

## una possibile funzione unidirezionale

- moltiplicare due interi a  $n$  bit è **facile** (in  $\mathcal{O}(n^2)$  con l'algoritmo usuale)
- trovare un primo a  $n$  bit, e verificare che è primo, è **facile**
- fattorizzare un numero a  $n$  bit è **difficile** ( $2^{cn^{1/3}}$ )
- si può costruire un crittosistema a chiave pubblica basato su questa osservazione?

## crittosistema RSA

- Sia  $N = pq$ ,  $p, q$  primi. Sia  $\mathcal{P} = \mathcal{C} = \mathbb{Z}_N$ .
- Lo spazio delle chiavi è

$$\mathcal{K} = \{(N, p, q, d, e) \mid de \equiv 1 \pmod{\phi(N)}\}.$$

- Se  $k = (N, p, q, d, e)$  è una chiave, poniamo
- $e_k(x) = x^e \pmod{N}$
- $N$  e  $e$  sono la **chiave pubblica**
- $d_k(y) = y^d \pmod{N}$
- $p, q, d$  sono la **chiave privata**

## RSA – funzione trapdoor unidirezionale

- una trapdoor (botola) one-way function è una funzione unidirezionale che diventa facile da invertire, se si conosce un'informazione supplementare
- la nostra lo è: abbiamo visto che si può invertire la f.ne  $f_e : x \rightarrow x^e \pmod{N}$  usando la funzione  $f_d : y \rightarrow y^d \pmod{N}$ , se  $ed \equiv 1 \pmod{\phi(N)}$
- l' informazione supplementare è l'esponente  $d$  – difficile da ricavare dalle informazioni pubbliche

## fattorizzazione

- la difficoltà di fattorizzare interi grandi non va comunque sopravvalutata
- nel '77, Rivest, Shamir e Adelman hanno proposto una sfida nella rubrica di Martin Gardner su *Scientific American*
- bisognava decifrare un testo cifrato con l'RSA-129 – con chiave pubblica  $e = 9007$  e

$N = 114381625757888867669235779976146$   
6120102182967212423625625618429357  
0693524573389783059712356395870505  
8989075147599290026879543541

- Rivest stimava che per fattorizzare  $N$  ci sarebbero voluti 40 quadrillioni di anni (1 quadrillione  $=10^{15}$ )
- il testo è stato decrittato nel 1994, da un team coordinato da Derek Atkins, Michael Graff, Arjen Lenstra, Paul Leyland usando il calcolo distribuito – il calcolo è durato sei mesi, ha coinvolto 1600 macchine
- il testo in chiaro era

*the magic words are squeamish ossifrage*

- questo successo è dovuto essenzialmente al miglioramento degli algoritmi di fattorizzazione

## algoritmi di fattorizzazione

$n$  = lunghezza dell'intero  $N$

- **divisioni successive** bisogna fare  $< \sqrt{N}$  divisioni - è in  $\mathcal{O}(2^{n/2})$ ; se  $N = p \cdot q$  e  $p$  e  $q$  sono vicini, bisogna farle quasi tutte
- **fattorizzazione alla Fermat** idea - se ho  $x$  e  $y \in \mathbb{N}$  tali che  $N + y^2 = x^2$  - ho la fattorizzazione  $N = (x + y)(x - y)$ . L'algoritmo controlla per piccoli valori di  $y$  se  $N + y^2$  è un quadrato perfetto.
- Esempio: sia  $N = 5609$ :
  - $y = 1$   $N + y^2 = 5610$  non è un quadrato
  - $y = 2$   $N + y^2 = 5613$  non è un quadrato
  - $y = 3$   $N + y^2 = 5618$  non è un quadrato
  - $y = 4$   $N + y^2 = 5625 = 75^2$
 quindi  $5609 = 75^2 - 4^2 = (75 + 4)(75 - 4) = 79 \cdot 71$   
 funziona meglio delle divisioni successive se  $N$  è il prodotto di due primi della stessa grandezza

# algoritmi di fattorizzazione

$n$  = lunghezza dell'intero  $N$

- **algoritmo di Lehman** combina i due metodi precedenti – è in  $\mathcal{O}(2^{n/3})$
- **$p - 1$  di Pollard** è in  $\mathcal{O}(2^{n/4})$
- **$p - 1$  di Pollard** funziona bene se  $N$  ha un divisore primo  $p$  tale che  $p - 1$  ha tutti i fattori primi piccoli
- **crivello quadratico di Pomerance** – è il più rapido - e il più usato - sotto le 100 cifre decimali
- **general number field sieve** (crivello generale) il più rapido in assoluto – molto complesso
- poi c'è l'**algoritmo di Shor** (1994) – in un computer quantistico, la fattorizzazione ha complessità polinomiale

## scelta dei primi

per evitare che la fattorizzazione abbia successo, si scelgono **primi forti**

- $p$  e  $q$  devono avere all'incirca la stessa lunghezza
- $p - 1$  e  $q - 1$  devono avere un fattore primo grande **altrimenti si può usare l'algoritmo  $p - 1$  di Pollard**
- la **lunghezza minima raccomandata** per  $N$  è **1024** bit (ca 300 cifre decimali) – RSA-1024 – comincia a sembrare un po' poco
- con un  $N$  a **2048** bit (ca 600 cifre decimali) si sta tranquilli (?) fino al 2030
- sempre più spesso si arriva a **4096** bit

## scelta di $e$

- si cerca di scegliere  $e$  non troppo grande e tale che nella scrittura binaria di  $e$  ci siano pochi 1
- $e$  piccolo = cifratura più veloce
  - nell'alg. square and multiply bisogna calcolare pochi quadrati
- meno 1 = cifratura più veloce
  - il numero di 1 è collegato al numero di moltiplicazioni nell'alg. square and multiply
- valori usati sono  $e = 3$  (inizialmente, ma dà problemi),  
 $e = 2^{16} + 1 = 65537$
- per il valore  $2^{16} + 1$  sono necessarie 17 moltiplicazioni - per un  $b < \phi(N)$  casuale, la stima è di circa 1000 moltiplicazioni
- non vuol dire che anche  $d$  sarà piccolo

## low exponent attack

- $d$  non deve essere piccolo
- se  $3d < N^{1/4}$  e  $q < p < 2q$  c'è un attacco dovuto a Wiener
- permette di ricavare rapidamente  $d$
- se  $N$  ha 1024 bit,  $d$  deve avere almeno 256 bit
- la decifratura è computazionalmente pesante - questo è un problema per esempio per le smartcard
- si può usare il teorema cinese del resto per velocizzare la decifratura

## RSA – side channel attack

- sono attacchi all'implementazione dell'algoritmo, non all'algoritmo
- uno è il **timing attack**
- analogia: uno scassinatore che cerca di capire la combinazione di una cassaforte dal tempo impiegato a “girare le rotelle”
- per l’RSA, si cerca di capire qual è la scrittura binaria dell’esponente di decifratura  $d$  osservando il tempo impiegato a decifrare
- nell’algoritmo square and multiply, si fa una moltiplicazione solo quando nella scrittura binaria di  $d$  c’è un 1
- posso risalire al numero di 1 in  $d$
- un altro è il **power monitoring attack**

## come utilizzare un PKCS

- abbiamo visto che affinché un CS sia sicuro, la cifratura deve essere **randomizzata**
- in un cifrario simmetrico (per esempio AES) si sceglia un’opportuna **modalità di funzionamento** (per esempio CBC)
- e in un PKCS?
- sia  $f : A \rightarrow B$  una funzione trapdoor one-way (come quella RSA)
- $f$  si utilizza per generare la chiave

## come utilizzare un PKCS

- per cifrare si usa  $f : A \rightarrow B$  funzione trapdoor one-way
- insieme a un CS simmetrico in modalità sicura (CBC-AES)
- Alice sceglie in modo casuale  $a \in A$  e calcola  $f(a) = b$  (facile - pubblico)
- Alice cifra il messaggio  $x$  con la chiave  $a$  e ottiene il CT  $y$  (usando AES:  $y = e_a(x)$ )
- trasmette a Bob la coppia  $(b, y)$

## come utilizzare un PKCS

- Bob riceve la coppia  $(b, y)$
- per decifrare deve conoscere  $a = f^{-1}(b)$
- è il solo a possedere l'informazione segreta che gli permette di invertire la  $f$
- può ottenere  $a$  e decifrare con AES:  $x = d_a(y)$
- la randomizzazione "viene" dall'AES
- per maggiore sicurezza, in genere la chiave non è  $a$  ma è  $h(a)$  dove  $h$  è una funzione hash

## problemi della chiave pubblica

- la crittografia a PK è lenta - molto più lenta della crittografia simmetrica
- in genere viene usata all'inizio di una sessione fra due utenti per stabilire una chiave da usare con un algoritmo simmetrico (*key exchange*)
- e per gli schemi di *firma digitale*

## problemi di sicurezza

- bisogna avere un modo per verificare la *corrispondenza fra utenti e chiavi*
- Alice scrive a Bob usando quella che crede essere la chiave pubblica di Bob
- se invece fosse quella di Eve?
- inoltre, in un CS simmetrico solo Alice e Bob conoscono la chiave
- se Bob riceve un messaggio di Alice e la decifrazione del messaggio ha senso, il messaggio *proviene certamente* da Alice
- in un PKCS, chiunque può scrivere un messaggio cifrato a Bob affermando di essere Alice
- serve una firma digitale