CSE 484 (Winter 2010)

Asymmetric Cryptography

Tadayoshi Kohno

Thanks to Dan Boneh, Dieter Gollmann, John Manferdelli, John Mitchell, Vitaly Shmatikov, Bennet Yee, and many others for sample slides and materials ...

Goals for Today

- Asymmetric Cryptography

Last Time

RSA Cryptosystem

[Rivest, Shamir, Adleman 1977]

- Key generation:
  - Generate large primes p, q
    - Say, 1024 bits each (need primality testing, too)
  - Compute $n=pq$ and $\varphi(n)=(p-1)(q-1)$
  - Choose small e, relatively prime to $\varphi(n)$
    - Typically, $e=3$ or $e=2^{16}+1=65537$ (why?)
  - Compute unique d such that $ed = 1 \bmod \varphi(n)$
  - Public key = (e,n); private key = (d,n)
- Encryption of m: $c = m^e \bmod n$
  - Modular exponentiation by repeated squaring
- Decryption of c: $c^d \bmod n = (m^e)^d \bmod n = m$

On PK encryption

- Encrypted message needs to be in interpreted as an integer less than n
  - Reason: Otherwise can't decrypt.
  - Message is very often a symmetric encryption key.

Last Time

Caveats

- e =3 is a common exponent
  - If $m < n^{1/3}$, then $c = m^3 < n$ and can just take the cube root of c to recover m
    - Even problems if “pad” m in some ways [Hastad]
  - Let $c_i = m^3 \bmod n_i$ - same message is encrypted to three people
    - Adversary can compute $m^3 \bmod n_1 n_2 n_3$ (using CRT)
    - Then take ordinary cube root to recover m
- Don’t use RSA directly for privacy!

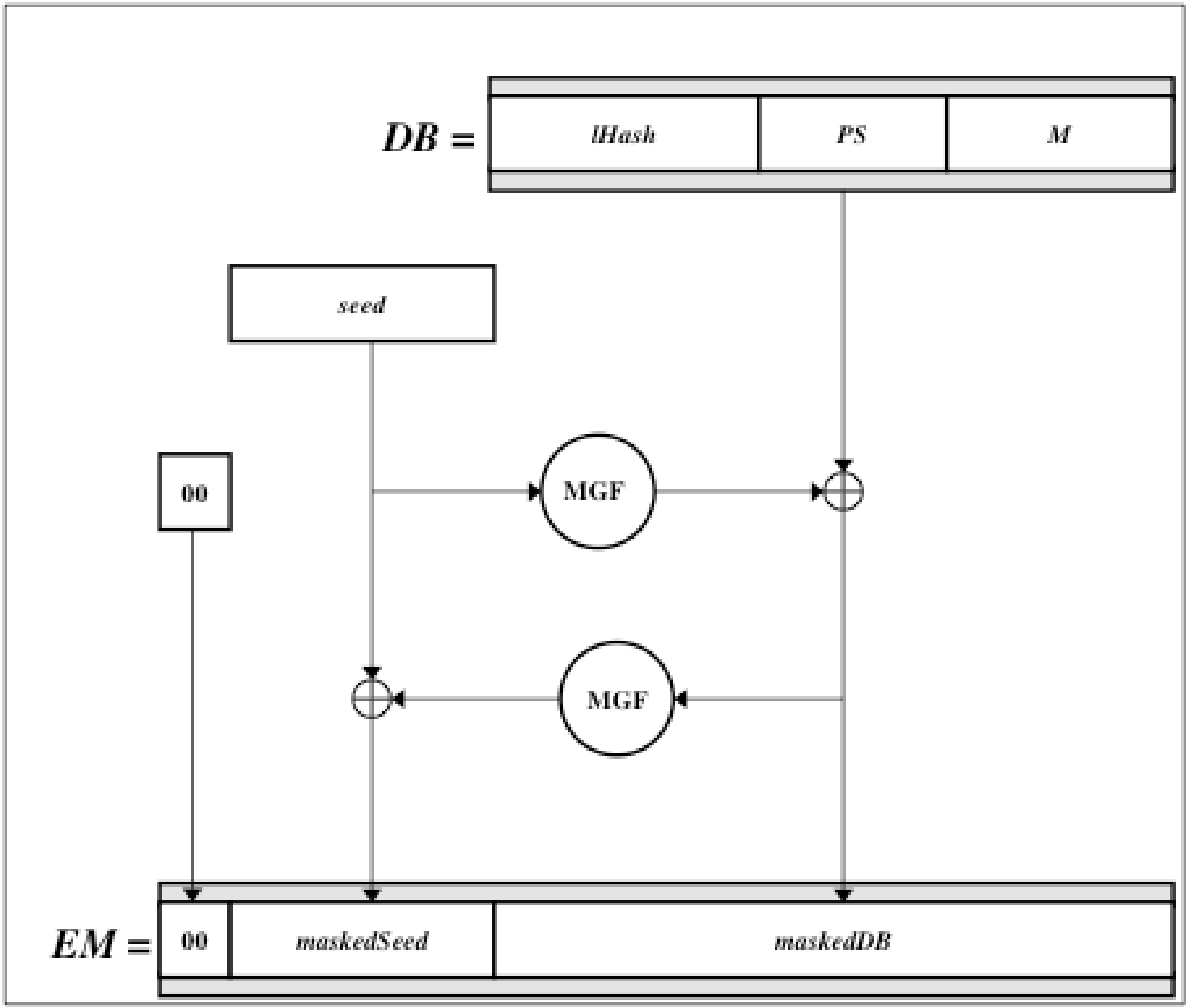
Last Time

Integrity in RSA Encryption

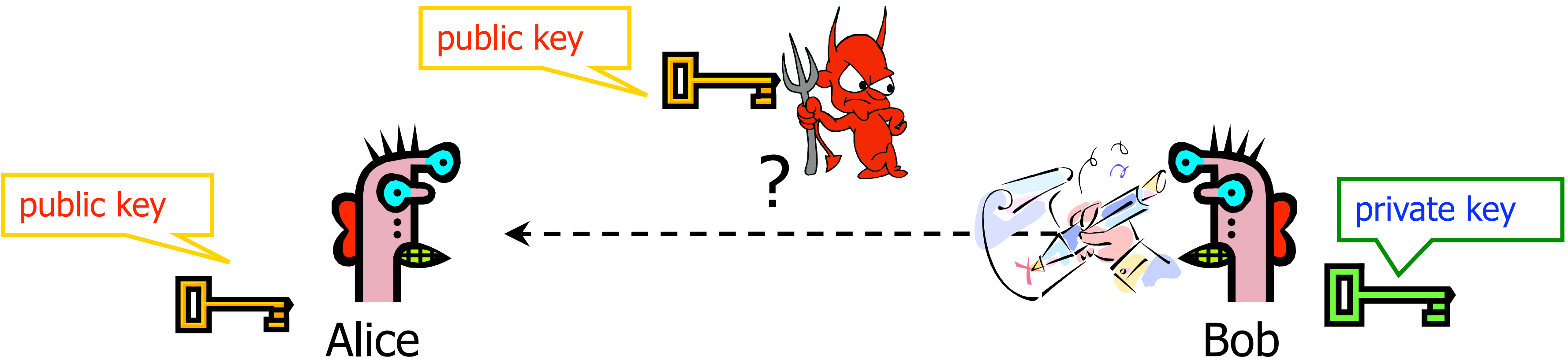
- Plain RSA does not provide integrity
  - Given encryptions of m_1 and m_2, attacker can create encryption of $m_1 \cdot m_2$
    - $(m_1^e) \cdot (m_2^e) \bmod n = (m_1 \cdot m_2)^e \bmod n$
  - Attacker can convert m into m^k without decrypting
    - $(m_1^e)^k \bmod n = (m^k)^e \bmod n$
- In practice, OAEP is used: instead of encrypting M, encrypt $M \oplus G(r) \,;\, r \oplus H(M \oplus G(r))$
  - r is random and fresh, G and H are hash functions
  - Resulting encryption is plaintext-aware: infeasible to compute a valid encryption without knowing plaintext
    - … if hash functions are "good" and RSA problem is hard

Last Time

OAEP (image from PKCS #1 v2.1)



Digital Signatures: Basic Idea



Given: Everybody knows Bob's public key

Only Bob knows the corresponding private key

Goal: Bob sends a "digitally signed" message

1. To compute a signature, must know the private key
2. To verify a signature, enough to know the public key

RSA Signatures

- Public key is (n,e), private key is d
- To sign message m: $s = m^d \bmod n$
  - Signing and decryption are the same **underlying** operation in RSA
  - It's infeasible to compute s on m if you don't know d
- To verify signature s on message m:

  $s^e \bmod n = (m^d)^e \bmod n = m$
  - Just like encryption
  - Anyone who knows n and e (public key) can verify signatures produced with d (private key)
- In practice, also need padding & hashing
  - Standard padding/hashing schemes exist for RSA signatures

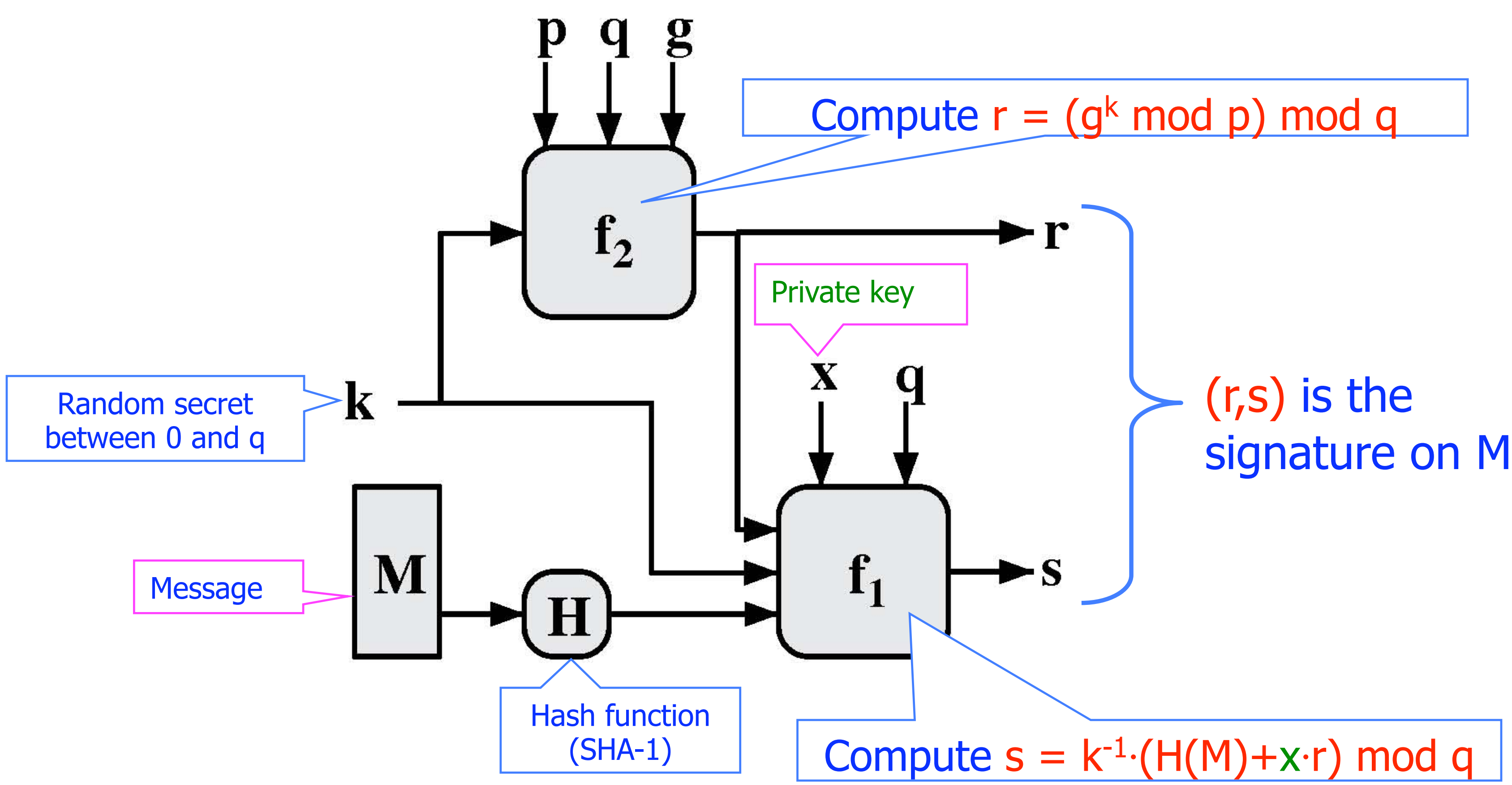
Encryption and Signatures

- Often people think: Encryption and decryption are inverses.
- That's a common view
  - True for the RSA **primitive (underlying component)**
- But not one we'll take
  - To really use RSA, we need padding
  - And there are many other decryption methods

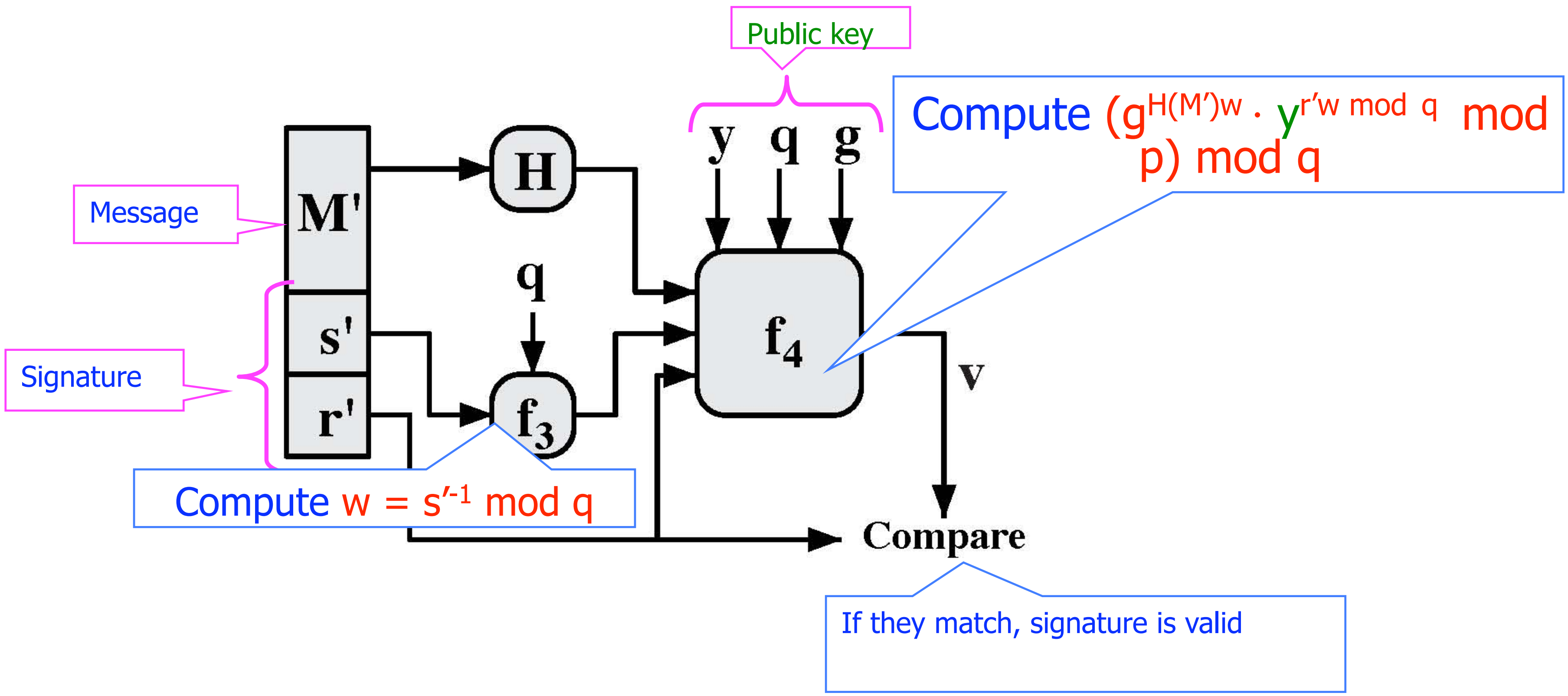
Digital Signature Standard (DSS)

- U.S. government standard (1991-94)
  - Modification of the ElGamal signature scheme (1985)
- Key generation:
  - Generate large primes p, q such that q divides p-1
    - $2^{159} < q < 2^{160}$, $2^{511+64t} < p < 2^{512+64t}$ where $0 \leq t \leq 8$
  - Select $h \in Z_p^*$ and compute $g = h^{(p-1)/q} \bmod p$
  - Select random x such $1 \leq x \leq q-1$, compute $y = g^x \bmod p$
- Public key: $(p, q, g, y = g^x \bmod p)$, private key: x
- Security of DSS requires hardness of discrete log
  - If could solve discrete logarithm problem, would extract x (private key) from $g^x \bmod p$ (public key)

DSS: Signing a Message (Skim)



DSS: Verifying a Signature (Skim)



Why DSS Verification Works (Skim)

- If (r,s) is a legitimate signature, then

  $r = (g^k \text{ mod } p) \text{ mod } q$; $s = k^{-1} \cdot (H(M)+x \cdot r) \text{ mod } q$

- Thus $H(M) = -x \cdot r + k \cdot s \text{ mod } q$
  - Multiply both sides by $w = s^{-1} \text{ mod } q$
- $H(M) \cdot w + x \cdot r \cdot w = k \text{ mod } q$
  - Exponentiate g to both sides
- $(g^{H(M) \cdot w + x \cdot r \cdot w} = g^k) \text{ mod } p \text{ mod } q$
  - In a valid signature, $g^k \text{ mod } p \text{ mod } q = r$, $g^x \text{ mod } p = y$
- Verify $g^{H(M) \cdot w} \cdot y^{r \cdot w} = r \text{ mod } p \text{ mod } q$

Security of DSS

- Can’t create a valid signature without private key
- Given a signature, hard to recover private key
- Can’t change or tamper with signed message
- If the same message is signed twice, signatures are different
  - Each signature is based in part on random secret k
- Secret k must be different for each signature!
  - If k is leaked or if two messages re-use the same k, attacker can recover secret key x and forge any signature from then on
  - Example problem scenario: rebooted VMs; restarted embedded machines

Advantages of Public-Key Crypto

- Confidentiality without shared secrets
  - Very useful in open environments
  - No "chicken-and-egg" key establishment problem
    - With symmetric crypto, two parties must share a secret before they can exchange secret messages
    - Caveats to come
- Authentication without shared secrets
  - Use digital signatures to prove the origin of messages
- Reduce protection of information to protection of authenticity of public keys
  - No need to keep public keys secret, but must be sure that Alice's public key is really her true public key

Disadvantages of Public-Key Crypto

- Calculations are 2-3 orders of magnitude slower
  - Modular exponentiation is an expensive computation
  - Typical usage: use public-key cryptography to establish a shared secret, then switch to symmetric crypto
    - E.g., IPsec, SSL, SSH, ...
- Keys are longer
  - 1024+ bits (RSA) rather than 128 bits (AES)
- Relies on unproven number-theoretic assumptions
  - What if factoring is easy?
    - Factoring is believed to be neither P, nor NP-complete
  - (Of course, symmetric crypto also rests on unproven assumptions)

Exponentiation

- How to compute $M^x \bmod N$?
- Say, x = 13
- Sums of power of 2, $x = 8+4+1 = 2^3+2^2+2^0$
- Can also write x in binary, e.g., x = 1101
- Can solve by repeated squaring
  - y = 1;
  - $y = y^2 * M \bmod N$ // y = M
  - $y = y^2 * M \bmod N$ // $y = M^2 * M = M^{2+1} = M^3$
  - $y = y^2 \bmod N$ // $y = (M^3)^2 = M^6$
  - $y = y^2 * M \bmod N$ // $y = (M^6)^2 * M = M^{12+1} = M^{13} = M^x$

Timing attacks

Collect timings for exponentiation with a bunch of messages M1, M2, ... (e.g., RSA signing operations with a private exponent)

Assume (inductively) know $b_3=1$, $b_2=1$, guess $b_1=1$

i	$b_i = 0$	$b_i = 1$	Comp	Meas
3	$y = y^2 \bmod N$	$y = y^2 * M1 \bmod N$		
2	$y = y^2 \bmod N$	$y = y^2 * M1 \bmod N$		
1	$y = y^2 \bmod N$	$y = y^2 * M1 \bmod N$	X1 secs	
0	$y = y^2 \bmod N$	$y = y^2 * M1 \bmod N$		Y1 secs

i	$b_i = 0$	$b_i = 1$	Comp	Meas
3	$y = y^2 \bmod N$	$y = y^2 * M2 \bmod N$		
2	$y = y^2 \bmod N$	$y = y^2 * M2 \bmod N$		
1	$y = y^2 \bmod N$	$y = y^2 * M2 \bmod N$	X2 secs	
0	$y = y^2 \bmod N$	$y = y^2 * M2 \bmod N$		Y2 secs

Timing attacks

- If b_1 = 1, then set of { Yj - Xj | j in {1,2, ..} } has distribution with "small" variance (due to time for final step, i=0)
  - "Guess" was correct when we computed X1, X2, ...
- If b_1 = 0, then set of { Yj - Xj | j in {1,2, ..} } has distribution with "large" variance (due to time for final step, i=0, and incorrect guess for b_1)
  - "Guess" was incorrect when we computed X1, X2, ...
  - So time computation wrong (Xj computed as large, but really small, ...)
- Strategy: Force user to sign large number of messages M1, M2, ... Record timings for signing.
- Iteratively learn bits of key by using above property.