Number Theory Part VI

Factoring game, RSA

Factoring Assumption

- Factoring Experiment, $\text{Fact}_{A,\text{GenModulus}}(n)$:

GenModulus (1^n) —> (N,p,q), such that N=pq; p,q are n-bit primes

Poly-time algorithm A: A(N) —> p′,q′

If {p,q} = {p′,q′}, output 1, else 0

- Factoring is hard w.r.t. GenModulus, if, for all PPT algorithms A, there exists a negligible function, negl, s.t.:

$$\Pr[\text{Fact}_{A,\text{GenModulus}}(n) = 1] \leq \text{negl}(n)$$

Factoring Assumption

- Factoring is hard, but *does not* yield a practical cryptosystem
  - Have to somehow relate hardness of guessing SK to assumption hardness, etc…

- Led people to explore existence of other hard problems *related* to factoring

- Most famous: RSA cryptosystem by Rivest, Shamir, Adleman, 1978

RSA Assumption

- Given:
  - $N = pq$
  - Euler's totient function, $\phi(N) = (p-1)(q-1)$,
  - some $e \in Z+$, $e > 2$, s.t., $\gcd(e, \phi(N)) = 1$
  - Ciphertext, $C = y^e \bmod N$, for some $y \in Z_N^*$
- Finding $y^{1/e} \bmod N$, without knowing p,q, $\phi(N)$ is *hard*

- If RSA is hard, factoring is hard: proven
- Other way: open question…

RSA Experiment

GenRSA (1^n) —> (N,e,d)

  GenModulus(1^n) —>(N,p,q)

  compute $\phi(N) = (p-1)(q-1)$

  pick $e > 2$, s.t., $\gcd(e, \phi(N)) = 1$

  /* Find mult. inverse of e mod $\phi(N)$ */

  compute $d \equiv e^{-1} \bmod \phi(N)$

  return (N,e,d)

RSA Example

- Let $p,q = 11,17$
- $N = 187$, $\phi(N) = 160$
- Pick $e = 7$ (check: $\gcd(7, 160) = 1$

- Hard part: find a d, s.t., $d \equiv e^{-1} \bmod \phi(N)$
- $d = 23$ (check: $7 \bullet 23 \bmod 160 = 1$)

- Now, encrypt something…

RSA Example

- Say, $y = 64$ (message)

- Encryption:
  - $C = y^e \bmod N = 64^7 \bmod 187 = 4$

- Decryption:
  - $y = 4^d \bmod N = 4^{23} \bmod 187 = 64$
  - Without knowing d, p, q, or $\phi(N)$, hard to find y
  - Any of (d, p, q, $\phi(N)$) can be used to efficiently compute the others

Choice of e

- Need a prime number with low *Hamming weight*
  - Hamming weight: no. of 1's in binary rep.
- e is public exponent; y^e mod N
- Modular exponentiation is (computationally) expensive!
- Efficient algorithm: *square-and-multiply*

Modular Exponentiation

- Square-and-multiply algorithm:

- $a^b \bmod N$ =
  - $(a^{b/2})^2 \bmod N$; when b is even
  - $a \bullet (a^{(b-1)/2})^2 \bmod N$; when b is odd

- Complexity: O(log b + hamming weight (b))
- Naive method would've been O(b)
- Plug in some numbers: e.g., 4096 steps vs. 12 + hw(b) steps!

Choice of e

- Coming back to e:

- Popular choice: e = 3
  - Careful of "low-range exponent attacks"
  - Pick p,q, s.t.,(p mod 3)≠1, (q mod 3)≠1

- Another choice: $e = 65537 = 2^{16}+1$

- Don't choose small d and "reverse-engineer" e ($e \equiv d^{-1} \bmod \phi(N)$): bad idea[1]

1: if d's range is small, d vulnerable to brute-force attacks