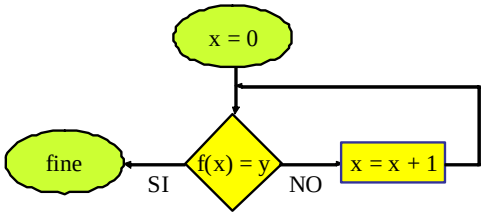
Cap. 2: ampliamenti e modifiche

Livelli di sicurezza (pag. 21)

Nella valutazione della sicurezza si può però anche fare a meno degli anni MIPS, introducendo il concetto di **livello di sicurezza**. Questa diversa unità di misura della robustezza assume come riferimento il tempo d'esecuzione dell'algoritmo di **ricerca esauriente**.

L'algoritmo di ricerca esauriente

PROBLEMA – **Per quale intero x, con $0 < x < 2^n-1$, vale la proprietà f(x) = y?**



Nel caso peggiore occorrono 2^n passi: O(exp(n))

La prima cosa da capire bene è il funzionamento di tale algoritmo, cosa che possiamo fare agevolmente ponendoci il seguente problema:
"*individuare nell'intervallo* ***0 ÷ 2^n-1*** *un numero* ***x*** *tale che* ***f(x) = C*** *senza avere a disposizione un algoritmo efficiente in grado di calcolare* ***f^{-1}***".

Per trovare **x** è sufficiente generare uno dopo l'altro, a partire dal più piccolo, i numeri binari di n bit controllando ogni volta se per il valore corrente si ha **f(x) = C**: in caso affermativo l'algoritmo termina, in caso negativo si prosegue nella generazione incrementando di un'unità l'ultimo numero che non ha superato il test.

Nel caso peggiore l'algoritmo termina dopo 2^n passi ed ha quindi complessità **O(exp(n))**.

La seconda cosa da capire è che l'algoritmo di ricerca esauriente è in grado di risolvere tutti i problemi della Crittanalisi (in questo contesto è detto **attacco con forza bruta**): una trasformazione segreta appartiene sempre ad un insieme finito di trasformazioni possibili e chi le prova tutte dispone sempre di un test per verificare se ha trovato quella buona.

La terza ed ultima cosa da osservare è che in moltissimi casi d'attacco esistono algoritmi più efficienti. Consideriamo dunque "il migliore" di questi algoritmi per un certo attacco e supponiamo di aver verificato che può essere condotto con successo eseguendo al più **N passi**.

Per misurarne la pericolosità è sufficiente individuare un **p** tale che

$$\mathbf{2^p \leq N < 2^p.}$$

ed affermare che il sistema attaccato ha **un livello di sicurezza di p bit**, cioè che è attaccabile solo con uno sforzo computazionale pari a quello richiesto dalla ricerca esauriente di un numero di p bit.

Il vantaggio di impiegare il livello di sicurezza al posto degli anni MIPS è quello di non fissare la velocità del calcolatore che esegue l'attacco e quindi di prescindere dall'evoluzione tecnologia. Fare riferimento al numero di passi piuttosto che al numero di operazioni costituisce inoltre un'utile semplificazione della valutazione di robustezza. Naturalmente è possibile passare dalla misura in anni MIPS a quella in livello di sicurezza e viceversa

ESEMPIO - $\mathbf{10^{12}}$ anni MIPS corrispondono ad un livello di sicurezza di 85 bit.

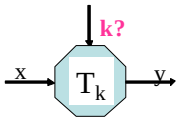
E' opinione largamente condivisa che i meccanismi progettati ed impiegati attualmente debbano avere un livello di sicurezza il più possibile vicino a **128 bit**.

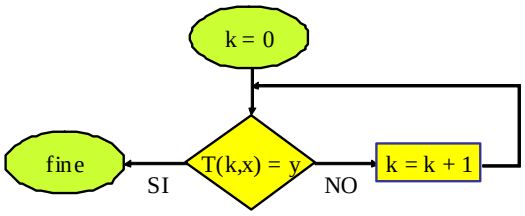
Eliminare R5bis e R7bis (pag. 23-24) e sostituire con:

Attacco con forza bruta alla segretezza di una chiave

IPOTESI:

1. **$0 \leq k \leq 2^n-1$, $y = T(k,x)$**
2. **E' nota almeno una coppia x, y**





Nel caso peggiore occorrono **2^n** passi: **O(exp(n))**

n = 128

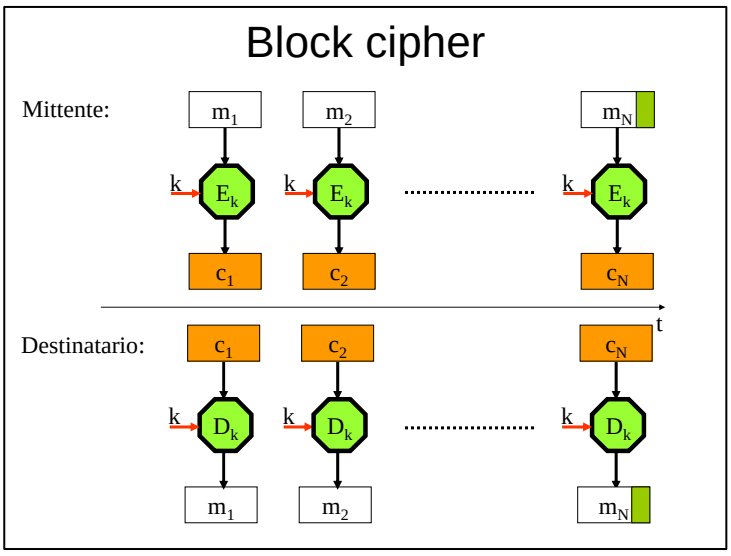
Lo spazio delle chiavi di un algoritmo crittografico deve essere molto grande. Per avere in testa una prima stima di questa dimensione basta pensare che l'**attacco con forza bruta** è la più semplice idea che può venire in testa ad un intruso in possesso di una coppia di testo in chiaro e di testo cifrato.

In figura è mostrato che occorrono al più 2^n passi per individuare una chiave di n bit. Per quanto abbiamo osservato nel paragrafo 2.1.5 oggi deve essere

$$80 < n \leq 128.$$

Più avanti vedremo che per certi algoritmi l'intruso dispone di ulteriori informazioni sulla chiave e può sfruttarle per condurre attacchi più efficienti (di complessità sub-esponenziale). In questi casi un livello di sicurezza di 128 bit richiede chiavi di più di 1000 bit.

Cap. 4: ampliamenti e modifiche



4.3 Cifrari a blocchi (pag. 65)

Il testo da cifrare viene suddiviso in blocchi $m_1, m_2, .., m_N$ di lunghezza prefissata L.

Se la lunghezza del testo in chiaro non è un multiplo intero della lunghezza del blocco, il mittente aggiunge all'ultimo blocco simboli di riempimento privi di significato (*padding*).

La regola di trasformazione dei blocchi, fissata dalla chiave **k**, è una sostituzione mono-alfabetica, ma la grossa dimensione dei blocchi (nel caso di un testo letterario codificato in ASCII un blocco include la codifica di almeno otto caratteri) la rende immune da un attacco con statistiche.

Anche la decifrazione procede a blocchi ed è fissata dalla chiave **k**. In conclusione si ha:

CIFRATURA: $c_i = E(m_i, k)$, per i =1,2, ..
DECIFRAZIONE: $D(c_i, k) = m_i$, per i =1,2, ..

Sono in uso diversi standard per consentire alla macchina del destinatario di eliminare automaticamente i bit di padding aggiunti dalla macchina del mittente. PKCS#7, ad esempio, prevede che il completamento dell'ultimo blocco sia fatto da **n byte di valore n**, se **8 × n** è il numero di bit che mancano nel testo in chiaro.

Ampliamento e modifica del testo da pag. 78 a pag.82

4.6.3 Esponenziazione in GF(p): $y = b^e \bmod p = (..(((b)b)b)..b) \bmod p$

La tabella riporta i risultati di $\mathbf{b^e \bmod m}$ nel caso semplice di $\mathbf{p = 11}$ e di $\mathbf{0 < e, b < p}$.

	e									
b	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
1	1	1	1	1	1	1	1	1	1	1
2	2	4	8	5	10	9	7	3	6	1
3	3	9	5	4	1	3	9	5	4	1
4	4	5	9	3	1	4	5	9	3	1
5	5	3	4	9	1	5	3	4	9	1
6	6	3	7	9	10	5	8	4	2	1
7	7	5	2	3	10	4	6	9	8	1
8	8	9	6	4	10	3	2	5	7	1
9	9	4	3	5	1	9	4	3	5	1
10	10	1	10	1	10	1	10	1	10	1

$b^e \bmod 11$

Proprietà notevoli dell'esponenziazione modulo un primo p

- **T4** (piccolo teorema di Fermat): "per ogni primo **p** e per ogni $\mathbf{x} \in \mathbf{Z^*_p}$ si ha $\mathbf{x^{p-1} \bmod p = 1}$, cioè $\mathbf{x^{p-1} \equiv 1 \pmod p}$ o anche".

ESEMPIO – Si noti la presenza del solo numero **1** nella colonna e = 10 (N.B. **p-1**) della precedente tabella.

Corollari

T4.1: Per ogni $\mathbf{x} \in \mathbf{Z^*_p}$, se $\mathbf{s \equiv r \pmod{p-1}}$ allora si ha anche $\mathbf{x^s \equiv x^r \pmod p}$.

COMMENTI - Per ogni **e>p-1**, si ha dunque $\mathbf{x^e \equiv x^{e \bmod p-1} \pmod p}$. All'aumentare di **e**, la tabella si ripete ed è quindi inutilmente pesante calcolare l'esponenziazione modulo p per esponenti maggiori di **p-1**. Si noti che è anche inutilmente oneroso calcolare l'esponenziazione modulo p per basi maggiori di p-1: $\mathbf{b^x \bmod p = (b \bmod p)^x \bmod p}$

T4.2: $\mathbf{x^{(p-1)/2} \bmod p}$ fornisce o **+1**, o **–1**, cioè le due radici quadrate di 1; si noti in tabella la colonna e = 5.

In GF(p) è possibile eseguire divisioni: esiste infatti l'inverso moltiplicativo di ogni elemento $\mathbf{x} \in \mathbf{Z^*_p}$, quale risulta definito dalla congruenza $\mathbf{x^{-1}\, x \equiv 1 \pmod p}$

- **T5**: "l'inverso moltiplicativo di ogni $\mathbf{x} \in \mathbf{Z^*_p}$ è valutabile con l'esponenziazione $\mathbf{x^{-1} = x^{p-2} \bmod p}$".

ESEMPIO – Moltiplicando mod 11 ogni elemento nella colonna 9 (N.B. **p-2**) per quello che denomina la riga in cui si trova, si ottiene sempre 1. Più avanti vedremo che è però più efficiente impiegare l'**algoritmo esteso di Euclide**.

Non tutte le righe sono **permutazioni** di $\mathbf{Z^*_{11}}$; la proprietà vale solo per b = 2, 6, 7, 8. In generale si ha:

- **T6**: "per ogni primo **p**, esiste almeno un **g < p,** detto **generatore mod p**, o **radice primitiva** di p, le cui potenze $\mathbf{g^1 \bmod p}$, $\mathbf{g^2 \bmod p}$, … $\mathbf{g^{p-1} \bmod p}$ forniscono tutti gli interi compresi tra **1** e **p-1** . Le radici primitive di **p** sono esattamente $\mathbf{\Phi(p-1)}$".

ESEMPIO – Trovare i generatori di un primo è un problema **difficile**. Facile invece è controllare se un elemento è un generatore. 2 è una radice primitiva di 11 (e di moltissimi altri numeri primi): $2^1 \bmod 11=2$, $2^2 \bmod 11=4$, $2^3 \bmod 11=8$, $2^4 \bmod 11=5$, $2^5 \bmod 11=10$, $2^6 \bmod 11=9$, $2^7 \bmod 11=7$, $2^8 \bmod 11=3$, $2^9 \bmod 11=6$, $2^{10} \bmod 11=1$.

- **T7**: "un $\alpha \in$ **GF(p)** è una radice primitiva se e solo se, per ogni **q** divisore primo di $\Phi(p)$, si ha $\alpha^{\Phi(p)/q} \neq 1$".

ESEMPIO - I generatori di **p = 11** sono quattro: $\mathbf{\Phi(10) = 10 \times (1\text{-}1/2) \times (1\text{-}1/5) = 4}$
Più precisamente i generatori sono:**2**, **6**, **7**,**8**
Verifica - $\Phi(11) = 10 = 2 \times 5$
In colonna e = 5 della tabella a lato si ha sempre $\mathbf{g^{(11-1)/2} = 10,}$ cioè –1. In colonna e = 2, $\mathbf{g^{(11-1)/5}}$ non è mai 1.

	e									
g	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	2	4	8	5	10	9	7	3	6	1
6	6	3	7	9	10	5	8	4	2	1
7	7	5	2	3	10	4	6	9	8	1
8	8	9	6	4	10	3	2	5	7	1

$\mathbf{g^e \bmod 11}$

Algoritmi per l'esponenziazione modulare

Esistono diversi algoritmi in grado di calcolare l'esponenziazione modulare

$\mathbf{y = b^e \bmod m}$ con $\mathbf{1 \le b,e \le m-1}$.

Il più inefficiente (ha tempo esponenziale!) si basa sulla definizione di esponenziazione:

$$\mathbf{b^e \bmod m = (..(((b)b)b)..b) \bmod m} \quad \text{(e volte)}$$

Tutti gli algoritmi con tempo polinomiale usano la **rappresentazione binaria** dell'esponente **e**.
Posto $\mathbf{t = \lceil \log_2 m \rceil}$, si ha $\mathbf{e = e_{t-1}\, 2^{t-1} + e_{t-2}\, 2^{t-2} + \ldots\ldots\ldots + e_1\, 2^1 + e_0\, 2^0}$ e quindi

$$y = b^e = \prod_{i=0}^{t-1} b^{e_i 2^i} = (b^{2^0})^{e_0} \times (b^{2^1})^{e_1} \times \ldots \times (b^{2^{t-1}})^{e_{t-1}}$$

Dalla formula precedente discende il seguente algoritmo.

A0: INPUT: $b \in Z_m$, $0 \le e < m$, $e = (e_{t-1} .. e_0)_2$
OUTPUT: $y = b^e \bmod m$
1. si calcolano **b**, $\mathbf{b^2}$, $\mathbf{b^4}$, $\mathbf{b^8}$ ecc.,
2. si moltiplicano tra loro i $\mathbf{b^i}$ per i quali $\mathbf{e_i = 1}$
3. si divide il risultato per **m** e si trattiene il resto.

Valutiamone la complessità. Occorrono **t** elevamenti al quadrato (per calcolare i $\mathbf{b^i}$), al più **t** moltiplicazioni (per calcolare il prodotto dei $\mathbf{b^i}$) ed una divisione. Supponendo, per semplicità, che l'elevamento al quadrato e la divisione richiedano lo stesso tempo di calcolo di una moltiplicazione, nel caso peggiore l'algoritmo richiede l'esecuzione di **2t+1** moltiplicazioni.

Quanto modo di procedere ha però un difetto: elevamenti al quadrato e moltiplicazioni generano numeri sempre più grandi e richiedono di conseguenza tempo d'esecuzione e memoria sempre più grandi. Per eliminarlo, basta fare di volta in volta la riduzione in modulo dei risultati parziali.

Gli algoritmi con ***repeated square-and-multiply*** (v. [2], pag. 71 e pag. 615), conseguono maggiore efficienza non separando gli elevamenti al quadrato dalle moltiplicazioni: A1 scandisce i bit di **e** dal meno significativo al più significativo, A2 procede in ordine inverso.

A1: INPUT: $b \in Z_m$, $0 \le e < m$, $e = (e_{t-1} .. e_0)_2$
OUTPUT: $y = b^e \bmod m$

1. Set $y \leftarrow 1$. If $e = 0$ then return (y)
2. Set $A \leftarrow b$
3. If $e_0 = 1$ then set $y \leftarrow b$
4. For i from 1 to t-1 do the following:
   4.1 Set $A \leftarrow A^2 \bmod m$
   4.2 If $e_i = 1$ then set $y \leftarrow A \times y \bmod m$
5. Return (y)

A2: INPUT: $b \in Z_m$, $0 \le e < m$, $e = (e_{t-1} .. e_0)_2$
OUTPUT: $y = b^e \bmod m$

1. $y \leftarrow 1$
2. For i from t-1 down to 0 do the following:
   2.1 $y \leftarrow y^2 \bmod m$
   2.2 If $e_i = 1$ then $y \leftarrow y \times b \bmod m$
3. Return (y)

ESEMPIO A1 - **m = 11, b = 7, e = 5 =** (**101**)$_2$

	y	**A**
	1	**7**
e_0	**7**	7
e_1	7	**7 × 7 mod 11 = 5**
e_2	**3 × 7 mod 11 = 10**	**5 × 5 mod 11 = 3**

ESEMPIO A2 - **m = 11, b = 7, e = 5 =** (**101**)$_2$

	y
	1
e_2	**7**
e_1	**7 × 7 mod 11 = 5**
e_0	**5 × 5 mod 11 = 3** **3 × 7 mod 11 = 10**

La complessità di tali algoritmi è di **$O(\log_2 m)$** moltiplicazioni e quindi di **$O((\log m)^3)$** operazioni binarie.

NOTA – A parità di **b** e di **m**, il tempo di esecuzione dell'esponenziazione è proporzionale al n° di "uni" presenti in **e**; indicato con **t** il n° di bit di **e** e con **1<n≤t** il n° di bit con valore 1**,** il numero di moltiplicazioni modulari da eseguire è, infatti, **t+n** .

4.6.4 Logaritmo discreto

Nella precedente tabulazione di **$g^i \bmod 11$** si può notare che ogni elemento di **Z^*_{11}** compare in ogni riga una ed una sola volta. La proprietà notevole vale per qualsiasi primo e per qualsiasi sua **radice primitiva**.

- **T8**: "dati un primo **p**, un generatore **g** ed un qualunque intero **y** maggiore di 0 e minore di **p**, esiste un unico **x** (detto *logaritmo discreto* di **y** rispetto alla base **g**, modulo **p**) tale che **$y = g^x \bmod p$** con **$0 \le x \le (p-1)$** ".

ESEMPIO - Consideriamo **y = 3**: nella tabulazione di **$g^i \bmod 11$** tale valore è presente in ogni riga, una volta sola ed in una colonna sempre diversa.

	x									
g	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	2	4	8	5	10	9	7	3	6	1
6	6	3	7	9	10	5	8	4	2	1
7	7	5	2	3	10	4	6	9	8	1
8	8	9	6	4	10	3	2	5	7	1
	$g^x \bmod 11$									

Tutti gli algoritmi per il calcolo del logaritmo discreto finora individuati sono **non polinomiali**.

Ricerca esauriente – Si calcola **$g^x \bmod p$** per **x** = 0, 1, 2, .. fino a quando non si trova **y**. Il tempo atteso di esecuzione è **$2^{\lceil \log p \rceil - 1}$** esponenziazioni modulari.

La ricerca esauriente non è l'attacco più pericoloso.

Diversi anni fa è stata data una diversa formulazione al problema del logaritmo discreto: al posto della congruenza $\mathbf{y \equiv g^{x}}$ (mod p), si considera la congruenza $\mathbf{y \equiv g^{q\,t\,+\,r}}$ (mod p), in cui **q** e **r** sono, come indica la formula, il **quoziente** ed il **resto** della divisione di **x** per un certo intero **t < x**.

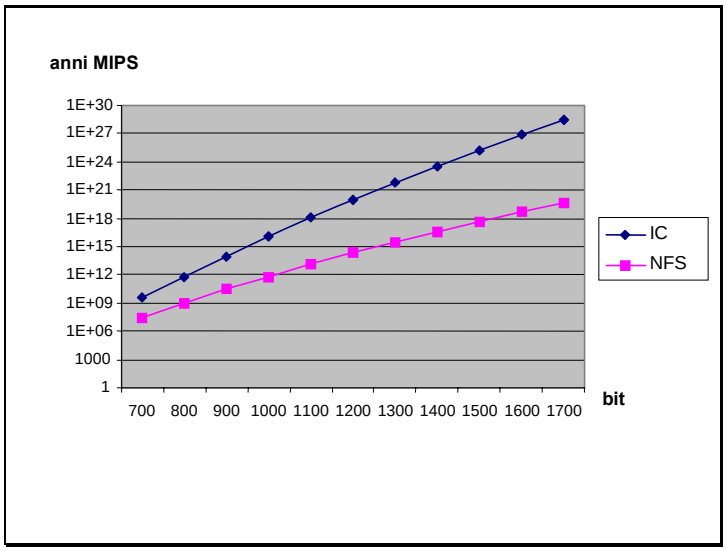
Il vantaggio rispetto alla forza bruta è che si deve eseguire un numero più basso di esponenziazioni con esponenti più piccoli di **x**. La contropartita è che occorre memorizzare **t** risultati parziali.

Un giusto equilibrio tra questi due aspetti si ottiene con $\mathbf{t = \lceil p^{1/2} \rceil}$.

Algoritmo passo del bambino-passo del gigante - Dalla precedente congruenza si ottiene $\mathbf{g^{r} \equiv y \times g^{-q\,t}}$ (mod p) e quindi anche $\mathbf{g^{r} \equiv y \times (g^{-t})^{q}}$ (mod p).

1. Si calcola $\mathbf{g^{r}\ mod\ p}$ per tutti i possibili valori di **r** ($\mathbf{0 \le r < t}$) e si costruisce una tabella in cui i risultati ottenuti, affiancati dal valore di **r** che li ha determinati, sono ordinati per valore crescente.
2. Si controlla se **y** è contenuto in una riga della tabella. Se ciò capita, si prende atto del valore di **r** che lo affianca, si restituisce **x = r** ed il calcolo termina; in caso contrario si pone **i = 1** e si calcola $\mathbf{z = y \times (g^{-t})\ mod\ p}$.
3. Si controlla se **z** è contenuto in una riga della tabella. Se ciò capita, si prende atto del valore di **r** che lo affianca, si restituisce $\mathbf{x = i \times t + r}$ ed il calcolo termina; in caso contrario si ripete con **i = i +1** e $\mathbf{z = z \times (g^{-t})\ mod\ p}$.

Sia $\mathbf{t = \lceil p^{1/2} \rceil}$. Dal punto di vista della complessità computazionale l'algoritmo richiede in questo caso **O(exp ((log p)/2))** operazioni ed altrettanti dati da memorizzare[1].



Algoritmi individuati successivamente[2] sono riusciti dapprima a ridurre le esigenze di memoria (Pohlig-Hellman, Pollard-ρ) e poi a rendere sub-esponenziale il tempo di esecuzione (Index-Calculus, Number Field Sieve, Function Field Sieve).

Con Index Calculus il tempo atteso di esecuzione ha ordine di grandezza asintotico

$$e^{((\ln p)^{1/2}(\ln(\ln p))^{1/2})}$$

Con Number Field Sieve si è successivamente riusciti ad ottenere

$$e^{((\ln p)^{1/3}(\ln(\ln p))^{2/3})}$$

Le curve quantitative indicate in figura vogliono evidenziare da un lato la differenza di questi tempi di esecuzione, da un altro lato l'evoluzione continua degli algoritmi di rottura.

COMMENTO – Per rendere impossibile il calcolo del logaritmo discreto oggi si usano primi di 1000 - 4000 bit.

Gli utenti di uno scambio DH devono però fare attenzione ai dati che impiegano: esistono, infatti, casi particolari in cui l'estrazione del logaritmo diventa facile!.

La scelta del numero segreto X può essere fatta a caso nell'intervallo 1 ÷ p-1, ma bisogna escludere i due estremi (v. pag. 78). Il motivo è molto semplice: se l'intruso intercettasse Y = g o Y = 1 non avrebbe bisogno di estrarre il logaritmo per arrivare alla conclusione che i corrispondenti valori di X sono 1 e p-1.

Anche la scelta di p è critica: Pohlig e Hellman hanno dimostrato che X può essere individuato con solo $2(\log p)^2$ moltiplicazioni, se i divisori di p-1 sono tutti numeri piccoli (v. [3] pag. 121)

4.6.5 Residui quadratici e sottogruppi ciclici

Per motivi di **efficienza** e di **sicurezza**, nelle applicazioni si impiega oggi una differente forma dello scambio di Diffie-Hellman. Il problema d'efficienza nasce dal fatto che per fronteggiare la crescente potenza di calcolo a disposizione dell'attaccante occorre via via i bit del modulo, ma purtroppo tale semplice accorgimento rende anche **sempre più oneroso** il calcolo che devono fare gli utenti. Passando ad esempio da 1024 bit di modulo (una dimensione oggi ancora in uso) a 2048 (il valore attualmente consigliato) il tempo di esecuzione dell'esponenziazione diventa otto volte più grande!

[1] Per la dimostrazione si veda [3] pag. 130. Per fare qualche valutazione si può sfruttare il notebook DL.nb predisposto da Ivo Muccioli.

[2] A.M.Odlyzko, "Discrete logarithms: The past and future", July 1999.

Il problema di sicurezza discende da una sottile vulnerabilità della versione originale: **chi intercetta il dato pubblico Y è in grado d'individuare agevolmente il valore del bit meno significativo del dato privato X**.

Entrambi i problemi sono risolti in maniera soddisfacente se si lavora su un **opportuno sottoinsieme** di Z^*_p: per giustificare tale affermazione, dobbiamo approfondire le nostre nozioni sulla Teoria dei numeri.

Come prima cosa, ci serve una definizione: un numero **q** è detto **residuo quadratico modulo p** se esiste un intero **x** tale che $\mathbf{x^2 \equiv q \pmod p}$; in caso contrario, q è detto essere **non-residuo quadratico**.

Metà dei numeri minori di p sono residui quadratici, metà sono non-residui quadratici.

ESEMPIO – I residui quadratici modulo 11 sono 1,3,4,5,9 e si trovano tutti nella colonna e=2 della tabella di pag.80. I non-residui quadratici non compaiono nella colonna e=2 e sono 2,6,7,8,10

Come seconda cosa, ci serve una deduzione: **ogni radice primitiva modulo p è un non-residuo quadratico**. Se così non fosse, infatti, l'operazione $\mathbf{g^X \bmod p}$ per x = 0, 1, 2, .., p-1 genererebbe solo residui quadratici e quindi solo la metà degli elementi del campo.

ESEMPIO – I quattro generatori di 11 sono non-residui quadratici: 2,6,7,8 (v. pag.81).

Come terza cosa, dobbiamo precisare meglio una proprietà già messa in evidenza in T4.2: se **a** è un residuo quadratico modulo p, allora $\mathbf{a^{(p-1)/2} \bmod p}$ vale **1**; in caso contrario vale **-1**.

Su tale proprietà, si basa un **test**, rappresentato dal simbolo (a/p) introdotto da Legendre, che consente di sapere se un numero minore di p è, o non è, un residuo quadratico modulo p. Per eseguire il test non è necessario eseguire l'esponenziazione: un algoritmo con complessità $O((\lg p)^2)$ è indicato in [2], pag. 73.

Da queste premesse discende che chi intercetta un $\mathbf{Y \equiv g^X} \pmod p$, può sapere agevolmente se è, o non è, un residuo quadratico modulo **p**. Nel primo caso **X** è necessariamente **pari**, nel secondo **dispari**.

ESEMPIO – Riprendiamo in considerazione le radici di p = 11 Nelle colonne e = 2,4,6,8,10 appaiono solo i residui quadratici 1,3,4,5,9. Nelle colonne e = 1,3,5,7,9 appaiono solo i non-residui quadratici 2,6,7,8,10.

	e									
g	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
2	2	4	8	5	10	9	7	3	6	1
6	6	3	7	9	10	5	8	4	2	1
7	7	5	2	3	10	4	6	9	8	1
8	8	9	6	4	10	3	2	5	7	1

$g^e \bmod 11$

La contromisura è **non impiegare radici primitive** come base dell'esponenziazione**.**

Scelto un qualsiasi elemento **h** di GF(p), le esponenziazioni $h^1 \bmod p$, $h^2 \bmod p$, … $h^x \bmod p$ forniscono elementi di GF(p). Sia **q** il più piccolo esponente per cui si ha $\mathbf{h^q \bmod p = 1}$.

Per un tale **h**, detto **generatore di ordine q**, valgono alcune proprietà notevoli:

- per ogni $1 \le i \le q$ e per ogni intero k si ha $\mathbf{h^{i+kq} \bmod p = h^i \bmod p}$;
- l'insieme $\{h^1 \bmod p , h^2 \bmod p , \ldots h^q \bmod p\}$ è un **sottogruppo moltiplicativo di Z^*_p** e contiene **q** elementi;
- **q** è un **divisore** di p-1.

E' dunque possibile adoperare come base dell'esponenziazione DH un residuo quadratico, ma il suo ordine deve essere "grande" per rendere impossibile il citato attacco di Pohlig-Hellman.

Scelto un **p** grande (e quindi dispari), esiste certamente un **h** di ordine **q = (p-1)/2**. Per non stare a cercarlo, i crittografi hanno scoperto che è più semplice cercare un primo di forma particolare.

E' detto *safe prime* un primo della forma **p = 2q + 1**, ove **q** è anch'esso un numero primo[3]. Per tali primi esistono soltanto due sottogruppi, uno di dimensione 2 (che ovviamente non interessa) ed uno di dimensione q (generato da uno qualsiasi dei residui quadratici modulo p, ad esclusione di 1).

ESEMPIO – p = 11 = 2×5 + 1 è un safe prime. L'esponenziazione con base b=3,4,5,9 e con esponente e=1,2,3,4,5 fornisce risultati tutti inclusi nell'unico sottogruppo {1,3,4,5,9}

	e									
b	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
3	3	9	5	4	1	3	9	5	4	1
4	4	5	9	3	1	4	5	9	5	1
5	5	3	4	9	1	5	3	4	9	1
9	9	4	3	5	1	9	4	3	5	1

$b^e \bmod 1$

[3] Nella Teoria dei numeri, q è noto come primo di Sophie Germain

Se il *safe prime* p è formato da **n** bit, **q** ed **e** ne richiedono **n-1**, ma questa piccola diminuzione della dimensione dell'esponente non risolve il problema dell'efficienza citato all'inizio di questo paragrafo.

A tal fine occorre impiegare sottogruppi più piccoli, ma pur sempre troppo grandi per consentire all'intruso di condurre un attacco con forza bruta. In [4], pag. 215, è indicato il seguente metodo per costruirsene uno:

1. si sceglie un primo **q** di dimensione non troppo grande (ad esempio 256 bit);
2. si costruisce un primo **p** della forma **Nq+1**, con N pari e più grande di 2;
3. si individua in Z^*_p un generatore di ordine q della forma $\mathbf{g=\alpha^N \bmod p}$, cercando a caso un α tale che $g \neq 1$ e $\mathbf{g^q \bmod p = 1}$.

ESEMPIO – q = 3, N = 4 e quindi p = 13. Il sottogruppo {1,3,9} ha due generatori: 3 e 9.

Ai fini dello scambio devono essere noti **p**, **g** e **q** o, perlomeno, $\lceil \mathbf{\log q} \rceil$ (v. standard PKCS#3); ogni utente, dopo aver scelto a caso un **X<q** (o di $\lceil \mathbf{\log q} \rceil$ bit, calcola e invia $\mathbf{Y= g^X \bmod p}$.

Quando **q** è noto, l'utente può anche verificare che il dato pubblico dell'altro appartenga realmente al sottogruppo di ordine q (o con il test di Legendre, o calcolando $\mathbf{Y^q \bmod p}$ per verificare se vale 1).

4.7 Numeri primi

Per la sicurezza dello scambio DH occorrono dunque **numeri primi molto grandi**. Nel prossimo capitolo vedremo che questa esigenza è molto sentita in tutta la Crittografia asimmetrica.

4.7.1 Distribuzione dei numeri primi

Lo studio delle proprietà dei numeri primi ha da sempre affascinato i matematici.

ESEMPI – **Euclide** ha dimostrato che i numeri primi sono infiniti.
Eratostene ha indicato un metodo per individuare tutti i primi minori o uguali ad un certo **N**:
1 - si costruisce una tavola dei numeri naturali inclusi nell'intervallo **2 ÷ N**;
2 - si eliminano dapprima tutti i **multipli di 2**, poi **di 3**, poi **di 5** e così via fino a quando non si trova un numero il cui quadrato sia maggiore di **N**. A questo punto la tavola contiene solo numeri primi.
Fermat ha fatto discendere da T4 un test che consente di verificare se un certo intero **x** è primo o composto:

```
for i from 1 to x-1
    y = i^(x-1) mod x
    if y ≠ 1 then return "x è composto"
return "x è primo"
```

Gauss ha formulato una congettura, dimostratasi poi molto ben approssimata, sulla distribuzione dei numeri primi.
Riemann ha studiato una funzione di variabile complessa che ha congetturato avere una profonda connessione con la distribuzione dei primi: la connessione è stata verificata in più casi, ma non è stata ancora dimostrata.

Due teoremi della Teoria dei numeri forniscono un'utile indicazione sulla distribuzione dei numeri primi.

- **T9**: "$\lim_{x \to \infty} \frac{\pi(x)}{x / \ln x} = 1$ *ove* $\pi(x)$ *è il n° di primi* $\leq x$ "

ESEMPI – Una buona approssimazione di $\pi(\mathbf{n})$, valida anche per **n** piccoli, è dunque **n/ln n**; ad esempio per $\pi(\mathbf{30})$, che vale **10**, si ha **30/ln 30** = **8,82**.

Nell'interessante sito http://primes.utm.edu, tutto dedicato alle proprietà dei numeri primi, è indicato che è ancora migliore l'approssimazione $\pi(\mathbf{x}) \cong \mathbf{x/(\ln x - 1)}$.

Da T9 discende che l'ennesimo numero primo è approssimativamente fornito dalla formula $\mathbf{p_n \cong n \ln n}$.

- **T10** (Dirichlet): "se **MCD(a, n) = 1**, allora la successione **p = a + k × n**, con k= 1,2,3,… contiene infiniti numeri primi congruenti ad **a** modulo **n**".

ESEMPI – Attentamente studiate dai Crittografi sono state le progressioni **4k±1** (che forniscono tutti i numeri dispari e quindi tutti i primi escluso 2), **6k±1** (che individuano, con maggiore efficienza, tutti i primi esclusi 2 e 3), **8k±1** e **8k±3** (che individuano, con ancora maggiore efficienza, tutti i primi ad esclusione di 2, 3 e 5), ecc..

La progressione **4k-1** contiene tutti i numeri primi congruenti a 3 modulo 4 (v. pag. 31), la progressione **6k-1** contiene i $p \equiv 5 \pmod 6$ ed in particolate tutti i primi di Sophie Germain e tutti i safe prime (v. pag.84).

4.7.2 Ricerca di numeri primi grandi

Scelto a caso un **x** grande, è dunque molto probabile trovare un primo in un suo intorno di ampiezza **ln x**. Per individuarlo servono solo un buon **test di primalità** e l'esecuzione del seguente algoritmo.

Algoritmo per la ricerca di un grande numero primo -
1. si genera a caso un numero intero grande e dispari;
2. si sottopone il numero ad un test di primalità;
3. se il test lo dichiara primo, si termina; in caso contrario o si torna al passo 1, o si incrementa il numero di 2 e si ritorna al passo 2.

I test di primalità di cui oggi disponiamo sono classificabili in due categorie.
1. Test **deterministici**: se **n** non lo supera è **composto**, se lo supera è **primo**.
2. Test **probabilistici**: se **n** fallisce il test è **composto**; se lo supera è **probabilmente** primo.

I test deterministici hanno complessità esponenziale[4], quelli probabilistici, polinomiale. Questi ultimi devono però essere ripetuti più e più volte per far tendere a 1 la probabilità di avere realmente individuato un primo.

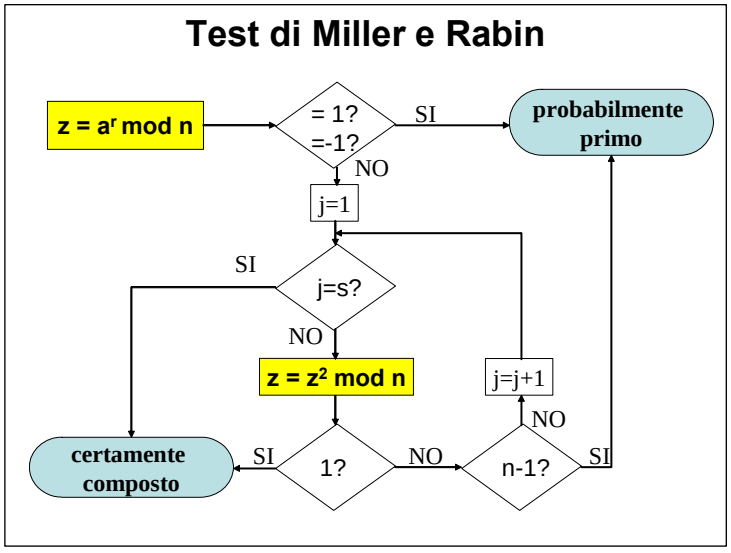
Il più usato è il **test di Miller-Rabin**. Il principio è di scegliere a caso un intero **a**, con **1 <a <n** e di calcolare $\mathbf{a^{n-1}}$ **mod n**: se il risultato è diverso da 1, allora **n** è certamente composto; se il risultato vale 1, **n** è detto **pseudoprimo in base a** e può anche essere un numero primo (per poter affermare con certezza che **n** è primo bisognerebbe verificare **T4** per ogni possibile **a**, come abbiamo visto nel test di Fermat citato in precedenza).

Sia **n** un numero dispari. Essendo **n-1** pari, si pone $\mathbf{n-1 = r \times 2^s}$ con **r** dispari e $\mathbf{s \geq 1}$. Si ha dunque:

$$a^{n-1} \bmod n = (a^r)^{2^s} \bmod n$$

Il test inizia con il calcolo di un'esponenziazione "più semplice" che quella richiesta dal test di Fermat:

$\mathbf{z = a^r \bmod n}$.



Se si ottiene **z = 1** o **z = n-1** (N.B. cioè –1) il test termina e **n** è dichiarato **probabilmente primo**: l'elevamento di z alla potenza 2^s è inutile essendo già certo che il risultato sarà 1.

In caso contrario si prosegue iterando al più s-1 volte il calcolo

$\mathbf{z = z^2 \bmod n}$.

Se in una qualsiasi delle iterazioni si ottiene **z = n-1** il test termina per le ragioni sopradette e **n** è dichiarato **probabilmente primo**.

Il test termina, anche se in una qualsiasi iterazione si ottiene $\mathbf{z^2 = 1}$: è, infatti, certo che solo un **numero composto**, diverso da 1 e da -1 può avere il quadrato uguale a 1 (v. [2], Fact 3.18). La stessa conclusione si può trarre se si arriva all'ultimo passo senza aver mai trovato né 1, né -1.

Per poter confidare sulla primalità di **n**, bisogna ripetere il test più volte con valori di **a** scelti a caso: dopo **t** test, tutti superati con la dichiarazione che **n** è probabilmente primo, la probabilità che non lo sia è più piccola di $\mathbf{2^{-2t}}$.

ESEMPIO – In [2], nell'ipotesi che il numero da testare sia stato generato a caso dall'utente, si dimostra che sono sufficienti 4-5 iterazioni.

In [4] si esamina il contesto di un primo ricevuto da una terza parte (che potrebbe essere un malintenzionato), ad esempio prima di procedere ad uno scambio DH: in questo caso viene suggerito di ripetere il test 64 volte per garantirsi un livello di sicurezza di 128 bit (se ci si fida e si usa come modulo un numero non primo, il dato privato non sarebbe, infatti, più protetto dall'impossibilità di calcolo del logaritmo discreto e la successiva chiave condivisa diventerebbe nota anche all'avversario).

[4] Nel 2002 Manindra Agrawal, Neeraj Kayal e Nitin Safena, tre ricercatori dell'Indian Institute of Technology di Kanpur, hanno per la prima volta individuato un algoritmo con complessità polinomiale. Studi successivi su AKS ne hanno ulteriormente ridotto la complessità computazionale.

Cap.5: ampliamenti e modifiche

Pag. 86

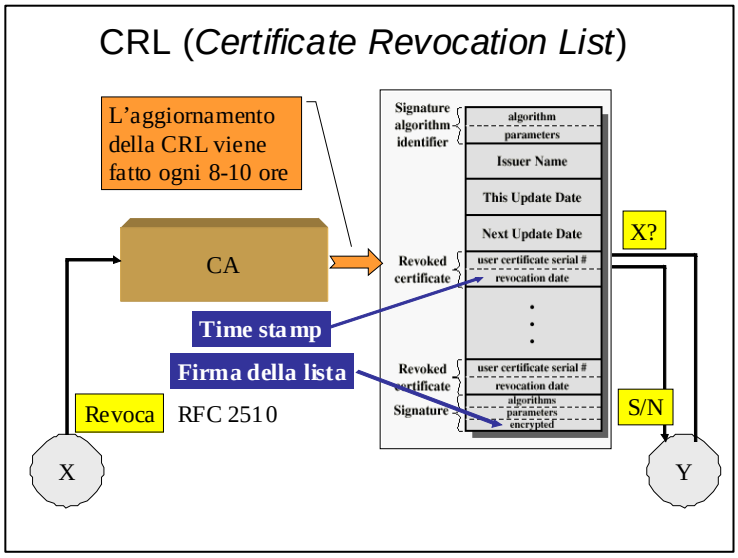
- **P1: problema del logaritmo discreto su un campo di Galois** - Dato un primo **p**, un generatore **g** ed un intero $c \in Z^*_p$, trovare l'intero **x**, $1 \leq x \leq p-2$, tale che $g^x \bmod p = c$, o anche $\mathbf{g^x \equiv c \pmod p}$.

COMMENTI – Abbiamo già visto che su P1 si basa la sicurezza dello scambio DH e che l'impossibilità di calcolo si ottiene oggi con numeri primi formati da almeno un migliaio di bit. Si ricordi che lo scambio, con la stessa robustezza, può essere definito su GF(p), su un suo sottogruppo di grande ordine e su $GF(p^n)$.

Vedremo tra poco che l'intrattabilità di P1 è il presupposto che rende pseudo-unidirezionale la funzione di cifratura del Cifrario di ElGamal e la funzione di verifica dello Schema di firma digitale DSS.

Pag. 91

Può essere necessario rinunciare ad usare una chiave, anche se non è stata compromessa: ciò capita quando l'utente, per una qualsiasi ragione, non ha più lo "stato" o il "ruolo" descritto nel certificato.



Il **ripudio** è l'azione con cui un utente, prima della scadenza, spezza l'associazione chiave pubblica/proprietario sancita da un certificato. E' indispensabile che questo evento sia **tempestivamente** notificato a tutti gli altri utenti.

La raccolta delle notifiche di ripudio è fatta dalla **RA**; alla **CA** compete la successiva azione di **revoca del certificato**.

Per rendere edotti tutti gli altri utenti che una chiave non deve essere più usata, **CA** mantiene *on line* una lista **autenticata** dei certificati revocati (**CRL**).

Chiunque è dunque così in grado di sapere se è ancora valida la chiave pubblica di chiunque altro. Fino a poco tempo fa occorreva reperire la CRL (dove è indicato nel certificato), scaricarla e poi interrogarla; ora alcune CA supportano *query* dirette.

Presupposto essenziale è che a CA non passi mai per la testa l'idea di fare "l'uomo in mezzo".

Pag. 96-97

1 – L'algoritmo *E*

La trasformazione operata sul testo in chiaro **m** è l'esponenziazione modulare:

$$\mathbf{c = m^e \bmod n}$$

ove $\mathbf{n = p \times q}$ è il prodotto di due distinti **numeri primi p** e **q** ed **e** è **coprimo** con $\Phi(\mathbf{n}) = (\mathbf{p-1}) \times (\mathbf{q-1})$.

N.B. – La precedente formulazione dell'algoritmo non è l'unica possibile. Per trattare valori più piccoli dell'esponente, si può anche imporre **e** coprimo con $\lambda(\mathbf{n}) = \mathbf{mcm(p-1, q-1)} = \Phi(\mathbf{n})/\mathbf{MCD(p-1,q-1)}$. Per rendere più veloci i calcoli di decifrazione (v. pag. 105) **n** può essere il prodotto di più di due primi.

La coppia di numeri $\mathbf{P = \{e, n\}}$ è quindi la **chiave pubblica del destinatario**. Per l'esecuzione dei calcoli si ricorre usualmente ad uno dei due algoritmi con *repeated square-and-multiply* (v. pag. 82).

2- Aritmetica modulo $\mathbf{n = p \times q}$, con p e q numeri primi

Per comprendere il funzionamento di RSA sono utili alcuni teoremi della Teoria dei numeri. Alla base di tutto c'è la generalizzazione, individuata da Eulero, del piccolo teorema di Fermat (v. T4, pag.80).

- **T11:** "per ogni **n** (N.B. non importa se primo o composto) e per ogni **x coprimo** con **n** si ha
$$\mathbf{x^{\Phi(n)} \equiv 1 \pmod n}$$ ".

Corollari

T11.1: "se $\mathbf{n = p \times q}$ e se $\mathbf{r \equiv s \pmod{\Phi(n)}}$, allora $\mathbf{a^r \equiv a^s \pmod n}$ per ogni intero **a**".

COMMENTO – Per non complicare inutilmente i calcoli, in RSA si deve quindi scegliere un $\mathbf{e \leq \Phi(n)}$.

T11.2: "per ogni $\mathbf{n = p \times q}$, con p e q **primi,** e per **ogni x > 0** si ha
$$\mathbf{x^{\Phi(n)+1} \equiv x \pmod n}$$ ".

COMMENTO – Per T11.1 si ha anche $\mathbf{x^{k\,\Phi(n)+1} \equiv x \pmod n}$.

- **T12:** "per ogni **n**, $\mathbf{Z^*_n}$ e la **moltiplicazione mod n** formano un **gruppo moltiplicativo**; se $\mathbf{n = p \times q}$ ed **e** è **coprimo con** $\Phi(\mathbf{n})$, $\mathbf{m^e \bmod n}$, calcolata per ogni $\mathbf{m \in Z_n}$, produce una permutazione di $\mathbf{Z_n}$".

COMMENTO – Per la decifrabilità di RSA, ad ogni testo cifrato deve corrispondere un solo testo in chiaro. Per ottenere una permutazione di $\mathbf{Z_n}$ deve dunque essere $\mathbf{e \in Z^*_{\Phi(n)}}$; è ovvio che il caso **e = 1** non interessa.

E' sicuramente utile rivedere le precedenti proprietà in un caso semplice[5].

[5] Per fare verifiche su RSA con Mathematica, è possibile scaricare dal sito del corso RSA.nb e RivestShamirAdleman.nb

3 – L'algoritmo *D* (pag. 98-101)

L'operazione inversa della cifratura RSA è l'**estrazione della radice e-esima modulo n del testo cifrato c**. Per eseguirla occorre saper risolvere il problema difficile P2, ma abbiamo già detto che il calcolo è attualmente impossibile se **n** è formato da almeno un migliaio di bit.

Il destinatario del messaggio riservato, cioè chi ha generato la chiave pubblica **P**, dispone di una **trapdoor**.

Sia **S = {d, n}** la sua chiave privata, con **$d = e^{-1} \bmod \Phi(n)$** i**nverso moltiplicativo** di **e** modulo **$\Phi(n)$**. Per ottenere il testo in chiaro gli è sufficiente eseguire un'esponenziazione; si ha, infatti:

$$c^d \bmod n = (m^e)^d \bmod n = m^{e\,d} \bmod n.$$

ESEMPIO – Riprendiamo il caso **n = 33** e tabuliamo il **prodotto mod 20** tra numeri minori di **$\Phi(33)$** e coprimi con esso.
Si noti ad esempio che **3** e **7** sono uno l'inverso moltiplicativo dell'altro modulo 20. Dalla tabella di pag. 97 si ottiene

m	$c = m^3 \bmod 33$	$c^7 \bmod 33$
1	1	1
2	8	2
.....	.....	.....
31	25	31
32	32	32

*	1	3	7	9	11	13	17	19
1	1	3	7	9	11	13	17	19
3	3	9	1	7	13	19	11	17
7	7	1	9	3	17	11	19	13
9	9	7	3	1	19	17	13	11
11	11	13	17	19	1	3	7	9
13	13	19	11	17	3	9	1	7
17	17	11	19	13	7	1	9	3
19	19	17	13	11	9	7	3	1

Il corretto funzionamento di RSA può anche essere verificato[6] formalmente.

Per $n = p \times q$ si ha: **$\Phi(n) = \Phi(p \times q) = \Phi(p) \times \Phi(q) = (p-1) \times (q-1)$**.
Da T11.2 si ha: **$m^{k\,\Phi(n)+1} = m^{k\,(p-1)(q-1)\,+1} \equiv m \pmod n$** per ogni $0 < m < n$.

Per la decifrabilità di ogni c deve essere: **$c^d \bmod n = m^{e\,d} \bmod n = m$** con $m \in Z_n$.

La precedente congruenza è vera se: **$e\,d = k(p-1)(q-1)+1$**,
o anche **$e\,d - k(p-1)(q-1) = 1$** (*)

Un teorema, dovuto a Bézout, evidenzia che il massimo comune divisore di due numeri interi **a** e **b** può sempre essere espresso da una loro combinazione lineare a coefficienti interi **x** e **y**:

MCD(a, b) = x a + y b.

La (*) equivale dunque a **$MCD(e, \Phi(n)) = 1$**
ed operando mod $\Phi(n)$ anche a **$e\,d \equiv 1 \pmod{\Phi(n)}$**

In altre parole il corretto funzionamento di RSA è garantito se e solo se il numero **e** è relativamente primo con **$\Phi(n)$** ed il numero **d** è il suo inverso moltiplicativo in **$Z^*_{\Phi(n)}$**.
Ma queste sono proprio le ipotesi da cui siamo partiti nella presentazione degli algoritmi E, D.

4 – L'algoritmo *G*

Ogni utente **U** di RSA deve eseguire il seguente algoritmo.

Generazione delle chiavi di RSA:
1. si generano a caso due grandi numeri primi **p**, **q** di circa egual dimensione
2. si calcola **$n = p \times q$**
3. si calcola **$\Phi(n) = \Phi(p) \times \Phi(q) = (p-1)(q-1)$**
4. si sceglie un intero **e**, **$1 < e < \Phi(n)$**, tale che **$MCD(\Phi(n), e) = 1$**
5. si calcola **$d = e^{-1} \bmod \Phi(n)$**
6. si rende di dominio pubblico la chiave **{e, n}**
7. si impiega come privata la chiave **{d, n}**

Tutti questi calcoli hanno complessità polinomiale. Verifichiamolo in due casi.

CALCOLO DELL'ESPONENTE PUBBLICO **e**: occorre scegliere un **e** tale che **$MCD(e, \Phi(n)) = 1$**.

L'**algoritmo di Euclide** è un modo efficiente per calcolare il massimo comun divisore di due interi **a** e **b**: la complessità è di **$O((\lg n)^2)$** operazioni su bit.

[6] Giulio C. Barozzi "I codici crittografici a chiave pubblica" Conferenza, Dallas, 2000.

MCD(a,b) o anche solo (a,b)
a, b interi positivi; a≥b>0
Finché b>0
calcola r = a mod b
poni a = b
poni b = r
Restituisci **a**

ESEMPIO: MCD(240,42) = 6

passo	a	b	r
1	240	42	30
2	42	30	12
3	30	12	6
4	12	6	0
	6		

Per il calcolo di **e** si può dunque scegliere a caso un **r < Φ(n)** e poi verificare se **MCD(r, Φ(n)) = 1**. In caso contrario si itera il procedimento. Di solito bastano poche prove, essendo

(*Prob*(MCD(r, Φ(n) =1) = Φ(Φ(n))/Φ(n).

Usualmente però l'esponente **e** viene **fissato** in modo da rendere semplice il calcolo di m^e mod n: a tal fine deve essere un numero binario con "pochi uni" [7]. Valori in uso sono **3** e $\mathbf{2^{16}+1}$ (ritenuto più sicuro).

CALCOLO DELL'ESPONENTE PRIVATO **d:** deve essere $\mathbf{d = e^{-1} \bmod \Phi(n)}$.

Dati due interi qualsiasi a, b il calcolo del loro massimo comun divisore e dei due coefficienti x,y della combinazione lineare che lo individua (th. di Bézout) può essere efficientemente fatto con l'algoritmo **esteso di Euclide**. La complessità è $\mathbf{O((\lg n)^2)}$ operazioni su bit:

MCD(a,b) = x a + y b
a, b interi positivi; a≥b>0
Poni $x_2 = 1$, $x_1 = 0$, $y_2 = 0$, $y_1 = 1$
Finché b>0
calcola $q = \lfloor a/b \rfloor$, $r = a - qb$
calcola $x = x_2 - qx_1$, $y = y_2 - qy_1$
poni $a = b$, $b = r$, $x_2 = x_1$, $x_1 = x$, $y_2 = y_1$, $y_1 = y$
Restituisci **MCD(a,b) = a, x = x_2 , y = y_2**

ESEMPIO – MCD(240,42) = 6 = 3×240 – 17×42

passo	a	b	x_2	x_1	y_2	y_1	q	r	x	y
1	240	42	1	0	0	1	5	30	1	-5
2	42	30	0	1	1	-5	1	12	-1	6
3	30	12	1	-1	-5	6	2	6	3	-17
4	12	6	-1	3	6	-17	2	0		
	6		**3**		**-17**					

Nel caso particolare di RSA, occorre dunque soltanto fornire in ingresso a tale algoritmo **Φ(n), e**; in uscita si otterrà (**Φ(n)**,**e**) = 1 (ma lo si sapeva già), **d** (il coefficiente di **e**) ed un intero che non interessa.

5 - Decifrazione con il teorema cinese dei resti

Dal punto di vista della sicurezza, sarebbe molto opportuno che chi ha generato una coppia di chiavi RSA eliminasse ogni traccia dei due numeri primi impiegati.

Dal punto di vista dell'efficienza è invece utile che li conservi: esiste, infatti, un metodo, basato proprio sulla conoscenza di **p** e di **q**, che consente di ridurre di **circa quattro volte** il carico di lavoro della decifrazione.

Per giustificarlo è necessario richiamare il **teorema cinese dei resti**, particolarmente utile anche in altre elaborazioni richieste dalla Crittografia asimmetrica.

- **T13: "**se gli interi $\mathbf{n_1}$, $\mathbf{n_2}$,..$\mathbf{n_k}$ sono **coprimi** a due a due, allora il sistema di congruenze
  $\mathbf{x \equiv a_1 \pmod{n_1}}$,
  $\mathbf{x \equiv a_2 \pmod{n_2}}$,
  … ,
  $\mathbf{x \equiv a_k \pmod{n_k}}$
  ha un'unica soluzione modulo $\mathbf{n = n_1 \times n_2 \times \ldots \times n_k}$ ".

[7] In questi casi la coppia **e**,**n** può essere assegnata con una stringa binaria più lunga di quella che rappresenta il solo n, ma più corta di 2 log n: ecco la spiegazione dell'incongruenza segnalata a pag. 93.

Una conseguenza importante. Quando si deve elaborare un certo numero **y** è possibile
1. passare alla sua **rappresentazione modulare** $\mathbf{v(y) = (y \bmod n_1, y \bmod n_2, ..., y \bmod n_k)}$,
2. elaborare **separatamente** le k coordinate,
3. restituire un risultato con la **rappresentazione originaria**.

CASO DI STUDIO - Nella decifrazione di RSA, la nuova rappresentazione di **c** si ottiene impiegando come moduli p e q (primi e quindi anche coprimi): $\mathbf{v(c) = (c \bmod p, c \bmod q)}$

In tale rappresentazione il calcolo di $\mathbf{c^d \bmod n}$ richiede due esponenziazioni modulari:

$$\mathbf{v(c^d) = ((c \bmod p)^d \bmod p, (c \bmod q)^d \bmod q)}$$

Il già citato vantaggio computazionale di procedere in questo modo è tutto nel fatto che i numeri da trattare sono di dimensione più piccola (½ log n). La complessità del calcolo è quindi $2(\frac{1}{2} \log n)^3 = \frac{1}{4} (\log n)^3$

Esistono due metodi per passare da una rappresentazione modulare **v(y)** alla rappresentazione originaria di **y**. Il primo, dovuto a Gauss, richiede di calcolare una combinazione lineare a coefficienti interi delle coordinare

CASO DI STUDIO **–** Indichiamo con **a** e con **b** i coefficienti che rimettono in chiaro ciò che è stato decifrato nella rappresentazione modulare. Deve dunque essere:

$$\mathbf{m = c^d \bmod n = \{a [(c \bmod p)^d \bmod p] + b [(c \bmod q)^d \bmod q]\} \bmod n}$$

I due coefficienti possono essere agevolmente calcolati con **T13** dovendo soddisfare le congruenze

$$\mathbf{a \equiv 1 \pmod p} \qquad \mathbf{b \equiv 0 \pmod p}$$
$$\mathbf{a \equiv 0 \pmod q} \qquad \mathbf{b \equiv 1 \pmod q}$$

ESEMPIO – Sia p = 3, q = 11 e n = 33. Per x = 17 si ha v(x) = (2,6).
Verifichiamo che a = 22, b = 12 soddisfano le congruenze indicate nel caso di studio:
22 mod 3 = 1, 22 mod 11 = 0, 12 mod 3 = 0, 12 mod 11 = 1.
Si ha inoltre: (22×2 + 12×6) mod 33 = 116 mod 33 = 17

Il passaggio dalla rappresentazione modulare alla rappresentazione originaria può essere oggi fatto in modo ancora più efficiente.

A**lgoritmo di Garner** - Sia $\mathbf{n = p \times q}$, con **p** e **q** primi
Sia inoltre assegnata la rappresentazione modulare di **y**

$$\mathbf{v(y) = (v1 = y \bmod p, v2 = y \bmod q)}, \text{ con } \mathbf{0 \le y < n}.$$

Il calcolo di **y** richiede tre passi:
1. $\mathbf{C = p^{-1} \bmod q}$
2. $\mathbf{u = ((v2 - v1) \times C) \bmod q}$
3. $\mathbf{y = v1 + u \times p}$

Formula di Garner – Per sostituzione si ha anche: $\mathbf{y = v1 + (((v2 - v1) (p^{-1} \bmod q) \bmod q) \times p}$

CASO DI STUDIO **–** Nelle implementazioni di RSA basate sul teorema cinese del resto (CRT) viene oggi usato l'algoritmo di Garner. Per rendere i calcoli efficienti, la chiave privata è formata da 5 componenti:
- **n**, il modulo dell'esponenziazione
- **p**, il primo fattore di n
- **q**, il secondo fattore di n
- $\mathbf{C = p^{-1} \bmod q}$, la costante chiamata in causa dall'algoritmo
- **d1= d mod (p-1)**, l'esponente più piccolo per il calcolo della prima coordinata (v. T4.1)
- **d2= d mod (q-1)**, l'esponente più piccolo per il calcolo della seconda coordinata

La decifrazione di un testo cifrato **c** richiede dunque i seguenti calcoli:

$\mathbf{v1 = c^{d1} \bmod p}$
$\mathbf{v2 = c^{d2} \bmod q}$
$\mathbf{u = ((v2 - v1) \times C) \bmod q}$
$\mathbf{m = v1 + u \times p}$

ESEMPIO – Sia n = 33, p = 3, q = 11, e = 3, d = 7.
I parametri della chiave privata sono: C = 4, d1 = 1, d2 = 7.
Per il testo in chiaro m =8 la cifratura ha generato il testo cifrato c = 17
In decifrazione si ha v1 = 2, v2 = 8, u = (6 x 4) mod q = 2 e quindi m = 2 + 2 x 3 = 8.

NOTA IMPORTANTE - Lo standard PKCS#1 v2 prevede che gli utenti possano anche usare **più di due primi** nel calcolo del modulo di RSA. Ciò risponde a due differenti esigenze:

- poter fare calcoli rispetto ad un modulo sempre più grande per fronteggiare la probabile evoluzione degli algoritmi di fattorizzazione e delle macchine che li eseguono,
- poter ottenere dall'uso del teorema cinese del resto un'efficienza ancora maggiore quando si deve o decifrare un messaggio riservato o, come vedremo, apporre la firma digitale ad un documento. L'algoritmo di Garner, con poche modifiche (v. [2], pag. 612), si applica infatti anche a questo caso.

6- Robustezza di RSA

Ipotesi fondamentale per la sicurezza di RSA è che il proprietario della chiave privata **S = {d, n}** tenga **segreti d** ed i fattori **p** e **q** di **n**.

Vediamo quali attacchi può condurre un avversario e quali contromisure si devono adottare per fronteggiarli.

6.1 - Chi conosce la sola chiave pubblica di un utente di RSA può individuare la corrispondente chiave privata fattorizzando **n** e calcolando poi $\Phi(n)$ e $d = e^{-1} \bmod \Phi(n)$. La presunta intrattabilità di P3 e la scelta di primi adeguatamente "grandi" rendono questo attacco computazionalmente impossibile.

Altri attacchi basati sulle proprietà matematiche di RSA hanno tutti mostrato la stessa complessità computazionale della fattorizzazione.

Sfruttare la conoscenza di un testo cifrato intercettato non fornisce alcun vantaggio computazionale.

ESEMPI – L'estrazione della radice e-esima ha la stessa complessità computazionale della fattorizzazione. Di pari complessità computazionale è il "**cycling attack**", che prevede di iterare la cifratura a partire da un testo intercettato **c** fino a quando non lo si riottene: se dopo **k** passi si ha $(..((c^e)^e)..^e) = c$, allora al passo precedente si ha **m**.

6.2 - RSA è un Cifrario **deterministico** ed ha quindi la vulnerabilità che a blocchi identici di testo in chiaro corrispondono blocchi identici di testo cifrato. Presenta inoltre una ulteriore vulnerabilità: per alcuni testi in chiaro (in realtà sono pochissimi e ben noti), si ha $m \equiv m^e \pmod n$.

ESEMPIO - Per e = 3 (v. tabella di pag. 101) sono "non nascosti" i messaggi 1, 10, 11, 12, 21, 22, 23, 32. Gli stessi messaggi sono non nascosti anche per e = 7.

La contromisura è, come si è detto a pag. 99, quella di rendere RSA **probabilistico**. A tal fine gli utenti, prima della cifratura, devono concatenare un numero a caso al testo in chiaro; quest'ultimo deve quindi avere una dimensione adeguatamente inferiore a quella del modulo.

PKCS#1 prevede due soluzioni che riassumiamo qui succintamente[8].

La prima premette a **m** un padding di 88 bit: 0x**00 ||** 0x**02 || r ||** 0x**00** || **m** , con **r** stringa di 64 bit casuali.

La seconda, detta OAEP (*Optimal Asymmetric Encryption Padding*) è decisamente più complessa, ma è anche considerata più sicura; il padding è ottenuto concatenando al prefisso 00 due dati casuali ottenuti sottoponendo **m** e **r** a funzioni hash ed a somme modulo due: 0x**00** || **maskedSeed** || **maskedDB**

6.3 - RSA può essere rotto con un attacco "con testo cifrato scelto". In questo contesto si suppone che l'intruso possa richiedere al proprietario della chiave privata la decifrazione di qualsiasi messaggio di sua scelta, ad esclusione del messaggio **c** di suo interesse.

L'attacco si basa sulla **proprietà moltiplicativa** di RSA.

Consideriamo un messaggio $m = m_1 \times m_2$. La sua cifratura è:

$$c = m^e \bmod n = ((m_1^e \bmod n) \times (m_2^e \bmod n)) \bmod n$$

La cifratura del prodotto di due messaggi è uguale al prodotto dei due testi cifrati. Analogamente si ha che **la decifrazione di un testo cifrato ottenuto moltiplicando due testi cifrati è uguale al prodotto dei due corrispondenti testi in chiaro.**

Un intruso che ha intercettato **c** può dunque generare un numero a caso **r**, costruire $c1 = c \times r^e$, calcolare $m2 = r^{-1}$, ottenere la decifrazione **m1** di **c1** e moltiplicare infine **m1** e **m2**.

Pur non avendo inviato **c**, l'intruso ha così ottenuto la decifrazione che gli interessa!

[8] Per maggiori dettagli, è sufficiente consultare lo standard.

Un caso particolare, detto *side-channel attack*, si ha quando l'intruso può richiedere la decifrazione di messaggi di sua scelta ed ottenere informazioni sui calcoli che vengono eseguiti dal destinatario (ad esempio il tempo di esecuzione, o il consumo di potenza, o l'emissione di onde elettromagnetiche, o la generazione di avvisi sonori). In queste situazioni oggetto dell'attacco non è l'algoritmo crittografico, ma la sua implementazione.

A prima vista lo svolgimento di questi attacchi può sembrare estremamente difficile, ma non è così: basti pensare alla facilità con cui un intruso può misurare le radiazioni emesse dallo schermo di un PC, o il consumo di potenza di una smart card lasciata incustodita, o il tempo di risposta di un server che esegue un protocollo interattivo in situazioni di basso carico.

ESEMPIO - Il "**timing attack**" si basa sulla misura del tempo di esecuzione per dedurre il parametro segreto di un algoritmo crittografico[9]. Nel caso di RSA (o anche dello scambio DH) si sfrutta il fatto che l'esponenziazione modulare è di norma realizzata con uno degli algoritmi con *repeated square and multiply*: Ciò consente all'attaccante di discriminare due casi: se uno qualsiasi dei bit dell'esponente segreto è 0, l'algoritmo esegue solo un elevamento al quadrato, se è 1, viene eseguita anche una moltiplicazione. In seguito, in dipendenza dal risultato ottenuto e quindi dalla base dell'esponenziazione, venire eseguita o meno una riduzione in modulo.

E' stato dimostrato che sulla base di queste differenze di comportamento è possibile preparare un certo numero di testi cifrati, richiederne la decifrazione ed usare la misura del tempo di esecuzione per ricostruire, bit a bit, l'intero esponente.

Ciò ha imposto di correre ai ripari. La contromisura attualmente più diffusa è quella di impiegare la proprietà moltiplicativa di RSA: prima della decifrazione si moltiplica **c** per $\mathbf{r}^{e}$, con **r** numero a caso; dopo la decifrazione si moltiplica il risultato per $\mathbf{r}^{-1}$.

5.4.2 Il Cifrario di ElGamal

Pag. 108-109

2 – L'algoritmo di firma DSS

DSS, anch'esso **probabilistico** e **con appendice**, discende dall'algoritmo di firma di ElGamal e basa quindi la sua sicurezza sul problema difficile del logaritmo discreto.

[9] P. Kocher "*Timing attacks on implementation ..* ecc." in http://www.cryptography.com/resources/whitepapers/TimingAttacks.pdf

Le differenze sono tre.

1. La dimensione di **p** è fissata dallo standard: $2^{512} < \mathbf{p} < 2^{1024}$.
2. I calcoli di **R** e di **S** sono fatti **mod q**, ove **q** è il più grande fattore primo di **p-1** tale che $2^{159} < \mathbf{q} < 2^{160}$: le due etichette sono dunque formate ciascuna da 160 bit.
3. L'impronta del messaggio **H(m)** deve essere di 160 bit e deve essere calcolata con **SHA-1**.

Per ottenere minore onerosità del calcolo dell'esponenziazione, l'algoritmo opera su un **sottogruppo moltiplicativo** di $\mathbf{Z_p}$ di ordine **q** (v. pag. 85).

Nel caso del DSS il sottogruppo ciclico di ordine **q** è unico.

Algoritmo *G* - Ogni utente **U** deve

1. scegliere a caso il suo primo **q**,
2. individuare un primo **p** tale che **q** sia un divisore di **p-1** (esiste in proposito un algoritmo suggerito dal NIST: v. [2], pag. 452),
3. individuare un generatore **g** del sottogruppo di ordine **q**,
4. scegliere a caso una chiave privata **a**, con $\mathbf{1 \le a \le q-1}$,
5. calcolare $\mathbf{y = g^a \bmod p}$ e rendere di dominio pubblico la chiave pubblica {**p**, **q**, **g , y**}.

Algoritmo *S* - Per ogni messaggio **m** da firmare (la lunghezza è arbitraria, dato che è autenticata la sola impronta), **U** deve scegliere un numero a caso **k**, con $\mathbf{1 \le k \le q-1}$ e calcolare due etichette di 160 bit ciascuna:

$$\mathbf{R = (g^k \bmod p) \bmod q}$$
$$\mathbf{S = [k^{-1} \times (H(m) + a \times R)] \bmod q}$$

Una firma autentica **m** deve dunque soddisfare l'equazione congruenziale

$$\mathbf{H(m) \equiv - a \times R + k \times S \ (mod\ q)}$$

Algoritmo *V* - Per verificare una firma di **U** occorre conoscere in modo sicuro **PU** (tipicamente estraendola da un certificato) e considerare la terna <**m**, **S**, **R**>.

La prima operazione da fare è controllare che entrambe le etichette ricevute siano maggiori di **0** e minori di **q**: in caso contrario la firma è sicuramente falsa.

Si devono poi fare i seguenti calcoli:

$$\mathbf{w = S^{-1} \bmod q}$$
$$\mathbf{u1 = [H(m) \times w] \bmod q}$$
$$\mathbf{u2 = (R \times w) \bmod q}$$
$$\mathbf{v = [(g^{u1} \times y^{u2}) \bmod p] \bmod q}.$$

Se si ottiene **v = R,** allora si è certi che **U** è il firmatario di **m**.

Giustificazione[10] – Supponiamo che le due etichette siano autentiche.

Dalla precedente equazione di firma si ha:

$$\mathbf{H(m) \equiv - a \times R + k \times S \ (mod\ q)}.$$

Dopo aver moltiplicato i due membri per **w** si ottiene:

$$\mathbf{w \times H(m) + w \times a \times R \equiv k \ (mod\ q)} \text{ e quindi}$$
$$\mathbf{u1 + a \times u2 \equiv k \ (mod\ q).}$$

Impiegando le due parti come esponenti di g si ha:

$$\mathbf{g^{(u1 + a\ u2) \bmod q} \equiv g^{k \bmod q} \ (mod\ p)} \text{ e quindi}$$
$$\mathbf{(g^{u1} \times y^{u2} \bmod p) \bmod q \equiv (g^k \bmod p) \bmod q,} \text{ cioè}$$
$$\mathbf{v = R.}$$

[10] Utili verifiche possono essere fatte anche con DSS.nb, disponibile nel sito del corso