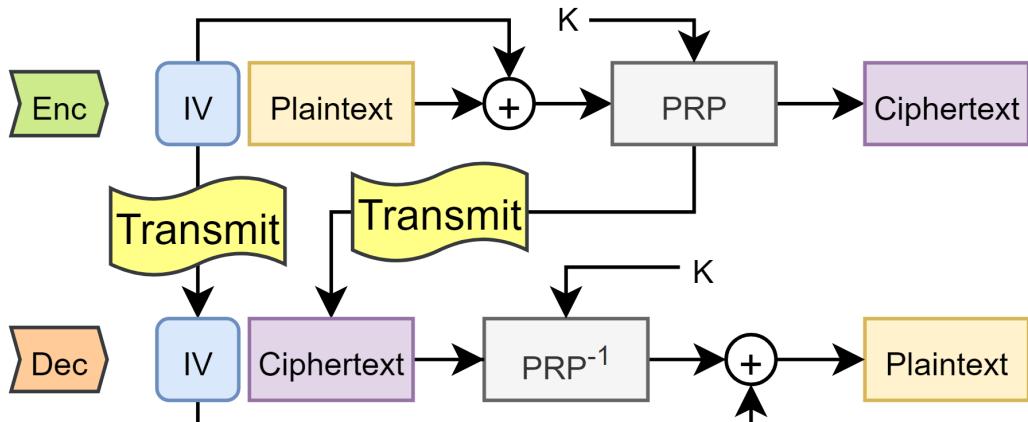


Figura 7.2: Block Cipher with IV

Le proprietà che adesso dobbiamo richiedere per gli IV sono due:

**Teorema 7.1 (Semantic Security for ECB)**

*Se le seguenti proprietà sono soddisfatte abbiamo sicurezza semantica ECB.*

**Proprietà [IV properties in Block Ciphers]**

- Gli IV **NON** devono **MAI RIPETERSI**.
- Gli IV **DEVONO** essere **NON PREDICIBILI**.

A patto che:

- Il messaggio è più piccolo di un blocco e non si ripete mai.
- Gli IV per messaggi ripetuti hanno valori casuali e sono della stessa dimensione di un blocco.
- Per messaggi più lunghi di un blocco bisogna combinare i blocchi per non costruire diversi ciphertext.

**Osservazione:** L'inconveniente è che dobbiamo trasmettere un messaggio di dimensione doppia:  $n$  bit di ciphertext più  $n$  di IV.



## 7.2 Modes of Operations

I block ciphers vengono suddivisi in categorie in base alla loro "Modalità di Operazione". Le più usate sono **CBC** e **CTR** mentre quelle consigliate da usare sono **CFB** e **OFB** per alcune particolari proprietà. Ci sono anche due modalità avanzate che forniscono *authenticated encryption*: **GCM** e **OCB**.



**Nota** Ricordiamo che il problema fondamentale è che gli IV non devono mai ripetersi, quindi è necessario ridurne la generazione, oltre che l'overhead sulla trasmissione.

### 7.2.1 Cipher Block Chaining: CBC

Questa modalità si basa su un sistema di block-chain descritto nel modo seguente:

#### Definizione 7.1 (Encryption with Cipher Block Chaining Mode)

- 1: Divide the plaintext in  $n$  chunks.
- 2:  $c[0] = PRP(K, IV \oplus m[0])$
- 3: **for**  $i = 1$  to  $n - 1$  **do**
- 4:      $c[i] = PRP(K, c[i - 1] \oplus m[i])$
- 5: **end for**
- 6: Send IV along with the ciphertexts.

▷ The IV must be truly rnd

▷  $c[i - 1]$  is used as IV for  $c[i]$

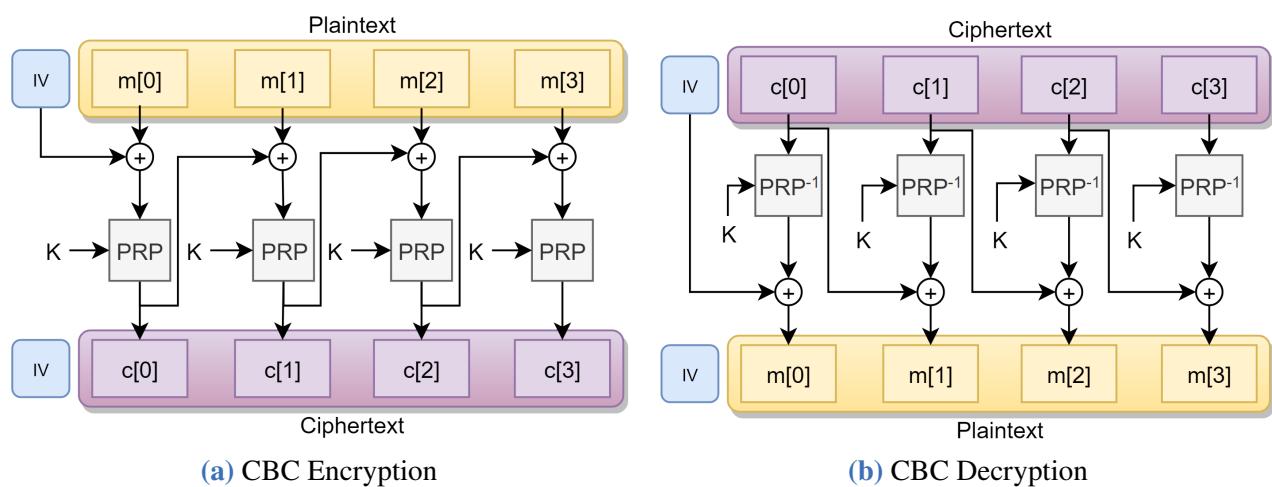
Se l'IV è **truly random** il meccanismo è **semantic secure** in quanto il primo ciphertext generato sarà una quantità che può essere vista come casuale e tale randomicità verrà trasportata anche ai ciphertext successivi. Pertanto, **con un solo IV inviato possiamo garantire sicurezza semantica per tutto il messaggio**.

Tuttavia le **cifrature non sono parallelizzabili** in quanto le cifrature  $i$ -esime dipendono da quella successiva e l'ordine delle cifrature è quindi fondamentale.

#### Definizione 7.2 (Decryption with Cipher Block Chaining Mode)

- 1:  $m[0] = IV \oplus PRP(K, c[0])$
- 2: **for**  $i = 1$  to  $n - 1$  **do**
- 3:      $m[i] = c[i - 1] \oplus PRP(K, c[i])$
- 4: **end for**

Osserviamo che il discorso sulla parallelizzazione non sussiste per la decrittazione, in quanto con il solo IV e il messaggio è possibile decifrare tutto il ciphertext in parallelo.



**Figura 7.3:** CBC Mode



**Nota** CBC è il sistema più comune e usato per cifrare dei messaggi, ma è sicuro sempre se e solo se la nonce è **truly random e non predicable**.

**Osservazione:** IV predicibili possono portare a CPA, come successo in **BEAST** nel 2011.

 **Nota** L'implementazione di CBC richiede due circuiti differenti per cifrare e decifrare i messaggi. Poiché questi meccanismi sono spesso inseriti a livello hardware, usare due circuiti significa usare il doppio dello spazio per la realizzazione del sistema di cifratura.

 **Nota** Poiché lo xor impone che le dimensioni dei chunk siano di un numero fissato, è necessario fare un padding per messaggi che non sono multipli del numero di byte scelto. Il padding viene calcolato secondo degli standard, un esempio è **PKCS#7**.

### 7.2.2 Cipher Feedback Mode e Output Feedback Mode

Le modalità CFB e OFB offrono un punto di vista differente rispetto a CBC. In particolare, tendono ad imitare gli stream ciphers, producendo un keystream utile alla cifratura del plaintext, permettendo di usare un solo circuito, sia per  $PRP$  che per  $PRP^{-1}$ .

#### Definizione 7.3 (Cipher Feedback Mode - CFB [Encryption])

- 1: Assume we have  $n$  plaintexts  $m[i] i = 1, \dots, n$ .
- 2:  $K_s = ENC(key, IV)$  ▷ The IV must be truly rnd
- 3:  $c[0] = m[0] \oplus K_s$  ▷ encrypt as stream ciphers with keystream
- 4: **for**  $i = 1$  to  $n - 1$  **do**
- 5:    $K_s = ENC(key, c[0])$
- 6:    $c[i] = c[i - 1] \oplus K_s$  ▷  $c[i - 1]$  is used as IV for  $c[i]$
- 7: **end for**
- 8: Send **IV** along with the **ciphertexts**.

#### Definizione 7.4 (Output Feedback Mode - OFB [Encryption])

- 1: Assume we have  $n$  plaintexts  $m[i] i = 1, \dots, n$ .
- 2:  $K_s[0] = ENC(key, IV)$  ▷ The IV must be truly rnd
- 3: **for**  $i = 0$  to  $n$  **do**
- 4:    $c[i] = m[0] \oplus K_s$  ▷ encrypt as stream ciphers with keystream
- 5:    $K_s[i] = ENC(key, K_s[i - 1])$  ▷  $K_s[i - 1]$  is used as IV for  $K_s[i]$
- 6: **end for**
- 7: Send **IV** along with the **ciphertexts**.

Sia CFB che OFB cifrano il messaggio mascherandolo con un keystream generato tramite un **block cipher encryption** senza la necessità di fare padding. L'**encryption non è parallelizzabile** in nessuno dei due casi ma **OFB permette** di fare **preprocessing** in quanto lo stato del messaggio cifrato non dipende da quello precedente e i keystream possono essere calcolati in blocco. Per fare decryption è possibile usare lo stesso circuito di encryption, dove **CFB** è in vantaggio perché **permette di parallelizzare** il processo.

**Definizione 7.5 (CFB Decryption)**

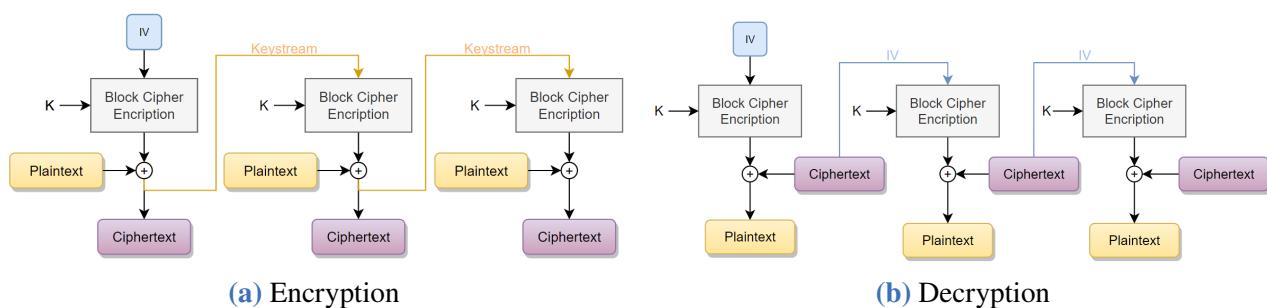
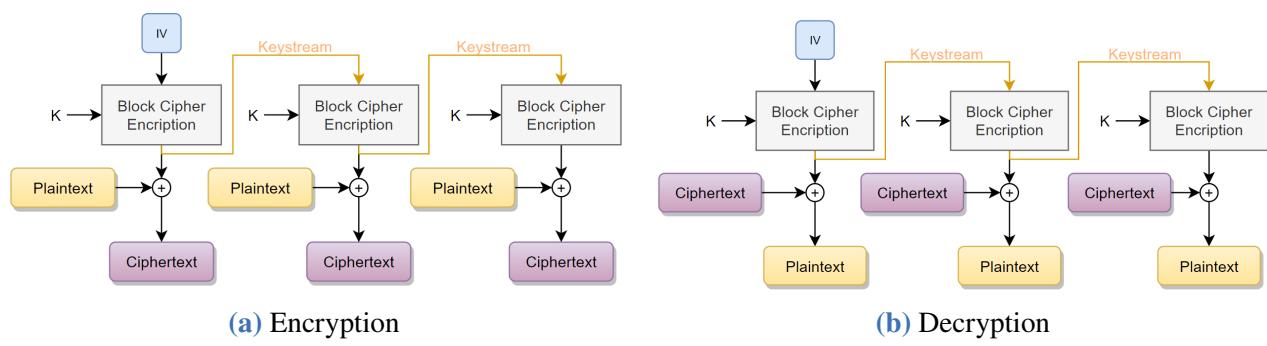
- 1: Assume we have  $n$  ciphertexts.
- 2: Take the IV from arriving message.
- 3:  $K_s[0] = ENC(IV, k)$
- 4: **for all ciphertexts do**
- 5:  $m[i] = K_s[i - 1] \oplus c[i]$  ▷ Get the Plaintext
- 6:  $K_s[i] = ENC(c[i - 1], k)$  ▷ Generate Keystream from  $c[i - 1]$
- 7: **end for**

□

**Definizione 7.6 (OFB Decryption)**

- 1: Assume we have  $n$  ciphertexts.
- 2: Take IV from arriving message.
- 3:  $K_s[0] = ENC(IV, k)$
- 4: **for all ciphertexts do**
- 5:  $m[i] = K_s[i - 1] \oplus c[i]$  ▷ Get the Plaintext
- 6:  $K_s = ENC(K_s[i - 1], K)$  ▷ Generate Keystream from  $K_s[i - 1]$
- 7: **end for**

□


**Figura 7.4:** Cipher Feedback Mode

**Figura 7.5:** Output Feedback Mode

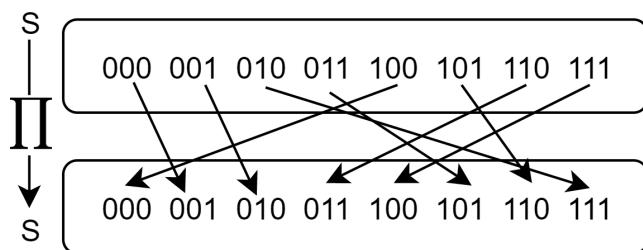
## 7.3 Short Cycle Problem

I cifratori a blocchi dipendono fortemente dal blocco di encryption che calcola la pseudorandom permutation. Questo implica che la funzione che scegliamo per calcolare la permutazione deve essere fatta bene e robusta anche a specifiche condizioni iniziali. In particolare, non possiamo usare funzioni che soffrono del:

### Definizione 7.7 (Short Cycle Problem)

*Condizione per la quale un algoritmo di cifratura, a partire da alcune specifiche condizioni iniziali, tendono a ripetere i valori di ciphertext dopo poco tempo.*

**Esempio 7.1** Consideriamo l'insieme  $S$  e la permutazione seguente  $\Pi$ .



**Figura 7.6:** PRP function for 3-bit blocks

1. **OFB** con  $IV = 010$ :  $C = 010\ 111\ 100\ 001\ \textcolor{red}{010\ 111}\ \dots$   
I valori di  $C$  cominciano a ripetersi dopo 5 iterazioni.
2. **OFB** con  $IV = 011$ :  $C = 011\ 101\ 110\ \textcolor{red}{011\ 101\ 110}\ \dots$   
I valori di  $C$  cominciano a ripetersi dopo 3 iterazioni.

**Esempio 7.2** Vediamo che anche **CBC** presenta gli stessi problemi se cifra un testo con delle ripetizioni:

- $P = 011\ 011\ 011$  con  $IV : 010$ :  $C = (\textcolor{red}{010})\ 010\ 010\ 010$

## 7.4 Counter Mode

Poiché le modalità precedenti sono soggette allo *short cycle problem* una modalità resistente è necessaria. In particolare, poiché il problema è intrinseco all' $IV$  scelto per avviare la cifratura, se questo fosse sviluppato dinamicamente e sempre diverso, non potremmo avere cicli all'interno di una sequenza di cifratura.

Assumendo l' $IV$  come un **contatore** incrementato ad ogni nuovo blocco, è possibile dimostrare che se la PRP è sicura, allora il risultato pseudorandomico generato è sicuro.



**Nota** La modalità **CTR** gode della maggior parte dei vantaggi:

- ✓ Trasforma i block-cipher in stream-cipher.
- ✓ Combina i vantaggi di **CFB**(def 7.3) e **OFB**(def 7.4).

- ✓ Permette un'implementazione efficiente (sia HW che SW) utilizzando parallelizzazione sia in encryption che decryption.
- ✓ Richiede l'implementazione di un singolo blocco di cifratura per entrambe le operazioni.
- ✓ Permette di attuare meccanismi di **Random Access** (ad esempio a sistemi di memoria cifrata) poiché la decrittazione del blocco *i-esimo* non dipende dai precedenti.
- ✓ E' sicuro se usiamo accortezze sui contatori.
- ✓ Viene garantita robustezza allo Short Cycle Problem (def 7.7).

### Definizione 7.8 (Counter Mode Encryption/Decryption)

- 1: Assume we have  $n$  plaintext/ciphertext
- 2: Initialize a counter  $ctr$  as standard specifies<sup>1</sup>.
- 3: **for all** plaintexts **do**
- 4:      $K_s[i] = ENC(ctr, K)$
- 5:      $c[i] = K_s[i] \oplus m[i]$
- 6:      $ctr = ctr + 1$
- 7: **end for**

<sup>1</sup>Ad esempio AES-CTR specifica che per l'**Initial Counter Block** il contatore è formato da 128bit dove i **primi 96** sono costituiti dall'**IV** e i **restanti 32** dal **contatore, partendo da 1**.

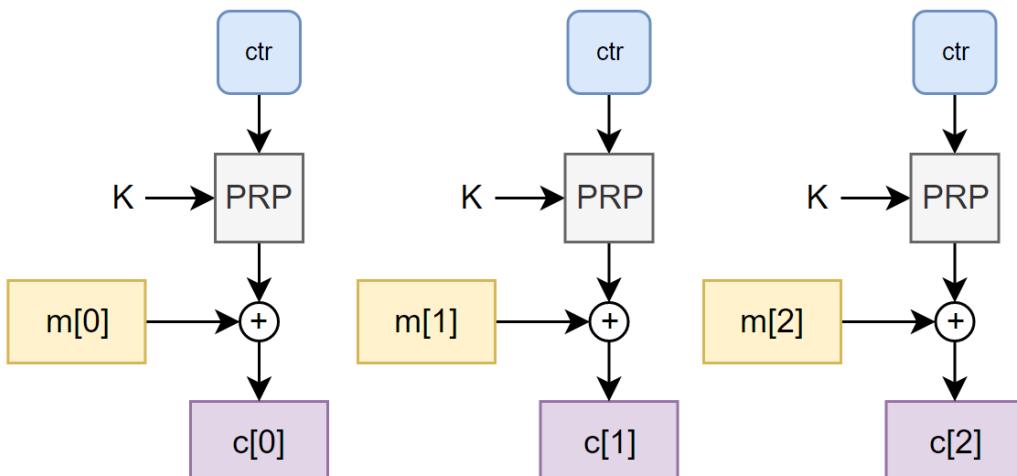


Figura 7.7: Counter Mode Block Cipher

# Capitolo Authenticated Encryption with Associated Data

Sappiamo che un cifratore ben strutturato può garantire *confidentiality*, ovvero abbiamo semantic security contro attacchi *CPA*. Tuttavia, per garantire *integrity* abbiamo fatto fino ad ora un discorso separato, basato su un codice generato tramite una funzione hash.

**Osservazione:** Se un attaccante dovesse essere "attivo", ovvero in grado di creare un testo cifrato il cui MAC sia valido, non avremmo modo di garantire *integrity*. ■

## Definizione 8.1 (Chosen Ciphertext Attack)

*Scenario di attacco nel quale un hacker può fare, oltre all'eavesdropping, anche una cifratura di un plaintext di sua creazione che è in grado di modificare un messaggio.*

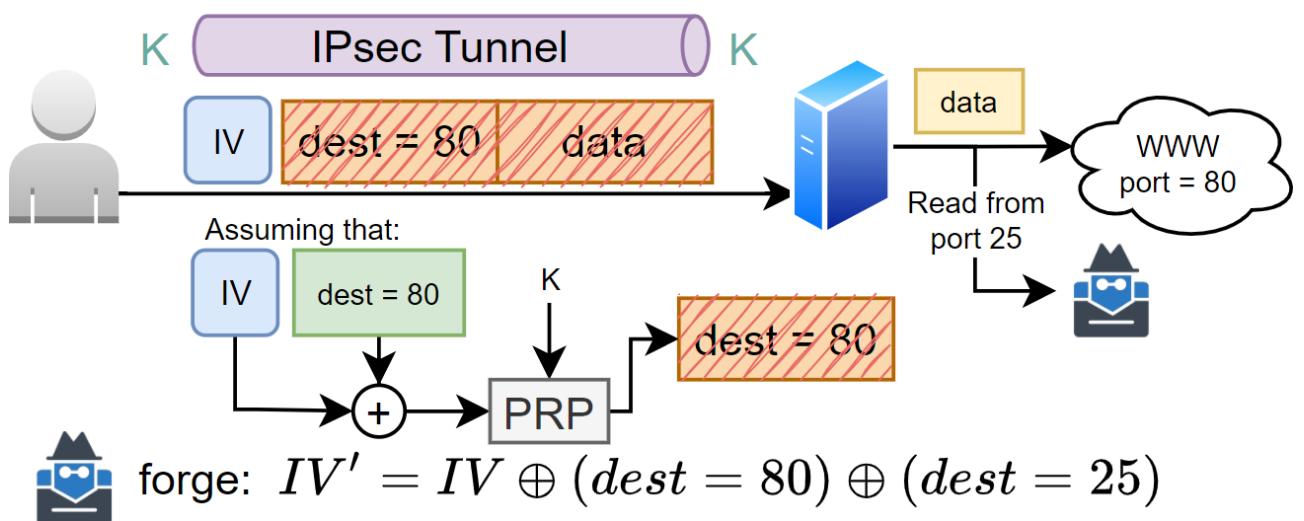
*Rompendo, di fatto, sia confidentiality che integrity.* □

Vediamo due esempi di come **rompere confidentiality** porti ad una **rottura di integrity**.

**Esempio 8.1** Consideriamo l'invio di un pacchetto *TCP/IP* da **A** verso **B** passando per un web-server che gestisce lo stack *TCP/IP*. Facciamo le seguenti supposizioni:

1. Il pacchetto attraversa un server sul quale **E** può intercettare i processi su una porta specifica. Per semplicità il **pacchetto** sia **diretto sulla porta 80 di B**, mentre **E** può **leggere** dalla **porta 25**.
2. Il canale di trasferimento è **protetto con sola encryption** (garantiamo solo *confidentiality*).
3. L'attaccante è in grado di generare un cipher-text il cui contenuto cambia la porta da 80 a 25.

Poiché non ci stiamo preoccupando di integrity (e quindi non usiamo un MAC), **E** può leggere i dati in chiaro dal server dalla porta 25. In particolare, se viene usato **AES-CBC** è semplice in quanto basta fare uno xor. Vediamo lo schema:



**Figura 8.1:** Broken Confidentiality

Inserendo l' $IV'$ , nel processo di cifratura possiamo cambiare la porta. Infatti: ■

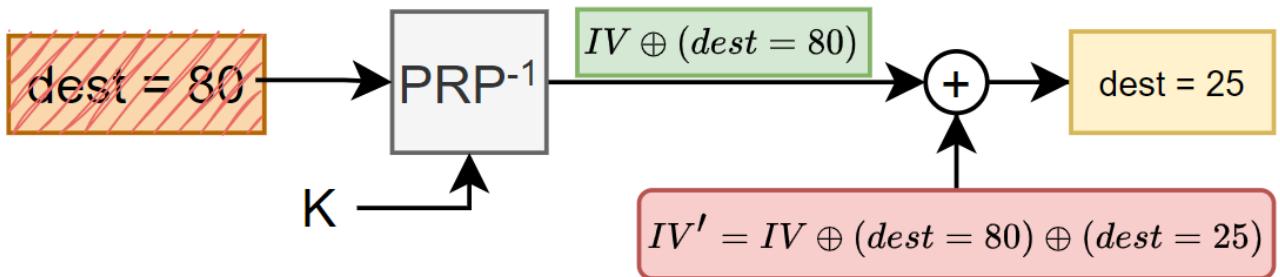


Figura 8.2: Description

**Esempio 8.2 ACK-Oracle:** Supponiamo che A stia usando una *remote terminal app*, dove ogni keystroke viene cifrato in CTR e inviato verso un server. I "poteri" di E sono i seguenti:

1. **E non ha accesso al backend.**
2. La cifratura viene fatta in CTR mode.
3. L'header è tipicamente noto poiché la struttura è definita da standard e i valori possono essere predetti.
4. E può copiare il messaggio cifrato di A.

**FACT:** Lo stack TCP/IP processa solo pacchetti validi, rispondendo con un *ACK* solo ai pacchetti con checksum valido. Il checksum è calcolato in funzione dell'header del pacchetto e del carattere inviato D.

**Nota** Il protocollo restituisce pertanto **DUE** risultati, indicandoci quali sono i pacchetti corretti e non rispondendo nulla se il pacchetto non lo è.

Supponiamo di fare molte copie del messaggio e di provare  $\forall t, s$  ad inviare verso il server una copia modificata nel campo checksum e carattere calcolando  $checksum \oplus t$  e  $D \oplus s$ . Per ogni volta in cui il server dello stack TCP/IP risponderà con un *ACK*, otteniamo un'equazione valida del tipo:

$$checksum(pkt - hdr, D \oplus s) = t \oplus checksum(pkt - hdr, D)$$

Poiché l'header del pacchetto è noto (punto 3), mentre la coppia checksum, D non lo è, il sistema è risolubile. Una volta trovata una soluzione, è facile risalire al checksum e a D in base alla struttura con cui è calcolato il checksum. ■

**Osservazione:** Dagli esempi capiamo che non bisogna **MAI** usare un meccanismo di encryption da solo ma bisogna sempre aggiungere un servizio di integrity, in particolare, **SI DEVE** usare **SOLO** authenticated encryption. ■

## 8.1 Meccanismi di AE

Gli algoritmi di AE producono un ciphertext di lunghezza maggiore del plaintext di partenza in quanto oltre al testo cifrato devono includere **anche** il tag di auth.

**Nota** E' impossibile, per un meccanismo di AE, produrre un ciphertext di lunghezza pari al plaintext.

Per costruzione gli algoritmi di AE verificano se il messaggio è valido e, successivamente, rispondono in due modi:

<sup>o</sup>A meno di alcuni casi speciali chiamati **Homomorphic Encryption**.

- ✓ Il messaggio originale.
- ✗ Un messaggio di **REJECT**.

### Proposizione 8.1

Gli algoritmi di Authenticated Encryption decifrano il messaggio se solo se il tag è valido. □

Possiamo dare una definizione di sicurezza per un algoritmo di authenticated encryption:

### Definizione 8.2 (Sicurezza per AE)

Un algoritmo di Authenticated Encryption è sicuro se:

- E' semantically secure sotto CPA.
- Garantisce integrity del ciphertext: la probabilità che un messaggio cifrato generato dall'avversario venga decifrato<sup>1</sup> deve essere trascurabile.

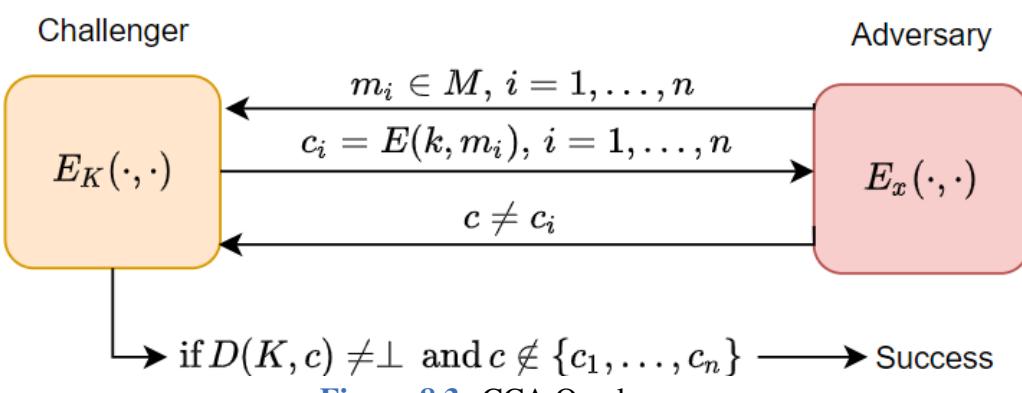
<sup>1</sup>Consideriamo un successo anche se il messaggio cifrato viene decifrato in un messaggio completamente casuale. □



**Nota** Il meccanismo **Encrypt-than-MAC** soddisfa def 8.2 ma non è di tipo AEAD

### Esempio 8.3 Semantic security against Chosen Ciphertext Attack Model

Consideriamo un esempio simile a quello di fig. 8.3 dove questa volta il meccanismo di risposta è diverso in quanto il risultato della decrittazione del chosen ciphertext può essere un success o un reject. Supponiamo ovviamente che l'avversario non conosca la chiave di cifratura ma scegliendo i plaintext di una challenge è in grado di ottenere dei ciphertext in risposta, come se il challenger fosse un oracolo. La mappa ottenuta permette all'avversario di generare un ciphertext diverso dai precedenti (ad esempio facendo combinazioni lineari/non lineari dei  $c_i$  precedenti), che una volta inviato come nuovo messaggio se viene decifrato in una forma qualsiasi di plaintext (sensato o meno) costituisce un indizio per l'avversario in quanto ha generato un messaggio valido. ■



**Figura 8.3:** CCA Oracle game

### 8.1.1 Associated Data

Quando trasmettiamo dei dati c'è spesso la necessità di trasmettere una parte dei dati in chiaro, ad esempio in un pacchetto TCP/IP il cui header può restare in chiaro. Quello che vorremmo però è che l'autenticazione venga fatta **su tutto** il pacchetto.

Possiamo avere due casi limite, tipicamente:

- No Associated Data: è sufficiente CCA-Secure Encryption, quindi tutto è cifrato.
- No Encrypted Data: è sufficiente un MAC costruito in modo sicuro (def 3.4).

Gli algoritmi di AEAD possono lavorare in modo diverso:

- Two-Layer: prima cifro e poi applico MAC (*es: AES-GCM*). Poiché servono due iterazioni, è un meccanismo più lento e non parallelizzabile poiché per produrre il MAC serve il ciphertext.
- One-Layer: faccio cifratura e MAC contemporaneamente (*es: OCB*). Più veloce (*online cipher*).



**Nota** Le performance sono fondamentali al giorno d'oggi visto che file sono grandi giga e giga.

Ad ogni modo, possiamo osservare due problemi fondamentali:

**Problema 8.1 Dove posizionare la parte di Associated Data?** Tipicamente non c'è una risposta, poiché dipende dal protocollo. Ad esempio se parliamo di header dovrà andare in testa, in altri casi alla fine o anche un mix tra la parte di cifratura e in chiaro.

**Problema 8.2 Misuse Resistance** Quanto è robusto un meccanismo di AEAD quando un IV si ripete?

## 8.2 AES-GCM: Galois Counter Mode

Algoritmo usato nella maggior parte dei protocolli di sicurezza come IPsec e TLS, che usa una struttura **Enc-than-MAC**.

- **Encryption:** AES-CTR (def 7.8).
- **MAC:** GHASH function. Non una crypto-hash ma una struttura algebrica per fare prodotti tra polinomi in modo molto veloce. **Non garantisce sicurezza crittografica.**

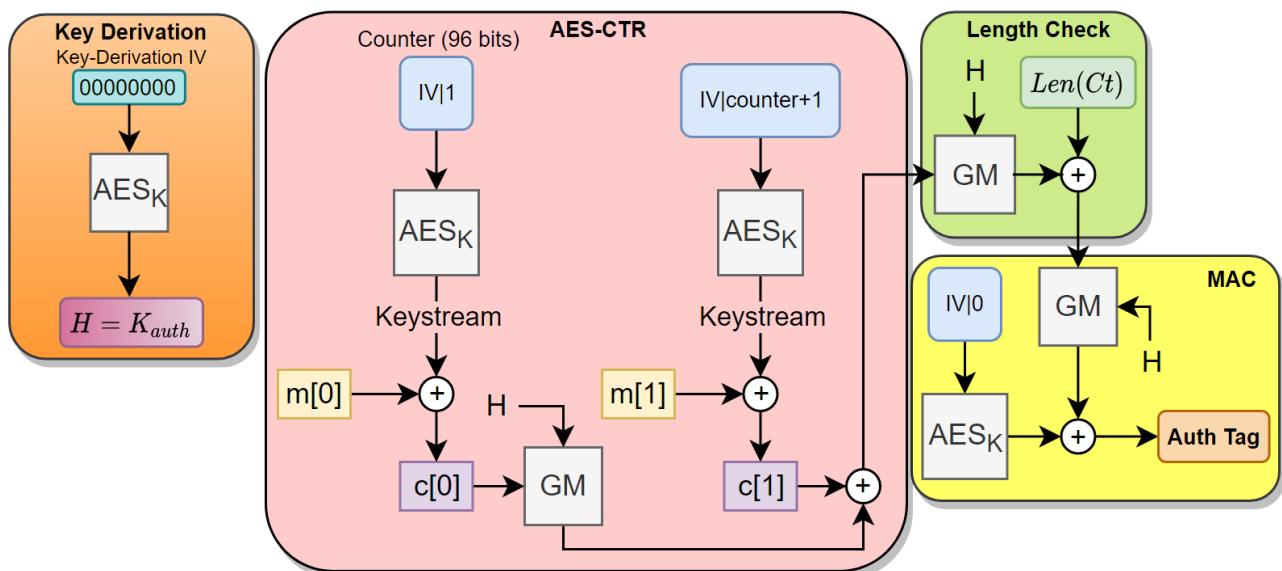


Figura 8.4: AES-GCM scheme

<sup>1</sup>Galois Field

**Definizione 8.3 (AES-GCM - Galois Counter Mode)**

Data la chiave  $K$ , AES-GCM opera nel seguente modo:

- **Key-Derivation:** Poiché la **chiave di cifratura DEVE essere diversa da quella di autenticazione**, produciamo una chiave  $H$ , usata per produrre il MAC.
- **AES-CTR:** La parte di cifratura del messaggio viene svolta come in CTR<sup>2</sup>(def 7.8) **partendo da 1**, ma ogni ciphertext ottenuto produce un MAC ottenuto tramite il blocco  $GM(H)$  per mezzo di prodotti tra polinomi. Il MAC ottenuto va in xor con il ciphertext successivo per produrre un nuovo MAC.
- **Length-Check:** Alla fine della cifratura, l'ultimo MAC prodotto viene messo in xor con la lunghezza del ciphertext complessivo per produrre un nuovo MAC.
-  **Nota** L'aggiunta del controllo sulla lunghezza protegge la costruzione del MAC da un expansion attack.
- **Wegman-Carter MAC:** Il MAC della lunghezza viene messo in xor con un keystream ottenuto in CTR-mode ma con contatore 0 per produrre un MAC sicuro.
-  **Nota** L'IV usato a questo step sarà **SEMPRE** diverso in quanto il contatore iniziale era 1.

<sup>2</sup>Il blocco  $AES_k$  è una Pseudo-Random Function (PRF).

**Corollario 8.1 (Wegman-Carter Construction)**

Il MAC costruito tramite GCM è una costruzione di tipo Wegman-Carter. Questa costruzione ci fornisce un modo di creare un codice di autenticazione anche quando **non usiamo** delle funzioni crittografiche. Abbiamo però bisogno di due elementi fondamentali:

- **Funzioni Hash Universali.**
- **Pseudo Random Functions.**



Le funzioni hash universali sono una famiglia di funzioni dette "**Key-Hash Function**" il cui output **dipende** dalla specifica chiave con cui la funzione è inizializzata. In particolare:

**Definizione 8.4 (Universal Hash Functions)**

Una famiglia di Key-Hash Function è detta universale se è difficile, per un attaccante che **non conosce la chiave**, trovare una collisione tale che  $H_k(M_1) = H_k(M_2)$ .

Analogamente:  $\forall x, y : x \neq y \text{ Prob}\{H_x(M) = H_y(M)\} \leq 1/m$ , con  $m$  dimensione del digest.



Rispetto ad una crypto hash, le differenze principali stano nel fatto che

- **se la chiave è nota** trovare una collisione **può essere facile**.
- L'output **potrebbe non essere** pseudorandomico.
- Dalla seconda proprietà: Un cambio di chiave dovrebbe provocare un cambio nel digest in modo causale e quindi ridurre drasticamente la probabilità di collisione.
- Dalla seconda proprietà: Non dovrebbero esserci coppie di messaggi che producono lo stesso hash per chiavi diverse.



### Nota Come tratto la parte di Dati Associati?

*Semplice: Per ogni blocco di AD produco un MAC che va in xor con il blocco successivo per produrre un nuovo MAC. Alla fine della parte di AD, il MAC va in xor con il primo ciphertext e si continua come indicato in fig. 8.4. La differenza sostanziale sta nell'allungare la dimensione del messaggio complessivo, che ora deve comprendere anche la parte di AD.*

## 8.2.1 Misuse Resistance per GCM

La costruzione trovata è utile ma ha un problema **fondamentale: E' LINEARE**, su due aspetti.

- Il tag è lineare in xor in quanto:  $tag = GHASH_H(CT) \oplus AES_K(IV, 0)$ . Pertanto, dati due tag

$$tag_1 = GHASH_H(CT_1) \oplus AES_K(IV, 0)$$

$$tag_2 = GHASH_H(CT_2) \oplus AES_K(IV, 0)$$

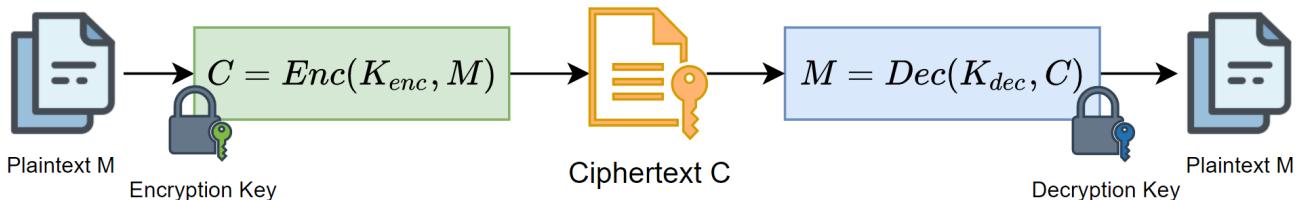
$$tag_1 \oplus tag_2 = GHASH_H(CT_1) \oplus GHASH_H(CT_2)$$

- Poiché la GHASH function usa una moltiplicazione tra polinomi, un attaccante può sempre recuperare la chiave di auth e forgiare dei messaggi validi (*la chiave K comunque rimane segreta*).

Esistono chiaramente delle costruzioni più robuste quando si verifica il riutilizzo di una nonce, ma non le trattiamo in questo momento.

# Capitolo Asymmetric Cryptography

Per poter instaurare una **comunicazione sicura** tra due entità che **non** possiedono un **segreto comune** è **necessario** utilizzare algoritmi che non rivelino ad entità terze in rete alcun segreto critico. Questi algoritmi sono detti a **Chiave Asimmetrica**, ovvero algoritmi dove le chiavi di encryption e decryption sono **legate ma differenti**. Distinguiamo in **chiave pubblica e privata**.



**Figura 9.1:** Asymmetric Encryption Base Scheme

C'è un **importante MA**: gli algoritmi a chiave asimmetrica sono **computazionalmente onerosi** e non sarebbero praticabili per l'**intera** sessione di comunicazione cifrata.

C'è bisogno di un approccio **ibrido**:

- Algoritmi Asimmetrici per **inizializzare** la comunicazione **sicura**
- Algoritmi Simmetrici per il **trasferimento sicuro** di informazioni.

## Definizione 9.1 (Hybrid Approach for Persistent Communication)

1. **Handshake/Session Key Exchange:** Viene utilizzata la **criptazione asimmetrica** per scambiarsi **informazioni sulla crittografia simmetrica** da utilizzare successivamente e **impostare dei segreti comuni (Key Agreement)** tra le due parti.

**Osservazione:** Dai segreti comuni vengono derivate le chiavi simmetriche, dette **Chiavi di Sessione**, usate nella **criptazione simmetrica** e nel **controllo di integrità**. ■

2. **Data Transfer:** le due parti **comunicano** in maniera sicura tra loro, tramite **meccanismi a chiave simmetrica**.

3. **Rekeying:** eventualmente vengono **derivate nuove chiavi simmetriche** dagli stessi segreti comuni impostati precedentemente. ■

## 9.1 Public Key Encryption & Digital Signature

Gli algoritmi asimmetrici utilizzano la **chiave** che viene condivisa, ovvero resa **pubblica**, durante la trasmissione dei dati. Poiché le chiavi pubbliche e private sono legate, vale la seguente proprietà:

### Proposizione 9.1 (Pubkey Encryption)

*In uno schema di cifratura con chiave pubblica e chiave privata, **chiunque** può cifrare un messaggio  $M$ , ma soltanto il vero destinatario può decifrarlo.* ■

La firma digitale è un concetto ben diverso, che sfrutta l'ipotesi che le funzioni di  $Enc$  e  $Dec$  siano commutative e quindi:

$$Dec(Enc(M)) = Enc(Dec(M)) \quad (9.1)$$

Questo implica che, in un sistema a **firma digitale**, abbiamo la seguente proprietà:

### Proposizione 9.2 (Digital Signature)

*Il meccanismo di firma digitale fornisce sia message integrity che l'autenticazione della sorgente/mittente (non repudiant). Viene realizzato utilizzando una funzione hash crittografica  $H$ , un algoritmo di criptazione asimmetrico e le chiavi privata e pubblica del mittente, per generare una firma del messaggio. In particolare:*

- Il **mittente** genera la sua firma  $sign_M = Dec_{src}(H(M))$ , criptando la sua **chiave privata** il digest del messaggio  $M$ . Invia quindi  $\{M, sign_M\}$  al destinatario.
- Il **destinatario** legge la firma e ottiene il digest decriptando con la **chiave pubblica del mittente**.

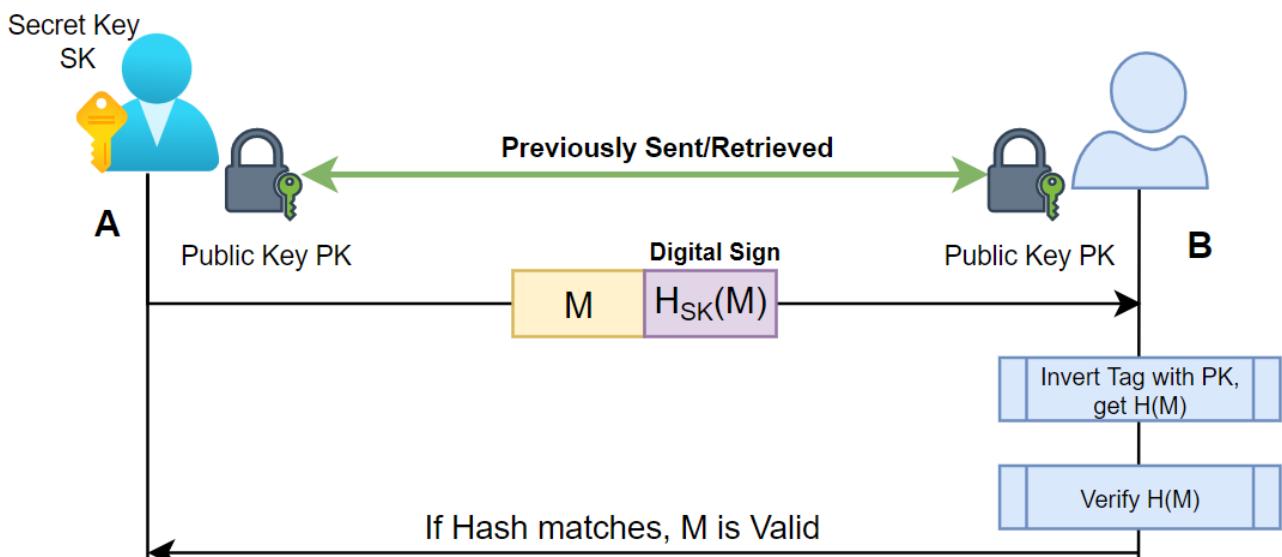
$$H(M) = Enc_{dest}(sign_M)$$

Ricalcola quindi il digest dal messaggio ricevuto e lo confronta con quello ottenuto dalla firma.

- Se i due digest corrispondono, il messaggio è integro e proviene con certezza dal mittente. □



**Nota** La funzione hash viene utilizzata senza chiave simmetrica in quanto ha il solo scopo di ridurre la dimensione dell'oggetto da criptare asimmetricamente, così da rendere il meccanismo efficiente. Inoltre, il **destinatario**, ovvero colui che ha necessità di verificare l'integrità del messaggio, **ha bisogno solamente della chiave pubblica del mittente**, ovvero di nessun segreto condiviso, a differenza del MAC che richiede una chiave simmetrica, ovvero un segreto noto a entrambe le parti.



**Figura 9.2:** Simple digital sign scheme

La firma digitale permette di avere un sistema più sicuro rispetto ad un semplice tag, in quanto siamo **certi** che chi ha firmato il documento è il vero autore. Difatti, in crittografia simmetrica, non è

possibile dedurre l'autore del messaggio a partire dal tag, perché chiunque potrebbe produrlo. Nel caso della digital signature, vale invece la seguente proprietà

### Definizione 9.2 (Non-Repudiation Property)

*E' un concetto di sicurezza dove viene garantita l'integrità del messaggio che viene trasmesso, fornendo un identificatore per l'autore della firma.*

## 9.2 Key Agreement Algorithms

Il problema fondamentale di un meccanismo a chiave asimmetrica è che è necessario sviluppare un modo per condividere le chiavi ed instaurare il processo di *Key Agreement*, ovvero un punto nel protocollo nel quale vengono fissate le chiavi da usare per scambiare i messaggi.

Supponiamo di suddividere la chiave  $K$  in due parti  $k_1, k_2$ , allora il problema di ricostruire la chiave è facile se le due parti sono note, ma è difficile data la chiave capire quali sono le parti che la costituiscono. Gli algoritmi principali di key agreement sono due: RSA e Diffie-Hellman.

Un attaccante che vuole trovare la chiave in un sistema che usa questi algoritmi è di fronte a due problemi:

### Proposizione 9.3 (Discrete Logarithm Problem in Prime Field)

*Dati tre numeri primi GRANDI  $p, g, x$  è facile calcolare:  $y = g^x \pmod p$ .*

*Dato  $y = g^x \pmod p$  è difficile calcolare:  $x = \log_g(y) \pmod p$*

### Proposizione 9.4 (Product Factorization of Two Large Prime Numbers)

*Dati due numeri primi GRANDI  $p, q$  è facile calcolare:  $N = p \cdot q$ .*

*Dato  $N$  è difficile trovare la fattorizzazione in numeri primi di  $p$  e  $q$ .*

**Osservazione:** Per "facile/difficile" intendiamo che esiste o meno un algoritmo per il calcolo **efficiente** di quel valore.

Vediamo perché calcolare  $g^x$  è facile:

La complessità dell'algoritmo è ovviamente  $O(\log_2(x)) = O(n)$  per i quadrati e  $O(\log_2(x)/2) = O(n/2)$  per le moltiplicazioni, con  $n$  numero di bit necessari a costruire l'esponente.

L'algoritmo rappresenta un metodo veloce per calcolare un qualsiasi esponenziale, ma lavorando con il modulo, abbiamo una sovrapposizione notevole di valori per il quale rende difficile il problema del calcolo inverso.

Vediamo un esempio grafico del perché è difficile calcolare l'inversa di un esponenziale in modulo, graficando i punti di  $y = 3^x \pmod{104729}$ , con  $x \in \{0, 3000\}$

Vediamo i due principali algoritmi di key-agreement.

**Algorithm 1** Square and Multiply

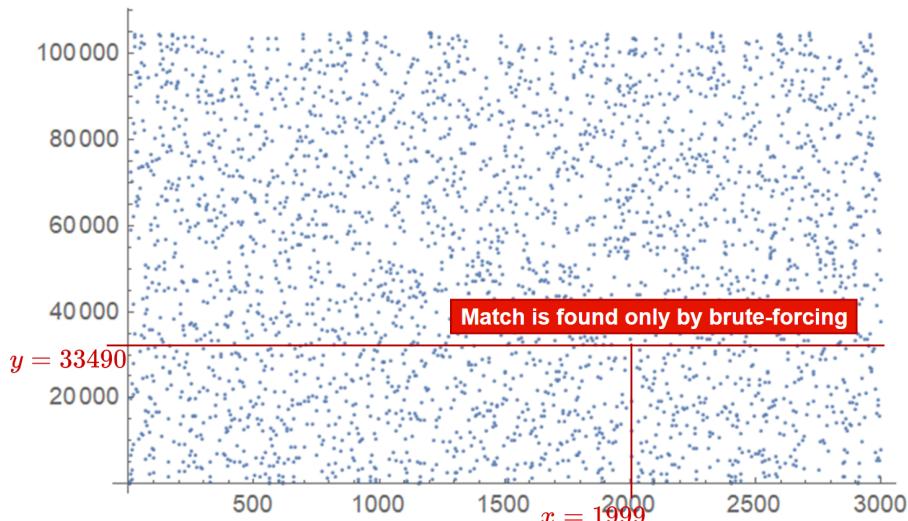
---

```

1: procedure EXP_BY_SQUARING( $g, x$ )                                ▷ Fast compute  $g^x$ 
2:   if  $x = 0$  then return 1
3:   else if  $x = 1$  then return  $g$ 
4:   else
5:      $exp \leftarrow x_2[1 :]$                                          ▷ Take exponential in base 2 and cut the msb
6:      $r \leftarrow g$ 
7:     for all  $b$  in  $exp$  do                                     ▷ Loop from 2nd msb to the lsb of  $exp$ 
8:        $r \leftarrow r^2$                                          ▷ For every bit, square  $g$ 
9:       if  $b[i] == 1$  then                               ▷ If current bit is 1, multiply by  $g$ 
10:       $r \leftarrow r \times g$ 
11:    end if
12:   end for
13: end if
14: end procedure

```

---



**Figura 9.3:** Descrete Logarithm Problem: How to invert the exponential?

## 9.3 Diffie Hellman

Lo scopo dell'algoritmo è **esclusivamente** lo scambio di chiavi. L'idea si basa sul calcolo dell'esponenziale in modulo dei numeri primi. Questo permette di rendere arduo il calcolo inverso. L'algoritmo lavora al seguente modo:

### Definizione 9.3 ((Vanilla) Diffie-Hellman)

Consideriamo due utenti A, che genera un rnd  $x$ , B che genera un rnd  $y$ . Queste sono le chiavi private degli utenti.

Assumiamo che vengano precedentemente scambiati i parametri  $g, p^a$ , rispettivamente base dell'esponenziale e numero primo GRANDE.

1. A invia  $g^x \bmod p$
2. B invia  $g^x \bmod p$
3. A calcola  $K = (g^y)^x \bmod p$
4. B calcola  $K = (g^x)^y \bmod p$

Adesso i due utenti hanno le chiavi necessarie.

<sup>a</sup>In realtà  $p$  deve avere delle proprietà particolare che vedremo poi



**Osservazione:** Nonostante è vero che un attaccante che intercetta i due messaggi non può calcolare  $g^{xy}$  con facilità, vedremo che non bisogna MAI usare Diffie-Hellman così come è fatto in quanto un MITM è dietro l'angolo.



**Nota** DH NON SUPPORTA la firma digitale e lo scambio di messaggi.

#### 9.3.1 Implementazioni di DH

L'algoritmo di Key Agreement di DH è stato implementato in tre modi. Nella sua versione **Vanilla** viene detto anche **Anonymous DH**, perché i valori  $x$  e  $y$  sono generati dinamicamente ma non autenticati. Tuttavia questo metodo risulta poco sicuro a fronte di attacchi MITM come possiamo vedere in fig. 9.4

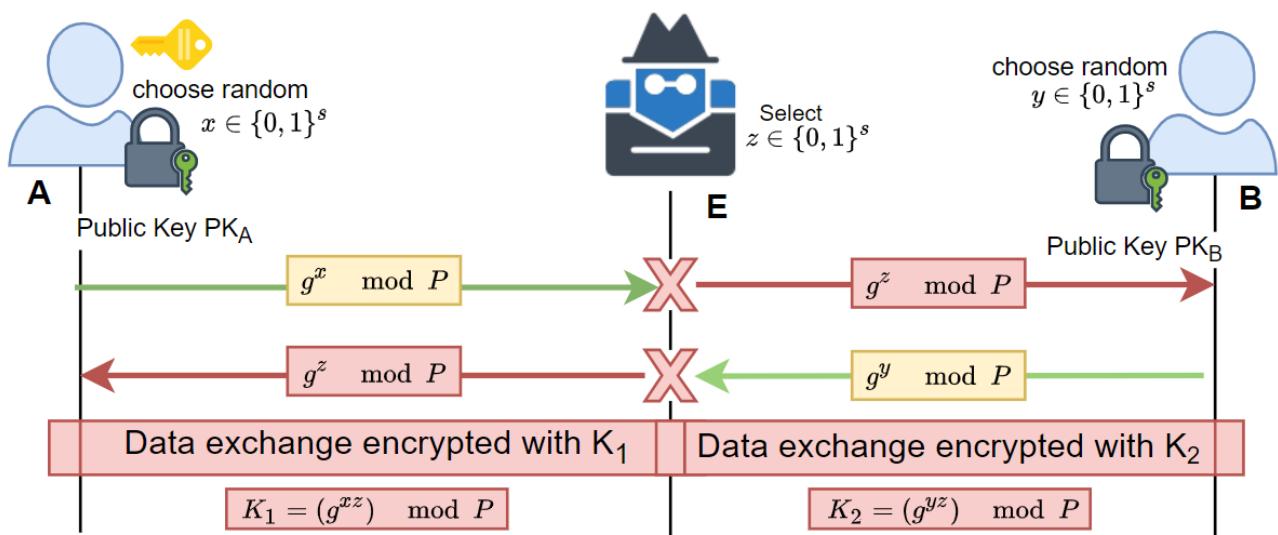


Figura 9.4: MITM in DH Key Agreement

Se i messaggi fossero stati autenticati tramite firma, questo non sarebbe successo. Per sopperire a questa vulnerabilità abbiamo due implementazioni aggiuntive:

**Definizione 9.4 (Fixed DH)**

A genera un numero  $x$ , B un numero  $y$ . Vengono allora calcolati rispettivamente  $g^x \pmod{P}$  e  $g^y \pmod{P}$ .

I due coefficienti pubblici vengono **mantenuti statici e firmate** da una Certification Authority (def 9.10). A questo punto si procede con il key agreement in versione standard.

**Corollario 9.1 (Vulnerabilities of Fixed DH)**

Il segreto  $g^{xy} \pmod{P}$  è ora sempre lo stesso e hardcoded nel certificato, pertanto andrà **sempre usato quello** per le future connessioni fino alla sua scadenza (sezione 9.5.1). Questo genera problemi nel caso di attacchi particolari, che sfruttano una mole di dati che si estende per un lasso temporale molto ampio (anche anni).

**Conseguenza:** Esposizione a *dictionary attacks*.

**Definizione 9.5 (Ephemeral DH)**

A genera un numero  $x$ , B un numero  $y$ . Vengono allora calcolati rispettivamente  $g^x \pmod{P}$  e  $g^y \pmod{P}$  e auto-firmati da loro stessi. Durante il processo di scambio insieme ai coefficienti pubblici vengono inviati anche i rispettivi certificati, firmati da una CA. Vengono scambiati quindi i seguenti pacchetti:

<b>A invia</b>	<b>B invia</b>
$(A, g^x)_{A\_sign}$	$(A, g^y)_{B\_sign}$
$(A, PK_A)_{CA\_sign}$	$(B, PK_B)_{CA\_sign}$

**Osservazione:** Questa variante permette di cambiare il coefficiente pubblico a piacere, in quanto l'identità delle entità coinvolte sarà certificata dalla CA

**Corollario 9.2 (Vulnerabilities of Ephemeral DH)**

Gli attacchi MITM sono **bloccati** dal fatto che i coefficienti pubblici sono firmati da una CA e il coefficiente può essere generato localmente quante volte si vuole.

## 9.4 RSA

Algoritmo sviluppato da Rivest, Shamir, Adleman nel '77 e brevettato fino al 2000, RSA è tra gli algoritmi di cifratura più famosi al mondo. Si basa sul problema della fattorizzazione di numeri primi (prop 9.4) e sulla **modular exponentiation** per cifrare e decifrare i messaggi.

**Proposizione 9.5 (RSA Encryption)**

Dato un messaggio  $m$ , una **chiave pubblica**  $K_{pub}$  e un numero primo grande  $N$

$$C = (m)^{K_{pub}} \pmod{N} \quad (9.2)$$

**Proposizione 9.6 (RSA Encryption)**

Dato un messaggio cifrato  $C$ , una **chiave privata**  $K_{sec}$  e un numero primo grande  $N$

$$m = (C)^{K_{sec}} \mod N \quad (9.3)$$

Vediamo però un esempio:

**Esempio 9.1** Poiché per costruzione l'algebra modulare comporta l'introduzione di un **periodo** lungo quanto il numero rispetto cui viene calcolato,  $m^x \mod N$  è periodica e abbiamo un problema di **ciclicità**.

$$3^x \mod 10 = 3, 9, 7, 1, 3, 9, 7, 1, \dots$$

$$7^x \mod 10 = 7, 9, 3, 1, 7, 9, 3, 1, \dots$$

$$9^x \mod 10 = 9, 1, 9, 1, 9, 1, \dots$$

**Osservazione:** RSA supporta sia firma digitale che la cifratura.

Il teorema di *Eulero-Fermat* permette di calcolare il **periodo** dell'esponente modulare:

**Proposizione 9.7 (Euler's Totient Function)**

Sia  $\Phi(N) = E$  il massimo periodo. Se  $m, N$  sono coprimi ( $\text{GCD}(m, N) = 1$ ), allora

$$m^{\Phi(N)} \mod N = 1 \quad (9.4)$$

Calcoliamo la funzione eq. (9.4) nel caso di RSA.

- Sia  $p$  un numero primo, allora  $\Phi(p) = p - 1$  per costruzione.
- Siano  $p, q$  primi. Allora

$$\Phi(p \cdot q) = \Phi(p) \cdot \Phi(q) = (p - 1)(q - 1)$$

- Sia  $p^k$  un numero primo, allora

$$\Phi(p^k) = \Phi(p) \cdot p^{k-1} = (p - 1) \cdot p^{k-1}$$

- Generalmente, se  $N = \prod_{i=1}^z p_i^{k_i}$ , allora

$$\Phi(N) = \prod_{i=1}^z (p_i - 1) p_i^{k_i - 1}$$

Il caso di RSA è ovviamente  $\Phi(p \cdot q)$ . La conseguenza della periodicità in RSA è dimostrabile con un esempio:

**Esempio 9.2** Consideriamo  $2^x \mod 11 = \{2, 4, 8, 5, 10, 9, 7, 3, 6, 1\}$ . Poiché  $p = 11$  è primo, allora  $\Phi(11) = 10$ . Supponiamo di calcolare  $9 \cdot 7 \mod 11$ .

- **Moltiplicazioni mod11:**  $9 \cdot 7 \mod 11 = 63 \mod 11 = 8$

- **Approccio alternativo:** Osserviamo che  $9 = 2^6 \mod 11$  mentre  $7 = 2^7 \mod 11$ . Allora:

$$9 \cdot 7 \mod 11 = 2^6 \cdot 2^7 \mod 11 = 2^{13} \mod 11 = 2^{10+3} \mod 11 = 1 \cdot 2^3 \mod 11 = 8$$



**Nota** La conseguenza quindi è che se dobbiamo calcolare l'esponente ad una potenza elevata qualsiasi, possiamo calcolare direttamente  $g^x \mod \Phi(N) \mod N$ , invece di calcolare il vero esponente.

### 9.4.1 RSA Key Transport

Vediamo ora come l'algoritmo costruisce le sue chiavi.

#### Definizione 9.6 (RSA Key Generation)

1. *Generare due numeri primi grandi:  $p, q$  che devono restare segreti*
2. *Calcolare l'RSA module:  $N = p \cdot q$ . Questo numero è reso pubblico.*
3. *Calcolare  $\Phi(N) = (p - 1)(q - 1)$ , che deve restare segreto.*
4. *Generare una chiave pubblica e che sia coprima con  $\Phi(N)$ . Ovvero:*

$$1 < e < \Phi(N)$$

5. *Generare una chiave privata  $d$  tale che*

$$\begin{aligned} e \cdot d &= 1 \pmod{\Phi(N)} \\ d &= e^{-1} \pmod{\Phi(N)} \end{aligned}$$

**Osservazione:** Generare una private-key in questo modo è possibile se solo se  $e$  è invertibile in  $\pmod{\Phi(N)}$ , per questo serve che siano coprimi. Se  $\Phi(N)$  è segreto, è ovviamente difficile calcolare  $d$  partendo da  $e$ .

Vediamo l'effetto di questa generazione delle chiavi:

$$\begin{aligned} C &= M^e \pmod{N} \\ M &= C^d \pmod{N} = (M^e)^d \pmod{N} = M^{ed} \pmod{N} = M^1 \pmod{N} = M \end{aligned}$$

#### Definizione 9.7 (Algoritmo di Euclide Esteso)

Consideriamo l'**identità di Bézout**: Dati due interi  $a, b$ , allora  $\exists x, y$  coefficienti tale che

$$ax + by = MCD(a, b)$$

Per trovare  $x, y$ , costruiamo la seguente tabella:

$x$	$y$	$r$	$q$
1	0	$\max\{a, b\}$	-
0	1	$\min\{a, b\}$	$a/b$
...	...	...	...
$x_{i-2} - q_{i-1} \cdot x_{i-1}$	$y_{i-2} - q_{i-1} \cdot y_{i-1}$	$rest(q_{i-2}/q_{i-1})$	$q_{i-2}/q_{i-1}$
...	...	...	...
$x^*$	$y^*$	1	-

I coefficienti cercati sono nell'ultima riga della tabella, che conterrà tante righe quante sono necessarie a far apparire 1 nella colonna delle  $q$ .

**Esempio 9.3 Why RSA works?** Supponiamo di essere in uno scenario di *Decryption Challenge*: Dati  $N, e$  e il messaggio cifrato  $m^e$ , calcolare la chiave di decifrazione (quella privata)  $x$  tale che

$$(m^e)^x \pmod{N} = m$$

Normalmente sarebbe un problema difficile risolvere il problema perché l'unico modo è fare brute

force sui possibili valori di  $x$ . **Se conosciamo**  $\Phi(N)$  allora in tempo polinomiale è possibile trovare la soluzione della seguente equazione:

$$x = e^{-1} \pmod{\Phi(N)}$$



**Esempio 9.4 Compute RSA Inverse** Consideriamo uno scenario dove  $e = K_{pub} = 13$  e  $N = 77 = 11 \times 7$ . Allora  $\Phi(77) = 10 \times 6 = 60$ . Osserviamo che  $MCD(e, \Phi) = 1$  e quindi sono coprimi. Vogliamo calcolare la chiave privata  $d$ .

Innanzitutto calcoliamo i coefficienti  $a, b$  tali che  $\Phi a + eb = 1$

x	y	r	q
1	0	60	-
0	1	13	4
1	-4	8	1
-1	5	5	1
-2	-9	3	1
-3	14	2	1
5	-23	1	-

Allora:  $60 \times 5 + 13 \times (-23) = 1$ . Per cui:  $13^{-1} \pmod{60} = -23$  e quindi  $d = 60 - 23 = 37$ . ■

**Esempio 9.5 RSA toy example** Supponiamo di applicare RSA, con  $p = 11, q = 17, N = 11 \times 17 = 187$ . Abbiamo quindi la fattorizzazione in numeri primi e il numero su cui calcolare il modulo. Allora:

$$\Phi = (p-1) \times (q-1) = 10 \times 16 = 160$$

Scegliamo la chiave pubblica nell'intervallo  $\{1, 160\}$ :  $e = 7$ . Allora:

$$d = 7^{-1} \pmod{160} = 23$$

**Osservazione:**  $23 \times 7 \equiv 161 = 160 + 1 = 1 \pmod{160}$  ■

Abbiamo allora:

Enc:	$C = M^7 \pmod{187}$
Dec:	$M = (M^7)^{23} \pmod{187} = M^{7 \times 23} \pmod{187} = M$
Digital Sign:	$TAG = H(M^{23}) \pmod{187}$
Verify Sign:	$H(M) = (TAG)^7 \pmod{187} = (H(M^{23}))^7 \pmod{187}$

Per verificare l'integrità, basta controllare che l'hash calcolato su  $M$  sia quello dedotto dalla firma. ■

## 9.4.2 Bleichenbacher's Oracle

### Definizione 9.8 (Non-Malleability)

*Diremo che un protocollo **non è malleabile** se, dato un messaggio cifrato  $C$  di un plaintext  $M$ , un attaccante **NON** è in grado di creare un ciphertext differente  $C'$  la cui decriptazione porta ad un messaggio  $M'$  che è **in qualche modo legato** ad  $M$*



RSA presenta un problema fondamentale, è **malleabile**. il protocollo di key-transport di RSA (def 9.6) è vulnerabile al Bleichenbacher Oracle, ovvero un attacco **Adaptive CCA**, che sfrutta la costruzione del padding in RSA, la sua **malleabilità** ed eventuali messaggi di error signaling della *specifica implementazione* del protocollo, al fine di **ricostruire il plaintext**.

### Definizione 9.9 (Bleichenbacher's Oracle)

*Supponiamo che l'implementazione di RSA fornisca un modo per determinare il primo bit del plaintext  $M$ , generato dalla decrittazione all'arrivo di un ciphertext arbitrario<sup>1</sup>.*

L'attaccante può effettuare il seguente CCA:

1. Sia  $C = M^e \pmod{N}$  il **ciphertext obiettivo**.
2. Sia  $r = 1$  e calcoliamo

$$\begin{aligned} C' &= ((C \pmod{N}) \cdot (r^e \pmod{N})) \pmod{N} = \\ &= (C \cdot r^e) \pmod{N} \equiv (M \cdot r)^e \pmod{N} \end{aligned}$$

3. Inviamo al sistema  $C'$  per farlo decifrare. Per l'assunzione iniziale otteniamo il primo bit di  $(M \cdot r)$ .<sup>2</sup>
4. Moltiplicando per  $r^{-1}$  troviamo il bit di  $M$ .
5. Ponendo  $r = r \cdot 2$  possiamo ripetere il processo e scoprire tutto  $M$

<sup>1</sup>Stiamo chiedendo che esista un **oracolo** di **decrittazione** solamente per il primo bit.

<sup>2</sup>In particolare il primo bit di  $M$  coincide con quello di  $M \cdot r$  se shiftato a sinistra di  $\log_2(\#bits_r)$



**Nota** Con un bleichenbacher oracle servono al più  $\log_2(n)$  tentativi per decifrare  $M$ . con  $n = \log_2(M) \text{ bits}$



## 9.5 Pub-Key Infrastructure

Consideriamo uno scenario di key agreement (scambio di chiavi pubbliche e setup di chiavi private comuni per creare un canale sicuro). Fino ad ora non ci siamo preoccupati di verificare la validità della chiave pubblica rilasciata da un utente e l'abbiamo solo usata per decifrare il tag aggiunto al messaggio per ottenere l'hash di  $M$  nel caso della firma digitale, oppure per stabilire una chiave comune per la cifratura del canale di comunicazione con RSA.



**Nota** Se un attaccante rispondesse alla richiesta di un utente di scaricare una chiave pubblica e la sostituisse con la propria, allora ogni messaggio inviato da quel momento in poi dall'attaccante sarebbe considerato valido perché la public key è stata sostituita.

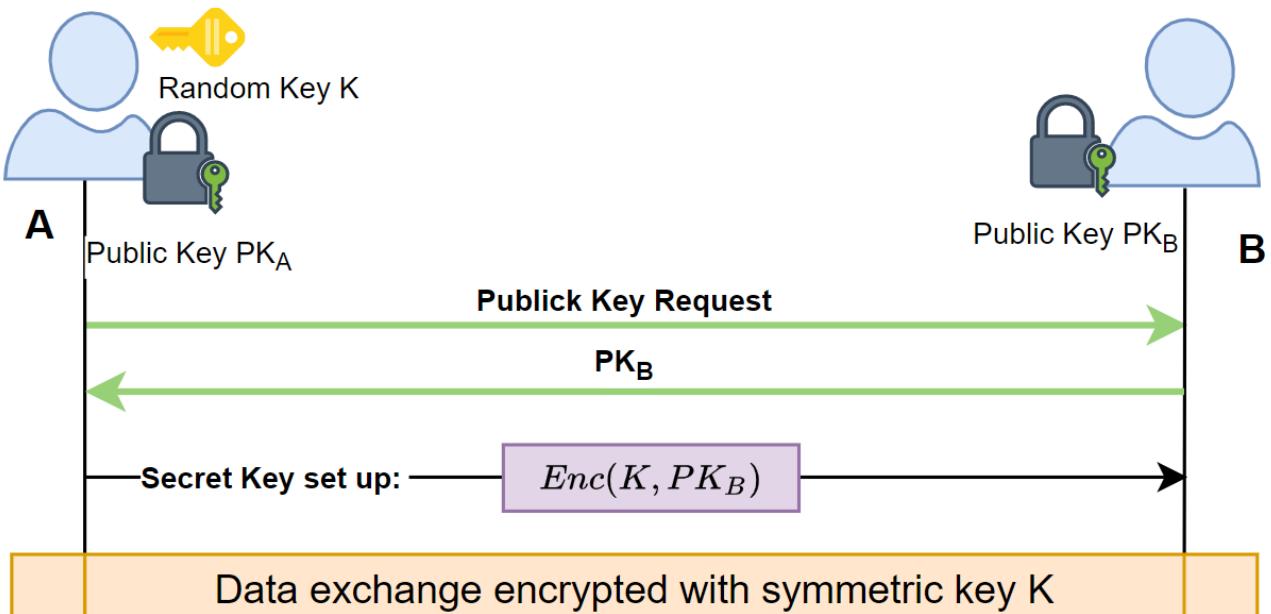
Vediamo lo schema di un possibile attacco ad RSA:

**Osservazione:** Diffie-Hellman non avrebbe comunque protetto da questo tipo di MITM, vedi fig. 9.4



### Proposizione 9.8

*Per scambiare le chiavi è necessario legare in modo crittografico la chiave pubblica all'identità.*

**Figura 9.5:** RSA Key Transport

Abbiamo bisogno di un'infrastruttura che, oltre a definire lo scambio di chiavi, definisca **protocolli, strategie e meccanismi** affinché **lo scambio sia sicuro**

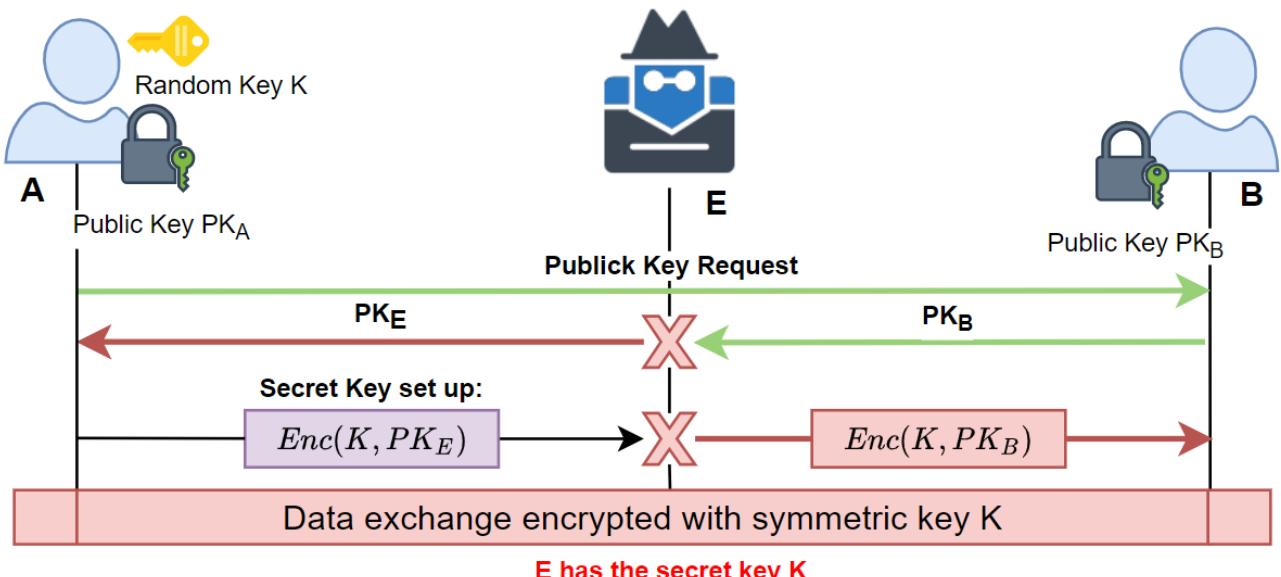


Figura 9.6: MITM in RSA Key Transport

### 9.5.1 Digital Certificate

Un modo per essere sicuri della chiave pubblica rilasciata da un utente è quella di **fidarsi di un'entità terza**, detta **Certification Authority**, che garantisce l'integrità del legame che c'è tra la chiave e l'identità.

#### Definizione 9.10 (Certification Authority)

*Una CA è una Trusted Third Party a cui altre entità si affidano per ricevere dei certificati di affidabilità.*

**Osservazione:** Una soluzione più semplice sarebbe quella di fare in modo che tutti conoscano le chiavi pubbliche degli utenti con i quali si vuole comunicare, ma questa soluzione **non è ovviamente scalabile**.



**Nota** I certificati digitali non risolvono il problema, perché potrebbe esserci un modo di falsificarli.

Il modo con cui una CA rilascia un certificato è il seguente:

#### Proposizione 9.9 (Emissione di un Certificato)

*Un'entità verifica (e.g. offline), presso una CA, la sua identità. Vengono svolte le seguenti operazioni:*

1. L'entità genera una  $PK$  e una  $S_K$ , che viene **mantenuta segreta**.
2. L'entità comunica la sua  $PK$  alla CA tramite un **canale sicuro**
3. La CA fa le verifiche del caso e se passano tutte emette un **certificato firmato**:

$$cert = (name_{entity}, PK)_{CA\_sign}$$

Il modo con cui un utente si accerta che un certificato è autentico è il seguente:

### Definizione 9.11 (Verifica della Validità di un Certificato)

Un'entità *A* vuole accertarsi che il **certificato** di *B* sia corretto e di sua proprietà. Vengono svolte le seguenti operazioni:

1. A contatta *B* e ne **richiede il certificato**.
2. *B* invia ad *A* la tripla:  $\{name_B, PK_B, CERT_B\}$ , dove

$$CERT_B = (name_B.PK_B)_{CA\_sign}$$

3. A verifica che  $CERT_B$  sia stato emesso da una **CA fidata** per lui e che la **firma sul certificato** sia corretta.
4. A si accerta dell'**identità** di *B* con una procedura "**chap-like**" (fig. 4.2) in uno dei modi seguenti:
  - (a). A invia una nonce a *B*
  - (b). *B* risponde ad *A* firmando la **nonce** con la sua **chiave segreta**  $SK_B$ .
  - (a). A invia una challenge contenente una cifratura a *B*

$$Chall = Enc(nonce)_{PK_B}$$

- (b). *B* risponde ad *A* con la nonce ottenuta decifrando la challenge con la sua  $SK_B$

$$nonce = Dec(Chall)_{SK_B}$$

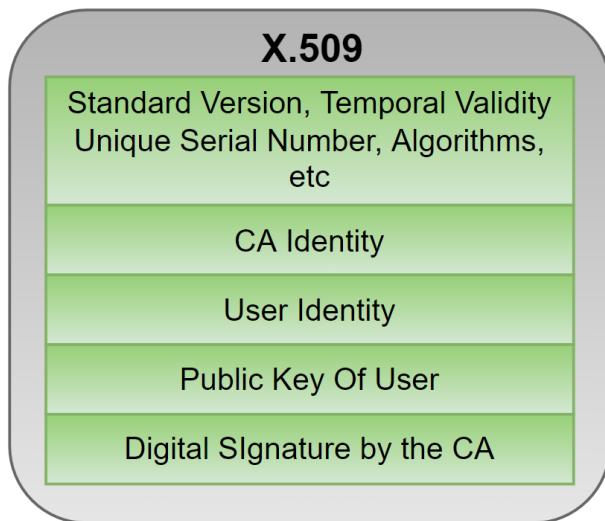


**Nota** Il metodo chap-like per verificare l'identità dell'entità con cui vogliamo collegarci è **necessario** per evitare replay attack sul certificato.

Le CA sono molte e dislocate in tutto il mondo, organizzate poi in una struttura gerarchica. Un'architettura PKI definisce proprio queste relazioni e anche gli standard a cui i certificati devono aderire. Un esempio di certificato è il seguente:

### Esempio 9.6 Certificate X.509 (High Level View)

- **Versione Standard:** Indica la versione del certificato.
- **Validità Temporale:** Data di inizio e fine validità.
- **Numero Seriale:** Identificatore univoco del certificato.
- **Algoritmi:** Algoritmi crittografici supportati dal certificato. Detta anche **Cipher Suite**
- **Identità CA:** Nome della CA che ha emesso il certificato.
- **Identità Utente:** nome dell'utente associato al certificato.
- **Chiave Pubblica Utente:** chiave pubblica dell'utente.
- **Firma Digitale:** firma del certificato generata dalla CA.



**Figura 9.7:** Certificate X.509

- **Estensioni:** (se X.509v3) campo addizionale che può contenere diversi tipi di attributi e permette di espandere il certificato con nuove feature.

### 9.5.2 Certificate Revocation List

Ogni PKI **dove** includere un meccanismo di *revoca* dei certificati, poiché questi possono essere **compromessi**. Ad esempio:

- La chiave privata è stata persa o divulgata.
- La chiave pubblica è obsoleta e non più utilizzata.
- Il certificato è scaduto.

Esistono 2 approcci per la revoca di un certificato:

- **Revocazione Implicita:** avviene se il **periodo di validità** di un certificato si esaurisce.
- **Revocazione Esplicita:** avviene quando effettuata per *Certificate Revocation List*

#### Definizione 9.12 (Certificate Revocation List)

Una CRL è una lista mantenuta da **OGNI CA**, che periodicamente **dove** essere **aggiornata** con i certificati che la CA ha **rimosso**.

Possiede il seguente formato:

- **Emitente (Issuer):** Indica la CA che ha **pubblicato** la lista.
- **Ultima Data di Aggiornamento:** ultima data in cui la lista è stata modificata.
- **Prossima Data di Aggiornamento:** data futura in cui verrà pubblicato il prossimo aggiornamento. Indica il periodo di validità della lista.
- **Elenco dei Seriali Revocati:** indica i numeri seriali dei certificati da revocare.
- **Elenco della Data di Revocazione.**
- **Firma Digitale:** firma generata dalla CA emittente della lista.



**Nota** Ogni entità che utilizza un certificato **dovrebbe sempre verificarne la validità** controllando la relativa CRL. Tuttavia, questa operazione è demandata ad applicazioni che svolgono un ruolo **critico** nella rete, per alleggerire il carico complessivo.

### 9.5.3 Public Key Criptography Standards

La PKI specifica molti aspetti legati alla **crittografica asimmetrica**. Gli standard sono definiti dalle:

#### Definizione 9.13 (PKCS)

*Public Key Criptography Standards*

Attualmente vanno dal numero 1 al numero 15, alcuni di questi sono:

- **PKCS#1:** RSA
- **PKCS#3:** DH
- **PKCS#6:** Estensione del formato **X.509**. Alcune delle estensioni più comuni sono:

#### Proposizione 9.10 (X.509 Extensions)

- **Authority Key Identifier:** identifica la chiave pubblica corrispondente alla chiave privata usata per firmare un certificato.
- **Subject Key Identifier:** Identifica i certificati che contengono al loro interno una determinata chiave pubblica.
- **Key Usage:** Specifica lo scopo della chiave contenuta nel certificato. Ovvero, per quale meccanismo può essere utilizzata.
- **Subject Alternative Name:** permette di associare più identità al soggetto del certificato, che possono aggiungersi o rimpiazzare le precedenti.
- **Extended Key Usage:** Specifica ulteriori scopi per la chiave pubblica contenuta nel certificato. che possono aggiungersi o rimpiazzare quello specificato nell'estensione precedente.

- **PKCS#10: CRSS (Certification Request Syntax Standard).** Specifica il formato dei messaggi di richiesta dei certificati. Un esempio è il seguente:

#### Proposizione 9.11 (Certification Signing Request - CSR)

*Messaggio inviato da parte di una CA al fine di applicarsi per un certificato della sua identità digitale. Le operazioni svolte dall'applicante sono:*

1. Genera una coppia di chiavi, pubblica e privata. L'ultima viene mantenuta segreta.
2. Genera un **CSR**, contenente le informazioni per identificarsi: Nome, Chiave pubblica e, eventualmente, alcune estensioni.
3. Il **CSR** viene **invia**to insieme ad altre credenziali o prove di identità richieste dalla CA. Inoltre, quest'ultima può contattare l'applicante per richiedere ulteriori informazioni.

- **PKCS#11: CTIS (Cryptographic Token Interface Standard).** Specifica l'API per **firmare e verificare** dati da parte di un dispositivo che mantiene chiavi.
- **PKCS#12: PIESS (Personal Information Exchange Syntax Standard).** Specifica il **formato** dei file utilizzati per **memorizzare i certificati** e le **chiavi private** usati per spostare informazioni

riservate tra diversi browsers.

- **PKCS#13: ECC (Elliptic Curve Cryptography)**

### 9.5.4 Certificate Chains

Per distribuire le responsabilità e il carico delle certificazioni digitali, le CA vengono **organizzate** secondo una **gerarchia ad albero** dove i nodi radice sono considerati, per definizione, i più sicuri. L'ovvio problema di questa struttura gerarchica è che, eccezion fatta per la radice, **non abbiamo garanzie** che un particolare nodo dell'albero sia **trusted**, ovvero che a sua volta la sua identità sia verificata. Sono necessari dei meccanismi aggiuntivi, detti proprio **certificate chains**.

**Osservazione:** I meccanismi aggiuntivi sono **necessari** in quanto su una singola macchina non è possibile salvare tutti i certificati di tutto il mondo, serve qualcosa di scalabile, come un database distribuito. □

#### Definizione 9.14 (Certificate Chain)

*Una catena è una lista di certificati seguiti da una o più CA, con le seguenti proprietà:*

- *L'entità che rilascia il certificato è collegata alla prossima in lista.*
- *Ogni certificato, a parte l'ultimo, è firmato dalla chiave segreta corrispondente al prossimo certificato nella catena.*
- *L'ultimo certificato in lista è un **trust anchor**, del quale l'utente si fida perché gli è stato fornito tramite una procedura attendibile. Dalle trust anchor inizia il processo di validazione.*

Una delle estensioni che è possibile trovare nel certificato è la seguente:

#### Teorema 9.1 (Basic Constraint)

*E' un valore booleano, che indica se l'entità legata a al certificato ha o meno il permesso di utilizzare la propria chiave privata per firmare altri certificati.* □

**Osservazione:** Permette di individuare un certificato illegittimo nella catena. □

 **Nota** Una compagnia può avere un cosiddetto **Wildcard Certificate**. E' un certificato jolly, che permette di certificare tutti i suoi sistemi, con nome diverso, in uno stesso dominio.

Il meccanismo di certificazione è simile a quello espresso in def 9.11, ma questa volta c'è almeno un nodo intermedio in più.

#### Definizione 9.15 (Certificate Chain Procedure)

*Supponiamo che A voglia autenticare B. Supponiamo che  $CA_B$  sia la certificante di B e che U sia la root della catena di certificazione di B ma che sia nella lista delle CA **universalmente verificate** di A.*

1. *B invia ad A il suo certificato che A controllerà per individuare la validità e, soprattutto, la CA che lo firma. Abbiamo due casi:*
  - **La CA di B è nella lista di A:** B è certificata.
  - **La CA di B NON è nella lista di A:** Servono ulteriori certificati.
2. *B richiede alla sua CA il suo certificato, che inoltra ad A.*
3. *A vede che il certificato di  $CA_B$  è del tipo:*

$$(CA_B, PK_{CA_B})_{U_{sign}}$$

*E poiché U è nella sua tabella e universalmente verificata, B è verificata.* □



**Nota** Tipicamente non va sempre così: sono necessari diversi certificati per risalire fino ad una CA effettivamente nella lista di A. In quel caso vengono inviati ad A tutti i certificati intermedi, per poter ricostruire la catena.

#### 9.5.5 Merkel's Tree

Consideriamo per un momento il problema di garantire l'integrità di un file.

#### Proposizione 9.12 (Best Practices for Integrity)

*Per garantire l'integrità di un file in maniera efficiente occorre provvedere a due aspetti:*

1. **Generare una segnatura (fingerprint) del file  $H(file)$  con una funzione crypto-hash  $H$ .**
2. **Proteggere la segnatura in uno dei seguenti modi:**
  - **Memorizzare la fingerprint in uno spazio di archiviazione sicuro** (una chiavetta, disco rimosso dalla rete ecc).
  - **Generare la firma digitale del file**  $H(file)_K$ .

💡 **Nota** Firmare il file da proteggere o il suo hash è equivalente perché l'hash preserva il contenuto del file da un punto di vista dell'integrità □

Come comportarsi però quando il **file** da proteggere è **troppo grande** e deve essere **scomposto** in diversi **blocchi (chunk)**? Sono chiaramente necessarie due operazioni per garantire quanto affermato in prop 9.12:

- **Verifica Indipendente della segnatura dei singoli chunk.** La verifica indipendente è necessaria poiché i chunks possono essere ricevuti fuori ordine e da mittenti differenti.
- **Verifica della segnatura dell'intero file riassemblato alla fine della ricezione, per impedire attacchi ai singoli chunk sfuggiti al controllo precedente.**

Al fine di applicare queste soluzioni, abbiamo due approcci:

- **Hash dell'Intero File:** si genera un unico digest per il file. Allora:
  - ✗ Serve che **tutti i chunk** sia stati ricevuti per ricostruire il file e poi produrre il digest.
  - ✗ La **verifica** di un **singolo chunk** può essere fatta **solo se si possiedono tutti gli altri**
  - ✗ Un attaccante può **inserire chunk malevoli** durante l'invio. (**fake chunk injection**)
- **Hash di ogni chunk:** per ogni chunk genero un **tag**. Allora:
  - ✗ **Grande spreco** di risorse computazionali e di archiviazione.
  - ✗ **L'intero file non è più verificabile.**
  - ✗ Un attaccante può **rimuovere chunk** dal canale di comunicazione. (**Strip-type attacks**)

Tali problematiche non sussistono se utilizziamo una struttura dati ad-hoc.

#### Definizione 9.16 (Merkle's Hash Trees)

*Un Merkle Tree è un albero binario i cui nodi interni sono costituiti da hash di chunk di un file e le foglie sono i chunk stessi. Per ogni nodo, tranne la radice, identifichiamo dei **siblings**, ovvero l'insieme dei fratelli ad ogni livello superiore al nodo considerato.*

*Questa costruzione permette di creare una fingerprint per singolo chunk che garantisce di:*

- *Verificare l'integrità utilizzando **soltanto i siblings** che si incontrano lungo il cammino dalla foglia alla radice dell'albero.*
- **Osservazione:** *Servono solo i  $\log_2(N)$  tag presenti nei nodi **opposti** ad uno stesso livello dell'albero fino alla root, che viene ricostruita man mano che si sale.* ■
- *Verificare l'integrità dell'intero file **ricalcolando** il tag posto nella **root**, utilizzando ogni chunk.* ■

**Esempio 9.7  $C_2$  Sign verify** Supponiamo di voler verificare l'integrità del chunk 2 in fig. 9.8. L'entità che esegue la verifica si fa inviare dal CTDB **tutti** i siblings necessari insieme al certificato e alla root, in questo caso sono necessari:  $S_1 = H(c_1)$ ,  $S_2 = H(H(c_3), H(c_4))$ .

L'entità calcola allora  $H(H(c_1), H(c_2))$ , con il quale a partire da  $S_2$  può calcolare la **sua versione della root**. Se questa coincide con quella ricevuta dal CTDB allora  $c_2$  è integro. ■

#### Definizione 9.17 (Merkle's Tree Time extension)

*Un merkle tree può essere esteso introducendo il concetto di tempo, usando uno schema ricorsivo nel quale alla root di un albero  $i$ -esimo costruito al tempo  $t$  associamo un'altra root di un albero  $j - esimo$  costruito al tempo  $t+1$  e creiamo una root a livello  $(i-1)$ -esimo contenente l'hash delle due root.* ■

E' immediato dedurre che per verificare l'integrità di un chunk con un albero esteso servono:

#### Proposizione 9.13 (Integrity check wiht Time Extension)

- *I siblings dell'albero a cui appartiene il chunk.*
- *La radice dello slot temporale precedente.*
- *Le radici di tutti gli slot temporali successivi.* ■

**Osservazione:** E' interessante notare come i merkle tree estesi permettano di costruire un sistema di **blockchain** per la verifica di certificati. ■

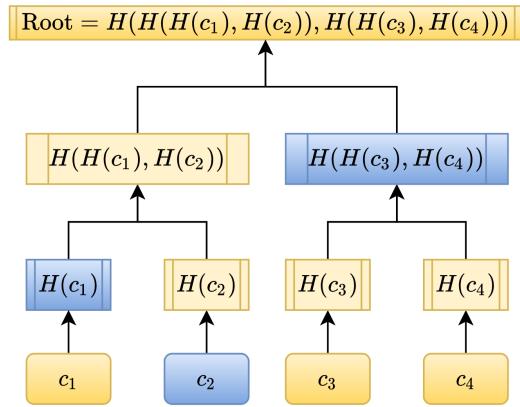
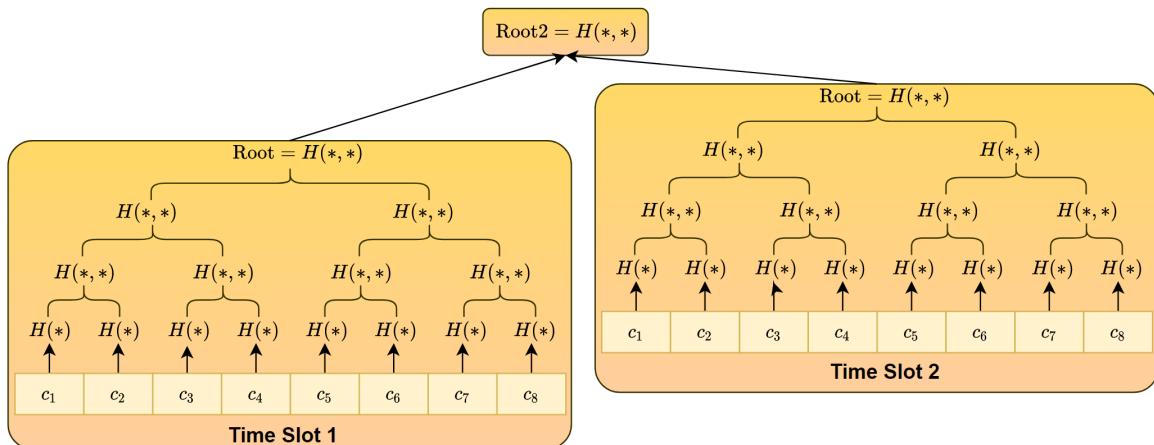
Figura 9.8: Merkle Tree example.  $c_2$ 's siblings

Figura 9.9: Merkle Tree with Time

### 9.5.6 Certificate Transparency

Il processo descritto nella def 9.15 è **debole** ad attacchi, perché se un **nodo** della catena **contiene** un **certificato falso**, allora il **sistema di autenticazione** è stato **rotto** a prescindere. Inoltre, **bisogna assicurarsi che la CA nella nostra lista possa rilasciare certificati**, come detto in thm 9.1.

Il problema descritto precedentemente può essere formalizzato così:

#### Definizione 9.18 (Fake Certificates Problem)

Una **CA intermedia** può generare **certificati malevoli** visti però come **validi** dalle entità, compromettendo i meccanismi di autenticazione e recupero delle chiavi pubbliche nella PKI. □

La soluzione venne proposta da Google nel 2013 e costituisce una mitigazione del problema, perché in ogni caso c'è una debolezza intrinseca alla base dell'infrastruttura.



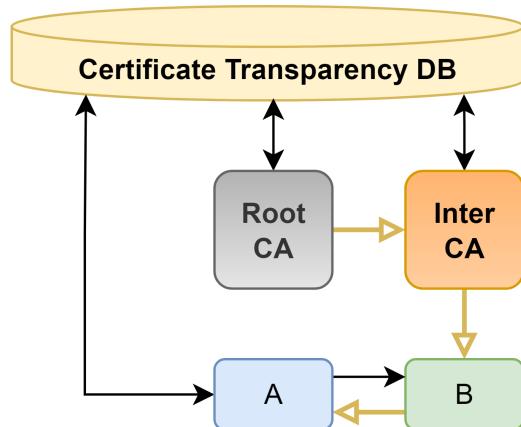
**Nota** Se una delle entità universalmente fidate (es. Governi) dovesse, per qualche motivo, generare certificati malevoli non potremmo accorgerci di niente.

L'idea è quella di creare un DB scalabile globalmente distribuito.

**Definizione 9.19 (CTDB)**

Le caratteristiche del CTDB sono:

- E' **globale, scalabile e sicuro** (non modificabile a posteriori).
- Può essere **amministrato da diversi stakeholders**.
- Tutte le CA possono inserire nel CTDB i certificati da loro emessi.
- Tutte le entità e le CA possono accedere e consultare il CTDB.



**Figura 9.10:** Certificate Transparency DB

Per soddisfare i requisiti funzionali il database è mantenuto in un **log server (unico e globalmente accessibile)**, protetto con uno schema derivante dai **Merkle's Tree esteso**, detto **Chron Tree**, caratterizzato dalle seguenti proprietà:

**Proposizione 9.14 (Chron Tree Properties)**

- Ogni certificato corrisponde ad un **chunk**. Inoltre, ad ognuno di questi è aggiunto un **timestamp** per evitare l'inserimento retroattivo di certificati
- L'albero è di tipo **Append Only**: l'albero viene **esteso ogni 24 ore** con un sottoalbero per la verifica della firma di tutti i certificati emessi dall'ultima estensione.
- **Consistency Proof**: ogni versione successiva del Merkle Tree **contiene per intero** la precedente come sottoalbero. Ciò rende **impossibile** la modifica di un certificato passato o l'inserimento di un **certificato retrodatato**.
- La verifica per un certificato necessita solamente del certificato stesso e dei **siblings-tag** all'interno dell'albero.

Le entità in gioco nel sistema descritto da fig. 9.10 possono eseguire le seguenti operazioni:

**Proposizione 9.15 (What a Requesting Entity can do)**

Prima di utilizzare il certificato di un'entità B, l'entità richiedente A effettua i seguenti controlli:

1. **Controlla se il certificato digitale è affidabile tramite certificate chain e se è valido.**
2. **Autentica B con un meccanismo CHAP-like**: A sottopone a B una prova per la verifica della conoscenza la chiave privata legata alla chiave pubblica nel certificato.
3. **Verifica che il certificato sia presente nel CTDB**: ricostruisce il tag root del sottoalbero del Merkle Tree esteso alla quale appartiene il certificato usando i siblings contenuti nel SCT (ricevuto insieme al certificato) e verifica la congruenza con quello presente nel log server.
4. **Verifica che il certificato non sia presente in nessuna CRL delle CAs coinvolte.**

### Proposizione 9.16 (What a CA can do)

- **Emettere un Certificato e Caricarlo nel CTDB.** In particolare vengono eseguite le seguenti operazioni:
  1. La CA genera un **precertificato temporaneo** che invia al log-server.
  2. Il **log-server genera**, a partire dal precertificato, il relativo **certificato definitivo** e lo invia alla CA, **insieme ad un SCT (Signed Certificate Timestamp)**<sup>3</sup>.
  3. La CA riceve la coppia {**certificato, SCT**} e la inoltra all'entità per la quale è stata generata. Per robustezza, sia CA che l'entità **mantengono localmente** il certificato e l'SCT.
- **Consultare il CRDB**<sup>4</sup> alla ricerca di certificati malevoli emessi a suo nome, così da inserirli nella sua CRL ed ottenerne la revocazione.

<sup>3</sup>l'SCT contiene info aggiuntive necessarie alla verifica della validità del certificato, tra cui i siblings del sottoalbero a cui appartiene.

<sup>4</sup>Certificate Revocation DB



Il sistema di certificate transparency non è completamente privo di vulnerabilità. In particolare una CA intermedi corrotta a favore di un'entità B (non richiedente) potrebbe:

- **Non aggiungere il certificato falso nel CTDB.** Tuttavia, l'entità A nota che il certificato per B non è presente nel CTDB, e lo scarta a priori. **L'attacco è perciò completamente neutralizzato.**
- **Aggiungere un certificato falso/rubato nel CTDB.** Tuttavia, la CA proprietaria dell'identità rubata si accorgerebbe subito del certificato falso inserito nel CTDB e lo aggiunge alla propria CRL.

L'attacco è perciò **mitigato**, in quanto può esistere **una finestra temporale** in cui il **certificato falso è nella CTDB ma non nella CRL**.



**Nota** La certificate transparency è diversa da un sistema a blockchain, sebbene siano entrambe protette da merkle tree. La differenza sta nel fatto che nelle **blockchain** le **transazioni** (i chunk) sono vere se appartengono alla blockchain.

Anche se è vero che un certificato viene considerato falso se non appartiene al CTDB, questo può ancora essere falso in quanto potrebbe esser stato revocato da una CA ed appartenere ad una CRL.

## 9.6 Authentication with Asymmetric Crypto

Nel contesto della **comunicazione sicura basata sulla crittografia asimmetrica e i certificati digitali**, un'entità può autenticarsi dimostrando di conoscere la **chiave privata associata alla chiave pubblica** dichiarata all'interno del suo certificato. Esistono 2 diversi approcci:

- **Firma Digitale:** il sistema autenticato dimostra la conoscenza della chiave privata generando la firma digitale per una challenge composta sia da una nonce che da testo arbitrario.
- **Decrittazione:** Il sistema autenticato dimostra la conoscenza della chiave privata, decrittando una challenge composta sia da una nonce che da un testo arbitrario. La decrittazione può avvenire in due modi:
  - **Asimmetrica:** Il sistema autenticato invia solamente la challenge criptata asimmetricamente.
  - **Simmetrica con chiave di Sessione:** l'autenticatore invia una chiave di sessione casuale, criptata asimmetricamente e la challenge criptata simmetricamente.

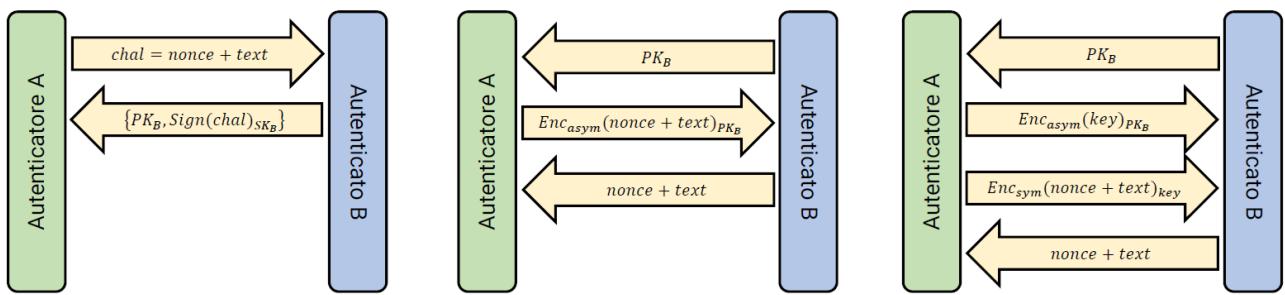


Figura 9.11: Authentication with Asymm Crypto

# Capitolo TLS - Transport Layer Security

I principali *network security protocol* sono **SSH**, **SSL** (*secure socket layer*), **TSL** e **IPsec**. Storicamente il primo pensato fu SSL, realizzato in diverse versioni ognuna sempre migliore, fino a quella che conosciamo oggi. Facciamo un breve recap storico:

- **(1994)** SSL v1 viene sviluppato da *Netscape* ma mai rilasciato.
- **(1995)** SSL v2 viene integrato in *Netscape 1.1* e verrà successivamente rotto.
- **(1996)** SSL v3 viene ridisegnato dalla base da *Netscape*.
- **(1996-1999)** TLS v1.0 viene sviluppato parallelamente a *SSL* dalla **IETF**, partendo dalla sua versione 3.1. Il protocollo è basato interamente su *TCP* per la parte di trasmissione e integra algoritmi di sicurezza **hardcoded**.
- **(2006)** TLS v1.1 viene rilasciato e viene aggiunto il supporto a *UDP*.
- **(2008)** TLS v1.2 viene rilasciato per disaccoppiare i sistemi di sicurezza deboli come *MD5/SHA-1* precedentemente hardcoded nel sistema. Introduce il concetto di *Negotiated-PRF*, impostando come default *SHA-256*.
- **(2018)** **TLS v1.3 viene rilasciato, costituendo un nuovo protocollo.** Vengono introdotti e ammessi solo cifratori basati su *AEAD* e viene aggiunto il *Three-way-Handshake*



**Nota** *TLS v1.3 rimuove anche RSA, in quanto il modo in cui viene usato può portare a diversi problemi.*  
In ultimo è utile ricordare che:

## Proposizione 10.1 (Livelli di Protezione)

*SSL/TLS sono protocolli di protezione che vanno sopra lo strato di trasmissione e sotto quello applicativo. Tuttavia non vanno presi come un miglioramento alla sicurezza di TCP in quanto i servizi offerti da TLS non sono limitati al garantire un trasporto sicuro di dati e descrivono una più generale relazione point-to-point.*

*Questo significa che SSL/TLS proteggono esclusivamente<sup>1</sup> il payload di un determinato pacchetto.*

<sup>1</sup>IPsec protegge invece tutto il pacchetto, sia payload che header.

## 10.1 Il Supporto Applicativo

Quando un servizio applicativo viene standardizzato bisogna specificare una serie di parametri come **IP address** e **port number** del servizio. TLS, avendo una *struttura a strati*, riesce a fornire un supporto ben strutturato a questa modalità di sviluppo poiché l'applicazione utilizza nel suo pacchetto la porta del servizio applicativo da usare (es: HTTP) mentre lo strato TLS incapsula il valore di porta specificando il valore del servizio su cui c'è sicurezza.

**Esempio 10.1** Supponiamo di inviare un pacchetto email tramite protocollo HTTP, che usa la porta 80. Poiché il servizio di mail offerto dal protocollo SMTP usa la porta 25, l'applicazione che utilizziamo per spedire il pacchetto specificherà nell'header la porta 80 e la porta 25. Se la connessione che usiamo è protetta da TLS, prima che il pacchetto venga passato allo strato di trasporto verranno specificate nuove porte, rispettivamente 443 per HTTPS e 587 per SMTPS, le due versioni sicure dei protocolli precedenti. ■

**Osservazione:** Le ragioni dietro questa scelta sono prettamente storiche. All'inizio nessuno usava TLS al di fuori delle comunicazioni di rete e si voleva garantire la scelta di utilizzare un servizio piuttosto che un altro, comportando una duplicazione delle porte per una medesima applicazione. ■

**Nota** IPsec usa un servizio chiamato *traffic flow confidentiality* che impedisce ad una persona intenta ad osservare il flusso dati in trasmissione quale protocollo/applicazione stiamo utilizzando. In TLS questo servizio **NON C'E'**, quindi l'applicazione e il protocollo appaiono in chiaro.

**Nota** Analizziamo da ora in poi TLS v1.2, evidenziando alcune differenze con la v1.3. ■

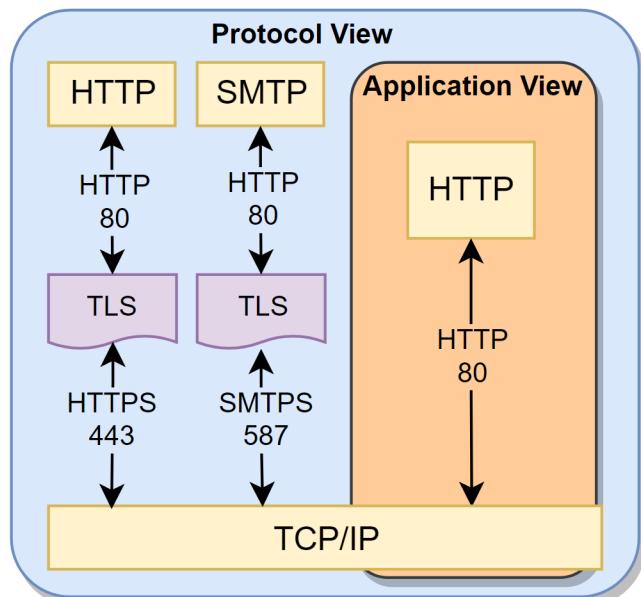


Figura 10.1: Port change made by TLS layer

## 10.2 TLS Protocol Stack

Abbiamo detto che TLS nasce per garantire una connessione sicura. Vediamo quali sono i servizi che dobbiamo garantire per averla e il modo con cui vengono forniti. TLS deve:

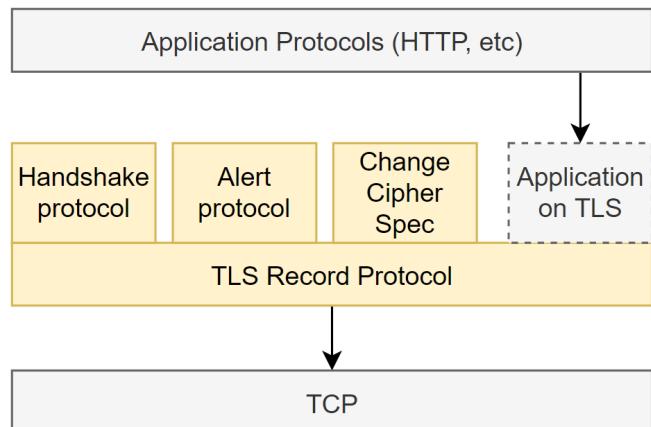
### Proposizione 10.2 (TLS Services)

1. Creare una sessione (*Handshake Phase*)
2. Condividere dei segreti.
3. permettere autenticazione degli utenti.
4. Trasferire dati su un canale sicuro (*Symmetric Encryption*).
5. Garantire integrità dei dati trasmessi (*HMAC*). ■

Il modo con cui TLS fornisce questi servizi è quello di fare due operazioni insieme, per motivi storici, a differenza di altri protocolli come IPsec che distinguono bene la fase di creazione della sessione da quella di trasferimento.

Lo stack di protocollo è quello visibile in fig. 10.2. Abbiamo detto che TLS fa un incapsulamento del pacchetto per rimappare la porta e specificare che si sta usando un servizio protetto; questo avviene nel *TLS Record Protocol*, al di sopra del quale possono essere eseguite applicazioni, l'handshake protocol e si possono gestire anche alert secondo protocolli variabili.

Il blocco relativo alle specifiche di cifratura serve per segnalare che il servizio di cifratura è attivo.



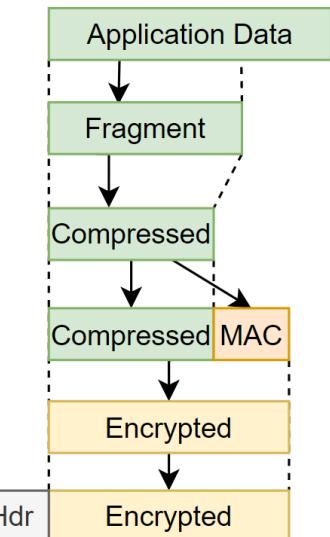
**Figura 10.2:** TLS Protocol Stack

### 10.2.1 TLS Record Protocol

Il **Record Protocol** si occupa di prendere il messaggio trasmesso dall'applicazione per trasformarlo in uno coerente con le specifiche del protocollo. Le operazioni sono le seguenti:

#### Teorema 10.1 (Record Protocol - Send)

- **Recupero Dati:** prelievo del messaggio generato dall'applicazione.
- **Frammentazione:** Divide i messaggi in frammenti di dimensioni gestibili.
- **Compressione (opzionale)**
- **Generazione MAC:** genera il MAC e appendilo in coda.
- **Criptazione:** Cifra il pacchetto e il MAC.
- **Aggiungi Header TCP:** aggiungere l'header con le informazioni protocollo in testa.



**Figura 10.3:** TLS Record Protocol

#### Teorema 10.2 (Record Protocol - Receive)

1. Decifra il Pacchetto.
2. Calcola il MAC e verifica con quello trasmesso.
3. (opzionale) Decomprimi.
4. Riassembra i frammenti.
5. Trasmetti al livello applicativo il messaggio.

Nel Record Protocol è possibile notare il primo grande errore delle versioni antecedenti<sup>2</sup> alla 1.3. Il meccanismo di **compress-than-MAC and encrypt** infatti rendeva possibile un particolare attacco (molto complesso) di cui parleremo più avanti. Analizziamo ora i punti del protocollo:

- La compressione è di tipo *loseless*<sup>3</sup> e viene fatta prima di cifrare, in quanto non avrebbe senso cercare una logica di compressione nel digest (serve un messaggio a bassa entropia).<sup>4</sup>
- Il MAC viene generato con una **HMAC-construction** (def 3.4) tramite chiave simmetrica scambiata durante il set-up dei parametri di sicurezza negoziati in fase di handshake, così come l'hash-function.
- La cifratura è applicata al singolo fragment e al suo MAC (possibilmente compressi) e non sarebbe possibile senza una negoziazione iniziale. L'algoritmo usato è un block o stream cipher (capitolo 7, capitolo 2), anch'esso deciso in fase di negoziazione ma la dimensione del digest non deve superare 1024 bytes.

La parte relativa ai dati trasmessi del Record Protocol (fig. 10.2) è una struttura dati con i seguenti campi:

#### Definizione 10.1 (Record Protocol Data Unit)

- **Header (5 bytes)**, costituito da:
  - **Content Type (1 byte)**: specifica il protocollo a livello superiore da attuare. Valori che può assumere sono: 20 per **Change Cipher spec**, 21 per un messaggio di **Alert**, 22 per la **fase di handshake**, 23 se sono dati applicativi.
  - **Versione (1 byte)**: vengono specificate la minima e massima versione supportata dal sistema.
  - **Lunghezza (2 byte)**.
  - **Ciphertext**, occasionalmente compresso.
  - **MAC**.

**Osservazione:** Poiché TLS si appoggia, generalmente, a TCP non è necessario includere un **SQN** in quanto il protocollo di trasporto garantisce che i pacchetti arrivino in ordine. ■ □



**Nota [Replay Attack]** Come detto nella prop 3.1 un **MAC** non protegge da **replay attack** a meno che questo non venga calcolato *on-the-fly* includendo una nonce. Nel caso di TLS venne usato un **SQN**, **inizializzato a 0 e con massimo**  $2^{64} - 1$ , mantenuto in modo separato agli estremi della connessione. Poiché come detto TLS si appoggia a TCP, per ottimizzare il protocollo non venne specificato un sequence-number ma venne usato direttamente quello di TCP per eseguire il calcolo del MAC senza mai includerlo nel pacchetto trasmesso.



**Nota [Cambiamento in DTLS]** Per includere UDP come protocollo di trasmissione è stato necessario specificare un **sequence di 8 bytes**.

**Osservazione:** Includere un sequence number come nonce permette di individuare dati extra o mancanti e di prevenire replay/reordering attack. ■

<sup>2</sup>Nella v1.0/SSL v3 venne considerata l'ipotesi della compressione, ma non venne applicata.

<sup>3</sup>Senza perdita di informazioni.

<sup>4</sup>Una compressione può essere tipo  $f(AAAA) = 4A$ . In un messaggio (pseudo)randomico non avrebbe senso.

## 10.3 Handshake Protocol

TLS si fonda sul protocollo di Handshake che permette di fare una *negoziazione* iniziale e di scambiare tutti i parametri necessari alla creazione di una **sessione sicura**, oppure per riavviare una sessione precedentemente aperta.

### Proposizione 10.3 (Session Init Steps (overview))

1. Effettuare la **mutua autenticazione** (sezione 4.5) di client e server.
2. Accordarsi sugli **algoritmi** da utilizzare.
3. **Scambio di valori casuali**
4. **Scambio di segreti e/o informazioni** necessarie a derivare le chiavi

### Proposizione 10.4 (Session Restart (overview))

Viene effettuato un **handshake abbreviato** per fare **rekeying**; devono essere **generati nuovi segreti** da cui **derivare nuove chiavi** di sessione.

L'obiettivo della procedura di handshake è quindi quella di fornire un protocollo unico che raccoglie i seguenti servizi:

### Teorema 10.3 (Handshake Protocol's Duties)

- **Negoziazione Sicura di Segreti Condivisi:** per garantire la **confidenzialità dei segreti condivisi** viene usata crittografia asimmetrica.
- **Negoziazione Affidabile:** un'attaccante **NON** deve essere in grado di **modificare/alterare una negoziazione** senza che venga rilevato dal protocollo.
- **Autenticazione Opzionale:** deve essere possibile **scegliere quali delle parti autenticare**.
- **Autenticazione Sicura:** il meccanismo di auth deve essere **robusto a MITM**.

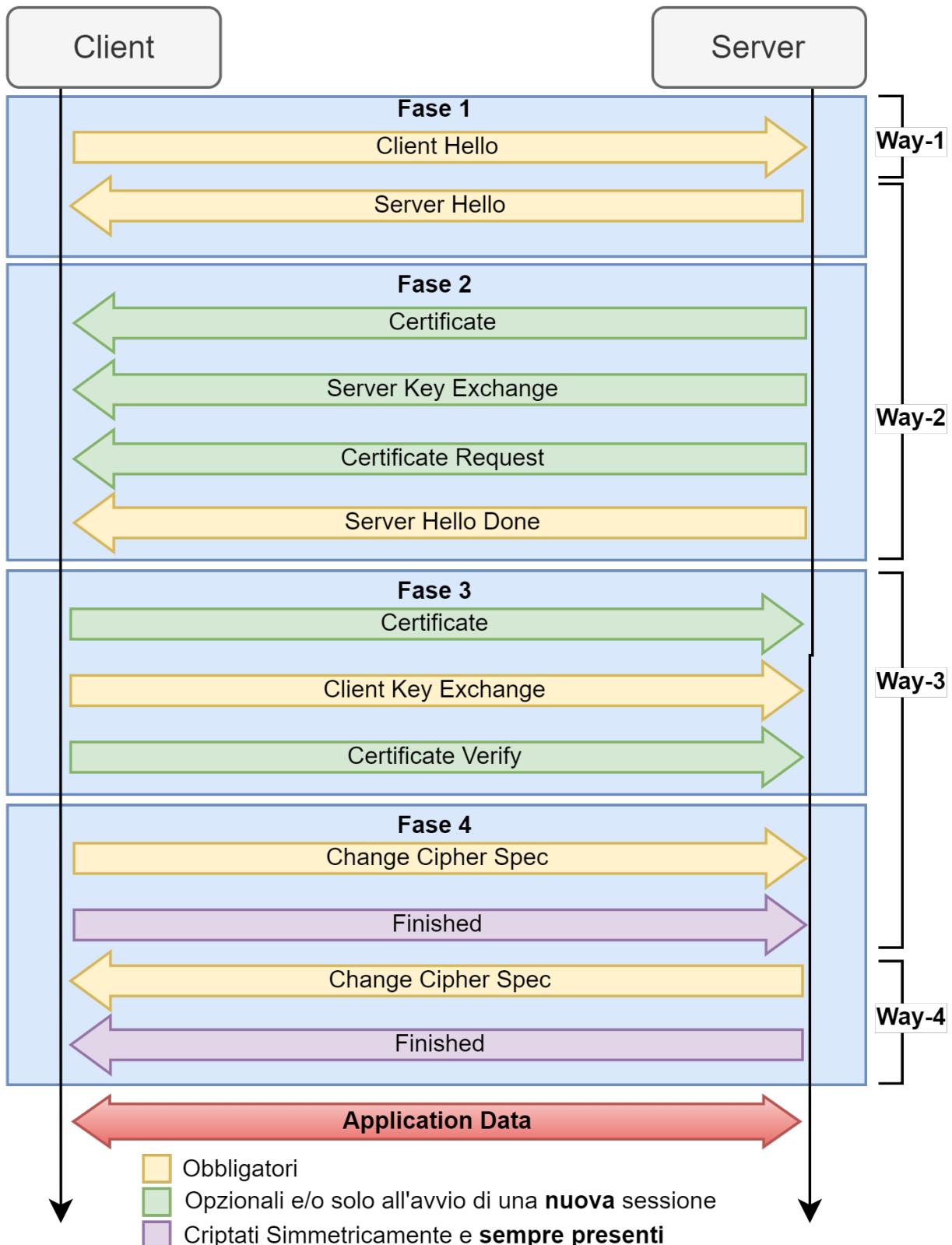
Il protocollo è **suddiviso in 4 fasi** ed è detto **4-way handshake**. Ogni messaggio ha il formato seguente:

### Definizione 10.2 (Handshake Message Format)

- **Header:** (4 byte) informazioni riguardanti il messaggio di handshake. Ha due campi:
  - **Handshake Type:** (1 byte) indica il tipo di messaggio.
  - **Length:** (3 byte) indica la lunghezza in byte del payload.
- **Payload:** contiene le informazioni del messaggio.

I messaggi di Handshake sono **incapsulati** come **payload** all'interno del **TLS Record** (fig. 10.3)

Il protocollo, nella sua interezza, è illustrato in fig. 10.4. Nelle sezioni successive vediamo passo per passo le 4 fasi.

**Figura 10.4:** TLS Handshake Protocol

### 10.3.1 Fase 1: Hello

La fase 1 dell'handshake ha lo scopo di creare una connessione TLS tra client e server. In particolare, svolge i seguenti compiti:

#### Proposizione 10.5 (Phase 1)

*La fase uno deve:*

- Accordare le parti sulla **versione di TLS** da utilizzare.
- Definire l'**ID della sessione**.
- Far scambiare a entrambe le parti una **nonce** costituita da un **timestamp** e da un **valore casuale**.
- Accorda le parti sugli **algoritmi di criptazione** da utilizzare.
- Accorda le parti sull'**algoritmo di compressione** da utilizzare.



Vengono scambiati i seguenti messaggi

#### Definizione 10.3 (Phase 1 Messages)

1. **Client Hello:** Viene inviato dal **client in chiaro**.
2. **Server Hello:** Viene inviato dal **server in chiaro**.



Vediamoli nel dettaglio:

#### Definizione 10.4 (Client Hello Message)

- **Handshake Type:** (1 byte) Indica che il messaggio è un **client hello**.
- **Length:** (3 byte) lunghezza in byte del messaggio.
- **Version:** (2 byte) indica la versione del protocollo TLS.
- **Nonce:** (32 byte) freschezza del messaggio. E' composta da:
  - **Timestamp:** (4 byte) preso all'istante di generazione del messaggio.
  - **Random Value:** (28 byte): valore casuale.
- **Session ID Length:** (1 byte) lunghezza in byte dell'**ID di sessione**.
- **Session ID:** (32 byte) identifica la sessione TLS. Se corrisponde all'**ID di una precedente sessione** allora il client sta **riavviando una vecchia sessione**.
- **Cipher Suites Length:** (2 byte) indica il numero di cipher suites disponibili.
- **Cipher Suites:** (2 byte  $\times N$ ) lista dei cipher algo disponibili.
- **Compression Length:** (1 byte) indica il numero di algoritmi di compressione disponibili.
- **Compression Algorithms:** (1 byte  $\times N$ ) lista degli algoritmi di compressione disponibili.



**Definizione 10.5 (Server Hello Message)**

- **Handshake Type:** (1 byte) Indica che il messaggio è un client hello.
- **Length:** (3 byte) lunghezza in byte del messaggio.
- **Version:** (2 byte) indica la versione del protocollo TLS.
- **Nonce:** (32 byte) freschezza del messaggio. E' composta da:
  - **Timestamp:** (4 byte) preso all'istante di generazione del messaggio.
  - **Random Value:** (28 byte): valore casuale.
- **Session ID Length:** (1 byte) lunghezza in byte dell'ID di sessione.
- **Session ID:** (32 byte) identifica la sessione TLS. Se l'**ID del client hello** è 0, o è di una sessione per il server **non riavviabile**, genera un nuovo ID. Altrimenti usa quello fornito dal client.
- **Cipher Suites:** (2 byte × N) lista dei cipher algo selezionati dal Client.
- **Compression Algorithms:** (1 byte × N) lista degli algoritmi di compressione selezionati dal client.

**Corollario 10.1 (Cipher Suite)**

Una suite di cifratura, denominata anche suite di crittografia o con il termine in inglese *cipher suite* è un insieme di algoritmi utilizzati per rendere sicuri i collegamenti di rete basati su TLS/SSL. L'insieme di algoritmi che costituisce una suite comprende tipicamente: un algoritmo per lo scambio delle chiavi crittografiche, un algoritmo di crittografia ed un algoritmo di message authentication code (MAC).

Il formato con cui una suite è definita è il seguente:

`TLS_ASYMALGO_WITH_SYMALGO_HASHALGO`

**Osservazione:** Non tutte le combinazioni di ASYMALGO, SYMALGO e HASHALGO sono possibili e/o proposte dal client.



### 10.3.2 Fase 2: Server Authentication

La fase 2 due scopi:

**Proposizione 10.6 (Phase 2)**

- **Autenticazione** (opzionale) del server, utilizzando il suo certificato.
- **Inviare la chiave pubblica** del server al client tramite **certificato** o messaggio extra apposito.



Vengono scambiati i seguenti messaggi:

**Definizione 10.6 (Phase 2 Messages)**

3. **Certificate:** (opzionale) inviato dal Server se richiesto dal client. Contiene il **certificato o la catena di certificati del server**.
4. **Server Key Exchange:** (opzionale) Inviato dal server nei casi in cui:
  - Il certificato non viene inviato dal server poiché non è richiesta la sua autenticazione.
  - La chiave contenuta nel certificato può essere usata solamente per firmare.
  - La chiave contenuta nel certificato non può essere usata per ragioni legali.
  - Viene utilizzato Anonymous DH o Ephemeral DH key agreement.
5. **Certificate Request:** (opzionale) inviato dal server se vuole autenticare il client. Contiene il tipo di certificato richiesto e una lista di Certification Authorities. Verificato tramite **certificate transparency** (fig. 9.10).
6. **Server Hello Done:** msg vuoto inviato dal server per indicare il **termine della fase 2**.

**10.3.3 Fase 3: Client Authentication**

La fase 3 deve:

**Proposizione 10.7 (Phase 3)**

- Autenticazione (opzionale) del **client** utilizzando il suo **certificato**.
- Inviare la **chiave pubblica** del client al server tramite il **certificato o messaggio extra** apposito.
- Far generare al **client** un **segreto condiviso** e farlo trasmettere in maniera sicura al **server**.

Vengono quindi scambiati i seguenti messaggi:

**Definizione 10.7 (Phase 3 Message)**

7. **Certificate:** (opzionale) inviato dal **client** se richiesto dal server. Contiene il **certificato o la catena di certificati del client**.
  8. **Client Key Exchange:** inviato dal **client** per permettere al server di **generare il master secret**. Contiene **criptate asimmetricamente** i seguenti campi:
    - La **versione TLS**. Serve per contrastare **early downgrade attacks**.
    - Un **pre-master secret** o delle **informazioni segrete**.
  9. **Certificate Verify** (opzionale) inviato dal **client** per **autenticarsi al server**, provando di conoscere la **chiave privata** associata al proprio certificato. Contiene anche la firma per tutti i **messaggi scambiati** tra client e server fino a questo messaggio compreso, così da evitare **tampering attacks**.
- Verificato tramite **certificate transparency** (fig. 9.10).



**Nota** Il messaggio di **Certificate Verify** è inviato come nono messaggio per includere nella **firma digitale** tutti i **parametri crittografici** scambiati durante l'**handshake**, così che vengano protetti:

- Tutti i **valori casuali** generati dal client e dal server.

- Il **Master Secret** comune, calcolato da client e server, che può essere **computato solamente dopo** che l'*handshake protocol* è arrivato a trasmettere il messaggio di **client Key Exchange**.

### 10.3.4 Fase 4: Finish Message

La fase 4, infine, deve:

#### Proposizione 10.8 (Phase 4)

- Cambiare lo stato della connessione TLS ad un nuovo stato di sicurezza, definito dai parametri scambiati nelle fasi precedenti. In particolare, tale stato comprende:
  - L'autenticazione del client e/o del server, se richieste.
  - L'avvio degli schemi di criptazione simmetrica concordati utilizzando le chiavi di sessione.
- Autenticare tutti i messaggi di handshake precedenti.

I messaggi scambiati sono i seguenti

#### Definizione 10.8 (Phase 4 Message)

10. **Change Cipher Spec**: Inviato dal **client** per indicare al server l'avvio della **criptazione simmetrica**. Il formato del messaggio è definito dal **change cipher spec protocol**.
11. **Finished**: inviato dal **client**. Contiene anche un **hash digest** generato da tutti i **messaggi scambiati** tra client e server **fino a questo messaggio compreso**. Inoltre, è già **criptato simmetricamente**.
12. **Change Cipher Spec**: inviato dal **server** per indicare al client l'avvio della **criptazione simmetrica**. Il formato del messaggio è definito dal **change cipher spec protocol**.
13. **Finished**: inviato dal **server**. Contiene anche un **hash digest** generato da tutti i **messaggi scambiati** tra client e server **fino a questo messaggio compreso**. Inoltre, è già **criptato simmetricamente**.

**Osservazione:** La **criptazione simmetrica** è immediatamente applicata ai messaggi di **Finished** per fornire un **ulteriore** controllo di sicurezza sia al client che al server durante l'handshake. Infatti, nel caso in cui uno dei due **non riesca a decifrare** il messaggio di **finished** ricevuto, l'intero handshake viene considerato **fallito**.

**Osservazione:** Gli **hash digest** contenuti nei messaggi **Finished** hanno lo stesso ruolo della firma contenuta nel messaggio di Client Key Exchange della fase 3 e permettono di **contrastare tampering attacks (MITM)**.

**Osservazione:** Con i messaggi di handshake di questa fase viene anche autenticato il server.

**Osservazione:** Per avere una **negoziazione affidabile** devono **sempre** essere **utilizzati** i meccanismi di **firma e message authentication**, così da **evitare tampering attacks**. I meccanismi di criptazione simmetrica invece possono essere disattivati nel caso in cui il client ed il server non sono interessati alla confidenzialità.

### 10.3.5 Handshake Abbreviato

L'handshake abbreviato permette di **riavviare** la sessione su una nuova connessione. Permette inoltre di **rigenerare** le **chiavi di sessione** senza dover riautenticare le parti e/o riscambiare il pre-master secret. E' a tre vie e composto dai messaggi in fig. 10.5

 **Nota** Nella parte equivalente alla fase 4 (prop 10.8) dell'handshake completo l'ordine di invio tra client e server è invertito, così da ottenere un handshake a 3 vie.

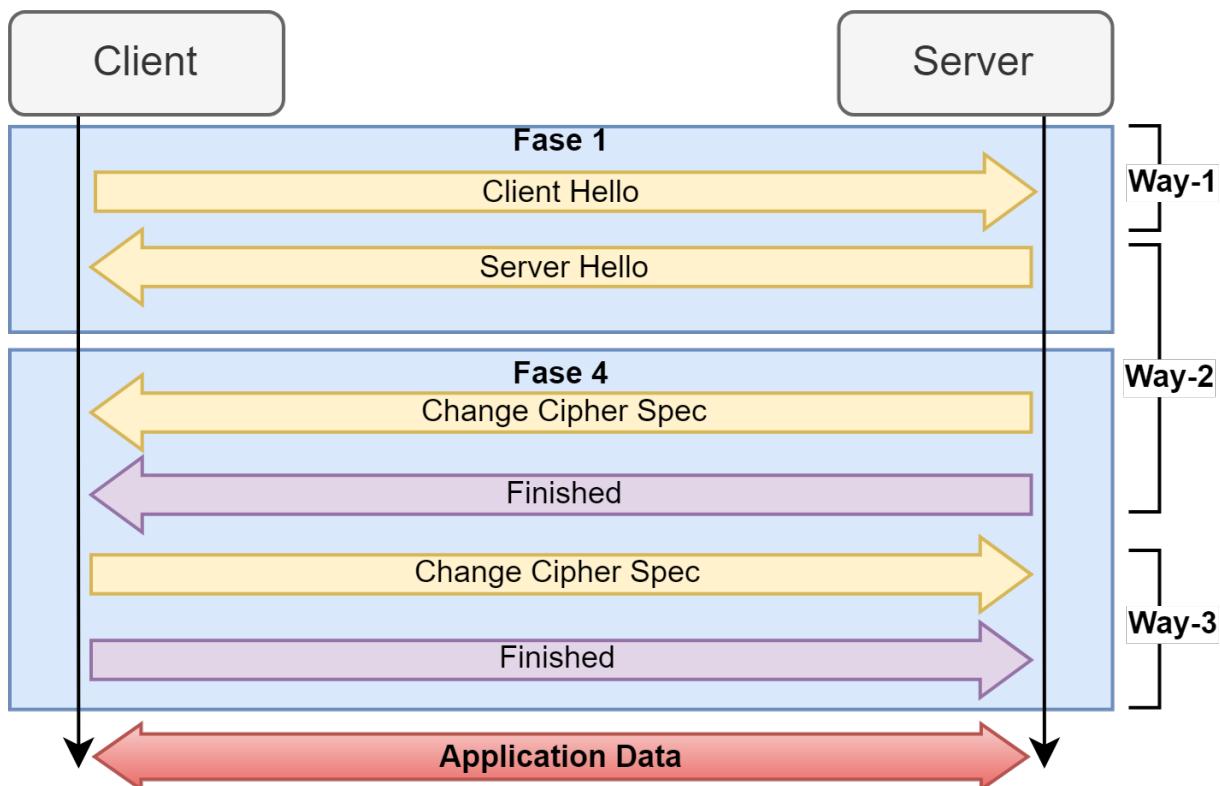


Figura 10.5: Handshake Abbreviato

## 10.4 Key Computation

Nel protocollo TLS sono **utilizzati diversi segreti e nonces per la generazione delle chiavi di sessione** necessarie ai meccanismi di sicurezza concordati durante l'handshake. In particolare abbiamo:

- **Nonces:** freschezza inserita sia da parte del client che da parte del server. Rispettivamente:
  - **Client Nonce:** Inviata tramite **Client Hello** (def 10.4), composta da timestamp e random value.
  - **Server Nonce:** Inviata tramite **Server Hello** (def 10.5), composta da timestamp e random value.
- **Shared Secrets:** Usati "a catena" per generare le chiavi di sessione. Sono di due tipi:
  - **Pre-Master Secret**

**Definizione 10.9 (Pre-Master Secret)**

*Il PMS viene generato **una volta** per sessione.*

- *Può essere generato dal client e comunicato al server con RSA Key Transport (fig. 9.5). In tal caso viene utilizzato solamente il Client Key Exchange message della fase 3 (prop 10.7).*
  - *Può essere calcolato tramite DH Key Agreement (def 9.3), utilizzando il Server Key Exchange della fase 2 (prop 10.6) e il Client Key Exchange della fase 3.*
- Osservazione:** Nel caso in cui venga usato Fixed DH (def 9.4) il valore del PMS è costante ad ogni nuova sessione, per una fissata coppia client-server

- **Master Secret (MS):** rigenerato ad ogni riavvio della sessione utilizzando il Pre Master Secret e le nonces del client e del server.
- **Connection State Keys**

**Definizione 10.10 (Connection State Keys)**

*chiavi di sessione rigenerate ad ogni riavvio e necessarie ai meccanismi di sicurezza negoziati durante l'handshake completo. Possono essere al più 6 chiavi, ovvero:*

- **Due Chiavi per la criptazione simmetrica**
  - $KC_C$ : chiave per la criptazione **dal client al server**
  - $KC_S$ : chiave per la criptazione **dal server al client**
- **Due chiavi per il controllo integrità**
  - $KI_C$ : chiave per la integrità **dal client al server**
  - $KI_S$ : chiave per la integrità **dal server al client**
- **Due Initialization Vectors, se necessari:**
  - $IV_C$ : iv per la criptazione **dal client al server**
  - $IV_S$ : iv per la criptazione **dal server al client**

Le chiavi di sessione vengono rigenerate ad ogni riavvio seguendo la strategia seguente:

**Definizione 10.11 (Extract-then-Expand)**

*La strategia EtE consta di due fasi:*

1. **Extract:** viene estratta una chiave psuedo-random (il MS) a partire da una chiave segreta (il PMS) ed un salt/seed (le nonces).

**Osservazione:** Serve ad aggiungere casualità nella generazione delle chiavi di sessione rendendo l'MS casuale con distribuzione uniforme. Il PMS infatti potrebbe essere **biased** o addirittura **costante**.

2. **Expand:** viene utilizzata una PRF per espandere arbitrariamente la chiave pseudo-random generata nella fase precedente.

**Osservazione:** Serve ad estendere un segreto limitato alla quantità di materiale crittografico desiderata, ovvero al numero di chiavi di sessione necessarie.

La funzione PRF usata in *EtE* può essere costruita utilizzando la costruzione HMAC (eq. (3.2)) con una funzione **crypto hash** qualsiasi. Alcuni dei modi usati sono i seguenti:

- Funzione di **Espansione Hash**:

$$P_H(secret, label, seed) = HMAC_{H,secret}(A_1||A_0)||HMAC_{H,secret}(A_2||A_0)||\dots$$

Dove  $A_i = HMAC_{H,secret}(A_{i-1})$  e  $A_0 = ctx\_str||seed$ .

 **Nota** Questa costruzione **non è sicura** in quanto non fornisce garanzie sull'assenza di collisioni sul risultato prodotto. Ad esempio potremmo andare in contro a **short cycle problem** (def 7.7)

- **HMAC-Based Key Derivation Function**:

#### Definizione 10.12 (HKDF)

Costruzione sicura in quanto utilizza un contatore per evitare cicli.

$$HKDF(secret, label, seed) = HMAC_{H,secret}(ctx\_str||0)||HMAC_{H,secret}(\cdot||1)||\dots$$



**Nota** Con *ctx\_str* indichiamo una **context string**, ovvero una stringa formata da un testo arbitrario aggiuntivo e un *seed*. Il suo contenuto è **esplicativo** del contesto nel quale stiamo calcolando la chiave e indica lo scopo della computazione, ad esempio:

$$PRF(master - secret, "client finished", MD5(all handshake msg)|SHA1(all handshake msg))$$

Con questi metodi si possono ottenere digest di lunghezza arbitraria e costruire le chiavi necessarie a TLS per operare correttamente ed utilizzare una chiave diversa per ogni servizio. Ogni versione del protocollo utilizza una propria PRF:

- **SSL V3.0**: utilizza una PRF che non soddisfa le proprietà richieste.
- **TLS v1.0/v1.1**: usa  $P_{H,secret}(secret, lbl, seed) = P_{MD5}(secret, lvl, sd) \oplus P_{SHA-1}(secret, lbl, sd)$ .
- Osservazione:** In queste versioni le funzioni hash erano hardcoded e l'uso di  $P_H$  introduce problemi di short cycling, quindi è un metodo non sicuro. Inoltre, venne fatto lo xor di due funzioni crittografiche differenti, generando una costruzione senza nessuna prova matematica che fosse sicura.
- **TLS v1.2**:  $P_H$  con funzioni hash negoziabili (default: SHA-256).
- **TLS v1.3**: Uso di **HKDF** con funzione hash negoziabile (default: SHA-256).

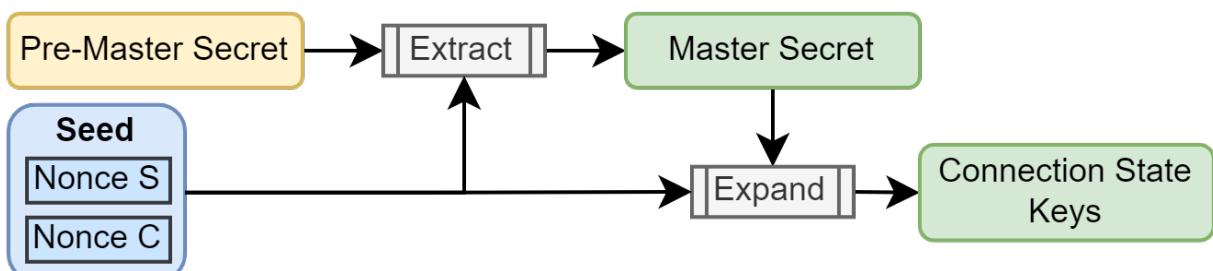


Figura 10.6: KDF Scheme

## 10.5 Other Minor Protocols

Ci sono ulteriori protocolli minori necessari al funzionamento e alla gestione della sessione, come la gestione degli errori o cambio di stato della sessione.

### 10.5.1 Change Chiper Spec Protocol

#### Definizione 10.13 (Change Cipher-Spec Protocol)

*Questo protocollo è definito come separato dall'Handshake Protocol per poter criptare immediatamente i messaggi successivi ad un Change Cipher Spec message. Questi messaggi sono dei messaggi vuoti costituiti da un byte costante, di valore 0 o 1 per indicare il cambio di stato del livello di sicurezza della sessione TLS.*

Inoltre:

#### Teorema 10.4 (TLS Message Aggregation Principle)

*Il principio di aggregazione dei messaggi TLS specifica che tutti i messaggi provenienti da uno stesso livello o superiore devono essere aggregati in un unico TLS record.*

*Inoltre, un TLS record deve essere criptato per intero e mai parzialmente.*

Con tale soluzione i ***finished message*** dell'handshake, poiché inviati **sempre dopo** i ***change cipher spec***, sono inviati sicuramente in un TLS record criptato **simmetricamente**.

Content type=0x14	Major Version	Minor Version	Length of Fragment 0x0001	Payload 0/1
-------------------	---------------	---------------	------------------------------	----------------

**Figura 10.7:** Change Cipher Spec Protocol Message

## 10.5.2 Alert Protocol

### Definizione 10.14 (Alert Protocol)

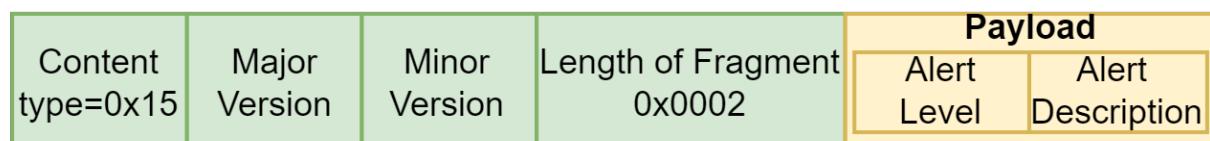
E' un protocollo che permette lo scambio di **avvertimenti** tra client e server, al fine di segnalare avvenimenti eccezionali e/o errori. A tale scopo vengono definiti messaggi speciali per comunicare e gestire tali casi.

### Definizione 10.15 (Alert Message Format)

Il formato degli alert è il seguente:

- **Alert Level** (1 byte) indica il livello di gravità che può essere:
  1. **Warning**: la connessione/sicurezza può essere instabile.
  2. **Fatal**: la connessione/sicurezza può essere stata compromessa, oppure si è verificato un errore **non recuperabile** nell'esecuzione del protocollo.
- **Alert Description**: (1 byte) Descrive il significato dell'alert, ne esistono 23 diversi. Alcuni sono:
  - **close\_notify**: Segnala la terminazione volontaria della connessione TLS.
  - **unexpected\_message**: E' stato ricevuto un messaggio inappropriato.
  - **bad\_record\_mac**: il MAC del TLS-Record non è corretto.
  - **record\_overflow**: la lunghezza del TLS-Record è maggiore a quella massima consentita.
  - **handshake\_failure**: impossibile negoziare un set accettabile di parametri di sicurezza dalle opzioni disponibili dichiarate dal client e/o dal server
  - **bad\_certificate**: il certificato è corrotto e/o la firma è errata.
  - **unsupported\_certificate**: il formato del certificato non è supportato.
  - **certificate\_revoked**: il certificato è stato revocato.
  - **certificate\_expired**: il certificato è scaduto.

Questi messaggi sono **incapsulati come payload all'interno di TLS Record**. Nel caso in cui venga ricevuto un **fatal alert**, la connessione viene **terminata** e viene **impedito il riavvio con gli stessi parametri di sicurezza**.



**Figura 10.8:** Alert Protocol Message



**Nota** Il warning di **close\_notify** viene utilizzato dal protocollo TLS per terminare la connessione in modo volontario da una delle due parti. Da questo momento in poi, una delle due parti specifica che **non verranno più trasmessi ulteriori dati applicativi**.

Questo meccanismo permette di **distinguere** tra le **connessioni terminate volontariamente in modo sicuro** e quelle **interrotte da cause esterne**, tra cui la **terminazione della** sottostante connessione TCP.

### 10.5.3 Renegotiation

La rinegoziazione è la possibilità di effettuare un handshake completo per una connessione TLS già esistente così da poter cambiare i parametri di sicurezza, come ad esempio il cipher suite utilizzato o autenticare una parte precedentemente non autenticata. Alcuni dettagli importanti sono:

#### Definizione 10.16 (Renegotiation (overview))

- *Viene generato un nuovo Session ID alla connessione TLS. In particolare, il client genera un Client Hello con Session ID pari a 0 così che il server utilizzi un Session ID differente dal precedente.*
- *Oltre alle chiavi di sessione, vengono anche rigenerati il pre-master secret ed il master secret.*
- *L'intero handshake per la rinegoziazione è criptato asimmetricamente tramite utilizzando il cipher suite in vigore, a differenza del primo handshake, il quale è effettuato in chiaro.*
- *La rinegoziazione ha precedenza sui dati applicativi. Eventuali dati generati dal server (il client è consapevole della rinegoziazione) durante l'esecuzione dell'handshake vengono bufferizzati localmente in chiaro in attesa che termini la rinegoziazione.*

Ne deriva che:

- La **rinegoziazione non è distinguibile** dalle **negoziacione precedenti**, escluso il fatto che è criptato usando la precedente cipher suite. Inoltre, non sono legate crittograficamente tra loro.
- Una **transazione** effettuata a **livello applicativo** potrebbe essere **spezzata** in due parti dall'esecuzione di una rinegoziazione, comportando differenza nella sicurezza applicata alle due parti.

 **Nota** *L'interruzione del livello applicativo genera delle vulnerabilità in TLS che non possono essere rilevate da client e server.*

## 10.6 DTLS

Il protocollo DTLS è una versione del **protocollo TLS che viene trasportata tramite il protocollo UDP**. Le principali differenze con la versione tradizionale trasportata tramite il protocollo TCP sono:

### Definizione 10.17 (DTLS Major Features)

- Viene utilizzato un **sequence number** nell'header dei DTLS Record per il riordinamento e dei **timeout** per rilevare la perdita di datagrammi UDP.  
→ TLS assume la consegna in ordine dei segmenti TCP.
- Sono aggiunti dei **meccanismi di frammentazione dei DTLS record**, così che possano essere trasportati in un solo DTLS record.  
→ In TLS possono essere generati dei TLS record di grandi dimensioni.
- La **connessione** viene esplicitamente **delimitata dall'handshake** ed i **messaggi di close notify**.  
→ In TLS la connessione viene assunta dall'uso del sottostante protocollo TCP.



## 10.7 Original Sin of TLS (up to v1.2)

I servizi di integrity possono essere visti spesso come l'unico requisito da implementare, in quanto è generalmente importante prevenire lo spoofing (injection) o il tampering (modification) dei messaggi. In parole povere vale il seguente "motto": *vedere ma non toccare*.

Allo stesso tempo, sappiamo che l'**encryption POTREBBE non garantire integrity**, a meno di usare **AEAD**<sup>5</sup> capitolo 8. I tre maggiori protocolli di sicurezza applicano i seguenti schemi di cifratura e autenticazione:

- **TLS (up to v1.2):** MAC-then-Encrypt. Invio  $ENC(Data, MAC(Data))$ . Costruzione sbagliata, dimostrata e che porta ad attacchi chosen ciphertext.
- **IPsec:** Encrypt-then-MAC. Invio  $ENC(Data)|MAC(ENC(Data))$ . L'unico modo giusto, ma **AEAD** (capitolo 8) è meglio.
- **SSH:** Encrypt-then-MAC. Invio  $ENC(Data)|MAC(Data)$ . Nonostante possa essere visto come più efficiente in quanto encryption e integrity possono essere svolte in parallelo, è formalmente sbagliato in quanto il MAC da solo non protegge la confidentiality, ed essendo deterministico (il mac per uno stesso messaggio deve essere lo stesso per definizione) risulta facilmente attaccabile appena c'è un messaggio ripetuto.



**Nota** Per le operazioni di decrittazione, è importante la dimensione dei pacchetti e, di riflesso, il padding usato. TLS specifica come definire il padding:

<sup>5</sup>Come fatto in TLS v1.3

### Definizione 10.18 (TLS Padding)

Assumiamo che  $L$  sia la dimensione di un blocco cifrato e che l'ultimo byte del blocco contenga proprio il valore di  $L$ . Se  $L$  è la dimensione del blocco meno 1, allora viene inserito un byte di padding il cui valore è 00. Se la dimensione è inferiore a quella di un blocco, allora come padding vengono inseriti tanti byte il cui valore è quello di  $L$ .

Se  $L$  è pari alla dimensione del blocco e non sarebbe necessario padding, si crea un nuovo blocco composto da solo byte di padding.

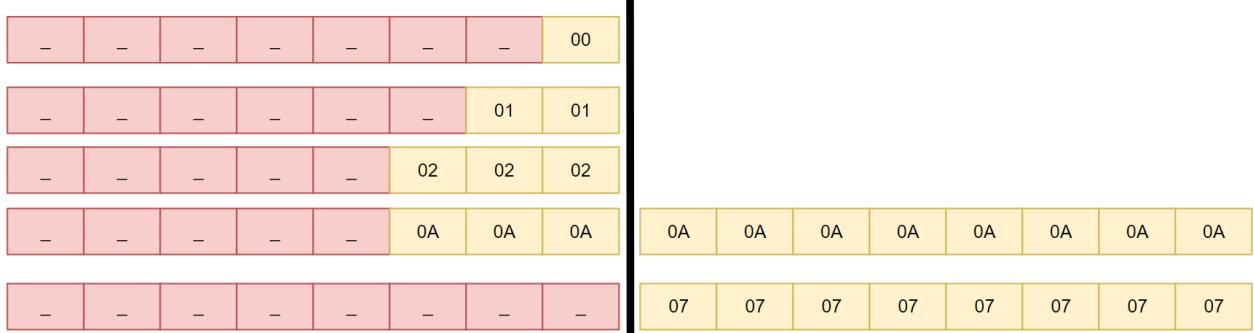


Figura 10.9: TLS Padding

#### 10.7.1 Attacks to Encryption - Padding Oracle

Nel capitolo 7 abbiamo visto i block cipher, che usano logiche "particolari" per cifrare i messaggi. TLS è fatto in modo di usare diversi cifratori e quindi gli attacchi sono *relativi* allo specifico algoritmo usato. Tralasciamo algoritmi deboli e/o bucati, e concentriamoci su quelle cattive implementazioni che ci sono state.

Consideriamo TLS v1.0 e il meccanismo di decrittazione usato da CBC (def 7.2). Riassumendo, il meccanismo si comporta nel seguente modo:

1. **Applica decrittazione:** se il numero di byte **NON** è multiplo della dimensione del blocco ritorna un messaggio di alert **decryption\_failed**.
2. **Rimuovi padding:** leggo dall'ultimo byte la lunghezza  $L$  e rimuovo gli ultimi  $L$  byte controllando che ogni blocco di padding sia uguale a  $L$ . Altrimenti ritorna un messaggio di alert: **decryption\_failed**.
3. **Controlla MAC:** il controllo è fatto sul messaggio decifrato senza pad. Se fallisce ritorna un alert message: **Bad\_record\_mac**

**Osservazione:** Il comportamento di error-signaling è tipico dei protocolli di rete, dove la **ragione dell'errore** viene sempre spiegata.

**Definizione 10.19 (Padding Oracle Attack<sup>6</sup>)**

*Consideriamo un sistema che accetta messaggi cifrati e che usi un block-cipher per cifrare e decifrare i messaggi.*

*Supponiamo che il sistema possa rispondere in più modi quando riceve un messaggio sbagliato, ad esempio in 2 possibili modi (vedi TLS v1.0):*

1. **Decryption Failed:** Il padding è sbagliato.
2. **Bad-MAC:** Il padding è giusto e il server è arrivato al terzo check.

*Supponiamo che l'attaccante disponga di un ciphertext C (ad esempio in risposta di una challenge) e che sia in grado di forgiare un ciphertext C' che inoltra al sistema, la cui probabilità di essere valido è trascurabile<sup>7</sup>.*

*Sapendo il modo con cui questo decrypta il messaggio e restituisce il codice d'errore l'attaccante ha a disposizione un oracle che, dato un messaggio cifrato in ingresso, è in grado di fornire informazioni sul plaintext.*

<sup>6</sup>Modello di attacco valido per ogni block-cipher.

<sup>7</sup>Anche se i ciphertext sono corretti con probabilità "negligible", l'errore può essere controllato a prescindere. □



**Nota** Non è necessario che il sistema risponda con un codice esplicito di errore. Spesso differenze nel tempo di elaborazione e nel comportamento che assume il sistema possono essere un indicatore per un attaccante in grado di intercettare questi dettagli.

**Esempio 10.2 Padding Oracle in TLS v1.0**

Consideriamo lo scenario di TLS e, per semplicità, supponiamo che stia usando CBC (fig. 7.3b) e che l'attaccante disponga di un ciphertext formato da  $c[0], c[1]$  e del relativo IV.

Per come funziona CBC in decryption abbiamo che:

$$\begin{aligned} m[0] &= PRP_k^{-1}(c[0]) \oplus IV \\ m[1] &= PRP_k^{-1}(c[1]) \oplus c[0] \end{aligned}$$

Poiché tale relazione è lineare nello xor, una modifica nell'i-esimo byte dell'IV implica una modifica nell'i-esimo byte di  $m[0]$  e, analogamente, una modifica nell'i-esimo byte di  $c[0]$  causa una modifica nell'i-esimo byte di  $c[1]$ .

Senza perdita di generalità supponiamo di voler conoscere  $m[1]$ <sup>8</sup>, quindi supponiamo di modificare l'ultimo byte di  $c[0]$  con un carattere da  $v$  da noi scelto in modo tale che:

$$c[0](n) = c[0](n) \oplus v \oplus 0 \times 00$$

Dove  $0 \times 00$  è il padding che ipotizziamo possa essere stato fatto in quanto stiamo indovinando l'ultimo carattere e, pertanto, potrebbe non esser stato necessario paddarlo.

Inviando questo messaggio all'oracolo riceveremo, nella maggior parte dei casi, un messaggio di errore:

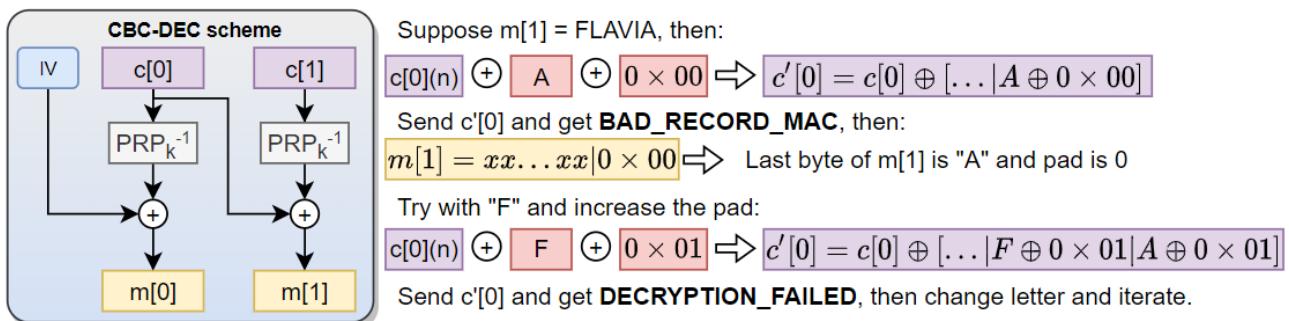
1. **Decryption Failed:** non resta che provare un nuovo carattere  $v$  poiché non abbiamo indovinato l'ultimo carattere.
2. **Bad-MAC:** allora il messaggio decifrato sarà  $m[1] = xx\ xx\ xx\ xx\ |0 \times 00$ . Tale messaggio produrrà un MAC diverso ma a noi non importa perché abbiamo scoperto il padding che era

<sup>8</sup>se avessimo voluto conoscere  $c[0]$  avremmo lavorato con l'IV.

stato fatto su  $m[1]$  e il valore del suo ultimo byte, ovvero:  $m[1] = xx\ xx\ xx\ xx\ v$ .

Dal caso due capiamo che l'attaccante è ora a conoscenza dell'ultimo carattere di  $m[1]$  e ora può iterare facendo brute-force su tutti i byte rimanenti, aumentando di 1 il valore del padding così da non alterare la dimensione del blocco che verrà controllato. In questo modo il sistema attaccato, controllando l'ultimo byte contenente la lunghezza del messaggio, arriverà sempre al terzo check.

 **Nota [Complessità dell'Attacco]** La complessità dell'attacco è  $\leq L$ , dove  $L$  è la lunghezza del blocco da decifrare. Con 8 un blocco da byte avremmo al più  $(2^8) * 8$  per decriptare un solo block.



**Figura 10.10:** Padding Oracle

 **Nota [Ricorsione per determinare il Padding]** E' possibile determinare il padding fatto sul blocco da decifrare partendo dal modificare uno alla volta i primi  $k$ -byte dell'IV relativo finché non otteniamo un **DECRYPTION\_FAILED**. Allora il padding fatto sarà  $n - k$  e possiamo provare l'attacco.

## 10.7.2 Solutions of next versions and Final Considerations

Dall'esempio 10.2 capiamo che MAC-than-Encrypt **NON E' MAI LA SCELTA MIGLIORE**, in quanto per proteggersi da un padding oracle sarebbe bastato autenticare il testo cifrato, così da bloccare ogni possibile modifica ad esso.

TLS 1.0 infatti venne patchato, selezionando come ritorno in caso di errore sempre e solo **BAD\_MAC**, soluzione poi standardizzata da TLS 1.1 e cambiata in **TLS v1.2**, dove se il **padding falliva**, il **MAC** era **validato in OGNI caso**. Questo perché fino alla v1.1 tramite side-channels era comunque possibile capire grazie al tempo di risposta del server in quale fase ci fosse stato il fallimento in quanto più tempo di risposta implica che l'errore è avvenuto in una fase successiva.

### Corollario 10.2 (Problem with TLS v1.2 workaround)

Validando il MAC in ogni caso, sorge il problema di **non sapere più quale dato viene validato**, poiché se il padding-check fallisce non abbiamo modo di conoscere la dimensione del messaggio rispetto a quella del padding. TLS v1.2 usava l'**intero messaggio** per convalidare, ma questo implicava l'uso di più dati e più tempo per calcolare l'HMAC.

 **Nota** L'unico modo per patchare definitivamente TLS fu quello di rimuovere ogni forma di combinazione di **ENC** e **MAC** e di usare **AEAD**.

### Proposizione 10.9 (Praticità dell'Attacco)

Un padding oracle potrebbe non essere applicabile in ogni scenario della realtà, in quanto TLS appena rileva un errore chiude la connessione e bisogna ricominciare daccapo. Tuttavia, resta uno strumento molto forte nelle mani di un hacker.



Nel 2003 è stato dimostrato che l'attacco è realizzabile su protocollo IMAP, in quanto i client IMAP fanno periodicamente login ogni 5 minuti, trasmettendo le credenziali che vengono inviate **sempre con lo stesso formato**. Facendo un DNS spoofing (MITM) è stato possibile effettuare un CCA, ottimizzando con strategie dictionary attack per scoprire le password.

## 10.8 Major Vulnerabilities

Una delle più grandi falte in TLS 1.0 fu la **scelta** di implementare hardcoded gli algoritmi di cifratura, compromettendone la resistenza *a lungo termine*.

Uno degli algoritmi implementati era CBC, il quale soffre di un problema di *IV-prediction*, Infatti per operare la cifratura di una serie di messaggi, soltanto il primo IV del primo gruppo di blocchi veniva generato "on-the-fly", perché per gli altri gruppi veniva usato l'ultimo cipherblock del gruppo precedente.

**Osservazione:** Se l'IV dipende dal blocco precedente, nonostante abbiamo la sicurezza semantica (IND-CPA), un attaccante può predire l'IV del prossimo processo di encryption e arrivare al plaintext.

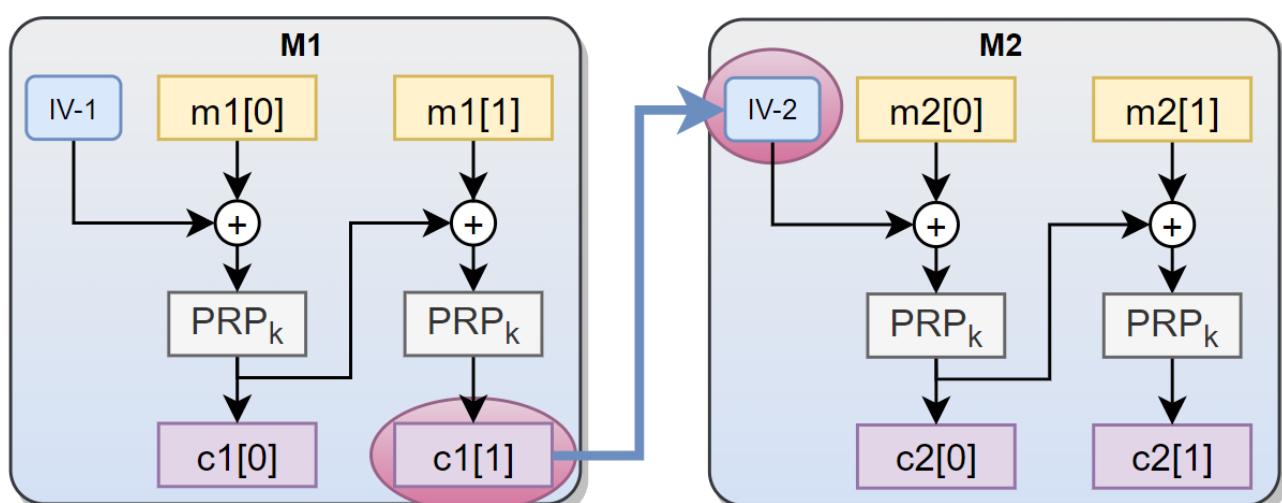


Figura 10.11: Last ciphertext-block of M1 used as IV for M2

Lo schema di attacco per un IV prevedibile è allora il seguente:

#### Definizione 10.20 (IV Prediction Attack)

*Supponiamo di sapere che l' $i$ -esima porzione di  $m$ ,  $m[i]$ , contenga una password e supponiamo per semplicità che la password possa essere A o B. Consideriamo lo schema di encryption di CBC (def 7.1). Poiché CBC è semantic secure, non è possibile usare un CPA, ma è possibile osservare  $c[i - 1]$ .*

*Sia  $X$  l'IV predetto, allora, forgiamo un ciphertext tale che:*

$$c'[i - 1] = X \oplus c[i - 1] \oplus \text{Guessed\_PWD}$$

*Se l'IV è stato predetto correttamente, allora basterà chiedere al sistema di cifrare per noi  $m[i]$  con IV pari a  $c'[i - 1]$ .*

*Se  $c'[i - 1] = c[i]$ , allora abbiamo individuato la password. Altrimenti dobbiamo provare l'altra password.* □



**Nota** Inizialmente si pensava che l'attacco fosse impossibile da realizzare davvero, ma la vulnerabilità venne exploitata davvero tramite le moderne tecnologie, come HTML5. Ad esempio, tramite l'inserimento forzato di chosen plaintext facendo girare codice direttamente sul browser dell'utente.

### 10.8.1 BEAST Attack: Chosen Boundary Attack

Consideriamo i cookie di un servizio on-line. Questi oggetti contengono in locale delle informazioni utili alle web-app per *velocizzare* l'utilizzo delle stesse, al fine di portare un vantaggio all'utente. Ad esempio sono utili per effettuare un "*login veloce*", in quanto il browser può auto-completare i campi e-mail e password di un form.

Tuttavia, è stato provato che è possibile decifrare i cookie di un browser e la complessità è lineare con la loro lunghezza. Questo lavoro è stato provato da Duong e Rizzo, nell'attacco BEAST<sup>9</sup>.

<sup>9</sup>BEAST

### Definizione 10.21 (Chosen Boundary Attack)

Supponiamo che un cookie contenga nell'ultimo blocco la password di un utente, supponiamo di sapere il punto in cui la password comincia, per semplicità<sup>10</sup>. Abbiamo allora a disposizione:  $IV = c[i - 1]$ ,  $c[i] = Enc_k(c[i - 1] \oplus P[i])$  e vogliamo indovinare  $P[i]$ .

1. Aggiungiamo davanti alla stringa da attaccare una stringa da noi scelta, per fare in modo di allineare **il primo carattere della pwd** all'ultimo byte di blocco considerato.
2. Lanciamo un brute-force attack, costringendo la vittima a lavorare come un encryption oracle. Il chosen plaintext che vogliamo inviare ha la struttura seguente:

$$P[i + 1] = c[i] \oplus \text{Guess} \oplus c[i - 1] \quad (10.1)$$

In questo modo quando quello che verrà cifrato sarà esattamente la stessa cosa che era stata cifrata all'inizio, ovvero:

$$\begin{aligned} c[i + 1] &= Enc_k(c[i] \oplus P[i + 1]) \\ &= Enc_k(c[i] \oplus c[i] \oplus \text{Guess} \oplus c[i - 1]) \\ &= Enc_k(\oplus \text{Guess} \oplus c[i - 1]) \end{aligned} \quad (10.2)$$

3. Se  $c[i + 1] = c[i]$ , allora significa che il **guess** è il plaintext cercato. Shiftando un carattere alla volta e modificando la stringa aggiunta all'inizio, possiamo far entrare un carattere alla volta nel blocco da decifrare e scoprire tutta la password.

<sup>10</sup>Altrimenti avremmo solo bisogno di più tempo per decifrare la password. □

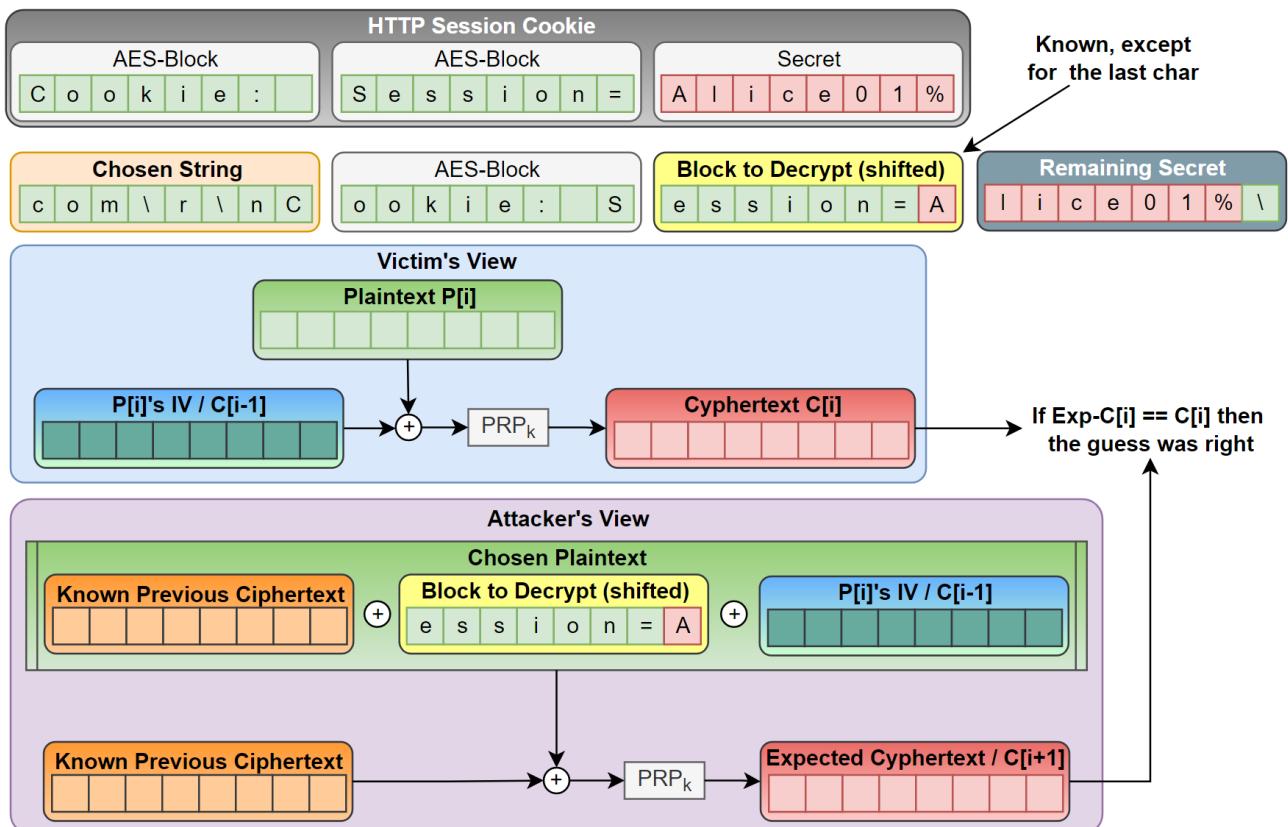


Figura 10.12: BEAST Attack

## 10.8.2 CRIME Attack: Compression after Encryption leakage

Nel 2012, Duong e Rizzo dimostrarono che la compressione messa in atto da TLS portava ad un leak di informazioni riguardo il plaintext. Saltò fuori che in realtà il problema è intrinseco a qualsiasi sistema di compressione. Infatti:

**Esempio 10.3 Compression Size Leak** Consideriamo una password di 6 byte: *AAAABC*. Applichiamo **Compress-than-Encrypt**, producendo prima *4ABC* e poi *X&%\$*.

**Osservazione:** La lunghezza della password è passata da 6 a 4B.

Consideriamo un'altra password da 6 byte, la cui entropia è maggiore: *ABCDEF*. Poiché l'entropia è massima, la compressione produrrà: *ABCDEF* e l'encryption *&\$£A£\$*.

**Osservazione:** La lunghezza della password è rimasta a 6B.

Un attaccante ha ora una chance, unendo il size-leaking con un plaintext-injection.

### Definizione 10.22 (Compression with CPA)

*Supponiamo di voler indovinare la password di un utente e supponiamo di conoscere la dimensione del pacchetto compresso e cifrato. Allora:*

- *Supponiamo di poter intercettare il messaggio contenente la password trasmessa e di poter modificare il messaggio trasmesso, ma di non poterlo leggere.*
- *Appendiamo un plaintext composto da lettere tutte uguali<sup>11</sup> all'inizio del testo sconosciuto.*
- *Il messaggio modificato verrà compresso e cifrato. Possono succedere 2 cose:*
  1. *len( $c'[i]$ ) = len( $c[i]$ ): La compressione non è stata utile perché l'entropia era troppo alta.*
  2. *len( $c'[i]$ ) < len( $c[i]$ ): Il testo che abbiamo aggiunto ha ridotto l'entropia ed è stato compresso.*

*Se la lunghezza è stata ridotta, significa che la nostra aggiunta era UGUALE (almeno) alla prima lettera della password.*

<sup>11</sup>set di  $n$  lettere tutte uguali: *AAA*



**Nota** Non potremmo comunque sapere tutta la password in questo modo, soltanto la prima o le prime lettere, se dovessero essere uguali.

Lo schema descritto venne adottato nell'attacco **CRIME**<sup>12</sup>, sempre ad opera di Duong e Rizzo nel 2012. Riutilizzando gli stessi tool di BEAST, riuscirono a fare plain-text injection per sfruttare una vulnerabilità di **DEFLATE**, un algoritmo di compressione basato sulla codifica **Huffman**<sup>13</sup> e su **LZ77**, una politica di compressione che sostituisce 3 o più caratteri ripetuti in una stringa con una coppia (**Offset, Size**) che costituisce un puntatore ad un punto precedente della stringa.

**Esempio 10.4 LZ77** Consideriamo la seguente stringa: *Giuseppe Bianchi and Marco Bianchini*  
Questa viene compressa in: *Giuseppe Bianchi and Marco (-18,7)ni*.

L'idea dello schema di CRIME è quindi il seguente:

<sup>12</sup>Compression Ratio Info-Leak Made Easy

<sup>13</sup>Huffman Encoding

**Definizione 10.23 (CRIME)**

Supponiamo di conoscere un testo generato da un utente, composto da una parte **conosciuta**, l'"**Header**" e una parte **non nota**, il **Secret**. Ad esempio:

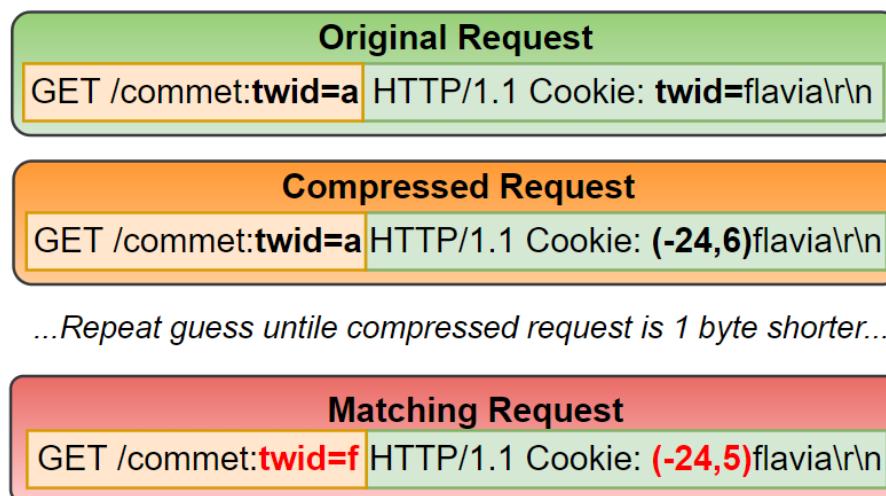
[*AuthenticationToken* : *passwd = alice%01*]

1. Creiamo multipli messaggi composti da:

*Att\_input + [header + secret]*

2. Inviamo e osserviamo il risultato della **compress-than-enc**:

*Len(Enc(Compress(*Att\_input + [header + secret]*)))*



**Figura 10.13:** CRIME example for auth token in HTTP session



**Nota** In realtà ci sono altri dettagli tecnici che andrebbero sviscerati, ma che non sono stati trattati, come il fatto che la compressione lavora sui singoli bit e non su byte, oppure che abbiamo completamente trascurato la codifica usata.

E' interessante osservare anche che l'attacco venne fatto su due protocolli diversi, TLS e SPDY (considerato HTTP2.0) e che il meccanismo è agnostico al sistema di cifratura.

### 10.8.3 Downgrade Attack

Il downgrade attack si basa su potenziali vulnerabilità **nella fase 1** dell'handshake protocol. Si tratta di un MITM con l'obiettivo di modificare la versione in uso di TLS tra client e server, affinché sia possibile sfruttare ulteriori falle.

Idealmente l'attacco è possibile in quanto per retrocompatibilità, stabilità ed affidabilità, diverse sistemi non vengono aggiornati a versioni recenti dei protocolli perché troppo costoso in termini di risorse (temporali ed economiche). Vediamone le caratteristiche:

### Definizione 10.24 (Downgrade Attack)

Il MITM sfrutta l'assenza di message authentication della fase 1, per modificare il contenuto di **Client Hello** e **Server Hello**. In particolare, viene cambiato il campo della versione del protocollo, con una arbitrariamente scelta dall'attaccante.

Se la versione scelta è supportata dal server, allora questo risponde con un server hello con la versione modificata.

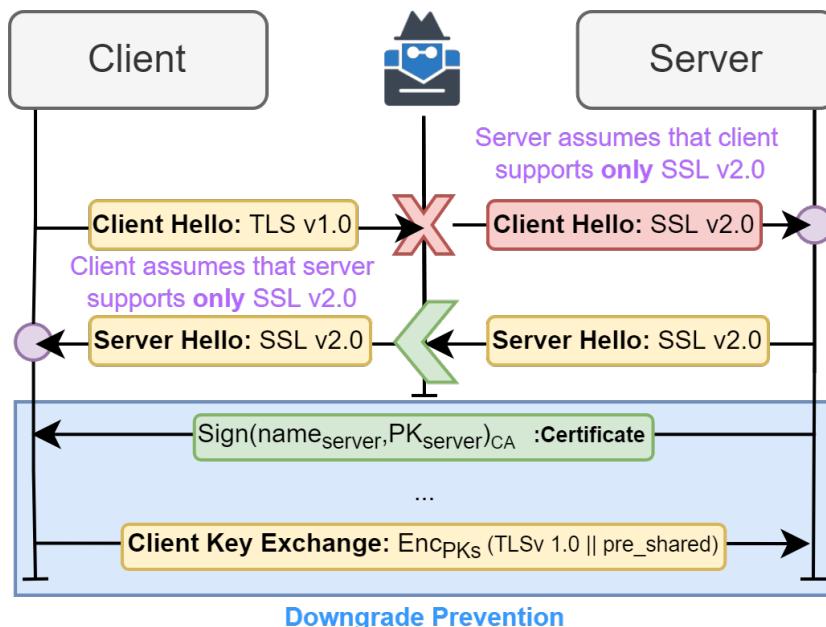


Figura 10.14: Downgrade Attack and Prevention

### Proposizione 10.10 (Downgrade Prevention)

Durante la fase 2 e 3 un downgrade attack viene intercettando in quanto:

- Il **certificate message** inviato dal server, firmato dalla CA, permette di **comunicare in maniera sicura** la chiave pubblica del **server**, con la quale il client cripterà asimmetricamente il contenuto dei suoi messaggi.

**Osservazione:** Questo avviene perché il certificato **non** può essere modificato dall'attaccante senza che non venga rilevato dal client, **impedendo la sostituzione** delle chiavi.



- Il **client key exchange** inviato dal client contiene al suo interno la versione di protocollo usata e dichiarata nel **Client Hello**. Il server può allora rilevare modifiche effettuate dall'attaccante.

**Osservazione:** L'attaccante non può neanche decriptare la **pre-master secret** inclusa nel messaggio, per cui non può inferire nulla sulle future chiavi di sessione.



**Nota** Si noti che il **Certificate message** non è obbligatorio e che il **Client Key Exchange** potrebbe essere utilizzato solamente per inviare la **pre-master secret** al server, ovvero non includere la versione **TLS** dichiarata dal client, rendendo di fatto il **Downgrade attack** realizzabile su alcune istanze **TLS**.

#### 10.8.4 Truncation Attack

Il **truncation attack** si basa sulla vulnerabilità a **DoS Attacks**<sup>14</sup> del sottostante protocollo TCP che si occupa di trasportare il **protocollo TLS**. In particolare, l'attacco è caratterizzato nel seguente modo:

- **Obiettivo:** Terminare una connessione **TLS** tra client e server
- **Abilità:** inviare segmenti **TCP contraffatti (spoofing attack)**

L'attaccante può inviare semneti TCP-FIN contraffatti per far terminare la connessione TCP sottostante che trasporta il protocollo TCP.

Due possibili correzioni per proteggere delle applicazioni non consapevoli dei **protocolli sottostanti utilizzati** si basano sull'uso della **tecnica del tunnelling**, e sono:

##### Teorema 10.5 (TCP over TLS over TCP)

*la connessione TCP utilizzata dall'applicazione viene trasportata all'interno del protocollo TLS, a sua volta trasportato da una seconda istanza TCP. L'attacco risulta perciò trasparente all'applicazione, in quanto viene interrotta la seconda istanza TCP ma non quella utilizzata dall'applicazione stessa.*



**Nota** L'uso di una doppia connessione TCP degrada le prestazioni della connessione complessiva in quanto gli algoritmi di congestion control delle diverse istanze del protocollo possono andare in conflitto tra loro. Inoltre, se la soluzione prevede anche differenti connessioni TCP tra i nodi intermedi della comunicazione, tale problematica viene amplificata.

##### Teorema 10.6 (TCP over DTLS over UDP)

*la connessione TCP utilizzata dall'applicazione viene trasportata all'interno del protocollo DTLS, a sua volta trasportato da una seconda istanza UDP. L'attacco viene vanificato dall'assenza di una connessione reale in quanto il trasportatore finale è il protocollo UDP.*



**Nota** L'uso del protocollo UDP rende inaffidabile la consegna dei segmenti UDP, sebbene permetta di evitare conflitti tra le varie istanze del protocollo e migliorare quindi le performance.

**Osservazione:** Il protocollo TLS attenua gli effetti che il Truncation attack può generare utilizzando i warning alert di **Close Notify**, così che le parti possano distinguere tra **connessioni TLS terminate legittimamente** e **connessioni TLS terminate per cause esterne**. Resta comunque debole a tale attacco.

<sup>14</sup>Denial of Service

### 10.8.5 Renegotiation Attack

Il Renegotiation attack si basa sulle vulnerabilità offerte dalla rinegoziazione TLS def 10.16. L'attacco è caratterizzato nel seguente modo:

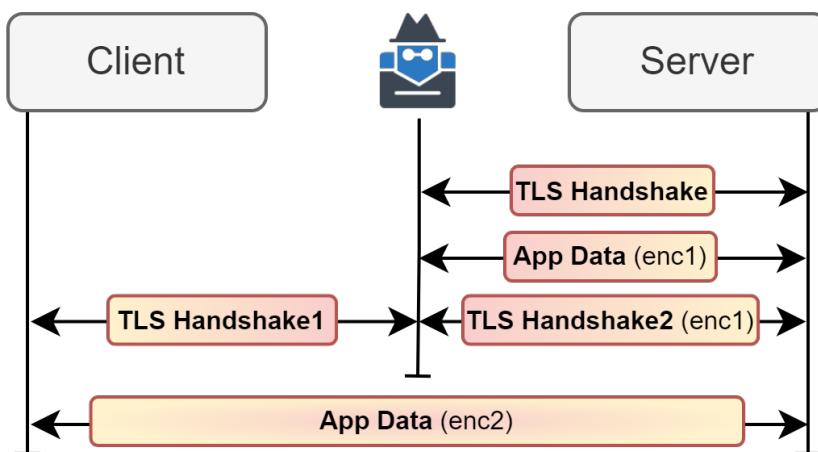
- **Obiettivo:** ottenere **informazioni sui dati applicativi** protetti tramite TLS.
- **Abilità:** è in grado di **Innescare una rinegoziazione (MITM)**, effettuare **plaintext injection attack al server (CPA)**.

#### Definizione 10.25 (Renegotiation Attack)

Assumiamo che:

1. Se il client invia un **client hello** prima dell'attaccante, quest'ultimo lo **intercetta e lo ritarda** fino al termine del **plaintext injection**.
2. L'attaccante avvia un **handshake** a nome del client, sfruttando l'assenza di cipher suite e l'opzionalità dell'autenticazione del client.
3. L'attaccante effettua (preable) **plaintext injection** sul server, inviando dati malevoli.
4. L'attaccante permette al client di effettuare l'**handshake** e di inviare i propri dati applicativi, che verranno accodati a quelli precedentemente inviati dall'attaccante.
5. L'attaccante visualizza l'esito dell'attacco<sup>15</sup>

<sup>15</sup>Varia in funzione dell'applicazione e dell'attacco stesso.



**Figura 10.15:** Renegotiation Attack

Il renegotiation attack è stato corretto in TLS v1.2 con la **TLS Renegotiation Extension**, che prevede l'aggiunta di un campo **reneg** all'interno del **client hello**, utilizzata nel seguente modo.

1. Nella **prima negoziazione**: il campo contiene un byte pari a 0 per indicare che è il **primo handshake completo** per la corrente sessione TLS.
2. Nelle **rinegoziazioni**: contiene l'ultimo **finished message** del precedente handshake, così da indicare implicitamente l'**avvenimento di una rinegoziazione** e legare crittograficamente tra loro le **rinegoziazioni successive**. Ciò permette di **rilevare** l'avvenimento di un attacco.

 **Nota** La rinegoziazione è stata **disabilitata** in TLS v1.3 poiché considerata troppo vulnerabile.

## 10.9 TLS v1.3

La versione più recente e sicura del protocollo TLS presenta delle differenze sostanziali, tant'è che potremmo definirla una 2.0. Le principali sono:

### Teorema 10.7 (TLS v1.3 "Changelog")

- E' ammesso **soltanto** l'uso di cipher suite AEAD, aventi formato: **TLS\_AEAD\_HASH**.
  - La funzione hash crittografica viene utilizzata per il PRF.
  - Neutralizza il Padding Oracle Attack poiché tutto viene autenticato.
- La funzione PRF utilizzata è di tipo **HKDF** (def 10.12)
- Per effettuare Key Exchange viene ammesso **soltanto** Ephemeral DH (def 9.5)
  - Permette di garantire **Perfect Forward Secrecy**
  - Neutralizza **BEAST Attack** (def 10.21)
  - Neutralizza **Bleichenbacher's Oracle Attack** (def 9.9)
- Non è ammessa rinegoziazione
  - Neutralizza **Renegotiation Attack** (def 10.25)
- Non è ammessa compressione
  - Neutralizza **CRIME Attack** (def 10.23)
- Ottimizza il processo di handshake, utilizzando un approccio 3-way.
- Viene rimosso il change cipher spec protocol
- Semplificazione nella gestione delle curve Ellittiche per la crittografia.
- Exported Key: Permette di esportare all'applicazione un ulteriore segreto condiviso.
  - Tale segreto è differente sia al pre-master che al master secret.
  - Permette di prevenire soluzioni proprietarie per la crittografia.



## 10.10 Perfect Forward Secrecy

La **PFS** è un requisito di sicurezza per il quale:

### Definizione 10.26 (Perfect Forward Secrecy)

- La compromissione di una chiave di sessione non deve compromettere le informazioni scambiate prima e dopo l'utilizzo della chiave stessa.
- La compromissione di una chiave privata (*long-term secret*) non deve compromettere le informazioni scambiate precedentemente alla compromissione stessa.



**Osservazione:** L'unione di questi due requisiti implica che le chiavi di sessione non sono compromesse neanche nel caso in cui venga compromesso il segreto a lungo termine utilizzato per la derivazione nella fase di handshaking.



La PFS offre quindi protezione verso un attaccante in grado di:

- **Memorizzare** una **grande quantità di dati**, ovvero in grado di salvare tutti i dati criptati scambiati durante la comunicazione e per un periodo di tempo molto lungo.
- **Rompere** eventualmente una **coppia di chiavi pubblica e privata**, ovvero di scoprire la chiave privata associata alla chiave pubblica utilizzata nella fase di handshaking.

#### Teorema 10.8 (PMS obtained by EDH)

*La PMS è garantita dall'uso di EDH (def 9.5).*



Infatti, nel caso in cui le chiavi segrete  $SK_C, SK_S$  vengono scoperte l'attaccante non ottiene informazioni utili a calcolare il pre-master secret, al quale vengono derivate le chiavi di sessione necessarie alla decriptazione.

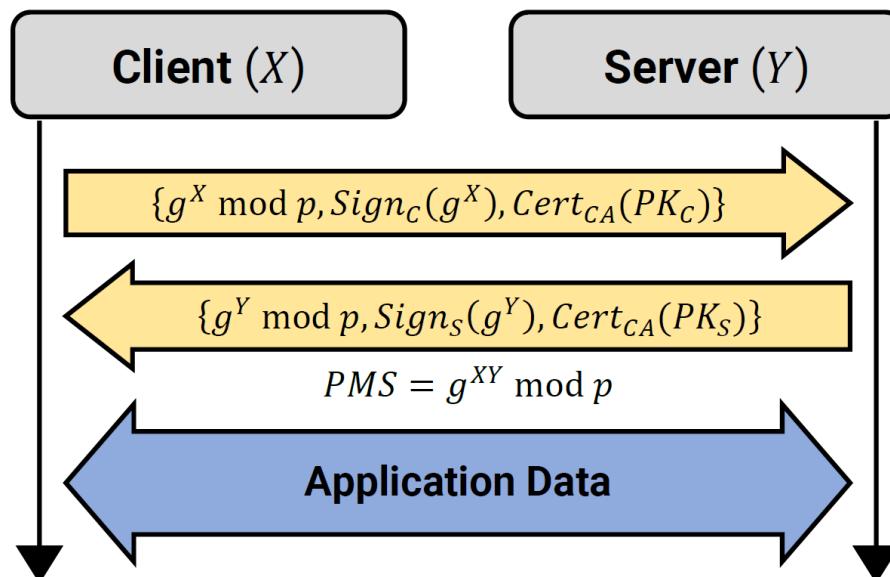
**Osservazione:** Se l'attaccante è passivo, non può ottenere le chiavi di sessioni, né passate né future. ■

**Osservazione:** Se l'attaccante è attivo, non può ottenere le chiavi di sessione passate, ma può effettuare attacchi MITM dopo la scoperta delle chiavi private. ■

Per quanto riguarda gli altri protocolli di key exchange, si ha che:

- **RSA Key Transport:** nel caso in cui la chiave privata  $SK_C$  del client viene scoperta, un attaccante può ottenere tutti i PMS generati nel tempo, in quanto criptati dalla relativa chiave pubblica  $PK_C$
- **Fixed DH Key Agreement:** il PMS è fisso, in quanto prodotto dai valori privati  $X, Y$  costanti. Nel caso in cui anche solo **uno** dei due valori privati venga scoperto, un attaccante è in grado di calcolare direttamente il PMS.

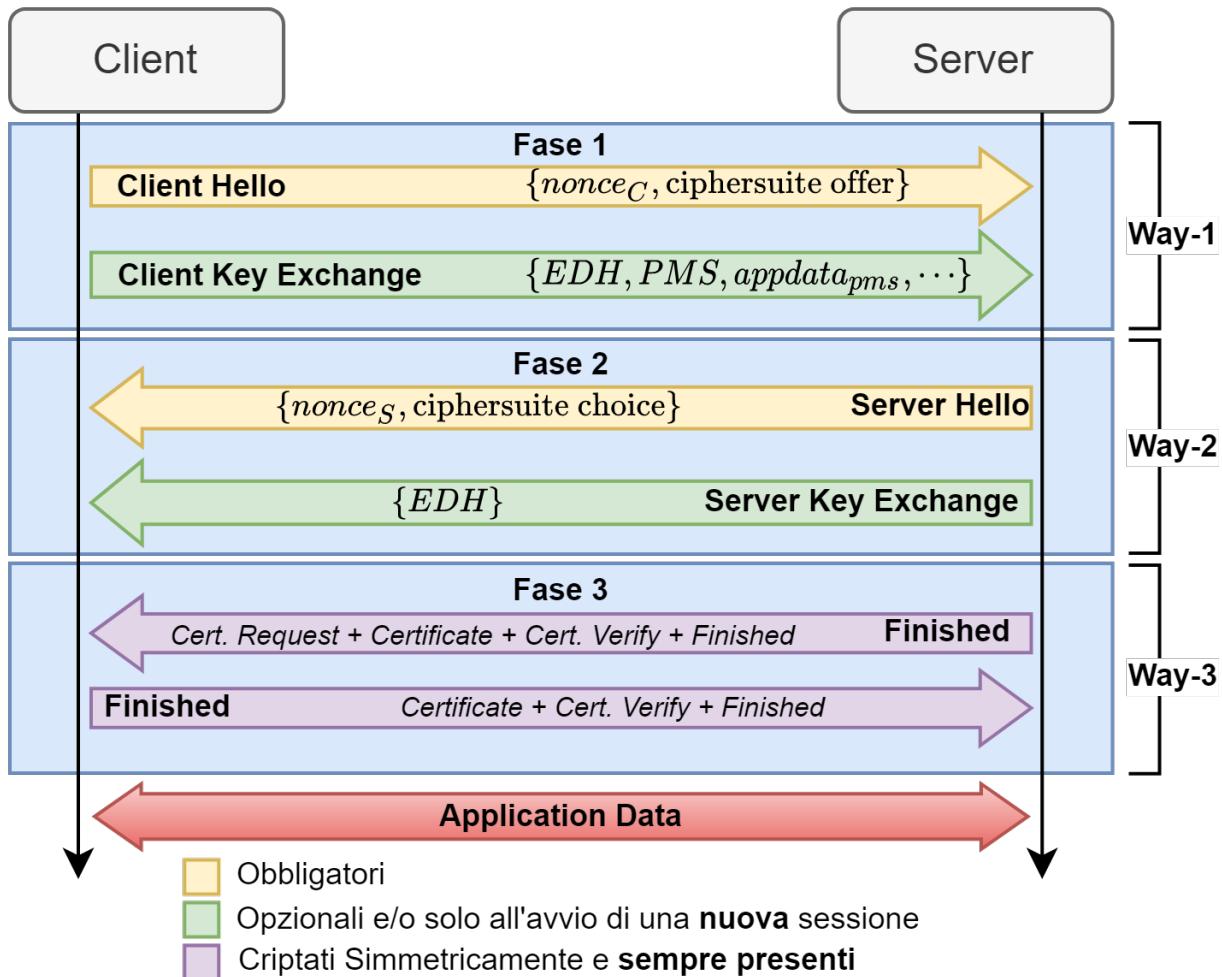
In Entrambi i casi, dato che le nonces sono inviate in chiaro e le funzioni PRF di dominio pubblico, una volta noto il PMS un attaccante può **ricostruire** tutte le chiavi di **sessioni passate e potenzialmente future** (a meno di fix eventuali) e **decriptare** l'intera comunicazione TLS.



**Figura 10.16:** Ephemeral Diffie Hellman in TLS v.1.3 during Key Exchange

### 10.10.1 Handshake

L'handshake completo in v1.3 è composto dai seguenti messaggi:



**Figura 10.17:** TLS v1.3 Handshake Protocol

**Osservazione:** Come possiamo vedere, avendo **obbligato** l'uso di **EDH** per il key agreement, l'**handshake** ne risulta più **snello e veloce** in quanto il **client non deve più aspettare la chiave pubblica del server**, ottenuta tramite il relativo certificato.

Inoltre, ci viene garantita la PMS e possiamo criptare immediatamente i certificati, per garantire *identity protection* ■

## 10.10.2 Perfect Forward Secrecy con Pre-Shared Key

TLS v1.3 permette di garantire PFS anche con uso esclusivo **pre-shared key**, ovvero, permette di **derivare chiavi di sessione** in maniera **sicura e senza usare** certificati e firme digitali.

### Teorema 10.9 (PMS with Pre-Shared Key)

*Sia PSK una pre-shared key nota sia al client che al server.<sup>16</sup>*

*Le chiavi di sessione vengono calcolate usando le nonces e Anonumous DH (def 9.3):*

$$\text{Keys} = \text{HKDF}(\text{PSK}, \text{label}, \{\text{nonce}_C, \text{nonce}_S, g^{xy} \bmod (P)\})$$

<sup>16</sup>Può essere installata offline o determinata alla prima connessione TLS tra le due parti. □



**Nota** Nel caso in cui venga scoperta la PSK, un attaccante **non è in grado** di recuperare le chiavi di sessioni passate in quanto i valori segreti  $X, Y$  cambiano ad ogni rekeying e **non è computazionalmente** in grado di **calcolarli** a partire dalla **conoscenza** di  $g^{xy} \bmod (P)$



**Nota** La PSK ha caratteristiche simili al PMS, ma:

- Viene utilizzato differentemente nella generazione delle chiavi.
- Non viene rigenerato ad ogni nuova sessione TLS.

## 10.10.3 0-RTT Data

Una caratteristica di TLSv1.3 è la **possibilità** di effettuare **0-RTT Data**:

### Definizione 10.27 (0-RTT Data)

E' una modalità di trasmissione che permette di inviare immediatamente dati applicativi nel primo messaggio di handshake, nel caso in cui sia già stata stabilita la pre-shared key PSK. □

Le **chiavi di sessioni** utilizzate per criptare i dati con 0-RTT vengono generati nel seguente modo:

### Definizione 10.28 (0-RTT KeyGen)

1. Durante la **way-1** dell'handshake le chiavi di sessione utilizzate sono calcolate come:

$$\text{Keys} = \text{HKDF}(\text{PSK}, \text{label}, \{\text{nonce}_C, \text{nonce}_S, g^x \bmod (P)\})$$

2. Dalla **way-2** dell'handshake in poi le chiavi di sessione utilizzate sono calcolate come:

$$\text{Keys} = \text{HKDF}(\text{PSK}, \text{label}, \{\text{nonce}_C, \text{nonce}_S, g^{xy} \bmod (P)\}) □$$

L'inclusione dei coefficienti DH permette di rispettare le condizioni necessarie alla PFS. Tuttavia, le **chiavi temporanee** di sessione utilizzate nel primo messaggio **non** includono la **nonce** ed il coefficiente del server, per cui la way-1 dell'handshake è vulnerabile a replay attack. Alcune possibili mitigazioni sono:

- **Inserire una finestra di validità** per la **pre-shared key** così da limitare l'intervallo temporale in cui possono avvenire replay attack sull'handshake.
- **Controllare il riuso di nonce** da parte del client, così da individuare replay attacks. Tuttavia, tale soluzione è poco scalabile.

# Capitolo IPSec: IP Security

IPSec è un altro protocollo di sicurezza che non lavora più sul garantire una sessione sicura, ma si impegna per proteggere **tutto** il pacchetto IP. Spesso il protocollo viene associato alle VPN, in quanto ne permette la costruzione. Ne vedremo poi una parentesi.

## 11.1 Protocol Structure

Il protocollo IPSec è detto anche **Network Layer Security** ed è un protocollo di **livello di rete** (livello 3) nell'architettura TCP/IP, che permette di **proteggere** sia l'**header** che il **payload** dei pacchetti IP. La protezione è quindi direttamente sull'**host** e basta sugli **indirizzi IP**.

Può essere implementato nei seguenti modi:

### Teorema 11.1 (IPSec implementation)

- **Native IP Code:** viene incluso all'interno del protocollo IP. Tipico delle soluzioni UNIX.
- **BITS (Bump in The Stack):** viene implementato come **protocollo intermedio** tra il livello di rete (protocollo IP) ed il livello di collegamento, agendo come driver intermedio.
- **BITW (Bump in The Wire):** viene implementato all'interno della **NIC** (Network Interface Card) del sistema dell'host, ovvero all'interno dell'**hardware**.

**Osservazione:** IPSec **coopera** e **coesiste** con il protocollo IP, per cui nella **comunicazione** tra due hosts vengono utilizzati sia **pacchetti IP protetti da IPsec** che **pacchetti IP non protetti**. E' inoltre **trasparente al livello applicativo** e non può proteggere direttamente le applicazioni. ■

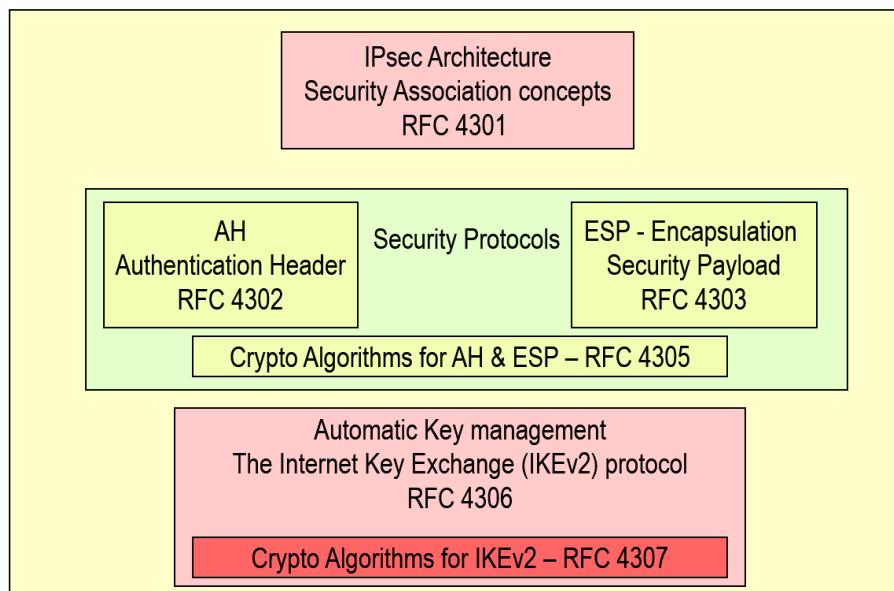


Figura 11.1: IPSec Architecture

### Definizione 11.1 (IPSec Architecture)

Nel protocollo IPSec distinguiamo le seguenti componenti: (fig. 11.1)

- **Security Association:** identifica una connessione da un host mittente ad un host destinatario. Definisce perciò l'architettura della comunicazione tra hosts.
- **Security Protocols:** protocolli per il trasferimento dei dati che garantiscono specifiche proprietà di sicurezza.
- **Key Management:** protocolli per l'instaurazione della connessione tra due hosts.



#### 11.1.1 Security Association

Una **Security Association (SA)** rappresenta una **connessione unidirezionale sicura** tra un **IP mittente** ed un **IP destinatario**, ovvero **definisce i confini entro i quali i pacchetti IP vengono protetti tramite criptazione e autenticazione**. Tali confini possono essere:

- Host to Host.
- Host to Intermediote Router (security Gateway).
- Security Gateway to Security Gateway.



**Nota** Nella SA non c'è trasmissione dati, ma soltanto una serie di azioni per inizializzare parametri di sicurezza.

### Proposizione 11.1 (SA Setup)

Le SA forniscono dei meccanismi di configurazione di due tipi:

- **Manualmente:** tutti i parametri della SA sono configurati in maniera manuale. Ne deriva che tale metodo è adatto solamente per VPN di piccole dimensioni.
- **Automaticamente:** vengono gestite tramite il protocollo IKEv2. In particolare, è possibile generare nuove SA su richiesta e gestire i parametri di sicurezza per sessioni.



Per poter gestire le SA e il traffico di dati, ogni entità mantiene:

### Definizione 11.2 (Un Security Association Database (SAD))

Il SAD mantiene le informazioni delle SA attive sia per il traffico entrante che uscente dall'entità.

Per ogni SA vengono memorizzati i seguenti parametri:

- **Security Parameters Index (SPI):** (32 bits) identifica univocamente la SA.
- **Lifetime:** tempo di vita della SA;
- **Indirizzo di Origine:** indirizzo IP del mittente
- **Indirizzo di Destinazione:** indirizzo IP del destinatario.
- **Security Protocol:** protocollo di sicurezza IPsec utilizzato per la SA.
- **Sequence Number Counter:** numero di sequenze per i pacchetti IPsec inviati sulla SA.
- **Anti-Replay Window:** definisce la finestra di pacchetti per la quale vengono contrastati replay attacks.
- **Algoritmi:** algoritmi utilizzati dalla SA per la criptazione ed il controllo di integrità.
- **Chiavi:** chiavi simmetriche utilizzate dalla SA per la criptazione ed il controllo di integrità.



**Definizione 11.3 (Un Security Policy Database (SPD))**

L'SPD mantiene un insieme di Security Policy (SP) per determinare le modalità con le quali un pacchetto che transita attraverso l'entità deve essere processato. In particolare, l'insieme di regole espresse all'interno del SPD permette di determinare se un pacchetto debba essere:

- **Ignorato:** il pacchetto IP/IPsec in transito **non** deve essere elaborato dal protocollo IPsec.
- **Proteggi:** il pacchetto IP/IPsec **dove** deve essere elaborato dal protocollo IPsec.
- **Scartato:** il pacchetto IP/IPsec **dove** deve essere scartato/bloccato dall'entità.

Per i Pacchetti protetti da IPsec, il SPD definisce le modalità di elaborazione in funzione delle caratteristiche del pacchetto (header IP e relazioni con il traffico di rete). Alcune di queste modalità sono:

- **Quale protocollo di sicurezza IPsec** deve essere utilizzato e con quale **modalità**.
- **La granularità** con la quale devono essere gestiti i pacchetti.
  - **Un singolo Tunnel** condiviso per tutte le connessioni.
  - **Un tunnel per ogni** differente connessione TCP.



## 11.1.2 Security Protocol

IPsec prevede due differenti protocolli di sicurezza:

**Definizione 11.4 (Authentication Header)**

Sia su payload che header dei pacchetti IP viene fatto controllo di integrità inserendo un tag tra IP header e payload.

Viene fornita **protezione contro replay attacks**

**Definizione 11.5 (Encapsulated Security Payload)**

Viene fornita autenticazione e confidenzialità con AEAD del solo payload del pacchetto IP, che viene rinchiuso da un nuovo header (inserito tra header e payload) e da un trailer più tag di auth.

Viene fornita protezione contro replay attacks e traffic flow confidentiality.



**Osservazione:** Nelle ultime RFC AH è stato declassato da "Must" a "May" poiché è stato visto che ESP copre la maggior parte dei requisiti necessari ai servizi di sicurezza. Raramente vengono usate anche in combinazione.

IPsec ha la caratteristica di poter operare in due modalità differenti:

**Definizione 11.6 (IPsec Transport)**

Il pacchetto IP protetto da IPsec viene elaborato direttamente dall'host.

**Osservazione:** Se viene usato AH e l'ip viene cambiato il pacchetto viene scartato.



### Definizione 11.7 (IPsec Tunnel)

Il pacchetto IP protetto da IPsec viene **incapsulato all'interno di un altro pacchetto IP** dopo aver aggiunto un tag che protegge tutto il pacchetto.

**Osservazione:** In tunneling l'IP del pacchetto che viene consegnato è quello del tunnel ed è diverso da quello realmente creato a livello di trasporto.

La modalità di lavoro del protocollo viene determinata in base all'esigenza dell'applicazione:

### Teorema 11.2 (IPsec mode selection)

- **Transport Mode** per connessioni **host-to-host o host-to-gateway**. In alcuni casi anche per connessioni **gateway-to-end o end-to-gateway**
- **Tunnel Mode** per connessioni **gateway-to-gateway**.

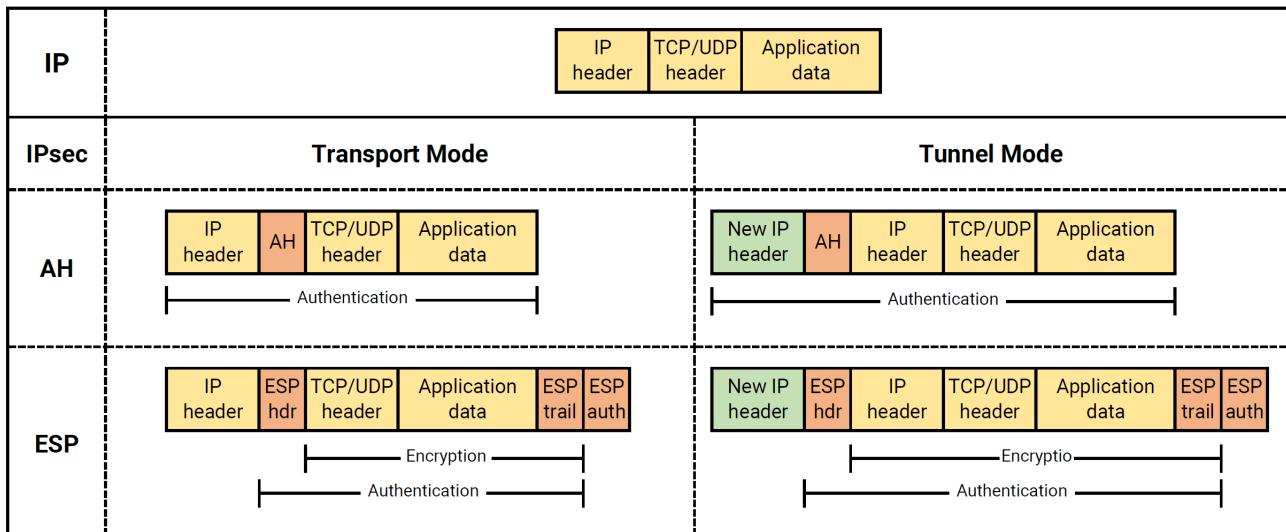


Figura 11.2: IPsec Mode of Operation

Version	Hdr len	Type of Service	Total Length					
Identification		Flags	Fragment Offset					
Time To Live	Protocol = 50		Header Checksum					
Source IP Address								
Destination IP Address								
Options (if any)								
Next header	AH payload len	Reserved						
Security Parameters Index (SPI)								
Sequence Number Field								
Integrity Check Value (ICV)								
Payload								

(a) Authentication Header

Version	Hdr len	Type of Service	Total Length					
Identification		Flags	Fragment Offset					
Time To Live	Protocol = 50		Header Checksum					
Source IP Address								
Destination IP Address								
Options (if any)								
Security Parameters Index (SPI)								
Sequence Number Field								
Payload...								
...Payload		Padding...						
...Padding		Pad length	Next header					
Integrity Check Value (ICV)								

(b) Encapsulated Security Payload

Figura 11.3: IPsec Packet Format

L'integrità dei pacchetti viene controllata nel seguente modo:

**Teorema 11.3 (IPsec Integrity Check)**

Viene generato un MAC contenuto nel campo **ICV (Integrity Check Value)** nell'header AH/trailer ESP nel seguente modo:

- Se viene usato AH, include tutti i campi non mutabili/prevedibili dell'header IP<sup>1</sup>
- Vengono inclusi per intero l'header AH/ESP ed il payload IP

<sup>1</sup>Ad esempio il Time-to-Live (TTL) e l'header checksum cambiano spesso. Per questi campi viene aggiunta una maschera di bit e poi calcolato il tag. Se viene usato AH, faccio MAC prima della frammentazione e ricontrollo al riassemblaggio.



**Nota** Al fine di proteggere IPsec da replay attacks sia AH che ESP usano un **extended sequence number (ESQN)** per numerare i pacchetti.

**Definizione 11.8 (ESQN)**

Numero complessivamente di 64bit, di cui:

- 32 bit meno significativi trasmessi con il **campo sequence number** nell'header AH/ESP.<sup>2</sup>
- 32 bit più significativi mantenuti e aggiornati autonomamente dagli estremi.
- Viene sempre inizializzato a 0 per una nuova SA
- Quando viene raggiunto il **massimo valore esprimibile**, la SA viene terminata.

<sup>2</sup>Il protocollo IP non prevede un campo apposito per tale funzione.

Viene poi introdotta una strategia anti-replay per determinare quali sequence number il destinatario reputa non validi.

**Teorema 11.4 (Anti-Replay Policy)**

Viene usata una **finestra scorrevole (sliding window)** così fatta:

- La finestra (di ricezione) ha **dimensione W** definita dal ricevitore.
- Il **margine destro** è dato dal **numero di sequenza più alto ricevuto**. Poiché W fissato, il margine sinistro è **univocamente determinato**.
- Alla **ricezione di un pacchetto** si applicano le seguenti regole:
  - **Duplicato o a sinistra di W**: scarto il pacchetto.
  - **All'interno di W e NON duplicato**: ricevo il pacchetto.
  - **A destra di W ed è contestualmente<sup>3</sup> possibile spostare in avanti W**: ricevo il pacchetto, altrimenti lo scarto.

<sup>3</sup>E' possibile scorrere W in avanti di n posizioni solamente se sono stati già ricevuti tutti i pacchetti nelle prime n posizioni di W.



**Nota** E' stato osservato che il traffico prodotto durante la comunicazione con un determinato server possiede una firma statistica che permette di distinguerlo dagli altri server. In particolare, esaminando le proprietà dei pacchetti trasmessi durante l'inizio di una nuova connessione con il server è possibile tracciare delle caratteristiche come:

- Dimensione dei pacchetti scambiati.
- Tempo di inter-arrivo tra i pacchetti.
- Presenza o assenza di determinati tipi di pacchetti.

Dalle firme statistiche del traffico, un attaccante può determinare con quale server il client comunica, solamente analizzando il traffico generato.

**Osservazione:** Sarebbe possibile generare dei collegamenti tra client e server che possono essere usati per strutturare attacchi più complessi. ■

Introduciamo quindi un nuovo requisito di sicurezza.

#### Definizione 11.9 (Traffic Flow Confidentiality)

E' un servizio di sicurezza che ha come obiettivo l'offuscamento della firma del traffico □

ESP ha due differenti meccanismi per contrastare l'analisi del traffico:

#### Proposizione 11.2 (ESP Traffic Flow Conf.)

- E' possibile alterare la dimensione dei pacchetti aggiungendo del padding.
  - ✓ facilmente gestibile utilizzando tunnel mode.
  - ✗ Il padding TCF potrebbe indebolire gli algoritmi di criptazione.
- E' possibile generare pacchetti fantoccio (dummy) da immettere



## 11.2 IKEv2 - Internet Key Exchange

#### Definizione 11.10 (IKEv2)

Il protocollo IKEv2 (Internet Key Exchange v2) permette di creare molteplici SA in maniera sicura tra due peers/hosts. In particolare, si distinguono:

- **Initiator:** host/peer che genera ed invia una richiesta. In genere corrisponde con il client.
- **Responder:** host/peer che riceve una richiesta IKE. In genere corrisponde con il server.
- **IKE SA:** Security Association creata ed utilizzata dal protocollo IKE per messaggi di controllo. Ne viene generata solamente una per istanza IKE.
- **Child SA:** Security Association creata per effettuare lo scambio dati tra i peers utilizzando il protocollo IPsec AH/ESP. Da un'unica istanza del protocollo IKE possono essere create molteplici Child SA per servire scopi differenti.



### 11.2.1 Funzionamento

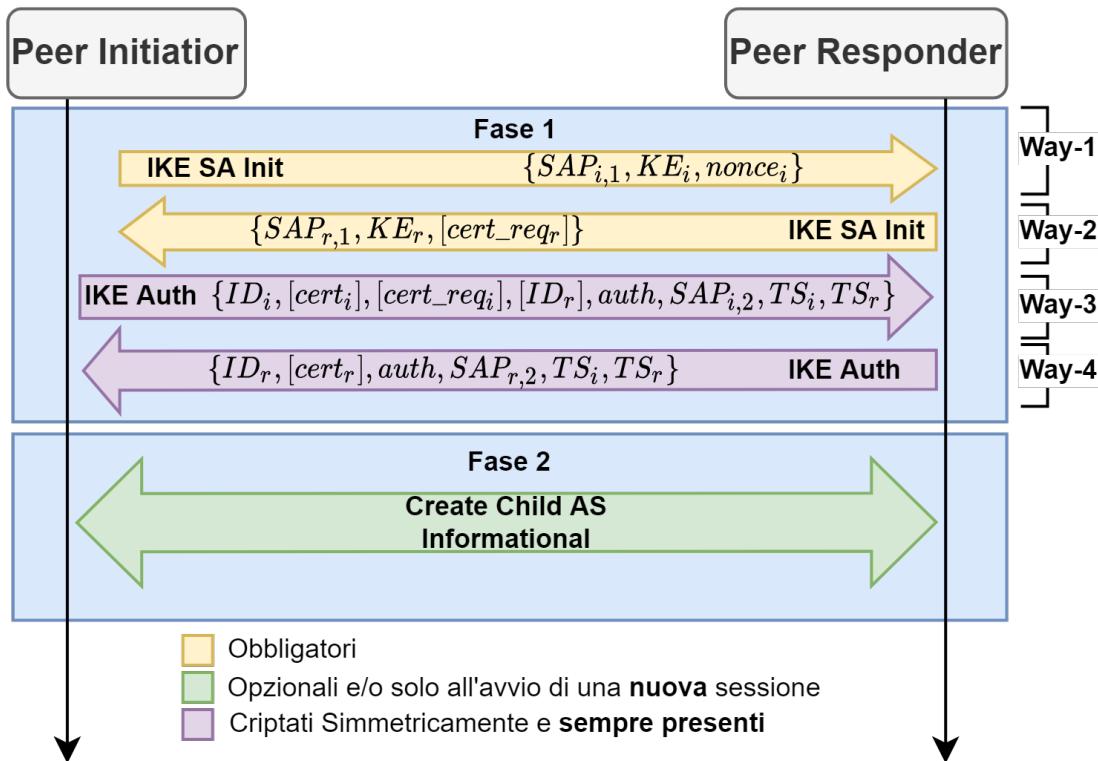
Il protocollo IKE è strutturato in 2 fasi:

#### Proposizione 11.3

1. **Handshake 4-way:** viene create la **IKE SA** e vengono scambiati due tipi di messaggi:
  - **IKE SA Init:** messaggi in chiaro utilizzati per **negoziare i parametri IKE**<sup>4</sup> e **inviare le nonces ed i valori DH**<sup>5</sup>
  - **IKE Auth:** messaggi **criptati simmetricamente e autenticati** per **confermare l'avvenuta negoziazione**. Permettono anche di **contrastare downgrade attacks**.
2. **Child SA:** (opzionale) vengono create delle **SA child** e vengono inviati messaggi IKE di controllo.

<sup>4</sup>Security Association Parameters: **SAP**

<sup>5</sup>Key Exchange Values **KE**



**Figura 11.4:** IKE 4-way Handshake

I messaggi hanno il seguente formato:

#### Definizione 11.11 (IKE Msg Format)

- **IKE Header:** indica il tipo di messaggio IKE.
- **IKE Payloads:** lista collegata di payload differenti. Ogni singolo payload possiede un proprio formato ed una propria funzione. Tale struttura flessibile permette di estendere rapidamente un messaggio IKE e, di fatto, le funzionalità del protocollo IKE stesso.

**Osservazione:** Questi messaggi sono trasportati tramite **UDP**, reso affidabile dal protocollo IKE stesso, ed inviati tramite protocollo IP.



### 11.2.2 Key Computation



**Nota** Nelle versioni più recenti vengono usate solo **HKDF** in quanto più sicure. Vediamo però come venivano costruite:

Le chiavi della **sessione IKE** vengono generate anche queste con approccio **Extract-than-Expand** def 10.11 tramite delle **PRF Negoziate** nella fase 1, utilizzando le **nonces** e i **dh coefficient** scambiati con i **messaggi di IKE SA Init**.

#### Definizione 11.12 (IKE Key Computation)

- **Extract:**  $SK_{seed} = PRF(nonce_i, nonce_r, g^{ir})^6$
- **Expand:**  $Keys = PRF^+(SK_{seed}, nonce_i, nonce_r, SPI_i, SPI_r)$

E vengono generate 7 chiavi di sessione:

- $SK_{ai}, SK_{ar}$  per il controllo di integrità.
- $SK_{ei}, SK_{er}$  per encryption e decryption.
- $SK_{pi}, SK_{pr}$  per la **generazione** di un **AUTH payload** nella fase **SA\_AUTH**.
- $SK_d$  per la derivazione di ulteriori chiavi per le **Child\_SA**.

<sup>6</sup> $g^{ir}$  è il DH shared key.



## 11.3 Vulnerabilità

IPsec non è immune ad attacchi e qui ne consideriamo due tipi:

- Downgrade Attacks
- DoS Attacks

Vediamo come IPsec si è protetto da questi attacchi.

### 11.3.1 Against Downgrade Attack

Viene introdotto un **version flag V** in tutti i messaggi scambiati. Tale flag indica se la **versione del protocollo negoziata** non è quella **massima disponibile**.

### Definizione 11.13 (IPsec Version Flag)

- $V = 0$  se il **mittente** del messaggio sta **negoziando la propria versione massima**.
- $V = 1$  altrimenti, ovvero se il **mittente** del messaggio sta **negoziando una versione inferiore alla propria versione massima disponibile**.

□

Un attaccante MITM può modificare sia la versione che il version flag all'interno dei messaggi di IKE SA Init per effettuare un downgrade. Tuttavia, l'attaccante non potrà alterare i messaggi di IKE SA Auth nel quale è contenuto il version flag realmente dichiarato, poiché protetti tramite criptazione e autenticazione.

**Osservazione:** Sia l'initiator che il responder possono **rilevare l'attacco** semplicemente **verificando** che i **version flag** sono per **entrambi** pari a 1, ovvero, **nessuno dei due sta utilizzando la propria versione massima**. La versione ottima per la sicurezza è infatti data dalla minima tra le due versioni massime disponibili.

■

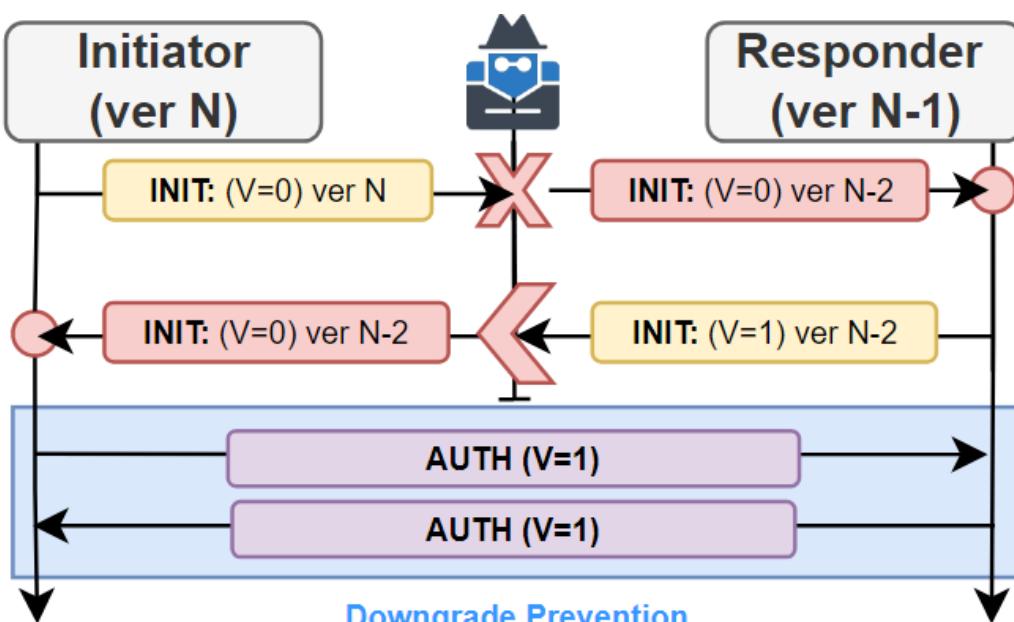


Figura 11.5: Downgrade Attack and Prevention in IKEv2

### 11.3.2 Against DoS Attacks

Il DoS (Denial of Service) attack utilizza delle **spoofed IKE SA Init** per **saturare le capacità computazionali del server** e renderlo **inaccessibile** agli utenti legittimi. Nel caso del protocollo IKE viene sfruttata l'**onerosità delle computazioni DH** svolte **dopo i primi due messaggi di IKE SA Init**. Il protocollo IKE viene protetto da DoS attack utilizzando un **Init handshake 4-way** basato sull'uso di **cookie**. In particolare, vengono inviati i seguenti messaggi

#### Definizione 11.14 (IKE DoS Prevention)

*Prima dell'init vengono aggiunti due messaggi:*

1. **Request:** il client invia la **richiesta** di connessione al server.
2. **Response:** il server risponde al client **ripetendo** a richiesta ed inviando un **cookie**.
3. **IKE SA Init (Initiator):** il client deve rispondere con un messaggio IKE SA Init in cui ripete la richiesta ed il cookie appena ricevuto dal server.
4. **IKE SA Init (Responder):** se la richiesta ed il cookie all'interno del messaggio IKE SA Init ricevuto corrispondono con quelli dei messaggi precedenti, il server continua con il protocollo IKE.

*Il server inizia le elaborazioni solamente al termine dell'Init handshake.* □



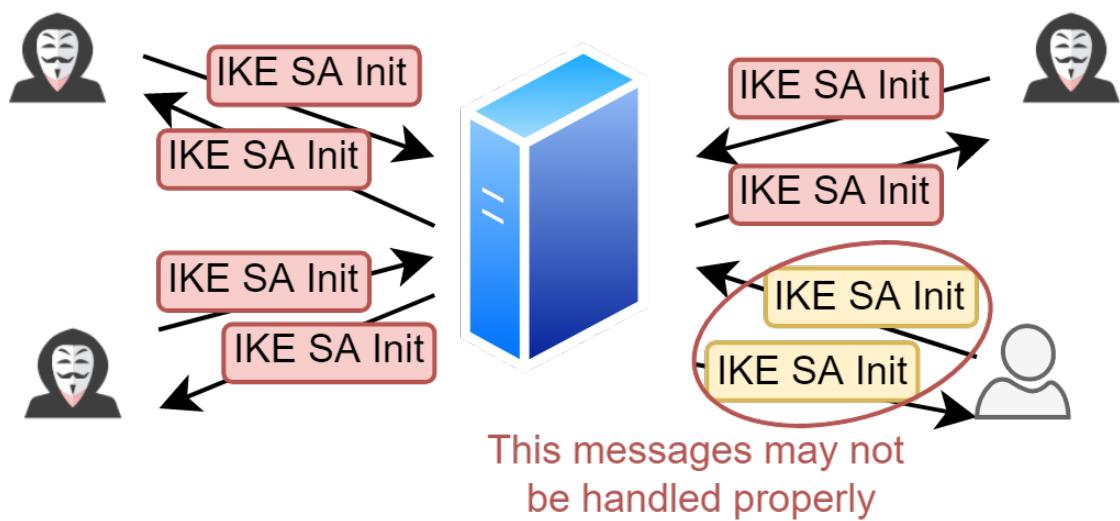
**Nota** Tale schema permette di contrastare attacchi DoS basati su IP spoofing in quanto l'attaccante non riceve il messaggio di **Response**, indirizzato verso l'indirizzo IP spoofed diverso dal suo, e non quindi può continuare l'handshake.

#### Corollario 11.1 (Cookie for Handshake 4-way)

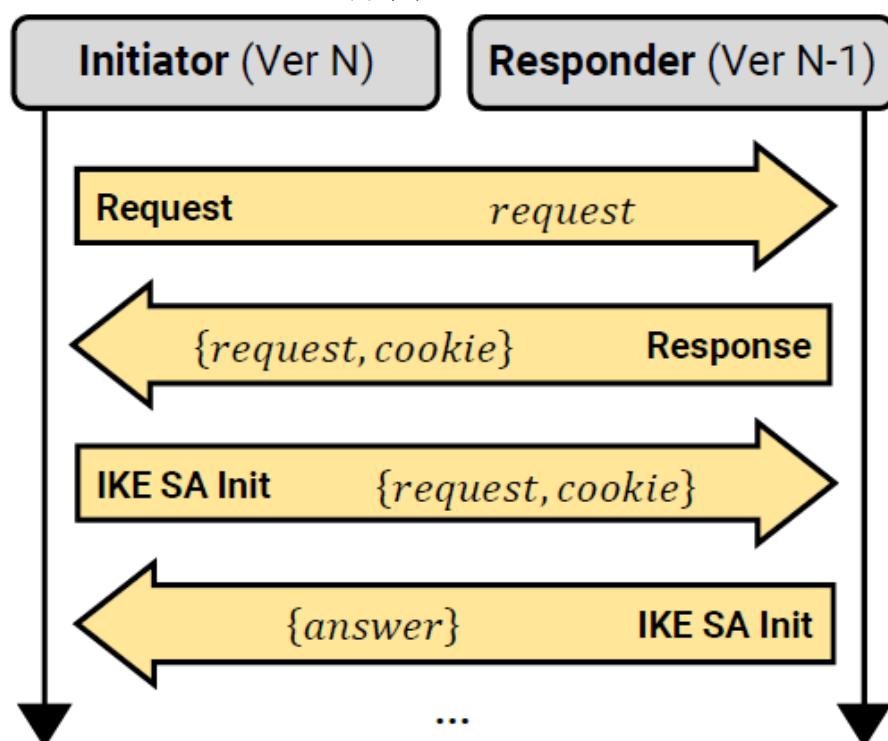
I **cookie** devono essere stateless: **casuali, non ripetuti e non predibili**. Sono possibili due approcci:

- **Generazione Diretta:** i cookie vengono generati come stringhe casuali.
  - **Poco Scalabile** in termini di **memoria**, in quanto occorre memorizzare tutti i cookie generati in precedenza per evitare di effettuare ripetizioni.
  - **E' prone ad errori implementativi** per i quali i cookie vengono ripetuti o sono predibili.
- **Generazione tramite funzione hash:**  $\text{cookie} = H_k(\text{request})$ 
  - I cookie vengono generati usando una funzione hash crittografica  $H$ , una chiave segreta  $K$ , nota solamente al server e l'intero messaggio di richiesta per i quali vengono generati.
 

Corrispondono al MAC per la message authentication dei messaggi.
  - La **chiave segreta K** può essere **cambiata periodicamente** per fornire **maggior robustezza**.



(a) (D)DoS Attack



(b) 4-way Handshake with Request and Response

## 11.4 VPN

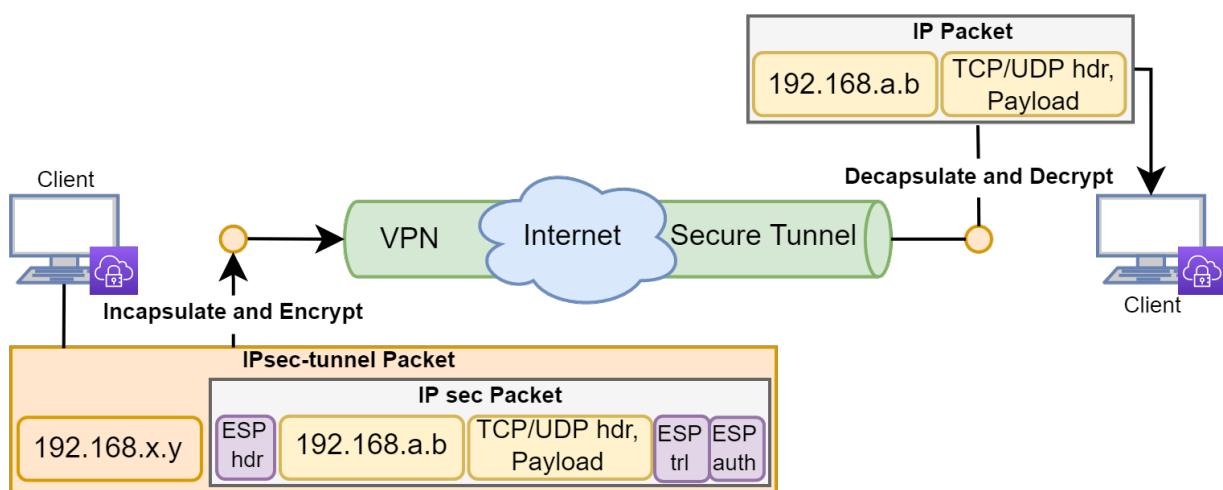
Una **Virtual Private Network** è una **rete virtuale privata** all'interno della rete pubblica che permette di

- **Proteggere i dati in transito nella VPN** dal resto della rete.
- **Connettere in maniera sicura** delle **sottoreti** geograficamente **distanti** tra loro.

Una VPN funziona con due meccanismi fondamentali:

- **Tunneling:** **incapsulare/trasportare pacchetti IP**, relativi alla **LAN**, all'interno di altri pacchetti IP, relativi alla **WAN**.
- **Encryption:** **criptare il payload** di pacchetti IP e quindi, **nel caso venga usato il tunneling**, criptare per **intero un pacchetto IP**.

Una VPN può chiaramente essere realizzata usando IPsec in modalità Tunnel (def 11.7).



**Figura 11.7:** VPN Scheme

# Capitolo Modern Secret Sharing

Nei precedenti capitoli abbiamo visto come condividere un segreto in modo *sicuro* usando **Diffie-Hellman** (def 9.3) oppure **RSA Key Transport** (def 9.6). In questi casi, la sicurezza proveniva dal fatto che i messaggi scambiati in rete implicavano una difficoltà enorme per un attaccante, che rendeva arduo il tentativo di determinare/rubare la chiave con cui i messaggi vengono cifrati prima di essere scambiati.

Vediamo come la crittografia può essere usata non solo per cifrare dei messaggi, ma anche per condividere un segreto.

**Esempio 12.1** Supponiamo di avere un *Dealer*  $D$  e due utenti  $A, B$ .  $D$  genera un segreto di 8 byte (un semplice numero), che spezza in due parti da consegnare agli utenti.

Per un segreto di questa dimensione, le probabilità di indovinare il segreto sono chiaramente  $1/2^8 = 1/256$ . Supponiamo ora che un *attaccante*  $E$  possa intercettare uno dei due messaggi e scoprire metà del segreto. Adesso le probabilità sono  $1/2^4 = 1/16$  ■

**Osservazione:** Dividendo il segreto originale e trasmettendone le due parti, ne riduciamo il livello di sicurezza. ■

Il metodo descritto nell'esempio 12.1 è tanto semplice quanto debole, pertanto è lecito chiedersi se sia possibile creare una versione robustificata.

## 12.1 Trivial Secret Sharing

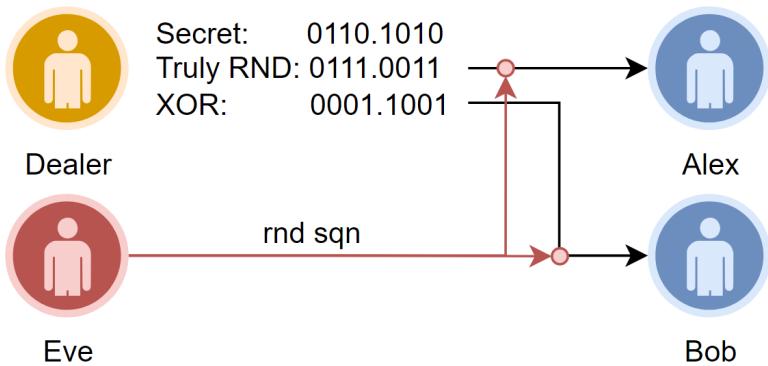
**Esempio 12.2** Consideriamo lo scenario dell'esempio 12.1 ma questa volta generiamo una sequenza **truly random** (una chiave) e facciamone lo xor con il segreto. Otteniamo così un **Vernam Cipher** (prop 1.2). Inviamo adesso ad  $A$  la chiave generata randomicamente e a  $B$  il risultato dello xor. Osserviamo che:

- Se  $E$  riuscisse ad intercettare il messaggio di  $A$ , non avrebbe alcun vantaggio in quanto il numero casuale da lui generato non contiene informazioni sul segreto originale.
- Se  $E$  riuscisse ad intercettare il messaggio di  $B$ , il quale contiene la chiave, non avrebbe alcun vantaggio in quanto l'operazione di xor effettuata con una quantità **truly random** genera un risultato altrettanto randomico, la cui probabilità di essere indovinato dopo aver ricevuto un messaggio non si abbassa e resta  $1/2^8 = 1/256$ . ■

Distinguiamo tra il segreto originale e le parti in cui viene suddiviso e consideriamo il teorema:

### Definizione 12.1 (Shares)

*Uno Share è una delle parti in cui viene suddiviso un segreto.* □

**Figura 12.1:** Trivial Secret Sharing**Teorema 12.1 (Modular Sum as XOR equivalent)**

L'operazione di XOR-ing non è necessaria. Si può dimostrare che al posto dello xor è possibile effettuare una **somma modulare**, poiché se uno degli addendi è truly random, il risultato sarà nuovamente truly random.

□

**Esempio 12.3** Consideriamo un secret  $S = 0010.1101 \rightarrow 45$  e un  $RAND \bmod 256 = 180$ .  $S - RAND \bmod 256 = -135 \bmod 256 = 121$ . A riceverà 180, B 121. Il segreto che le parti potranno ricomporre sarà quindi dato da:  $180 + 121 \bmod 256 = 45$ .

**Nota** Questo fatto è interessante poiché stiamo usando l'algebra già contenuta nei calcolatori moderni (a seconda della parola usata stiamo facendo somme in modulo 32, 64bit ecc.) ed è più facile da implementare rispetto ad una algebra bitwise.

Supponiamo ora di dover dividere il segreto in  $n$  parti. Possiamo condividere il segreto nella seguente modalità.

**Teorema 12.2 (Trivial Secret Sharing)**

Generiamo un segreto  $S$  all'interno di un range  $K$ . Per condividerlo con  $n$  parti dobbiamo:

1. Generare  $n - 1$  quantità  $r \bmod K$  **truly random**.
2. Inviare alle  $n - 1$  parti uno degli shares generati precedentemente.
3. Inviare all' $n$ -esima parte  $S - \sum_{i=1}^{n-1} r_i$ .

Finché  $n - 1$  shares vengono rilevati, un avversario ha **ancora** probabilità di indovinare il segreto pari a  $1/\#bit(S)$ . Per indovinare il segreto sono necessarie **tutte le**  $n$  parti.

□

## 12.2 Shamir Secret Sharing

La versione moderna e più utilizzata per condividere un segreto è attribuita a Shamir. Per affrontare questo schema di condivisione, definiamo:

**Definizione 12.2 (N out of N Secret Sharing)**

Uno schema di condivisione ( $n,n$ ) "N out of N" condivide il segreto con  $n$  persone e necessita che **tutte** ed  $n$  siano "presenti" per rivelarlo.

□

### Definizione 12.3 (T out of N Secret Sharing)

Uno schema di condivisione  $(t,n)$  "T out of N" condivide il segreto con  $n$  persone e necessita che solo  $t$  siano "presenti" per rivelarlo.

**Osservazione:**  $1 < t < n$ . Se  $t = 1$ , ogni parte conosce il segreto.



**Nota** Un esempio di schema  $(n,n)$  è il trivial secret sharing thm 12.2

Il secondo schema è molto più robusto poiché permette di ricostruire un segreto anche se non tutte le parti che lo conoscono sono presenti. L'idea alla base dello schema proposto da Shamir è la seguente:

### Esempio 12.4 (2,n) Secret Sharing

Consideriamo una retta in un piano cartesiano.

Quanti punti sono necessari per identificare univocamente tale retta?



**Nota** Sappiamo che servono 2 punti per identificare univocamente la retta e che per un punto ne passano infinite.

Sia  $f(x)$  la retta in questione. Se poniamo il **segreto** come  $S = f(0)$ , possiamo assegnare i diversi **shares** come **punti qualsiasi sulla retta**. Unendo due punti, la retta è univocamente determinata e calcolandone il valore nell'origine possiamo trovare il segreto. Se invece abbiamo solo uno share, è impossibile determinare il segreto poiché esistono infinite rette che passano per quel punto.

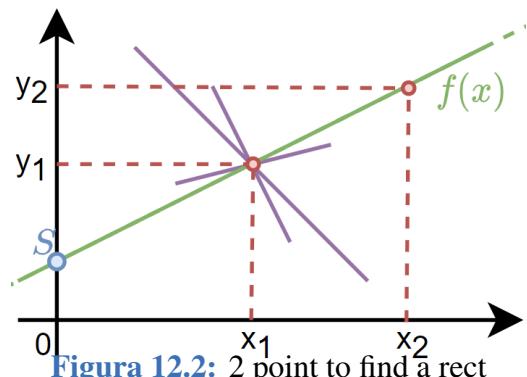


Figura 12.2: 2 point to find a rect

Possiamo riassumere l'idea di base con il seguente teorema:

### Teorema 12.3 ((2,n) Rect Secret Sharing)

1. Generare un coefficiente a truly random e un segreto  $S$ . La retta risultante sarà:

$$y = S + a \cdot x$$

2. Numerare i partecipanti in modo ordinato da 1 a  $n$ .
3. Scegliere a priori, in modo randomico ed esclusivo, le  $x_i$  da assegnare.
4. Comunicare ai partecipanti il loro numero e la corrispondente  $x_i$ .

Per ogni partecipante, avremo un punto  $p_i = (x_i, y_i)$ .<sup>1</sup>

Per ricostruire il segreto possiamo procedere per interpolazione:

1. Dati  $p_i = (x_i, y_i)$  e  $p_j = (x_j, y_j)$ , allora:

$$\frac{y - y_i}{y_j - y_i} = \frac{x - x_i}{x_j - x_i} \implies y = y_i + \frac{x - x_i}{x_j - x_i} (y_j - y_i)$$

2. Porre  $x = 0$  e trovare  $y = S$ :<sup>2</sup>

$$S = y_j \frac{-x_i}{x_j - x_i} + y_i \frac{-x_j}{x_j - x_i}$$

<sup>1</sup>Poiché il numero assegnato a ogni partecipante è noto a priori, è possibile salvare esclusivamente il valore  $y_i$  per semplicità, piuttosto che l'intero punto come share.

<sup>2</sup>Il segreto non verrà mai usato da nessuno dei partecipanti in quanto il valore  $x = 0$  è precluso agli altri.

E generalizzarla rispetto ad un polinomio di grado  $t - 1$ .

### Corollario 12.1 (Lagrange Interpolation)

*Un qualsiasi polinomio di grado  $t - 1$ , con  $t$  punti noti*

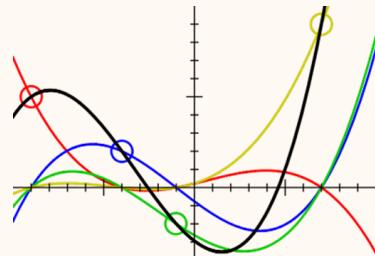
*$(x_1, y_1), \dots, (x_{t-1}, y_{t-1})$ , è sempre decomponibile nella forma:*

$$y = \sum_{i=1}^t y_i \Lambda_i(x) \quad (12.1)$$

*Dove  $\Lambda_i(x)$  è la base/polinomio di Lagrange definita come:*

$$\Lambda_i(x) = \prod_{m=1, m \neq i}^t \frac{x - x_m}{x_i - x_m} = \frac{x - x_1}{x_i - x_1} \cdots \frac{x - x_{i-1}}{x_i - x_{i-1}} \frac{x - x_{i+1}}{x_i - x_{i+1}} \cdots \frac{x - x_t}{x_i - x_t} \quad (12.2)$$

$$\Lambda_i(x_i) = 1, \quad \Lambda_i(x_m) = 0 \text{ for } m \neq i$$



**Figura 12.3:** Polynomials interpolation

**Osservazione:** In parole povere, una base di lagrange è un polinomio definito come 1 quando calcolato nello share da cui deriva, e 0 per gli altri shares. Questo perché quando calcoliamo una base in un altro share un numeratore sarà sicuramente nullo ed annullerà tutto il prodotto, quando calcolata nel suo share, invece, tutti i termini si semplificheranno. ■

### Teorema 12.4 ((t,n) Shamir's Secret Sharing Scheme)

Come Dealer:

1. Genera un polinomio  $p(x)$  di grado  $t - 1$  con **coefficienti truly random**  $a_i$ .
2. Seleziona un segreto  $S$  come termine noto del polinomio.
3. Distribuisci uno share ad ognuno degli  $n$  partecipanti, come:  $(x_i, y_i)$  con  $y_i = p(x_i)$

Per ricostruire il segreto:

1. Raccogli  $t$  degli  $n$  shares disponibili.
2. Calcola il segreto usando L'interpolazione di lagrange (eq. (12.1)) con  $x = 0$

$$S = \sum_{\text{shares } x_i} y_i \Lambda_{x_i} \text{ con } \Lambda_{x_i} = \Lambda_{x_i}(0) = \prod_{\text{shares } x_k \neq x_i} \frac{-x_k}{x_i - x_k} \quad (12.3)$$



**Nota** Il polinomio di lagrange può essere **precalcolato** in quanto non contiene alcun valore che non sia lo share  $i$ -esimo o gli altri  $k$  selezionati per ricostruire il segreto.

**Esempio 12.5** Consideriamo uno scenario (3,n). Il dealer deve generare un polinomio di grado  $t - 1 = 3 - 1 = 2$ , ovvero una parabola. Supponiamo:

$$y = 32 + 52x + 3x^2$$

Supponiamo di avere 4 partecipanti, pertanto avremo bisogno di generare e distribuire 4 shares fatti da  $(x_i, y_i)$ , ma soltanto 3 saranno necessari a ricostruire il segreto. Supponiamo di usare 1, 2, 3.

- $x = 1 \rightarrow$  share: (1, 87)
- $x = 2 \rightarrow$  share: (2, 148)
- $x = 3 \rightarrow$  share: (3, 215)
- $x = 4 \rightarrow$  share: (4, 288)

$$\begin{aligned}\Lambda_1 &= \frac{-x_2}{x_1 - x_2} \cdot \frac{-x_3}{x_1 - x_3} = \frac{-2}{1-2} \cdot \frac{-3}{1-3} = 3 \\ \Lambda_2 &= \frac{-x_1}{x_2 - x_1} \cdot \frac{-x_3}{x_2 - x_3} = \frac{-1}{2-1} \cdot \frac{-3}{2-3} = -3 \\ \Lambda_3 &= \frac{-x_1}{x_3 - x_1} \cdot \frac{-x_2}{x_3 - x_2} = \frac{-1}{3-1} \cdot \frac{-2}{3-2} = 1\end{aligned}$$

Ricostruiamo il segreto con:  $s = y_1\Lambda_1 + y_2\Lambda_2 + y_3\Lambda_3 = 87 \cdot 3 + 148 \cdot (-3) + 215 \cdot 1 = 32$

**Osservazione:** Il modello usato in questo esempio è sicuro, in quanto per ricalcolare il segreto **SERVE** che 3 shares vengano messi "insieme". █

**Osservazione:** Se avessimo calcolato il prodotto  $\sum_{i=1}^3 y_i\Lambda_i$  **senza sostituire**  $x = 0$  avremmo ritrovato il polinomio di partenza. █ █

### Proposizione 12.1 (Perfect Security of Shamir's Scheme)

*Lo schema di Shamir è sicuro a patto che i calcoli vengano fatti in aritmetica modulare. In particolare, sia il segreto che i coefficienti del polinomio devono essere presi nel campo dei numeri primi.<sup>3</sup>*

<sup>3</sup>Il numero primo scelto non deve essere **necessariamente grande**, ma sufficientemente grande per garantire di avere un segreto all'interno dell'intervallo scelto. Ad esempio, per  $s \in (1, 100)$   $p = 101$  è un buon numero primo. █

Infatti, prendendo una parabola, abbiamo soltanto un sottoinsieme di valori che possiamo utilizzare e questo restringe il campo di probabilità rispetto ad un brute-force. Rispetto all'esempio 12.5 supponiamo:

- $S \in [1, 100]$  distribuito in modo uniforme.
- Di **conoscere** il **primo** e il **terzo** share.

Per calcolare il secondo share, si può procedere per brute-force calcolando:

$$s = y_1\Lambda_1 + y_2D + y_3\Lambda_3 = 87 \cdot 3 + D \cdot (-3) + 215 \cdot 1 = 476 - 3D$$

Per ogni valore **intero**  $D$ . In particolare, si può vedere come per  $D \in (125, 158)$  il valore di  $S$  assume valori nell'intervallo  $[2, 98]$  con un passo di 3 per ogni tentativo effettuato. Questo significa che sull'intervallo aperto  $(1, 100)$  con probabilità  $p = 1/3$  il segreto può essere rivelato.



**Nota** La conoscenza di due shares su tre aumenta la probabilità di indovinare il segreto, fornendo informazioni all'attaccante.

Questo esempio ci permette di definire il concetto di sicurezza in due modi diversi:

**Definizione 12.4 (Computational Security)**

*Uno schema computational Secure non fornisce all'attaccante un modo efficiente<sup>4</sup> per calcolare il segreto.*

<sup>4</sup>Vedi **Discrete Logarithm Problem** o **Fattorizzazione RSA** (props 9.3 e 9.4). □

**Definizione 12.5 (Unconditionally Security)**

*Uno schema Unconditionally Secure non fornisce all'attaccante alcuna possibilità di calcolare un segreto se uno o più dati sono mancanti.* □

Lo schema di Shamir è **Unconditionally Secure**, perché matematicamente se uno share è mancante anche se un'attaccante ha sufficiente potenza di calcolo non potrebbe mai ricostruire il segreto. Vediamo ora cosa succede usando aritmetica modulare con numeri primi.

**Esempio 12.6** Consideriamo uno scenario (3,n). Il dealer deve generare un polinomio di grado  $t - 1 = 3 - 1 = 2$ , ovvero una parabola. Supponiamo:

$$y = 3x^2 + 52x + 32 \text{ mod } 101$$

Dove il segreto  $s = 32 \in (1, 100)$  e il numero primo scelto è  $p = 101$ . Con 4 partecipanti avremo bisogno di generare e distribuire 4 shares ma supponiamo di usare 1, 2, 4 per ricostruire il segreto. Per provare che è possibile non seguire l'ordine cardinale nell'assegnazione degli share, assegniamo  $x_4 = 6$ .

Facendo i calcoli abbiamo:

$$y_1 = y(x_1) = 87, \quad \Lambda_1 = (-x_2)(-x_4) \text{ mod } 101 \cdot [(x_1 - x_2)(x_1 - x_4)]^{-1} \text{ mod } 101 = 63$$

$$y_2 = y(x_2) = 47, \quad \Lambda_2 = (-x_1)(-x_4) \text{ mod } 101 \cdot [(x_2 - x_1)(x_2 - x_4)]^{-1} \text{ mod } 101 = 49$$

$$y_3 = y(x_3) = 13, \quad \text{Non va calcolato.}$$

$$y_4 = y(x_4) = 48, \quad \Lambda_4 = (-x_1)(-x_2) \text{ mod } 101 \cdot [(x_4 - x_1)(x_4 - x_2)]^{-1} \text{ mod } 101 = 91$$



**Nota** Ovviamente con aritmetica modulare non possiamo usare le divisioni, ma dobbiamo moltiplicare per l'inversa modulare.

A questo punto possiamo ricostruire il segreto semplicemente moltiplicando i tre fattori:

$$s = [y_1\Lambda_1 + y_2\Lambda_2 + y_3\Lambda_3] \text{ mod } 101 = [87 \cdot 63 + 47 \cdot 49 + 48 \cdot 91] \text{ mod } 101 = 32$$

**Osservazione:** Supponiamo ora di attaccare lo schema, conoscendo il secondo e il quarto share e il range di valori in cui il segreto può essere contenuto. Dovremmo risolvere l'equazione:

$$s = [\Lambda_1 \cdot D + \Lambda_2 \cdot Lambda_2 + \Lambda_3 \cdot y_3] \text{ mod } 101 = [5 + 63D] \text{ mod } 101$$

Se adesso proviamo un qualsiasi valore di  $D$  nell'intervallo in cui il segreto può esistere vedremmo che i risultati ottenuti sarebbero **tutti i numeri da 1 a 100** in modo **uniformemente distribuito**. Questa proprietà è dovuta ovviamente all'operazione di modulo e garantisce unconditional security anche se sono noti due share su tre. ■



**Nota** Quando utilizziamo schemi Computational Secure in realtà stiamo facendo **Computation Hiding**. Ovvero, nascondiamo un segreto tramite una funzione Hash (anche la più resistente al mondo), che dipende sempre e comunque dallo spazio dei valori assunti dal segreto. Più questo sarà grande, più sarà difficile per un attaccante riuscire ad indovinare il segreto tramite **enumeration attack**.

#### Corollario 12.2 (Ideality of Secret Sharing Scheme)

Uno schema di condivisione di segreti è **ideale** se tutti gli share hanno **dimensione almeno pari** a quella del segreto. Se gli share fossero più piccoli, una volta che almeno uno è noto sarebbe più facile ricostruire il segreto a causa dell'entropia ridotta.



**Nota** Lo schema di Shamir è ideale. Lo schema proposto da Blakley era invece basato su share di dimensione  $t$ -volte quella del segreto.



### 12.2.1 Secret Sharing for Multipart Computation

Introduciamo una proprietà di cui gode lo schema di Shamir:

#### Proposizione 12.2 (Homomorphic Property of Shares)

Poiché gli share in uno schema di Shamir sono generati a partire da polinomi, allora vale che: la somma degli shares è uguale allo share della somma dei segreti. Ovvero:

$$f_i + g_i = (f + g)_i \quad (12.4)$$

Dove  $f_i, g_i$  sono due polinomi che definiscono un segreto e gli share sono  $\{k_i, y_i(k_i)\}$



Questa proprietà permette di usare lo schema anche nel caso in cui invece di essere interessati alla ricostruzione di un segreto in particolare, siamo interessati alla ricostruzione della somma di più segreti.

Tipicamente l'approccio normale sarebbe il seguente:

1. Ricostruisco il segreto  $S_1$ .
2. Ricostruisco il segreto  $S_2$ .
3. Calcolo  $S_1 + S_2$ .

Per la prop 12.2 invece possiamo procedere come segue:

1. sia  $\alpha_1$  il segreto dato da  $f_i(k) + g_i(k)$ .
2. sia  $\alpha_2$  il segreto dato da  $f_i(v) + g_i(v)$ .
3. Usando eq. (12.4) possiamo ricostruire la somma  $\alpha_1 + \alpha_2$  **SENZA CONOSCERE** né il segreto  $\alpha_1$  né  $\alpha_2$ .

Questa modalità di calcolare un segreto è detta: **Secure Multiparty Computation** e costituisce una branca della crittografia utile ad esempio in scenari finanziari (Si vuole sapere chi ha fatto una donazione ma non quanto), medici, controllo del traffico ecc.

### Definizione 12.6 (Secure Multiparty Computation)

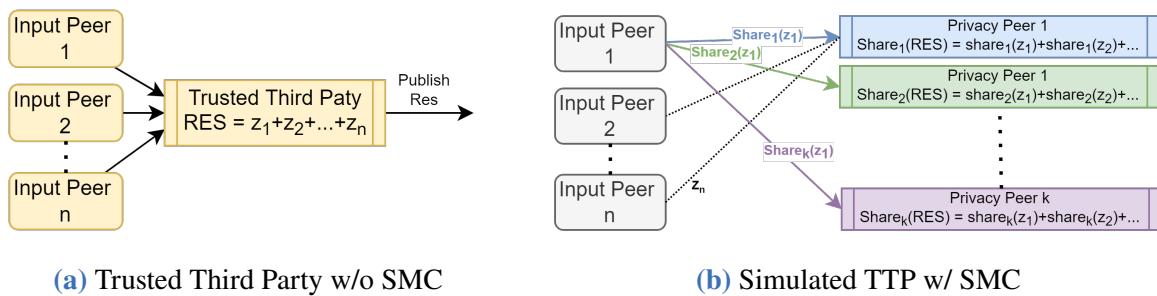
*Calcolo del risultato di una funzione senza rilevare i dati in input. Formalmente:*

- Per  $N$  partecipanti  $P_1, \dots, P_n$  associa un valore  $z_i$ .
- Calcola la funzione  $f(z_1, z_2, \dots, z_n)$  tale che il **risultato sia pubblico ma nessuna informazione su un qualsiasi  $z_i$  è deducibile dal risultato di  $f$ .**

**Osservazione:** Si può dimostrare che è possibile **effettuare qualsiasi calcolo** a patto di avere sufficiente potenza di calcolo.



Lo SMC è una tecnica che torna comoda quando utilizziamo una **somma pesata**, che può essere calcolata in modo **molto efficiente**. In particolare, per caso in cui vogliamo garantire una certa confidenzialità come ad esempio scenari di calcolo salariale medio di una popolazione o consumo energetico, possiamo suddividere un'unica grande trusted third party in diversi enti che possono garantire per la privacy dell'utenza che non comunicando tra loro (a meno di collusioni eventuali) possono oscurare il dato privato del singolo utente.



**Figura 12.4:** SMC example

Il vantaggio fondamentale è il deployment del sistema:

### Corollario 12.3 (SMC Deployment)

- Possiamo avere da  $3^5 a n$  input peer e migliaia di end-user.
- Possiamo avere da  $2 a k$  privacy peer, e l'aumentare di questo numero aumenta il livello di sicurezza del sistema.
- Possiamo usare uno schema di Shamir, per usare un numero qualsiasi di privacy peer, pagando il prezzo di aritmetica modulare con numeri primi.

<sup>5</sup>Se avessimo 2 input una delle parti potrebbe dedurre il valore privato per sottrazione.



### Teorema 12.5 (SMC with Shamir Scheme)

Come Input Peer  $i$ -esimo.<sup>6</sup>

- Per  $N$  partecipanti  $P_1, \dots, P_n$  associa un valore  $z_i$ .
- Genera polinomi  $p_i(x)$  di grado  $t - 1$ .
- Invia privatamente gli shares  $p_i(1), \dots, p_i(k)$  ai privacy peer  $1, \dots, k$ .

Come Privacy Peer  $m$ -esimo:

- Raccogli gli shares in input  $p_1(m), p_2(m), \dots, p_k(m)$ .
- Calcola  $RES(m) = p_1(m) + p_2(m) + \dots + p_k(m)$ .
- Pubblica il risultato aggregato  $RES(m)$ .

Come End-User, possiamo ricostruire il risultato tramite lagrange interpolation.

<sup>6</sup>Non serve ci sia coordinazione tra gli input peer. □

### Esempio 12.7 Input Peers as Privacy Peers

Un esempio pratico è quello nel quale vi è la necessità in un gruppo di utenti di scambiarsi dati in forma anonima, come ad esempio una votazione in una cerchia ristretta. Lo schema da adottare è  $(n,n)$  poiché tutti devono partecipare. In questo caso gli input peers possono agire come privacy peers a rotazione, senza che nessuno riveli mai il proprio segreto.

In particolare, possiamo procedere nel seguente modo:

- L'entità  $i$ -esima **fa da dealer** e genera  $n - 1$  share  $R_i$  randomicamente e **trattiene per se** lo share  $R_1$ .
- La stessa entità invia gli share alle altre entità, inviando all'ultima la quantità:

$$(S_i - \sum_{i=1}^{n-1} R_i) \bmod N$$

- **A rotazione**, tutti i partecipanti fanno da dealer, tranne l'ultimo.
- L'entità  $n$ -esima invia alle altre entità i suoi share e trattiene per se la quantità

$$(S_i - \sum_{i=1}^{n-1} R_i) \bmod N$$

Vediamo un esempio con schema  $(3,3)$  in fig. 12.5 ■

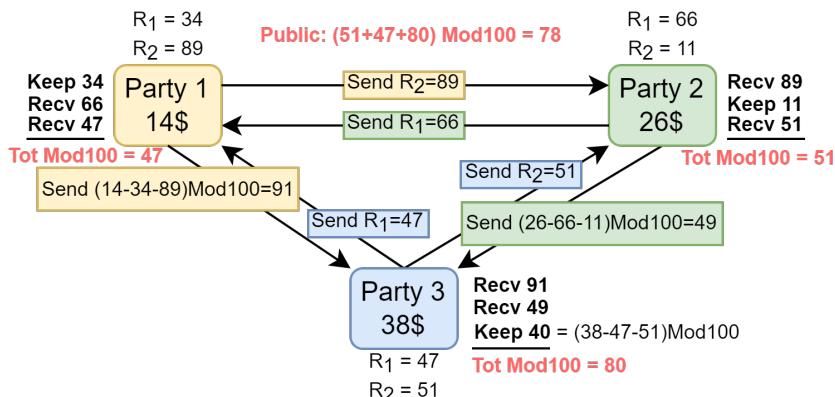


Figura 12.5: (3,3) SMC Scheme

## 12.3 Verifiable Secret Sharing

Supponiamo uno scenario in cui viene fatta una donazione di 10\$, ma viene comunicato che l'importo donato è di 1000\$. Lo schema di Shamir (thm 12.4) è vulnerabile sul lato autenticazione, perché un input peer/ trusted party potrebbe *"barare"* o, più semplicemente, non ha un modo per verificare che il segreto inserito è corretto.

Un modello di attacco che sorge in questo frangente, è il seguente:

### Definizione 12.7 (Honest-but-Curious Model)

*Modello di attacco nel quale si vuole scoprire il segreto di un sistema, seguendo le regole del sistema stesso.*



Abbiamo bisogno di un modo di controllare che le regole del protocollo vengano seguite per intercettare e bloccare eventuali cheater.

**Osservazione:** Sia Dealer che Player possono essere cheater. I primi comunicando segreti inconsistenti, i secondi comunicando risultati che possono essere usati per manipolare la ricostruzione finale. ■

Riassumendo, abbiamo bisogno di due proprietà:

- **Malicious Dealer:** Una qualsiasi parte deve verificare che lo **share condiviso** sia consistente.
- **Cheating Parties:** Le parti devono verificare se uno **share rivelato** è consistente.

L'idea alla base dei protocolli aggiuntivi che vedremo è quella di creare un binding tra il segreto e ciò che viene rivelato. Questo binding è detto **Crypto Commitment**

### Teorema 12.6 (Feldman Verifiable Secret Sharing)

*Come Dealer:*

1. Genera polinomio random  $p(x)$  di grado  $t - 1$  tale che  $p(0) = s$ .
2. Distribuisci uno share per ognuna delle  $n$  parti:  $(x_i, y_i)$ , con  $y_i = p(x_i)$ . Tipicamente  $x_i = i$ .
3. Per ogni coefficiente del polinomio, compreso il segreto, comunicare pubblicamente:

$$c_0 = g^s; c_1 = g^{a_1}; \dots, c_{t-1} = g^{a_{t-1}}$$

*Calcolati con algebra modulare<sup>7</sup> rispetto un numero primo.*

*Come Verifier:*

1. Ogni parte riceve il suo share  $(x_i, y_i)$ , con  $y_i = p(x_i)$ .
2. **Check Malicious Dealer:** Verifico che lo share ricevuto sia valido calcolando in mod  $P$

$$\begin{aligned} c_0 \cdot c_1^{x_i} \cdot c_2^{x_i^2} \cdot \dots \cdot c_{t-1}^{x_i^{t-1}} &= g^s \cdot \prod_{k=1}^{t-1} g^{a_k x_i^k} = \\ &= g^{s+a_1 x_i + a_2 x_i^2 + \dots + a_{t-1} x_i^{t-1}} = g^{p(x_i)} \end{aligned}$$

3. Controlla che il risultato ottenuto sia uguale a  $g^{y_i}$ , con  $y_i$  lo share della parte.

<sup>7</sup>Il calcolo modulare introduce il problema del discrete log, che rende difficile il calcolo dell'esponente.





**Nota** Stiamo usando la proprietà di omomorfismo dello schema di shamir, per verificare che non ci sono state alterazioni negli share inviati.

### Definizione 12.8 (Feldman Commitment)

L'"**impegno**" nello schema di Feldman è il coefficiente  $c = g^x \bmod P$ , che impedisce a chi riceve il segreto di conoscere il valore di  $x$ .

Al fine di essere robusto un commitment deve soddisfare:

- (**Computational**) **Hiding**: il ricevente **non** dovrebbe poter dedurre niente sul segreto inviato.
- (**Perfectly**) **Binding**: Chi invia il commitment non può trovare un altro  $x'$  tale che  $g^{x'} = c$ .<sup>8</sup>

<sup>8</sup>E' fondamentale ricordare che  $x$  non può essere scelto in modo qualunque, altrimenti è facile trovare una collisione, vedi eq. (9.4).



**Nota** Se il commitment fosse stato fatto con una funzione Hash, avremmo avuto la proprietà della computational hiding e perfectly binding se avessimo usato una funzione hash forte. Il problema è che siamo limitati a segreti di dimensione elevata per garantire sufficiente protezione, che comunque potrebbe non essere sufficiente nel tempo.

Vedremo successivamente dettagli aggiuntivi **necessari** affinché lo schema di Feldman possa funzionare. Per il momento, è importante capire che i valori di **g,p devono essere scelti in modo appropriato**:

**Osservazione:** Per come è definito il commitment di Feldman,  $c_0 = g^s$  lascia trapelare informazioni sul segreto. Se  $s$  è piccolo, un dictionary attack può trovare il suo valore:

```

1: for  $x$  in  $[1, 1000]$  do
2:   if  $g^x == c_0$  then
3:     Secret Found
4:   end if
5: end for

```



**Nota** E' dimostrato che è **impossibile** costruire dei commitment che siano contemporaneamente perfectly hiding e perfectly binding. Tuttavia, è possibile costruire commitment che siano perfectly hiding **al costo** di essere computational binding.

### 12.3.1 Pedersen Commitment

La costruzione di Pedersen nasce per migliorare quella di Feldman, la cui proprietà principale era che il commitment era omomorfico, infatti

$$\text{Commit}(a+b) = \text{Commit}(a) \cdot \text{Commit}(b) \rightarrow g^{a+b} = g^a \cdot g^b \pmod{p}$$

#### Teorema 12.7 (Pedersen Commitment)

Dati  $g, h$  pubblici, il commitment è definito come:

$$\text{Commit}(a, r) = g^a \cdot h^r \pmod{P} \quad (12.5)$$

E gode della proprietà di omomorfismo:

$$\text{Commit}(a+b, r_a+r_b) = g^{a+b} \cdot h^{r_a+r_b} \pmod{P} = g^a h^{r_a} \cdot g^b h^{r_b} = \text{Commit}(a, r_a) \cdot \text{Commit}(b, r_b)$$

Inoltre:

- **Perfectly Hiding:**  $\text{c}(a, r) = g^a \cdot h^r \pmod{P}$  può essere un commitment per qualsiasi valore.

Formalmente:  $\forall a' \neq a \exists! r' :$

$$\text{commit}(a', r') = g^{a'} \cdot h^{r'} \pmod{P} = g^a \cdot h^r \pmod{P} = \text{commit}(a, r)$$

- **Computationally Binding:** il Sender non dovrebbe essere in grado di trovare  $a'$  ed un  $r'$  associato, a meno di conoscere  $\log_h$ .

□

**Dimostrazione** Proviamo che il Perdersen Commitment è Computationaly Binding:

Sia  $h = g^w \rightarrow w = \log_g h$ . Supponiamo di conoscere  $a, r$  e di scegliere  $a'$ . Vogliamo trovare un  $r'$  associato ad  $a'$  tale che

$$g^a h^r = g^{a'} h^{r'} \quad (12.6)$$

Allora:

$$g^a g^{wr} = g^{a'} g^{wr'} \rightarrow g^{a+wr} = g^{a'+wr'}$$

L'ultima equazione è vera se solo se:  $a + wr = a' + wr'$ . Pertanto possiamo trovare  $r'$  tale che:

$$r' = w^{-1}(a - a' + wr)$$

Ricordando che le operazioni sono fatte in modulo, dato un  $w$ , possiamo calcolare qualsiasi collisione dei commitment, quindi non avremmo binding.

Per risolvere il problema, comunque, **devo poter invertire il discrete logarithm** e trovare  $w$ .

□

**Definizione 12.9 (Pedersen VSS)**

Come Dealer:

1. Genera due polinomi, shamir-like, tali che:

$$f(x) = s + a_1x + a_2x^2 + \dots + a_{t-1}x^{t-1}$$

$$f'(x) = r + b_1x + b_2x^2 + \dots + b_{t-1}x^{t-1}$$

Dove  $s, r$  sono i segreti da non rivelare,  $a_i, b_i$  sono i coefficienti random.

2. Invia alla parte  $i$ -esima un doppio share:  $(x_i, y_i, z_i)$  con  $y_i = f(x_i)$ ,  $z_i = f'(x_i)$ .
3. Pubblica commitment secondo Pedersen:  $c_0 = g^s h^r$ ,  $c_1 = g^{a_1} h^{b_1}$ , ...,  $c_{t-1} = g^{a_{t-1}} h^{b_{t-1}}$ .

Come un Verifier:

1. La parte  $i$ -esima riceve lo share  $(x_i, y_i, z_i)$  con  $y_i = f(x_i)$ ,  $z_i = f'(x_i)$ .
2. Verifico  $(\text{mod } P)$  come in Feldman (thm 12.6), ma verifico la quantità  $g^{y_i} h^{z_i}$ . □



**Nota** Applicando questo schema ad un esempio, senza prendere degli accorgimenti, non produrrebbe un risultato corretto. Dimostrazione tramite foglio di calcolo mathematica.

### 12.3.2 Prime for Dummies

Introduciamo la definizione di Gruppo:

**Definizione 12.10 (Group)**

Un gruppo  $(G, \circ)$  è un insieme nel quale viene definita un'operazione, indicata con  $\circ$ . Per un gruppo valgono le seguenti proprietà:

- **Chiusura:**  $\forall g_1, g_2 \in G \quad g_x = g_1 \circ g_2$ .
- **Identità:**  $\exists I \in G : g \circ I = I \circ g = g$ .
- **Inversa:**  $\forall g \in G, \exists g^{-1} \in G : g \circ g^{-1} = I$ .
- **Associativa:**  $\forall g_1, g_2, g_3 \in G$  vale che:  $(g_1 \circ g_2) \circ g_3 = g_1 \circ (g_2 \circ g_3)$  □

**Corollario 12.4 (Abelian Group)**

Se per un gruppo (def 12.10) vale la proprietà di commutatività, allora è detto Abeliano. □

Il gruppo usato nelle moltiplicazioni della costruzione di Pedersen è il gruppo  $\mathbf{Zp}^*$ .

**Definizione 12.11 ( $\mathbf{Zp}^*$ )**

Gruppo basato sulla moltiplicazione di numeri interi in modulo  $P$ , dove  $P$  è un numero primo.

**Osservazione:** Per costruzione presenta un numero finito di  $P - 1$  elementi da 1 a  $p - 1$ , vedi eq. (9.4). 0 è escluso. □ □



**Nota** Se avessimo avuto la commutatività, avremmo avuto un campo:  $\mathcal{F}_p$

**Osservazione:** Tre proprietà su quattro sono ovviamente soddisfatte, compresa la commutatività. Per quanto riguarda l'inversa però abbiamo il problema che in modulo non esiste sempre l'inversa, a meno che questo non sia fatto secondo un numero primo, per questo è necessario.<sup>9</sup> □

<sup>9</sup>Per calcolare l'inversa si può usare l'Algoritmo di Euclide Esteso (def 9.7)

Per un gruppo basato sulla moltiplicazione, un esponente equivale al prodotto di  $k$  volte della stessa base.

#### Definizione 12.12 (Generator of Group of Order m)

*Per un gruppo esiste sempre un elemento  $g$  tale che  $g^0, g^1, \dots, g^{m-1}$  è un insieme di  $m$  elementi appartenenti a  $G$ .*

**Proprietà [Prime-Order Group]** Un gruppo è di ordine primo se  $m$  è un numero primo. Questo implica che ogni suo elemento, a parte l'identità, è un generatore.

**Osservazione:** Un gruppo  $Z_p^*$  ha ordine  $p - 1$ , che è pari e non primo. □

**Esempio 12.8**  $Z_{11}^*$  Consideriamo un insieme di elementi da 1 a 10:  $G = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$  per il gruppo  $Z_{11}^*$ . Chi sono i generatori di questo gruppo?

- **g=2:**  $\{2, 4, 8, 5, 10, 9, 7, 3, 6, 1\}$ . E' un generatore.
- **g=3:**  $\{3, 9, 5, 4, 1, 3, 9, 5, 4, 1\}$ . **Non** è un generatore. C'è un sottogruppo di ordine 5.

... 3,4,5,9 non sono generatori, 6,7,8 si ...

- **g=10:**  $\{10, 1, 10, 1, 10, 1, 10, 1, 10, 1\}$ . **Non** è un generatore, sottogruppo di ordine 2.

**Nota** La dimensione dell'insieme è data dal prodotto della dimensione dei due sottogruppi:  $10 = 2 \cdot 5$ . Si potrebbe dimostrare, ma non lo facciamo. ■

#### Teorema 12.8 (Group Order)

L'ordine di un gruppo è dato dalla fattorizzazione della dimensione del gruppo. □

Quello che abbiamo descritto implica che gli stessi numeri primi che usiamo in crittografia devono soddisfare una proprietà che chiamiamo:

#### Definizione 12.13 (Strong Primes)

Un numero primo  $p$  è detto **Strong** se:

$$p = 2q + 1, \text{ } q \text{ è un numero primo} \quad (12.7) \quad \square$$

Quindi, il gruppo  $Z_p^*$  che ha ordine  $p - 1$ , in realtà ha ordine  $2q$ . Quindi, **qualsiasi** membro  $x$  del gruppo (**tranne** 1 e  $p - 1$ ) può fare una delle due cose:

- Genera l'intero gruppo.
- Genera un sottogruppo di ordine primo  $q$ .

**Nota** Ogni volta che vogliamo un numero primo **grande**, vogliamo che sia  $p$  che  $q$  siano grandi. Altrimenti  $p - 1$  può essere fattorizzato in numeri più piccoli.

**Osservazione:** C'è una proprietà molto importante che possiamo dedurre dall'esempio 12.8: si può vedere che i numeri che sono generatori del gruppo sono in realtà tutti quei numeri che non possono essere espressi come radici quadrate. ■

**Definizione 12.14 (Quadratic Residue)**

Un numero  $x \in Z_p^*$  è un residuo quadratico se ammette radice quadratica nel gruppo. Ovvero:

$$\exists a : a^2 \bmod p = x$$

**Definizione 12.15 (Quadratic Residue Subgroup QR)**

E' un sottogruppo formato da tutti gli elementi  $x \in Z_p^*$  che sono residui quadratici.

**Esempio 12.9** Per  $Z_{11}^*$  il  $QR = \{1, 3, 4, 5, 9\}$

Vale il seguente teorema:

**Teorema 12.9 (Subgroup Generator)**

Se  $x$  è un residuo quadratico, allora genera un sottogruppo di ordine  $q = \frac{p-1}{2}$ .

Inoltre, si può provare che:

$$a \in QR \iff a^{\frac{p-1}{2}} = 1$$

Questa operazione è detta **Legendre Symbol**

**Corollario 12.5**

Se  $p$  è un numero primo forte, allora,  $QR_p$  ha prime-order  $q$ .



**Nota** Questo significa che in uno schema di Pedersen se  $g \in QR_p$  allora quando calcoliamo l'esponenziale lo stiamo facendo in modulo  $q$ .

## 12.4 Distributed Secret Sharing

**Problema 12.1** E' possibile generare una coppia  $\{\text{Pub}_k, \text{Priv}_k\}$  tale che tutti conoscano la chiave pubblica e nessuno quella privata?

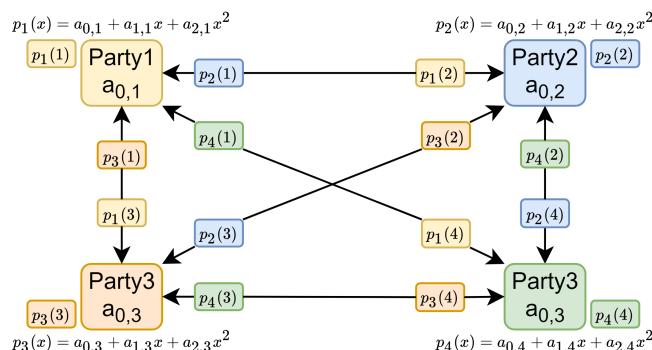
Questo scenario è coperto dai sistemi crittografici **DLog-based**, dove la private-key  $x$  è nascosta dalla public key  $g^x$ . Schemi del genere sono utili quando la chiave privata deve essere rivelata in un secondo momento (ad esempio in caso in cui si deve cifrare un messaggio che nessuno può aprire a meno di speciali condizioni), o non deve **mai** essere rivelata ma le sue caratteristiche sono necessarie. Lo schema di condivisione funziona nel seguente modo:

### Definizione 12.16 (Distributed Secret Sharing)

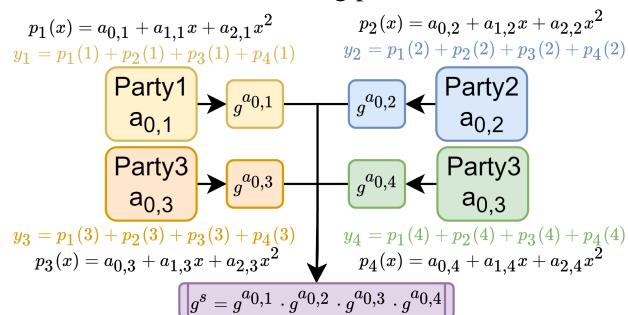
Consideriamo uno schema di secret sharing distribuito con  $x$  chiave privata e  $g^x$  chiave pubblica.

Procediamo nel seguente modo:

1. Ogni parte genera indipendentemente un segreto  $a_{0,i}$
2. Ogni parte genera indipendentemente il suo polinomio di grado  $t - 1$  con termine noto  $a_{0,i}$ .
3. Ogni parte invia uno share  $p_i(k)$  alle altre parti, e trattiene  $p_i(i)$ .
4. Ogni parte calcola la somma degli share e **la trattiene per se**. Questa somma sarà la somma dei segreti, che fino ad ora sono rimasti privati.
- Osservazione:** Se ogni parte comunicasse il proprio risultato  $y_i$  sarebbe possibile per interpolazione calcolare  $S = \sum_i a_{0,i}$  ■
5. Invio in broadcast un commitment per  $a_{0,i}$ :  $g^{a_{0,i}}$ . Moltiplicando i quattro commitment possiamo calcolare  $g^s$ . □



(a) Sharing phase



(b)  $g^s$  computation

**Figura 12.6:** DSS Example with (3,4) scheme

# Capitolo Threshold and Policy-Based Cryptography

Fino ad ora abbiamo visto come Diffie-Hellman ha usato il principio del *Dlog* per fare solo, ed esclusivamente, *key-agreement* mentre RSA ha risolto il problema di fare cifratura asimmetrica con una chiave pubblica, usando il principio della *fattorizzazione*.

Il principio alla base di DH è però più robusto e vorremmo poterlo usare per fare public-key encryption. El Gamal nel 1985 risolse il problema trasformando DH in un asymmetric cipher.

## Definizione 13.1 (ElGamal)

Sia  $p$  un numero **primo grande** e  $g$  un group-generator. Da questo momento in poi **ogni operazione avviene in modp**.

Sia  $s$  la **private key** e  $h = g^s$  la **public key**. Allora, preso un numero casuale  $r$ :

- **Encrypt:**

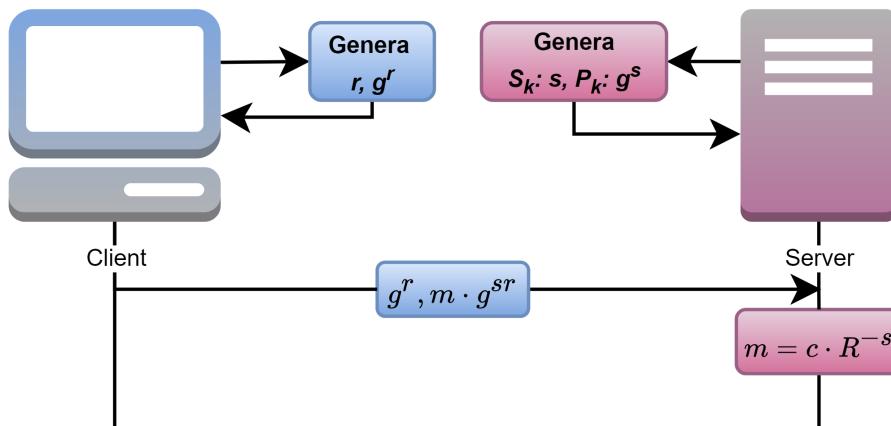
$$(R, c) = (g^r, m \cdot h^r) \quad (13.1)$$

- **Decrypt:**<sup>1</sup>

$$m = c \cdot R^{-s} = \frac{c}{(g^r)^s} = \frac{m \cdot h^r}{g^{rs}} = \frac{m \cdot g^{rs}}{g^{rs}} \quad (13.2)$$

**Osservazione:** Siamo costretti a **raddoppiare la dimensione** del testo che inviamo, poiché dobbiamo includere anche la chiave pubblica di cifratura  $R$ . ■

<sup>1</sup>L'inversa è **modulare**. □



**Figura 13.1:** ElGamal scheme



**Nota** Nell'esempio possiamo vedere come lo schema è molto simile a DH, in quanto  $g^s, g^r$  sono i coefficienti pubblici (rispettivamente per la chiave privata e per il ciphertext).  $s, r$  sono segreti noti esclusivamente al ricevente e al trasmittente rispettivamente, quindi **non c'è modo** per chiunque di ricostruire  $g^{sr}$ .

Tra l'altro,  $r$  è un segreto **effimero** che viene generato on-the-fly.

## 13.1 Asymmetric Ciphers and Hybrid Encryption

I cifratori asimmetrici devono inviare messaggi che siano all'interno del gruppo che viene generato dal numero primo scelto. Ovviamente, i messaggi possono essere molto più grandi della dimensione scelta per il numero primo  $p$ . Servono quindi schemi che possano lavorare fianco a fianco con quelli classici (es: AES) che si occuperanno di cifrare il messaggio, mentre i primi cifreranno la chiave con cui il messaggio è stato protetto. Questa modalità caratterizza gli schemi *ibridi*, usati ad esempio nei sistemi 5G quando l'utente deve necessariamente comunicare il proprio IMSI (la propria identità digitale). Un modo per creare uno schema ibrido è usare ElGamal.

**Esempio 13.1 IMSI share in Modern Cell Network** Consideriamo una SIM all'interno della quale è stata scritta la public-key della Home Network  $g^{HN}$  e la chiave privata dell'utente  $x$ .

Facciamo uno scambio di chiavi:

1. Viene generata la chiave pubblica  $g^x$ .
2. Viene derivata la chiave di cifratura con  $K = HKDF(g^{HN \cdot x})$ .
3. L'utente invia in modo cifrato il suo **SUPI**<sup>2</sup> con  $AES_K(SUPI)$  insieme ad un HMAC per la verifica e la sua chiave pubblica.
4. In ricezione la HN prende il pacchetto  $(g^x, AES_K(SUPI), HMAC(AES_K(SUPI)))$  e usa  $g^x$  per ricostruire la chiave con  $K = HKDF(g^{HN \cdot x})$ .
5. Viene usata la chiave ricostruita per decifrare e fare la verifica di integrità.

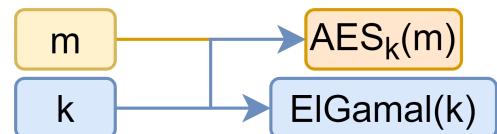


Figura 13.2: Hybrid Encryption

### 13.1.1 Threshold ElGamal

ElGamal è uno schema che torna utile anche per quei casi in cui vogliamo rendere la decriptazione di un messaggio possibile **solo se** un certo numero di persone riceventi coopera. Una prima idea di questo schema potrebbe essere la seguente:

1. Utilizzando uno schema di Pedersen  $(t,n)$  vengono condivisi share della chiave privata  $S$ .
2. Ricostruisco la chiave privata  $S$  quando le  $t$  parti sono riunite.

Il problema di questo schema è che una volta che la chiave viene ricostruita è disponibile a tutti e qualsiasi cosa potrà essere decifrato da quel momento in poi.

**Problema 13.1 Possiamo decifrare un messaggio e garantire confidenzialità per qualsiasi altri messaggi?**

La risposta è ovviamente sì e possiamo pensare di usare uno schema ElGamal-like, dove la chiave pubblica è generata on-the-fly per ogni nuovo messaggio e quindi in decryption il denominatore sarà qualcosa del tipo  $g^{r_i s}$  per il messaggio *i-esimo*. Lo schema corretto è il seguente:

<sup>2</sup>La nuova versione dell'IMSI.

### Definizione 13.2 (threshold ElGamal)

Consideriamo uno schema  $(t,n)$ . Sia  $p$  strong-prime, con  $q$  large prime. Sia  $g \in QR_p$  il generatore del sottogruppo. Presa secret-key  $s \in \mathbb{Z}_q$  random, la pub-key è  $h = g^s \bmod p$ .

**Come Dealer:**

1. Genero polinomio  $p(x) \bmod q$  di ordine  $t - 1$  e invio ad ogni parte  $(x_i, y_i)$  con  $y_i = p(x_i) \bmod q$ .
2. Faccio trasmissione del messaggio  $m$  tramite ElGamal (def 13.1), inviando

$$(R, c) = (g^r, m \cdot h^r) \bmod p$$

con  $r \in \mathbb{Z}_q$

**Come Receiver:**

1. Estraggo chiave pubblica  $g^r$  dal messaggio ricevuto.
2. Calcolo il coefficiente di Lagrange:

$$\Lambda_{x_i} = \prod_{\substack{\text{shares } x_k \neq x_i}} \frac{-x_k}{x_i - x_k} \bmod q$$

3. Calcolo l'esponenziale dello share per la decrittazione:  $w_i = (g^r)^{y_i \Lambda_i} \bmod p$
4. Raccogliendo  $t$  shares, possiamo ricostruire la chiave di decrittazione:

$$\prod_{i=1}^t w_i = \prod_{i=1}^t (g^r)^{y_i \Lambda_i} = (g^r)^{\sum_{i=1}^t y_i \Lambda_i} = (g^r)^s = g^{rs} \bmod p$$

5. La decrittazione è possibile:

$$m = c \cdot R^{-s} = \frac{c}{(g^r)^s} = \frac{m \cdot h^r}{g^{rs}} = \frac{m \cdot g^{rs}}{g^{rs}} \bmod p$$

**Osservazione:** La secret-key  $s$  non viene mai rivelata/ricostruita poiché per ogni messaggio  $m$  viene generata una nuova nonce  $r$  e quindi la chiave di decifratura  $g^{rs}$  sarà sempre diversa. ■ □

## 13.2 Threshold Signature

La threshold signature è uno use-case molto più importante rispetto alla threshold encryption perché permette a  $t$  membri di un gruppo di  $n$  elementi di firmare un messaggio. Questo consente di fare cose come:

- Verifica della validità della firma da molteplici fonti.
- Un membro del gruppo può essere certificato dagli altri  $t$  membri.
- Posizionare un segreto in più di una CA.
- Impossibilità di creare una firma se meno di  $t$  utenti non cooperano.

Le uniche accortezze a cui bisogna stare attenti sono le seguenti:

- Riutilizzare sistemi di firma esistenti.
- La dimensione della firma non deve scalare con il numero di partecipanti.

Uno dei sistemi di firma che possono essere fatti è tramite RSA, vediamone lo schema:

**Definizione 13.3 (RSA Signature)**

1. Scegliere numeri primi **grandi**  $p, q$  tale che  $N = pq$ . Allora  $\phi(N) = (p-1)(q-1)$
2. Scelgo public-key e coprimo con  $\phi(N)$  e calcolo la secret-key  $d = e^{-1} \bmod \phi(N)$
3. **Firmo:**  $\{m, H(m)^d\}$
4. **Controllo la firma:**  $\{H(M) = (H(m)^d)^e\}$

□

Per estendere questo schema ad uno a soglia, potremmo procedere con il solito approccio:

1. Come dealer, genero un polinomio  $f(x)$  tale che  $f(0) = d$ .
2. Invio lo share  $(x_i, y_i = f(x_i)) \bmod \phi(N)$
3. Invio messaggi firmati  $\{m, H(m)\}$
4. Share della firma:  $H(m)^{y_i \Lambda_i} \bmod N$
5. Verifico firma:  $\prod H(m)^{y_i \Lambda_i} = H(m)^{\sum y_i \Lambda_i} = H(m)^d$

Se provassimo ad eseguire questo schema, troveremmo che non funziona, perché:

**Dimostrazione** La parte i-esima deve calcolare:  $H(m)^{y_i \Lambda_i} \bmod N$ , dove

$$\Lambda_{x_i} = \prod_{\text{shares } x_k \neq x_i} \frac{-x_k}{x_i - x_k} = \frac{\alpha_{x_i}}{\beta_{x_i}} = \alpha_{x_i} \beta_{x_i}^{-1}$$

Quindi, la firma sullo share sarebbe:

$$H(m)^{y_i \Lambda_i} \bmod N = H(m)^{y_i \alpha_{x_i} \beta_{x_i}^{-1}} \bmod N$$

Tuttavia, il funzionamento di RSA è dovuto al fatto che l'esponenziale è calcolato rispetto una potenza **intera**, quindi non possiamo fare l'esponenziale di una frazione. Tra l'altro, questo significherebbe calcolare  $H(m)^{\frac{1}{e}}$  il che significherebbe **rompere RSA** perché la sua sicurezza **risiede** nel fatto che **non possiamo calcolare**  $e^{-1}$  senza conoscere  $\phi(n)$  perché equivarrebbe a **trovare una fattorizzazione**. ■

**Osservazione:** Se il termine  $\beta$  fosse pari, non potremmo neanche calcolare l'inversa modulare. ■

 **Nota** Uno schema  $(n,n)$ , non avendo la frazione (no termine di Lagrange), avrebbe funzionato.

**Definizione 13.4 (RSA ( $n,n$ ) Signature)**

- **RSA Setup:** seleziona  $p, q$  large primes tale che

$$N = p \cdot q \longrightarrow \phi(N) = (p-1)(q-1)$$

$$1 < e < \phi(N) : \gcd(e, \phi) = 1 \longrightarrow d = e^{-1} \bmod \phi$$

- **Dealer:**

- genera  $n$  shares truly random tale che:

$$y_i = \begin{cases} r_i \bmod \phi & \text{if } i \neq n \\ d - \sum_{i=1}^n r_i \bmod \phi & \text{if } i = n \end{cases}$$

- invia ad ogni peer il relativo share.

- **Receiver:** Ogni peer esegue

- Dato il messaggio  $m$  calcola  $H(m)$ .
- Si crea uno **share della firma**:  $w_i = H(m)^{y_i} \bmod N$
- Si comunica lo share agli altri peer.

- **Verifica della firma:** ogni peer raccoglie gli share delle altre parti e

- Si ricostruisce la firma condivisa:

$$\prod_{i=1}^n w_i = H(m)^{\sum_{i=1}^n y_i} = H(m)^{d - \sum_{i=1}^n r_i + \sum_{i=1}^n r_i} = H(m)^d \bmod N$$

- Si verifica la correttezza della firma calcolando  $H(m)$  e verificando che

$$H(m) = (H(m)^d)^e \bmod N$$

□

Vediamo una soluzione al problema, proposta da Victor Shoup: Consideriamo uno scenario con  $L$  player e assumiamo come sempre che lo share sia del tipo  $x_i = i$ . La base di lagrange che andrà calcolata nel caso di uno schema  $(L, L)$  sarà del tipo:

$$\Lambda_i = \prod_{\substack{\text{shares } k \neq i}} \frac{x - k}{i - k} = \frac{\text{something}(x)}{(i-1)(i-2)\dots(i(i-1))(i-(i+1))\dots(i-L)} \quad (13.3)$$

**Osservazione:** Si può vedere che il denominatore di eq. (13.3) è **sicuramente un divisore** di  $i!(L-i)!$  in quanto la **prima metà** è proprio  $(i-1)!$  e la **seconda metà** è  $(L-i)!$ . Se adesso consideriamo il coefficiente binomiale  $\binom{L}{i} = \frac{L!}{i!(L-i)!}$  possiamo vedere che  $L! \Lambda_i = \overline{\Lambda}_i(x)$  è sicuramente un numero intero porta a 1 il denominatore di  $\Lambda_i$ . ■

Quindi lo schema della soluzione potrebbe essere il seguente: Come **Dealer**:

1. Scegliere numeri primi **grandi**  $p, q$  tale che  $N = pq$ . Allora  $\phi(N) = (p-1)(q-1)$ .
2. Scelgo public-key  $e$  coprimo con  $\phi(N)$  e calcolo la secret-key  $d = e^{-1} \bmod \phi(N)$ .
3. Genero polinomio  $f(x)$  di grado  $t-1$  tale che  $f(0) = d$ .
4. Invio share  $i$ -esimo:  $(i, y_i) \bmod \phi(N)$
5. Firmo lo share<sup>2</sup> come:  $H(m)^{y_i \overline{\Lambda}_{x_i}} \bmod N$  dove

$$\overline{\Lambda}_{x_i} = L! \Lambda_i = L! \prod_{\substack{\text{shares } x_k \neq x_i}} \frac{-x_k}{x_i - x_k}$$

<sup>2</sup>Victor Shoup - RSA Threshdol

Come **Verifier** per ricostruire la firma faccio:

$$\prod H(m)^{y_i \bar{\Lambda}_{x_i}} = H(m)^{L! \sum y_i \Lambda_{x_i}} = H(m)^{d \cdot L!} \bmod N$$



**Nota Problema:** la firma creata non è la firma, ma la firma elevata ad una costante,  $L!$ . Potremmo decidere di dividere per la stessa costante, ma torneremmo al caso precedente, introducendo il calcolo dell'inversa in modulo  $\phi(N)$ , che non possiamo fare.

### 13.2.1 RSA Common Modulus Attack

La nota scritta prima non è necessariamente vera, poiché l'approccio "alla Shoup" descritto prima non è completo.

Introduciamo innanzitutto una vulnerabilità di RSA:

#### Proposizione 13.1 (Modulus Reuse)

In RSA, mai riutilizzare due volte lo stesso modulo  $N$  per due chiavi pubbliche differenti. □

Questo può succedere quando su diversi sistemi di cifratura (ad esempio due siti diversi gestiti dalla stessa entità) utilizziamo lo stesso numero primo  $N$  per due chiavi pubbliche differenti. Lo scenario è abbastanza comune, in quanto la parte difficile di RSA è trovare  $p, q$  che fattorizzano  $N$ , mentre la parte facile è derivare quante chiavi pubbliche vogliamo, purché  $e_i$  sia coprima con  $\phi(N) = (p - 1)(q - 1)$  e  $p - 1, q - 1$  siano **strong-primes**.

Utilizziamo un esempio per esprimere il concetto del **Common Modulus Attack**

#### Teorema 13.1 (RSA Common Modulus Attack)

- Supponiamo che Alice cifri un messaggio  $m$  con la public-key  $e_a$  e che Bob cifri lo stesso messaggio  $m$  con una chiave pubblica  $e_b$ .
- Supponiamo anche che il numero  $N$  su cui calcoliamo il modulo sia lo stesso.

**Osservazione:** Questo implica che le chiavi pubbliche sono coprime tra loro in quanto devono essere coprime con  $\phi(N)$ . □

Si può dimostrare che tramite **Ext. Euclidean Algorithm** si può decryptare il messaggio, ovvero:

Troviamo  $r, s$  interi tale che  $e_a \cdot r + e_b \cdot s = 1$

**Dimostrazione** Applichiamo l'Extended-GCD al nostro caso per trovare  $r, s$  interi e prendiamo i due ciphertext  $c_1 = m^{e_a}, c_2 = m^{e_b}$ .

Adesso calcoliamo:

$$c_1^r \cdot c_2^s = (m^{e_a})^r + (m^{e_b})^s = m^{e_a \cdot r + e_b \cdot s} = m \quad (13.4)$$

E abbiamo decrittato il messaggio senza conoscere la chiave di decrittazione/privata. □

**Osservazione:** RSA Common Modulus Attack è possibile quando:

- due sistemi RSA usano lo stesso modulo  $N$  per due differenti coppie  $(e, d)$  e viene cifrato lo stesso messaggio

<sup>3</sup>Adesso è possibile poiché esponente intero e non serve più l'inversa.



Utilizziamo ora l'eq. (13.4) a nostro vantaggio per eliminare il fattore  $L!$ .

### Definizione 13.5 (Shoup - RSA Threshold)

*Come Dealer:*

1. Scegliere numeri primi **grandi**  $p, q$  tale che  $N = pq$ . Allora  $\phi(N) = (p-1)(q-1)$ .
2. Scelgo public-key e coprimo con  $\phi(N)$  e calcolo la secret-key  $d = e^{-1} \bmod \phi(N)$ .
3. Genero polinomio  $f(x)$  di grado  $t-1$  tale che  $f(0) = d$ .
4. Invio share  $i$ -esimo:  $(i, y_i) \bmod \phi(N)$
5. Firmo lo share<sup>4</sup> come:  $H(m)^{y_i \bar{\Lambda}_{x_i}} \bmod N$  dove

$$\bar{\Lambda}_{x_i} = L! \Lambda_i = L! \prod_{\substack{\text{shares } x_k \neq x_i}} \frac{-x_k}{x_i - x_k}$$

*Come Verifier per ricostruire la firma faccio:*

$$\prod H(m)^{y_i \bar{\Lambda}_{x_i}} = H(m)^{L! \sum y_i \bar{\Lambda}_{x_i}} = H(m)^{d \cdot L!} \bmod N$$

**Osservazione:** Sia ora  $L! = \Delta$ . Allora:

$$H(m)^{d \cdot L!} = H(m)^{d \Delta} = v^\Delta$$

Osserviamo che la quantità  $H(m) = [H(m)^d]^e = v^e$  è nota<sup>5</sup>. Poiché vogliamo  $H(m)^d = v$  per controllare la firma, usando l'RSA Common Modulus Attack (thm 13.1) troviamo  $v$ , se  $\Delta$  ed **e sono coprimi**<sup>6</sup>:

- tramite  $\text{ExtGCD}(e, \Delta)$  troviamo  $(r, s)$ .
- Calcoliamo:  $(v^e)^r \cdot (v^\Delta)^s = v^{e \cdot r + \Delta \cdot s} = v$



<sup>4</sup>Adesso è possibile poiché esponente intero e non serve più l'inversa.

<sup>5</sup>La chiave di cifratura  $e$  è pubblica e può essere usata.

<sup>6</sup>Questo succede praticamente sempre se  $e$  è un numero pimo più grande di  $L$ . Altrimenti la firma con RSA resta uguale.



**Osservazione:** Se svolgiamo i calcoli sul sottogruppo dei **Residui Quadratici** (def 12.14) e usiamo primi forti (eq. (12.7))  $p = 2p'+1, q = 2q'+1$  allora il periodo di  $N = pq$  sarà  $\phi(N) = (p-1)(q-1) = 2p' \cdot 2q' = 4p'q'$ . Il periodo spennato da questo numero è **molto grande**, anche se non è primo. Quindi non c'è il problema di invertire il modulo a meno di voler invertire un numero del tipo  $1/p'$  ma non succede praticamente mai.



**Nota** Schoup dimostrò anche la verificabilità del secret sharing fatto con questa tecnica.

**Nota** Il problema che questo schema ha è che **almeno una** persona nel mondo conosce  $\phi(N)$ , il Dealer. Questo problema è stato risolto da Foque e Stern, nel 2001. Non lo vediamo, anche perché RSA al giorno d'oggi è obsoleto rispetto ad altri sistemi di sicurezza.

# Capitolo Mobile Devices Resilient to Capture

Quando un dispositivo mobile viene rubato, il proprietario non perde soltanto il valore economico del dispositivo ma anche i dati al suo interno. E' pertanto un problema non banale quello di tutelare gli utenti affinché dei dati "fisicamente" rubati, non possano essere acceduti.

Vorremmo creare uno schema che renda un device impossibile di essere usato senza il consenso del proprietario, ovvero:

## Definizione 14.1 (Capture Resilient Device)

*Dispositivo che può essere usato solo e soltanto dal proprietario.*



**Nota** Assumiamo che il "**core**" di questo dispositivo sia una secret-key.

Alcuni approcci possibili possono essere:

- password per bloccare/sbloccare (cifrare/decifrare) il dispositivo.
- proteggere il segreto in un hardware "tamper-proof" (hardware che non può essere manomesso).
- Download di una chiave da una repository in rete quando serve.

Sappiamo che una password è una soluzione **generalmente** debole per proteggere un segreto, una repository remota (o il suo gestore) deve essere affidabile o potrebbero esserci altri problemi e un hardware **realmente** impossibile da manomettere **per sempre** non è detto esista. Sembra quindi che siano necessarie altri sistemi.

## 14.1 Capture-Protection Server

MacKenzie e Raiter nel 2003 risolsero il problema, creando uno schema *a tre player* che coinvolgeva un **untrusted server** che poteva essere anche non fidato e due approcci possibili: standard cryptography o secret sharing.



**Nota** L'idea è che se un attaccante dovesse riuscire a rubare il segreto di uno dei tre attori o una combinazione di 2 attori su 3, **non è possibile** dedurre niente e rompere il sistema.

**Osservazione:** C'è una combinazione che può rompere il sistema: se un attaccante riuscisse a rubare sia la password che il device, allora sarebbe "digitalmente identico" al proprietario e non ci sarebbe niente da fare. **A meno di usare secret sharing**

## Definizione 14.2 (Capture-Protection Server)

Consideriamo:

- **Utente**  $U$ , detiene **password**  $P$ .
- **Dispositivo**  $D$ , detiene  $SK_D, PK_D$ .
- **Server Remoto**  $S$ , detiene  $SK_S, PK_S$ .

Assumiamo che il **device** sia **sempre connesso alla rete** e che la sua chiave sia **protetta da cifratura** e che sia decifrabile solo cooperando con il server.

Il protocollo si divide in due fasi:



**Definizione 14.3 (Capture-Protection Server: Device Init)**

1. L'utente registra, temporaneamente,  $P$  nel device.
2. Scarichiamo, temporaneamente, sul device la  $PK_S$ .
3. Generiamo due chiavi random  $v, a$  e calcoliamo

$$b = H(P)$$

4. Tramite stream-cipher<sup>1</sup> cifriamo la  $SK_D$

$$c = SK_D \oplus PRF(v, P)$$

Dove il keystream generato dalla PRF dipende da un valore generato da  $D$  e da un valore generato da  $U$ : **La ricostruzione è possibile solo se conosco entrambi**<sup>2</sup>.

5. Generiamo un ticket<sup>3</sup> tramite una funzione di envelope:

$$tkt = E_{PK_S}[a, b, c] = [a, H(real\ pswd), SK_S \oplus keystream]$$

e lo tengo in locale per un momento successivo.

6. Cancello dal device: la password  $P$  e il suo hash  $b$ , il ciphertext  $c$  e la  $SK_D$ .

<sup>1</sup>ad esempio AES-CTR (def 7.8)

<sup>2</sup>Questa costruzione permette di evitare il secret-sharing poiché il funzionamento è analogo: genero un segreto a partire da due valori, senza i quali non è rigenerabile.

<sup>3</sup>Poiché l'envelope è fatto in funzione della public-key del server, questo la potrà decriptare ma non verrà mai a conoscenza degli altri segreti poiché sono cifrati.

**Definizione 14.4 (Capture-Protection Server: Key Retrieval)**

1. Nel device troviamo:  $v, a, tkt, PK_D$ .
2. L'utente accede al dispositivo con  $P$ .
3. Calcoliamo l'hash della password inserita:

$$\beta = H(P)$$

se l'utente è legittimo  $\beta \equiv b$ .

4. Genero un random  $\rho$  per cifrare il canale di comunicazione.
5. Invio al remote-server il  $tkt$  e la sua autenticazione:

$$HMAC_a[tkt, E_{PK_S}[\beta, \rho]]$$

6. Il server decrypta il ticket e l'envelope, leggendo  $a, b, c, \beta, \rho$ .
7. Il server controlla l'HMAC ricevuto utilizzando la chiave  $a$  e autentica il device.
8. Il server decrypta l'hash della password  $\beta$  e la confronta con  $b$  per autenticare l'utente.
9. Il server invia  $\rho \oplus c$ , per proteggere il trasporto della secret-key (cifrata) del device.
10. Il device ottiene la sua chiave privata decifrando:

$$(\rho \oplus c) \oplus \rho \oplus PRF(v, P) = SK_D$$

Solo un utente che ha il device e la password può decifrare la  $SK_D$ .



**Nota** Un attaccante in possesso del device potrebbe vedere soltanto i valori di  $v, a$  e il  $tkt$ . Tuttavia, dovrebbe riuscire a rompere l'envelope e dopo invertire l'hash o rompere  $c$ .

Le vulnerabilità aperte su questo schema sono due:

- **L'attaccante viene in possesso di  $SK_D$ :** in questo caso è possibile ad esempio disconnettere il device dalla rete e accedere a tutto.
- **device rubato e password predicibile:** prima o poi sarà possibile accedere alla secret-key facendo brute-forcing.

Negli altri casi siamo protetti:

- **Server Cracked AND Password Known:** l'attaccante non può firmare/decifrare perché  $v$  è **tenuta segreta nel device**, quindi  $SK_D$  è cifrata con  $PRV(v, P)$  e non può essere ottenuta.
- **Device Cracked/Stolen:** l'attaccante può soltanto fare dictionary attack **online**. TUTTAVIA è facilmente individuabile perché anche se i **tentativi** di attacco sono **autenticati**, la password fallisce molte volte, a meno di avere una password difficile.
- **Device and Server Cracked:** l'attaccante può soltanto fare dictionary attack **offline** perché conosce  $b$  e cercare la password.

La vulnerabilità rimasta al protocollo può essere fixata introducendo un **threshold secret-sharing**, nascondendo la SK anche allo stesso device.

#### Corollario 14.1 (Threshold SS for Capture-Protection: Device Init)

*Supponiamo di utilizzare uno schema di Secret-Sharing (2,2) con RSA come sistema di firma:*

1. L'utente registra, temporaneamente,  $P$  nel device.
2. Scarichiamo, temporaneamente, sul device la  $PK_S$ .
3. Generiamo **due chiavi random**  $v, a$  e calcoliamo

$$b = H(P)$$

4. Assegniamo  $n = p \cdot q$ . Quindi  $\phi(n) = (p - 1)(q - 1)$  e seleziono  $e, d$  secondo **RSA** (def 9.6)
5. Genero Share 1:  $d_1 = PRF(v, P)$ .<sup>4</sup>
6. Genero Share 2:  $d_2 = d - d_1 \bmod \phi(n)$ .
7. Genero **Disabling-Key**:  $t$ . Questa chiave va **mantenuta offline**.
8. Genero **Disabling Value**:  $u = H(t)$ .
9. Genero ed invio un **ticket** tramite una funzione di **envelope**:

$$tkt = E_{PK_S}[a, b, u, d_2, N]$$

- $a$ : device auth.
- $b$ : pw auth.
- $d_2$ : second share.
- $N$ : RSA sign.

10. **Cancello** dal device tutto ciò che è in **rosso**.

<sup>4</sup>La ricostruzione dello share **a posteriori** è possibile **se solo se** conosco sia  $v$  che  $P$

**Corollario 14.2 (Threshold SS for Capture-Protection: Key Retrieval)**

Sia  $m$  il messaggio da firmare, **ogni volta** che avviene una transazione con il server.

1. Nel device troviamo:  $v, a, tkt, e, N$ .
2. L'utente **accede** al dispositivo con  $P$ .
3. Calcoliamo l'hash della password **inserita**:

$$\beta = H(P)$$

se l'utente è legittimo  $\beta \equiv b$ .

4. Genero un random  $\rho$  per cifrare il canale di comunicazione.
5. Invio al **remote-server** il  $tkt$  e la sua autenticazione:

$$\text{HMAC}_a[tkt, E_{PK_S}[H(m), \beta, \rho]]$$

6. Il server decrypta il ticket, leggendo  $a, b, d_2, u, N$ .
7. Il server **controlla** l'HMAC ricevuto utilizzando la chiave  $a$  e **autentica** il device.
8. Se solo se il device è autenticato il server decrypta il resto del messaggio leggendo  $H[m]\beta, \rho$ .
9. Il server confronta  $\beta$  con  $b$  per **autenticare** l'utente.
10. Il server controlla se  $u \neq H(t)$ , dove  $t$  è una delle **disabling-key** mantenute nella black-list del server. Se c'è un **match**, allora il device è stato rubato e ignora il resto del protocollo, altrimenti...
11. Il server invia la sua firma del messaggio  $\rho \oplus H[m]^{d_2}$ , per proteggere il trasporto della secret-key (cifrata) del device.
12. Il device decifra il messaggio ottenuto per ottenere

$$H[m]^{d_2} = (\rho \oplus H[m]^{d_2}) \oplus \rho$$

13. Il device **completa la firma**, calcolando a runtime:

$$d_1 = PRF[v, P]$$

$$H[m]^{d_2} H[m]^{d_1} \bmod N$$

□



**Nota** La signature-key  $d$  non è mai trasmessa in chiaro. Pertanto, se un attaccante ottiene sia la password che il device, non può ottenere  $d$  e firmare messaggi come fosse il proprietario.

Con questo metodo, possiamo ampliare il protocollo con una funzione di disattivazione:

**Corollario 14.3 (Threshold SS for Capture-Protection: Key Disabling)**

Il server mantiene un database dove salva le associazioni  $\{ticket, t\}$  che vengono black-listate. Se un utente **mantiene una copia off-line** di  $t$ , e il suo device viene rubato, può inviare al server in un qualsiasi momento un messaggio contenente la disabling-key e il server dopo aver autenticato l'utente inserirà il device nella black-list.

□

# Capitolo Linear Secret Sharing

Vediamo un modello di generalizzazione di secret sharing affinché invece di avere uno schema basato su threshold, la condizione di accesso al segreto sia un **predicato "arbitrario"** che specifichi una specifica logica.

Secondo lo schema di Shamir (thm 12.4), uno *share* è un *polinomio*, che può **sempre** essere visto come il **prodotto scalare** di due vettori: uno di **variabili simboliche**  $x_i^k$  e uno di **coefficienti randomici**  $r_k$ ,  $k \in [1, t - 1]$ . Questo ci porta a poter riscrivere lo schema di secret-sharing sottoforma di sistema lineare:

$$Ax = b$$

Dove  $A$  è una matrice  $n \times t$  **data**,  $x$  è un vettore di dimensione  $t$  e  $b$  uno di dimensione  $n$ . In particolare,  $x$  è tale che **la prima componente** rappresenta il segreto mentre le altre i coefficienti casuali.

## Esempio 15.1 (3,4) Secret-Sharing

$$\begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_3 & x_3^2 \\ 1 & x_4 & x_4^2 \end{bmatrix} \begin{bmatrix} s \\ r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

Questa scrittura di  $A$  è anche detta **Matrice di Vandermonde**.<sup>1</sup> Il segreto può essere ricostruito banalmente **risolvendo il sistema**:

$$\begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_4 & x_4^2 \end{bmatrix} \begin{bmatrix} s \\ r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} y_1 \\ y_2 \\ y_4 \end{bmatrix}$$

$$\begin{bmatrix} s \\ r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} 1 & x_1 & x_1^2 \\ 1 & x_2 & x_2^2 \\ 1 & x_4 & x_4^2 \end{bmatrix}^{-1} \begin{bmatrix} y_1 \\ y_2 \\ y_4 \end{bmatrix} = \begin{bmatrix} \frac{x_2x_4}{(x_2-x_1)(x_4-x_1)} & \frac{-x_1x_4}{(x_2-x_1)(x_4-x_2)} & \frac{x_1x_2}{(x_4-x_1)(x_4-x_2)} \\ \frac{-x_4+x_2}{(x_2-x_1)(x_4-x_1)} & \frac{x_4+x_1}{(x_2-x_1)(x_4-x_2)} & \frac{-x_2+x_1}{(x_4-x_1)(x_4-x_2)} \\ \frac{1}{(x_2-x_1)(x_4-x_1)} & \frac{1}{(x_2-x_1)(x_4-x_2)} & \frac{1}{(x_4-x_1)(x_4-x_2)} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_4 \end{bmatrix}$$

Eseguendo il prodotto per la prima riga troviamo:

$$s = y_1 \frac{x_2x_4}{(x_2-x_1)(x_4-x_1)} - y_2 \frac{x_1x_4}{(x_2-x_1)(x_4-x_2)} + y_4 \frac{x_1x_2}{(x_4-x_1)(x_4-x_2)}$$

Che coincide con l'interpolazione di Lagrange (eq. (12.1)).

**Osservazione:** Un'altra interpretazione è possibile!

---

<sup>1</sup>Vandermonde Matrix.

## 15.1 Monotone Span Programs

Consideriamo ogni *riga* di  $A$  come vettore in  $\mathbb{R}$  e, conseguentemente, il segreto  $s$  come un punto qualsiasi sull'asse  $x$ . Da teoremi di algebra lineare sappiamo che una soluzione per il sistema esiste se il rango di  $A$  è pieno, ovvero:

$$\mathbb{R}^3 = \text{span} \left\{ \begin{pmatrix} 1 \\ x_1 \\ x_1^2 \end{pmatrix}, \begin{pmatrix} 1 \\ x_2 \\ x_2^2 \end{pmatrix}, \begin{pmatrix} 1 \\ x_4 \\ x_4^2 \end{pmatrix} \right\}$$

Questa condizione ci dice che **è possibile ricostruire** il segreto **se solo se** esiste una combinazione lineare non banale dei vettori precedenti. Il che ci porta ad un'altra considerazione:

**Esempio 15.2** Verifichiamo che:

$$\begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix} \in \text{span} \left\{ \begin{pmatrix} 1 \\ x_1 \\ x_1^2 \end{pmatrix}, \begin{pmatrix} 1 \\ x_2 \\ x_2^2 \end{pmatrix}, \begin{pmatrix} 1 \\ x_4 \\ x_4^2 \end{pmatrix} \right\}$$

Consideriamo quindi tre coefficienti  $c_1, c_2, c_3$  tali che:

$$c_1(1, x_1, x_1^2) + c_2(1, x_2, x_2^2) + c_3(1, x_4, x_4^2) = (1, 0, 0) \quad (15.1)$$

Ovvero, vogliamo risolvere il sistema:

$$\begin{pmatrix} 1 & 1 & 1 \\ x_1 & x_2 & x_4 \\ x_1^2 & x_2^2 & x_4^2 \end{pmatrix} \begin{pmatrix} c_1 \\ c_2 \\ c_3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Risolvendo, troviamo che i coefficienti cercati sono proprio:

$$c_1 = \frac{x_2 x_4}{(x_2 - x_1)(x_4 - x_1)}; \quad c_2 = \frac{x_1 x_4}{(x_2 - x_1)(x_4 - x_2)}; \quad c_3 = \frac{x_1 x_2}{(x_4 - x_1)(x_4 - x_2)}$$



**Osservazione:** Avendo scelto  $s$  appartenente all'asse  $x$ , quello che abbiamo trovato sono proprio i coefficienti necessari ad ottenere  $s$  e **coincidono** con quelli di Lagrange. Questo significa che se moltiplichiamo la combinazione lineare in eq. (15.1) per il vettore  $(s, r_1, r_2)^\top$  eseguiamo nuovamente l'interpolazione:

$$c_1(1, x_1, x_1^2) + c_2(1, x_2, x_2^2) + c_3(1, x_4, x_4^2) \begin{pmatrix} s \\ r_1 \\ r_2 \end{pmatrix} = (1, 0, 0) \begin{pmatrix} s \\ r_1 \\ r_2 \end{pmatrix} = s$$

$$c_1 \underbrace{(1, x_1, x_1^2)}_{y_1} \begin{pmatrix} s \\ r_1 \\ r_2 \end{pmatrix} + c_2(1, x_2, x_2^2) \begin{pmatrix} s \\ r_1 \\ r_2 \end{pmatrix} + c_3(1, x_4, x_4^2) \begin{pmatrix} s \\ r_1 \\ r_2 \end{pmatrix} = s$$

$$c_1 y_1 + c_2 y_2 + c_3 y_4 = s$$

Dove  $y_i$  è lo share assegnato alla parte  $i$ -esima.



**Nota** Vedendo il problema dello sharing come problema di span, è possibile vedere ancora meglio come la mancanza di uno share renda impossibile il calcolo del segreto in quanto la matrice di  $A$  non

sarebbe a rango pieno e il problema non si potrebbe porre.

Salta fuori, inoltre, che:

### Teorema 15.1 (LSSS = MSP<sup>2</sup>)

Ogni Linear Secret Sharing Scheme è equivalente ad un problema di span.

<sup>2</sup>Monotone Span Programs



E' possibile quindi costruire uno schema che permetta di assegnare un valore ai coefficienti della matrice  $A$  per assegnare proprietà desiderate.

**Esempio 15.3 Trivial SS as LSSS** Consideriamo un SS (3,3). Un dealer genera  $s$  e assegna come share  $r_1, r_2, s - r_1 - r_2$  con  $r_1, r_2$  casuali secondo certe proprietà (thm 12.2). Allora:

$$\begin{pmatrix} 1 & -1 & -1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} s \\ r_1 \\ r_2 \end{pmatrix} = \begin{pmatrix} s - r_1 - r_2 \\ r_1 \\ r_2 \end{pmatrix}$$

Considerando il problema di span associato ("Monotone Span Program"):

$$c_1(1, -1, -1) + c_2(0, 1, 0) + c_3(0, 0, -1) = (1, 0, 0)$$

Troviamo che  $c_1 = c_2 = c_3 = 1$  e la ricostruzione del segreto è quindi banale:

$$c_1(s - r_1 + r_2) + c_2r_1 + c_3r_2 = (s - r_1 + r_2) + r_1 + r_2 = s$$



**Nota** Senza uno dei tre partecipanti, non sarebbe possibile ricostruire il segreto perché non ci sarebbe una combinazione lineare che soddisfa l'uguaglianza.



**Nota** Supponiamo di passare da uno schema (3,3) ad uno (3,4) e che il quarto share assegnato sia del tipo  $r_1 + r_2$ . Questo significa che la riga da aggiungere ad  $A$  sarebbe  $(0, 1, 1)$ . A questo punto basterebbero i player 1 e 4 per ricostruire il segreto, o meglio, per ricostruire lo spazio dove **risiede** il vettore  $e_x$  del quale  $s$  è multiplo.

$$c_1(1, -1, -1) + c_4(0, 1, 1) = (1, 0, 0) \longrightarrow c_1 = c_4 = 1$$

$$c_1(s - r_1 + r_2) + c_4(r_1 + r_2) = s$$

Si vede quindi che è possibile ricostruire il segreto in due condizioni:  $(P_1 \wedge P_3 \wedge P_3) \vee (P_1 \wedge P_4)$ . Questa condizione ci porta dai threshold scheme a **policy-based** scheme, nei quali un player potrebbe essere più potente di altri e permettere di costruire schemi più complessi e articolati.

Vale il seguente teorema:

### Teorema 15.2 (LSSS is Homomorphic)

Qualsiasi schema LSSS è omomorfico:

$$x_a = (s_a, r_{1a}, r_{2a}, \dots) \quad y_a = (share_{1a}, share_{2a}, \dots)$$

$$x_b = (s_b, r_{1b}, r_{2b}, \dots) \quad y_b = (share_{1b}, share_{2b}, \dots)$$

$$Ax_a = y_a \quad Ax_b = y_b$$

$$y_a + y_b = Ax_a + Ax_b = A(x_a + x_b) = A(s_a + s_b, rand, \dots)$$





**Nota** Tutti i ragionamenti fatti fino ad ora continuano a valere a prescindere dal campo rispetto al quale facciamo le operazioni!

### Esempio 15.4 Monotone Span Programs in Galois Field 2

Supponiamo di essere in **GF2** (somme o sottrazioni diventano xor e i coefficienti sono  $\{0,1\}$ ) e di avere **5 parti**. Supponendo di avere la matrice in fig. 15.1. Vediamo che alla parte 2 sono stati associati 2 share.

L'ultima riga è formata dal vettore che vogliamo ricostruire a partire dalla matrice. Questo è possibile se solo se è ottenibile tramite xor di due o più delle righe della matrice.

Si può vedere che:  $P_{21} \oplus P_{22} \oplus P_4 = (1, 0, 0, 0)$ .

Questo significa che con  $P_2$  e  $P_4$  è possibile ricostruire il vettore target.

Vediamo ora come ricostruire il segreto, associando il vettore di rand:  $[s, r_2, r_3, r_4]$

$P_2$	1	1	0	1
$P_2$	0	1	1	0
$P_1$	0	1	1	0
$P_3$	1	1	0	0
$P_4$	0	0	1	1
	1	0	0	0

$$\begin{array}{c|c}
\begin{matrix} s \\ r_2 \\ r_3 \\ r_4 \end{matrix} & = \begin{matrix} s + r_2 + r_4 \\ r_2 + r_3 \\ r_2 + r_3 \\ s + r_2 \\ r_3 + r_4 \end{matrix} \\
\hline
\end{array}$$

Figura 15.2: Secret Reconstruction with MSP



**Nota** Qualsiasi altra combinazione di  $P_i$ , per questo problema, non permetterebbe di ricostruire il segreto.



**Nota** Si può vedere che gli share da distribuire corrispondono al prodotto scalare della matrice per il vettore delle parti. Se ad esempio avessimo assunto  $(s, r_2, r_3, r_4) = (1, 0, 0, 1)$  gli share distribuiti sarebbero stati  $(0, 0, 0, 1, 1)$ , rispettivamente per le parti  $P_2, P_2, P_1, P_3, P_4$ .



Dall'esempio, si può vedere il senso della seguente definizione:

#### Definizione 15.1 (MSP Accept)

Un MSP accetta un set di parti  $B$  se le righe etichettate da  $B$  eseguono uno span del vettore target.



## 15.2 Access Structure

Vediamo come costruire un sistema di accesso basato su una certa policy che cambia a seconda delle parti coinvolte nel calcolo del segreto.

**Definizione 15.2 (Access Structure)**

Sia  $\mathcal{P}$  un insieme di entità. Una **Struttura di Accesso** è un insieme costituito da tutti quei sottoinsiemi  $X \subseteq \mathcal{P}$  che permettono di **accedere** ad un **segreto**, partendo dalla prossima definizione.

Diremo che una struttura d'accesso è **monotona** se la struttura **non supporta** l'operatore di negazione. □

**Esempio 15.5** Consideriamo un insieme di partecipanti  $P = \{P_1, P_2, P_3, P_4\}$ . Supponiamo che il segreto celato dai partecipanti possa essere ricostruito **se solo se** si incontrano  $P_1$  e  $P_2$  o  $P_1, P_3, P_4$ . La struttura d'accesso **monotona** che ne deriva è l'insieme

$$A = \{[P_1, P_2], [P_1, P_3, P_4]\}$$

Il quale equivalente booleano è:  $(P_1 \wedge P_2) \vee (P_1 \wedge P_2 \wedge P_3) = P_1 \wedge (P_2 \vee (P_3 \wedge P_4))$  ■

**Osservazione:** Gli schemi che costruiremo **potrebbero non essere ideali**, poiché una parte potrebbe ricevere più di uno share. ■

Questo fatto è osservabile costruendo effettivamente la matrice di accesso di controllo corrispondente alla politica da implementare. Consideriamo la struttura d'accesso dell'esempio 15.5:  $P_1 \wedge (P_2 \vee (P_3 \wedge P_4))$  e deriviamo la matrice corrispondente creando un **access control tree**.

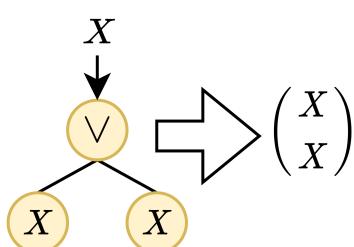
**Definizione 15.3 (Access Control Tree)**

Un **Access Control Tree** è un albero binario che implementa la politica di una struttura di accesso. I nodi sono gli operatori booleani della politica, le foglie le parti che la compongono. □

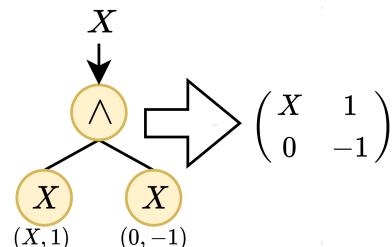
**Definizione 15.4 (Access Control Matrix)**

Una **Access Control Matrix** è una matrice **derivata** da un **ACT**, per implementare una Access Structure. In particolare:

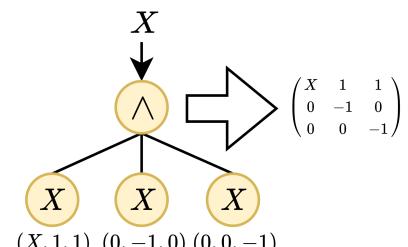
- Un nodo **OR** (fig. 15.3a) costruisce un vettore le cui **componenti** sono **pari al segreto in ingresso** al nodo.
- Un nodo **AND** (fig. 15.3b e 15.3c) **aumenta** la dimensione dello al numero di attori in gioco. La prima riga sarà del tipo  $(X, 1, \dots, 1)$ , le altre saranno **nulle** in ogni componente **tranne** che nella colonna  $i$ -esima, il cui valore dovrà annullare l' $1$  contenuto nella prima riga.



(a) OR-ACM from OR-ACT



(b) AND-ACM from AND-ACT

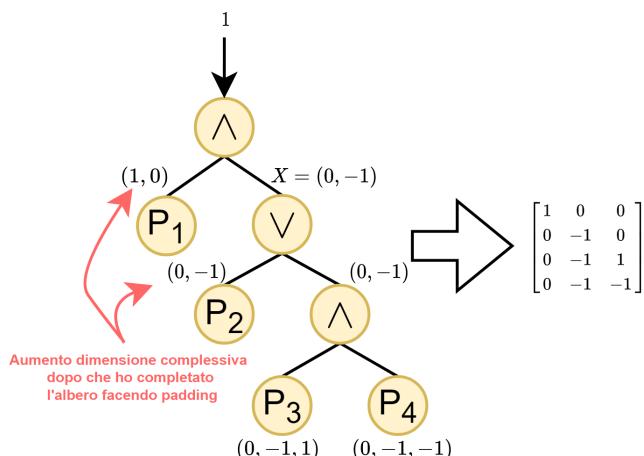


(c) AND-ACM from AND-ACT

**Figura 15.3:** From ACT to ACM

**Esempio 15.6 Access Control Matrix** Scriviamo la ACM equivalente alla seguente proposizione:

$$(P_1 \wedge P_2) \vee (P_1 \wedge P_2 \wedge P_3) = P_1 \wedge (P_2 \vee (P_3 \wedge P_4))$$



**Figura 15.4:** ACM from policy

**Nota** Se avessimo calcolato la ACM senza semplificarla, avremmo avuto più entry per P1 e P2 e questo non sarebbe stato uno schema **Ideale**. Tuttavia, la soluzione sub-ottimale è semplice e garantisce comunque sicurezza, senza necessariamente usare numeri primi, perché la sicurezza è garantita dalla policy.

### Corollario 15.1

Con una ACM non è possibile implementare schemi ideali. E' possibile modificare lo schema per aggiungere dei "**threshold Gates**"

**Esempio 15.7** Proviamo ad implementare la policy  $(A \wedge B) \vee (A \wedge C) \vee (B \wedge C)$  di uno schema (2,3).



# Capitolo Elliptic Curve Cryptography

Abbiamo visto come la sicurezza può essere affidata ad aspetti computazionali o logici (per teoremi matematici ecc). Per quanto riguarda la *computational security* abbiamo osservato due principi fondamentali: **Discrete-Logarithm** (prop 9.3) e **Prime-Factorization** (prop 9.4).

E' stato osservato che a seconda del gruppo usato nella crittografia, questi problemi possono essere più o meno complessi da risolvere. In particolare, alcuni dati sul tempo usato per risolvere il problema di **DLog** sono i seguenti:

- **Dlog in  $Z^*(p)$ :**  $\exp O(n^{\frac{1}{3}})$ .
- **Dlog in EC:**  $\exp O(n^{\frac{1}{2}})$ .
- **Dlog in  $(Z(p), +)$ :** polinomial-time



**Nota** Il fattore di scala  $n$  è la **dimensione** degli elementi nel campo. Le curve ellittiche sono quelle che scalano meglio, per questo vengono usate.

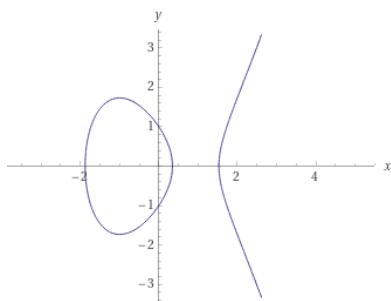
## 16.1 EC Minor Details

Le curve ellittiche non sono delle vere ellissi. Sono delle curve generate dalla **Espressione di Weierstrass**:

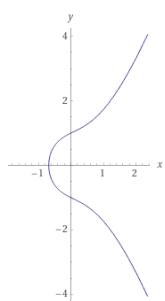
### Definizione 16.1 (Weierstrass Equation for EC)

Una curva ellittica è descritta dal seguente sistema:

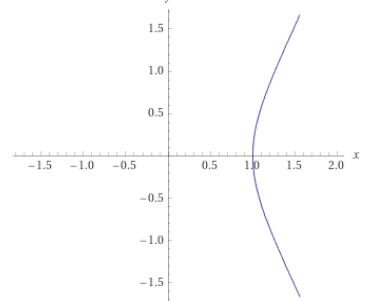
$$\begin{cases} y^2 = x^3 + ax + b \\ 4a^3 + 27b^2 \neq 0 \end{cases} \quad (16.1)$$



(a)  $y^2 = x^3 - 3x + 1$



(b)  $y^2 = x^3 + x + 1$



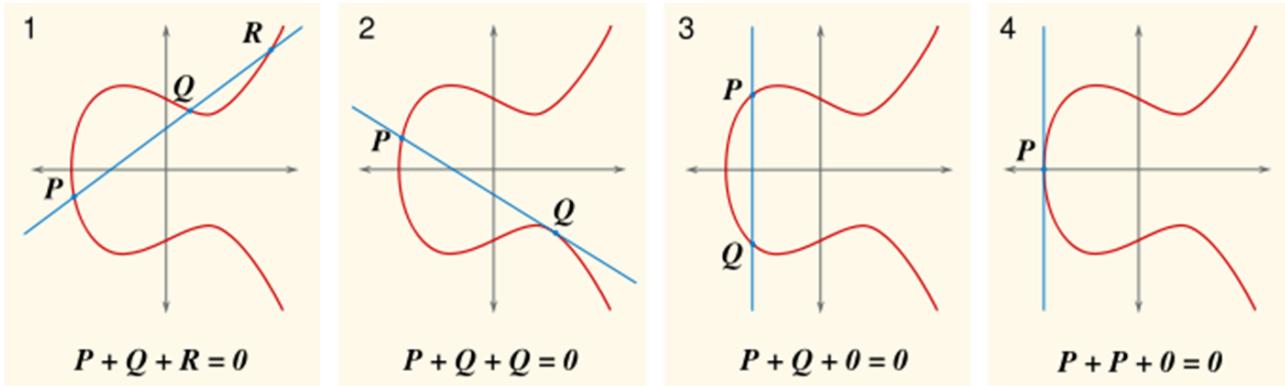
(c)  $y^2 = x^3 - 1$

**Figura 16.1:** ECC examples

Per trasformare le EC in un gruppo, definiamo un operatore che sia omomorfico, basandoci sul prossimo teorema:

**Teorema 16.1 (General Rule for EC)**

Se tre punti di una curva ellittica risiedono sulla stessa retta allora la loro somma è nulla. □

**Figura 16.2:** EC General Rule

Allora, definiamo l'operatore:

**Definizione 16.2 (EC "Sum" Operator)**

Sia  $+$  il nome dell'operatore e  $P, Q, R$  dei punti sulla curva tali che  $P + Q = R$ . Allora:

- $P \neq Q$ : tracciare la retta passante per  $P$  e  $Q$ . Il punto di intersezione con la curva è  $-R$ . Invertendo il segno otteniamo  $R$ .
  - $P = Q$ : tracciare la tangente alla curva in  $P$ .  $-R$  è il punto di intersezione con la curva. Invertendo il segno otteniamo  $R$ .
  - $-P = Q$ : completiamo l'insieme dei punti della curva con una "**Chiusura all'Infinito**", aggiungendo il punto 0. Allora  $P + 0 = P$
- 

Questa definizione ci permette di rispettare le richieste sulla chiusura dell'operatore.

Possiamo dedurre delle relazioni algebriche per definire in modo rigoroso l'operatore di somma.

**Teorema 16.2 (Algebraic Relationships in EC)**

Siano  $P = (x_1, y_1), Q = (x_2, y_2)$ . Allora  $R = P + Q = (x_3, y_3)$ . Allora:

$$\begin{aligned} x_3 &= \lambda^2 - x_1 - x_2 \\ y_3 &= \lambda(x_1 - x_3) + y_1 \\ \lambda &= \begin{cases} \frac{y_2 - y_1}{x_2 - x_1} & P \neq Q \\ \frac{3x_1^2 + a}{2y_1} & P = Q \end{cases} \end{aligned} \tag{16.2}$$
□

**Dimostrazione** Siano  $P = (x_1, y_1), Q = (x_2, y_2)$ . Allora  $R = P + Q = (x_3, y_3)$ .

- $P \neq Q$ : l'equazione della retta passante per due punti è:

$$y = y_1 + \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) = \lambda(x - x_1) + y_1 \tag{16.3}$$

Messa a sistema con la eq. (16.1) troviamo che:  $(\lambda(x - x_1) + y_1)^2 = x^3 + ax + b$ .

Espandendo e portando tutto ad un membro troviamo un'espressione del tipo:  $x^3 - \lambda^2 x^2 + \dots = 0$ .

Poiché il termine di sinistra è un polinomio di terzo grado e per costruzione del problema conosciamo 2 soluzioni su 3, allora possiamo affermare che avrà una fattorizzazione del tipo:

$$(x - x_1)(x - x_2)(x - x_3) = x^3 - (x_1 + x_2 + x_3)x^2 + \dots$$

Quindi:  $\lambda^2 = x_1 + x_2 + x_3 \rightarrow x_3 = \lambda^2 - x_1 - x_2$ . Trovare  $y_3$  è adesso elementare, basta sostituire in eq. (16.3) e cambiare il segno.

- $P = Q$ : L'equazione del fascio di rette per un punto è:

$$y = y_1 + k(x - x_1) = \lambda(x - x_1) + y_1 \quad (16.4)$$

Derivando l'equazione della curva troviamo:

$$2y \frac{dy}{dx} = 3x^2 + a \rightarrow \frac{dy}{dx} = \frac{3x^2 + a}{2y}$$

Allora:  $\lambda = k = \frac{3x_1^2 + a}{2y_1}$



## 16.2 EC over $\mathcal{Z}(p)$

L'interpretazione geometrica usata fino ad ora è valida per i numeri reali. Consideriamo adesso però l'insieme costituiti dagli *interi in modulo p*: le coordinate dei punti della curva saranno degli interi in  $\mathcal{Z}(p)$  tali che:

$$y^2 \bmod p = x^3 + ax + b \bmod p \quad (16.5)$$

Questo significa che il numero di punti ammissibili è un insieme limitato, non più  $p^2$  ma  $p - 1$  più lo zero, ovvero  $p$  punti.

**Esempio 16.1** Consideriamo il gruppo descritto da  $x^3 + x + 1 \bmod 5$ . Prendiamo allora le coppie  $(i, j)$ ,  $i, j \in [0, 4]$  tali che  $j^2 = i^3 + i + 1 \bmod 5$  e il punto 0.

Per  $x = 1$  troviamo  $y = 3$ . Se il punto  $P = (1, 3)$  è sulla curva, allora ne soddisfa l'equazione e ciò implicherebbe che esiste un numero **intero** il cui quadrato è 3:

$$3^2 \bmod 5 = 1^3 + 1 + 1 \bmod 5 \rightarrow 4 = 3$$

Poiché la relazione precedente è falsa deduciamo che il punto  $(1, 3)$  non appartiene al nostro gruppo.

Per  $x = 0$  invece troviamo  $0^3 + 0 + 1 = 1$  e la relazione della curva è banalmente verificata perché  $1^2 = 1$ , quindi il punto  $P = (0, 1)$  appartiene al gruppo.

Ciclando su tutte le coppie possiamo vedere che dei 25 punti possibili soltanto 8+1 sono i punti effettivamente nel gruppo. In particolare:

$$E(Z_5) = \{(0, 0), (0, 1), (0, 4), (1, 0), (1, 2), (1, 3), (1, 4), (2, 1), (2, 4)\}$$



**Nota** Il fatto più importante è che un punto qualsiasi del gruppo ne è generatore e possiamo trovare

tutti gli altri punti sommando  $P$  con se stesso, ovvero, calcolando  $kP = P + P + \dots$

$$P = (0, 1) \longrightarrow 2P = (0, 1) + (0, 1)$$

$$\lambda = \frac{3x_1^2 + 1}{2y_1} = 1 \cdot 2^{-1} \bmod 5 = 3$$

$$x_3 = \lambda^2 - x_1 - x_1 = 3^2 = 9 \bmod 5 = 4 \longrightarrow y_3 = \lambda(x_1 - x_3) - y_1 = 3(-4) - 1 = -13 \bmod 5 = 2$$

$$2P = (4, 2)$$

potremmo andare avanti fino a raggiungere un ciclo.



Possiamo concludere quindi che

### Teorema 16.3 ( $E(\mathbb{Z}(p), +)$ )

Il gruppo  $(E(\mathbb{Z}(p), +)$  è:

- Chiuso rispetto alla "somma" (def 16.2).
- L'"addizione" è commutativa.
- O è l'identità rispetto alla "somma".
- Ogni punto  $P \in E$  ha un inversa  $-P$ .
- La proprietà associativa è valida.



## 16.3 EC Crypto

Affrontiamo adesso gli schemi di crittografia basati su curve ellittiche. Il livello di sicurezza offerto da una curva si basa su quanto è difficile risolvere il problema di Dlog.

### Teorema 16.4 (Dlog problem for EC)

Dato  $P$  e  $k$  è **facile** calcolare  $kP^1$ . Dato  $P, kP$  è **difficile** trovare  $k$ .

Questo perché  $P$  è il punto della curva nel gruppo, mentre  $k$  è un numero intero che dipende dall'ordine del gruppo.

<sup>1</sup>Se avessimo usato una notazione moltiplicativa invece della somma, avremmo avuto esattamente il Dlog problem, perché avremmo calcolato  $P^k$ .



### Corollario 16.1 (Elliptic Curve Discrete Log Problem)

La difficoltà nel risolvere il problema è **strettamente correlato** al tipo di curva usato, per questo devono sempre essere usate curve definite dallo standard per gli algoritmi di cifratura.



Vediamo adesso un algoritmo simile allo **Square and Multiply** (algoritmo 1) per calcolare  $kP$ . Poiché non esiste una versione che permette di *invertire* il calcolo in maniera efficiente, il problema è considerato difficile.



**Nota** Ricordiamo inoltre che un vantaggio delle curve ellittiche è la possibilità di **ridurre** la dimensione della chiave, a parità del livello di sicurezza.

**Algorithm 2** Double and Sum

---

```

1: procedure MULT_BY_DUBLEUP( $k, P$ )                                ▷ Fast compute  $kP$ 
2:    $exp \leftarrow x_2[1 :]$                                          ▷ Take exponential in base 2 and cut the msb
3:    $r \leftarrow P$ 
4:   for all  $b$  in  $exp$  do                                         ▷ Loop from 2nd lsb to msb of  $exp$ 
5:      $r \leftarrow r \cdot 2$                                          ▷ For every bit, double  $g$ 
6:     if  $b[i] == 1$  then                                         ▷ If current bit is 1, add  $g$ 
7:        $r \leftarrow r + P$ 
8:     end if
9:   end for
10:  end procedure

```

---

Vediamo gli algoritmi che vennero **standardizzati** dalle entità Americane nella **Suite B**.

**Osservazione:** Non vediamo gli algoritmi di cifratura, già ampiamente trattati nel corso, tuttavia per pura nozione ricordiamo che vengono usati **AES 128/256** mentre per gli algoritmi di hashing **SHA-256** e **SHA-384**. ■

Per quanto riguarda lo scambio di chiavi, venne standardizzata una versione di DH per funzionare su curve ellittiche.

**Definizione 16.3 (Elliptic Curve Diffie Helman)**

Consideriamo un client  $C$  e un server  $S$  che vogliono stabilire una chiave per la cifratura della loro comunicazione.

1. **C sceglie** dalla sua suite, il **numero primo**  $p$  di  $\mathcal{Z}(p)$ , i **parametri della curva**  $a, b$  (eq. (16.1)) e l'ordine del gruppo  $q$ .
2. **C genera** un numero **casuale**  $x$  e calcola  $x \times G \bmod p$ , dove  $G$  è un **punto generatore della curva scelta**.
3. **C invia a S**  $\{G, x \times G \bmod p\}$ .
4. **S genera** un numero **casuale**  $y$ , e calcola  $y \times G \bmod p$  e il **segreto**  $s = yx \times G \bmod p$ .
5. **S invia a C**:  $y \times G \bmod p$ .
6. **C calcola il segreto**:  $s = xy \times G \bmod p$ .

Vengono usate curve su numeri primi a 256 e 384 bit. **Osservazione:** I prodotti sono intesi come somma ripetuta, seguendo la definizione di somma su curve ellittiche (def 16.2). ■

**Osservazione:** Le due entità si scambiano **punti della curva ellittica**, senza mai rivelarla. ■

 **Nota** ECDH è un algoritmo di Key-Agreement usato nelle moderne versioni di TLS.

per quanto riguarda l'algoritmo di firma digitale, vediamo come funziona un processo di firma basato su Dlog.

### Definizione 16.4 (Digital Signature Algorithm)

Come firmatario di  $m$ :

- Seleziono un numero  $p > 2$  primo forte ( $p = 2q + 1$ ,  $q$  molto grande).
- Seleziono un generatore  $g$  tale che  $g^x \bmod p$  genera tutti gli elementi del gruppo  $\mathbb{Z}^*(p)$ .
- Seleziono la chiave privata  $d$  e calcolo la public key  $e = g^d \bmod p$ .
- Seleziono un nonce  $k \in (1, q)$ .

Per firmare eseguo:

- Calcolo  $r = (g^k \bmod p) \bmod q$ .
- Calcolo<sup>2</sup>  $s = k^{-1}(H(m) + d \cdot r) \bmod q$ .
- La firma è:  $(r, s)$ .

Per verificare, assumendo di conoscere  $e$ , eseguo:

- Calcolo  $u_1 = s^{-1}H(m) \bmod q$ .
- Calcolo  $u_2 = s^{-1}r \bmod q$ .
- Verifico che:  $[(g^{u_1}e^{u_2}) \bmod p] \bmod q = r$ , dove  $e = g^d$ . In questo modo stiamo calcolando:

$$g^{\frac{k}{H(m)+rd}H(m)} \cdot g^{d\frac{k}{H(m)+rd}r} = g^{\frac{k(H(m)+rd)}{H(m)+rd}} = g^k \bmod p$$

<sup>2</sup>Dipende dalla nonce!



#### Assumptions:

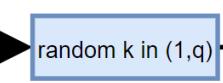
Select  $\mathbb{Z}^*(p)$  and  $g$  st:

$g^x \bmod p$  generates the group

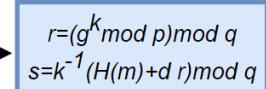
Select  $d$  st:  $e = g^d \bmod p$



Select



Compute

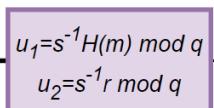


Send  $(r, s)$

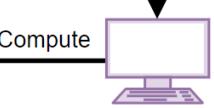


#### Assumptions:

Verifier



Verify



Compute

**Figura 16.3:** Digital Signature Algorithm

Il processo di firma sembra robusto poiché avendo legato firma alla nonce, un attaccante non potrà replicarla. Cosa succede però se la nonce dovesse essere predicibile?

**Corollario 16.2 (Predictable Nonce in DSA)**

*Supponiamo di poter conoscere il prossimo della nonce  $k$  usato per un messaggio valido  $(r,s)^3$ .*

*Allora:*

$$\begin{aligned}s &= k^{-1}[H(m) + rd] \longrightarrow sk = H(m) + rd \\ d &= [sk - H(m)]r^{-1}\end{aligned}$$

*La chiave privata è ora nota all'attaccante.*

<sup>3</sup>r ed s sono note, per questo posso risolvere l'equazione. □

Se la nonce dovesse invece ripetersi?

**Corollario 16.3 (RepeatedNonce in DSA)**

*Supponiamo di avere due messaggi validi  $(r, s_1), (r, s_2)$  dove viene usata la stessa nonce  $k$ .<sup>4</sup>*

*Allora è possibile risolvere il seguente sistema:*

$$\begin{aligned}\begin{cases} s_1 &= k^{-1}[H(m_1) + rd] \\ s_2 &= k^{-1}[H(m_2) + rd] \end{cases} &\longrightarrow \begin{cases} k &= s_1^{-1}[H(m_1) + rd] \\ k &= s_2^{-1}[H(m_2) + rd] \end{cases} \\ s_1^{-1}[H(m_1) + rd] &= s_2^{-1}[H(m_2) + rd] \\ s_2H(m_1) + s_2rd &= s_1H(m_2) - s_2H(m_1) \\ d(s_2 - s_1)r &= s_1H(m_2) - s_2H(m_1) \\ d &= [s_1H(m_2) - s_2H(m_1)](s_2 - s_1)^{-1}r^{-1}\end{aligned}$$

<sup>4</sup> $r = (g^k \bmod p) \bmod q$ . Per questo i due messaggi hanno stessa r □

**Osservazione:** Risulta evidente che la nonce  $k$  **deve** essere selezionata in modo molto accurato e deve rispettare la definizione di essere **unica**, altrimenti il sistema viene completamente rotto. ■

Esponiamo ora la versione di DSA sulle curve ellittiche:

### Definizione 16.5 (Elliptic Curve Digital Signature Algorithm)

**Come firmatario:**

- Scelgo  $E(\mathbb{Z}_p)$ , con  $p$  primo,  $n$  (primo, grande) l'ordine del gruppo,  $P$  il generatore del gruppo.
- Calcolo  $Q = [d]P$ , con  $d$  chiave privata presa come numero intero  $[1, n - 1]$  e  $Q$  la chiave pubblica.

**Osservazione:** La chiave privata è un intero, la pubblica è un punto della curva. ■

**Per firmare eseguo:**

- Scelgo  $k \in [1, n - 1]$  intero casuale, **unico e NON predicibile**.
- Calcolo il punto  $kP = (x_1, y_1)$ . Le sue coordinate sono **due interi modulo  $p$** .
- Per evitare che le coordinate siano **fuori dall'ordine del gruppo**<sup>5</sup> calcolo:

$$r = x_1 \bmod n$$

- Calcolo la firma come:  $s = k^{-1}(H(m) + rd) \bmod n$ .
- Invio  $\{m, r, s\}$

**Per verificare eseguo:** (assumendo tutti i calcoli in modn)

- Dal messaggio  $m$  calcolo l'hash  $H(m)$ .
- Da  $s$  calcolo:  $w = s^{-1} \bmod n = \frac{k}{H(m)+rd}$ .
- Calcolo (**interi**):  $u_1 = w \cdot H(m)$  e  $u_2 = r \cdot w$ .
- Calcolo il punto della curva:  $u_1 P + u_2 Q = (u_1 + u_2 d)P = (x_0, y_0)$ . Ovvero:

$$\begin{aligned} u_1 P + u_2 Q &= \frac{H(m)k}{H(m) + rd} + \frac{rk}{H(m) + rd} Q \\ &= \frac{H(m) \cdot k}{H(m) + rd} + \frac{rd \cdot k}{H(m) + rd} P = kP \end{aligned}$$

- Verifco che  $v = x_0 \bmod n = r$ .

**Osservazione:** Lo schema è basato su Dlog e non su factoring. ■

Vengono usati numeri primi a 256 e 384 bit per i moduli.

<sup>5</sup>L'ordine del gruppo può essere più piccolo, pari o maggiore di  $p$ , per questo è necessario. Tuttavia, nel 99% dei casi non si applica mai poiché  $n$  non è quasi mai maggiore di  $p$ . □

**Osservazione:** Il motivo per il quale spesso viene riutilizzata la nonce è perché le operazioni su EC costano a livello computazionale. Sarebbe possibile quindi ottimizzare il calcolo del processo di firma calcolando  $k$  una volta per tutte, ma esponendo lo schema ai problemi citati sopra. ■



**Nota** Per trasferire file viene usato un algoritmo chiamato ECIES, che non vediamo in questa sede. E' usato ad esempio in 5G.



**Nota** Al giorno d'oggi, quando bisogna fare trasmissione criptata vengono usati degli HSM (Hardware Security Modules) che implementano a livello hardware algoritmi di cifratura.

# Capitolo Pairing-Based Cryptography

Questo tipo di approccio crittografico utilizza lo strumento matematico delle **mappe bilineari** per trasformare punti di due gruppi crittografici in punti di un altro gruppo tramite una specifica funzione. È un approccio potente, perché permette ad esempio di ridurre un problema difficile in un gruppo ad un problema più facile da risolvere su un altro gruppo.

Questo campo di ricerca venne lanciato da Boneh e Franklin nel 2001, nel tentativo di creare uno schema di verifica dell'identità **senza** fare uso dei certificati.

## 17.1 Bilinear Maps

Diamo una definizione di una Mappa Bilineare:

### Definizione 17.1 (Bilinear Map)

Siano  $G_1, G_2, G_t$  gruppi **ciclici** dello stesso **ordine**. Allora, una mappa bilineare da  $G_1 \times G_2$  è una funzione e tale che:

$$e : G_1 \times G_2 \longrightarrow G_t : \forall u \in G_1 v \in G_2, a, b \in \mathbb{Z} \text{ s.t.:} \quad (17.1)$$
$$e(u^a, v^b) = e(u, v)^{ab}$$

**Osservazione:** Le mappe bilineari sono chiamate **pairings** (accoppiamenti) poiché associano coppie di elementi da  $G_1$  e  $G_2$  con elementi di  $G_t$ .



**Nota** La def 17.1 ammette anche mappe **degeneri**, ovvero che mappano tutto nell'identità di  $G_t$ .

Dobbiamo pertanto dare una seconda definizione, che ci dica se una mappa è utilizzabile per i nostri scopi:

### Definizione 17.2 (Admissible Bilinear Map)

Sia  $e : G_1 \times G_2 \longrightarrow G_t$  una mappa bilineare e siano  $g_1, g_2$  i **generatori** di  $G_1, G_2$  rispettivamente.

La mappa  $e$  è **una mappa bilineare ammissibile** se  $e(g_1, g_2)$  genera  $G_t$  ed è **calcolabile in modo efficiente**.



**Nota** A volte questo tipo di mappe vengono denominate con  $\hat{e}$ , ma noi continueremo ad usare  $e$ .

**Osservazione:** Da questo momento in poi, ogni volta che useremo una mappa si intenderà che è una mappa ammissibile.

Tipicamente, negli schemi crittografici i gruppi  $G_1, G_2$  vengono presi uguali, per questo da ora in poi faremo questa assunzione. Inoltre, l'ordine è solitamente un numero primo (la differenza sta nel modo in cui i gruppi funzionano e vengono usati).

**Osservazione:** Idealmente vorremmo anche cercare una mappa che trasformi  $G \times G \longrightarrow G$ , ma non ci sono studi in merito e non consideriamo questo caso.



**Nota** Le mappe bilineari tipicamente usate sono quelle di **Weil** e **Tate**, estremamente complicate ma non necessitano di essere capite per essere usate.

Il concetto fondamentale sulle mappe bilineari è come queste influenzano il **Decisional Diffie-Hellman Problem**:

### Definizione 17.3 (Decisional Diffie-Hellman Problem)

*Dati  $g, g^a, g^b$  e un valore  $\alpha$  che con probabilità  $1/2$  assume il valore  $g^{ab}$  oppure  $g^z$ , dove  $z$  è un numero intero casuale. Indovinare qual è il valore di  $\alpha$ .*

*Diremo che un DDH-problem è **hard**, se senza alcuno strumento la probabilità di indovinare è negligible.* □



**Nota** Se un attaccante fosse in grado di calcolare  $g^{ab}$ , ovvero è in grado di **rompere computational DH** (CDH), allora anche DDH viene rotto. Il viceversa potrebbe non essere vero.

**Esempio 17.1 How pairing breaks DDH** Supponiamo che un attaccante disponga di  $g, g^a, g^b$ , che corrispondono a tre punti di una curva ellittica. Avendo a disposizione un pairing del tipo  $e(g^a, g^b) = e(g, g)^{ab} = g_t^{ab}$ <sup>1</sup> che mappa due punti di una curva su un intero modulare, allora DDH è rotto, perché basta controllare che un valore  $\alpha = \{g^{ab}, g^z\}$  abbia lo stesso pairing, ovvero:

$$e(g, \alpha) = \begin{cases} e(g, g^{ab}) = g_t^{1 \cdot ab} & p = \frac{1}{2} \\ e(g, g^z) = g_t^z & p = \frac{1}{2} \end{cases}$$



**Nota** E' importante osservare che **non è stato calcolato** il valore di  $g^{ab}$ , ma è stato calcolato  $g_t^{ab}$  perché il primo è un punto di una EC, il secondo di  $Z_p^*$ . ■

**Osservazione:** L'esempio ci mostra come non è possibile rompere DDH nuovamente, poiché una volta fatto il primo pairing non siamo più in EC e non possiamo tornare indietro. ■

### 17.1.1 MOV Reduction

Ricordiamo il vantaggio di usare le curve ellittiche:

**Proprietà** Se stiamo usando come gruppo  $Z_p^*$  per gli algoritmi di cifratura, DH o per risolvere Dlog, allora le **chiavi** hanno **dimensione** di 2048bit. Se usiamo **curve ellittiche**, possiamo **ridurre la dimensione a 256bit**.

Supponiamo adesso di voler risolvere un Dlog-problem in un gruppo basato su EC. Un dealer ci consegna il generatore  $g$  e il risultato di  $g^x$ . Supponiamo che su questo gruppo  $G$  il problema sia difficile perché la chiave usata è a 256bit.

Supponendo però di avere a disposizione  $e : G \times G \longrightarrow G_t = Z_p^*$  con key-size di 256bit, possiamo calcolare:

$$e(g, g^x) = e(g, g)^x = g_t^x$$

E portare il problema su un gruppo più debole.

Questo problema è chiamato la **MOV Reduction** ed è **lo strumento principale** per rompere l'EC-crypto.

<sup>1</sup>E' stata usata la notazione moltiplicativa per il calcolo, dove  $\alpha * P = P^\alpha$ .

**Problema 17.13-party DH** Consideriamo uno scenario di key-agreement nel quale tre entità vogliono stabilire una chiave comune, dove:  $P_1, P_2, P_3$  hanno a disposizione le coppie  $(x, g^x), (y, g^y), (z, g^z)$  (supponendo che il generatore  $g$  sia stato precedentemente accordato).

 **Nota** Con schemi crittografici standard non possiamo risolvere il problema.

Nel 2000, un crittografo francese chiamato Joux, trovò una soluzione applicando tecniche di pairing:

### Soluzione

#### Teorema 17.1 (Joux's 3-party DH)

Siano  $X, Y, Z$  tre entità che effettuano un key-agreement, rispettivamente con  $(x, g^x), (y, g^y), (z, g^z)$ . Allora:

1. Ogni parte scambia la sua chiave pubblica con le altre due.
2. Ogni parte calcola  $(g^i)^j = g^{ij}$ ,  $i$  è la chiave della parte  $i$  – esima,  $j$  una delle altre due.
3. Successivamente tremite pairing:  $e(g^{ij}, g^k) = e(g, g)^{ijk} = g_t^{ijk}$ ,  $k$  è la chiave restante.

**Osservazione:** Poiché, tipicamente,  $g \in EC(\cdot)$ ,  $g_t \in Z_p^*$ , avremmo potuto anche calcolare prima il pairing e poi la potenza, guadagnando a livello di tempo computazionale:

1.  $e(g^x, g^y) = g_t^{xy}$ .
2.  $w = (g_t^{xy})^z$ .



**Nota** E' fondamentale notare che avendo eseguito un pairing, il gruppo d'arrivo non è quello di partenza in cui vengono calcolate le chiavi pubbliche, quindi abbiamo potuto "barare" una e una sola volta con DH. Questo significa che non è possibile eseguire uno schema a 4-parti.

## 17.2 Identity Based Encryption

**Problema 17.2** Vogliamo costruire un sistema crittografico a public-key dove la  $pk$  è un **nome** (una stringa leggibile).



**Nota** Proprio come con gli indirizzi IP, vogliamo cercare un metodo di associare ad un numero un nome che sia facilmente utilizzabile per una persona fisica.

Questa branca della crittografia viene definita Identity-Based Encryption (IBE). Nel 2001, Boneh propose una prima soluzione partendo dai risultati di Joux.

Il ragionamento fatto da Boneh e che è alla base dello schema, consiste nel fatto che è possibile istituire un 3-way DH, dove il PKG ha  $s, g^s$  e si occupa solo di distribuire la chiave a B, mentre A ( $r, g^r$ ) e B ( $t, g^t$ ) calcolano, rispettivamente  $e(g^s, g^t)^r = \hat{g}^{rst}$ ,  $e(g^s, g^r)^t = \hat{g}^{rst}$ .

Di base lo schema non potrebbe funzionare perché in realtà B **non ha** inizialmente la **chiave privata**  $t$ , quindi **non può svolgere il suo calcolo**. Tuttavia, il PKG può fornire a B direttamente il valore  $g^{st} = (g^t)^s = ("bob")^s$  piuttosto che  $\hat{g}^{st}$  e permettere di "barare" calcolando il pairing tra due punti di EC, come se  $t$  fosse il discrete-log di  $h_1("bob")$  base  $g$ .

#### Definizione 17.4 (Boneh & Franklin's IBE Scheme)

Sia  $G$  un gruppo di ordine primo  $q$ , sia  $e : G \times G \rightarrow G_t$  una mappa bilineare e  $g$  un generatore per  $G$ . Sia  $\hat{g} = e(g, g) \in G_t$  e siano  $h_1 : \{0, 1\}^* \rightarrow G$ ,  $h_2 : G_t \rightarrow \{0, 1\}^*$  delle funzioni hash che, rispettivamente, mappano una stringa in un numero su curva ellittica e viceversa. Tutti questi parametri sono pubblici.

Consideriamo tre attori, Alice, Bob e un **Private Key Generator**, e associamo ad ognuno di loro una chiave privata (random) e pubblica, estratta dallo stesso gruppo  $G$ .

- $A$ :  $r, g^r$ .
- $PKG$ :  $s, g^s$
- $B$ : "bob" (chiave pubblica).

Supponiamo che  $A$  voglia inviare un messaggio  $m$  a  $B$ . Allora  $A$  deve calcolare:

$$\begin{aligned} Enc(g, g^s, "bob", m) &= (g^r, m \oplus h_2(e(h_1("bob"), g^s)^r)) \\ &= (g^r, m \oplus h_2(e(h_1("bob"), g)^{sr})) \end{aligned} \quad (17.2)$$

e inviare:

$$(u, v) = (g^r, m \oplus h_2(e(h_1("bob"), g)^{sr}))$$

Adesso il PKG può calcolare la private-key di  $B$  come:

$$w = MakeKey(s, "bob") = h_1("bob")^s \quad (17.3)$$

Quindi  $B$  può decifrare nel seguente modo:

$$\begin{aligned} Dec(u, v, w) &= v \oplus h_2(e(w, u)) \\ &= m \oplus h_2(e(h_1("bob"), g)^{sr}) \oplus h_2(\underbrace{e(h_1("bob")^s, g^r)}_{w}, \underbrace{u}) \\ &= m \oplus h_2(e(h_1("bob"), g)^{sr}) \oplus h_2(e(h_1("bob"), g)^{sr}) = m \end{aligned} \quad (17.4)$$



**Osservazione:** Per la definizione data di  $h_1$ ,  $B$  distribuisce il suo come come un punto di una curva ellittica, così come  $g^s, g^r$ . Questo ci permette di costruire un **ElGamal-like 3-party DH**, perché possiamo sfruttare il pairing per fare encryption e decryption. ■

Per realizzare lo schema in modo effettivo però bisogna concentrarsi su tre aspetti fondamentali:

1. **Chi è il PKG?** Se il PKG fosse un'entità potrebbero esserci problemi di sicurezza legati all'affidabilità di essa, come con una qualsiasi CA, perché **può decifrare qualsiasi messaggio scambiato con questo schema** (rendendosi, **di fatto, più potente** di una CA), rendendo inutile la Public-Key-Infrastructure (PKI).
2. **Come costruirne uno?**
3. Usare come **chiave pubblica** una **stringa** (pochi caratteri, soliti problemi ecc), ha **davvero un vantaggio** rispetto al prezzo pagato?

Ovviamente, i primi due problemi possono essere risolti creando un **sistema distribuito** basato su secret-sharing, in modo tale che nessuna entità nella rete possa effettivamente decifrare tutto il traffico e mantenere i vantaggi di una IBE. Vediamone un esempio:

### Definizione 17.5 (Distributed PKG)

Consideriamo un sistema distribuito composta da  $n$  entità impegnate a realizzare un  $(t, n)$  distributed secret sharing scheme. Consideriamo il gruppo  $\mathbb{Z}_q$  di ordine primo  $q$ , con  $q$  numero primo grande e  $g, h$  due generatori per il gruppo. Sia  $H_1 : \{0, 1\}^* \rightarrow G$  una funzione hash per mappare stringhe in interi.

Sia  $P_i$  l' $i$ -esimo nodo PKG, allora:

- **Setup:**  $\forall i = 1, \dots, n$

1.  $P_i$  seleziona due polinomi random in  $\mathbb{Z}_q$

$$f_i(x) = \sum_{k=0}^{t-1} a_{ik} \bmod q \quad f'_i(x) = \sum_{k=0}^{t-1} b_{ik} \bmod q$$

2.  $P_i$  seleziona la sua chiave privata come  $s_i = f_i(0) = a_{i0}$  e chiave pubblica  $g^{s_i}$
3.  $P_i$  calcola ed invia su canale sicuro e autenticato alle relative  $P_j$ ,  $j = 1, \dots, n$  gli share:

$$y_{ij} = f_i(x_j) \bmod q \quad y'_{ij} = f'_i(x_j) \bmod q$$

4.  $P_i$  calcola ed invia in broadcast su canale sicuro e autenticato un pedersen commit

$$c_{ik} = g^{a_{ik}} \cdot h^{b_{ik}} \bmod q, \quad k = 0, \dots, t-1$$

- **Verifying PKGs:** Il nodo  $P_j$  ha ricevuto gli share  $\{x_j, y_{ij}, y'_{ij}\}$  e controlla l'affidabilità del nodo  $P_i$  calcolando:

$$g^{y_{ij}} h^{y'_{ij}} \stackrel{?}{=} \prod_{k=0}^{t-1} c_{ik}^{x_j^k} = g^{\sum_{k=0}^{t-1} a_{ik} x_j^k} h^{\sum_{k=0}^{t-1} b_{ik} x_j^k} = g^{f_i(x_j)} h^{f'_i(x_j)}$$

**Osservazione:** Se il controllo ha successo il nodo  $P_i$  è considerato **trusted** da  $P_j$ . Altrimenti, se per  $t+1$  nodi  $P_j$  il controllo fallisce,  $P_i$  è **untrusted** ed è escluso dal protocollo. ■

- **Private Key Extraction:** sia  $C$  un client che utilizza la sua identità "ID" come chiave pubblica e vuole ottenere la sua chiave privata dal Sistema DPKG

1.  $C$  contatta  $t$  delle  $n$  entità nel sistema DPKG per effettuare la richiesta.
2. I  $t$  nodi che rispondono alla richiesta autenticano  $C$  e gli inviano su canale sicuro e autenticato uno share del loro segreto insieme alla chiave pubblica:

$$\sigma_i = H_1(ID)^{y_i}, \quad y_i = f_i(x_i), \quad \forall i = 1, \dots, t$$

3.  $C$  può ricostruire il master-secret (senza esporlo) complessivo per calcolare la sua chiave privata  $\Sigma$  e la chiave pubblica complessiva necessaria per le operazioni di IBE  $g^s$ :

$$\begin{aligned} \Sigma &= \prod_{i=1}^{t-1} \sigma_i^{\Lambda_i} = H_1(ID)^{\sum_{i=1}^{t-1} y_i \Lambda_i} = H_1(ID)^s; \quad g^s = \prod_{i=1}^{t-1} g^{s_i} \\ \Lambda_i &= \prod_{k \neq i} \frac{-x_k}{x_i - x_k} \quad \forall i = 1, \dots, t \end{aligned}$$

4.  $C$  può verificare la validità del risultato ottenuto controllando che:

$$e(\Sigma, g) = e(H_1(ID)^s, g) \stackrel{?}{=} e(H_1(ID), g^s) = e(H_1(ID), g)^s$$

□

 **Nota** IBE, così come è stato concepito, in realtà non è così fondamentale. La sua generalizzazione, l'Attribute-Based-Encryption, è uno schema molto potente che permette di creare sistemi nei quali invece di distribuire chiavi, gli utenti possono accedere a contenuti cifrati per il semplice fatto di appartenere ad una particolare categoria, ovvero, perché posseggono **gli attributi necessari** per accedervi.