RSA

CS6025 Data Encoding

Yizong Cheng

3-24-15

Residues Relatively Prime to n in Z_n

- If a and b are relatively prime to n, then so is ab or ab mod n.
  - a is relatively prime to n if gcd(a, n) = 1.
- The number of elements in Z_n that are relatively prime to n is the totient number $\phi(n)$.
- If p is prime, then $\phi(p) = p-1$.
- If p and q are prime, then $\phi(pq) = pq – 1 – (p – 1) – (q – 1) = (p - 1)(q - 1)$
  - p-1 multiples of q and q-1 multiples of p
- $\phi(6) = 2$, $\phi(10) = 4$, $\phi(15) = 8$, $\phi(35) = 24$.

Z_8 Multiplication and the Group of the Relatively Prime

x 1 2 3 4 5 6 7 mod 8,	$\phi(8) = 4$	Power Table
1 1 2 3 4 5 6 7	1 3 5 7	1 1 1 1
2 2 4 6 0 2 4 6		
3 3 6 1 4 7 2 5	3 1 7 5	3 1 3 1
4 4 0 4 0 4 0 4		
5 5 2 7 4 1 6 3	5 7 1 3	5 1 5 1
6 6 4 2 0 6 4 2		
7 7 6 5 4 3 2 1	7 5 3 1	7 1 7 1

Euler's Theorem

- {1,3,5,7} form a multiplication group mod 8.
- $\phi(8) = 4$.
  - $(1)(3)(5)(7) = (\alpha*1)(\alpha*3)(\alpha*5)(\alpha*7) = \alpha^{\phi(8)}(1)(3)(5)(7)$
- $\alpha^{\phi(8)} = 1$ if α is relatively prime to 8.
- In general, $\alpha^{\phi(n)} = 1 \bmod n$ if α is relatively prime to n.
- When n is a prime, $\phi(n) = n - 1$, and Euler's Theorem becomes Fermat's Theorem.

RSA

- Choose private primes p and q and compute $n = pq$ and $\phi(n) = (p - 1)(q - 1)$.
- Choose e, relatively prime to $\phi(n)$, and compute its multiplicative inverse $d = e^{-1} \bmod \phi(n)$.
  - Common e = 3 or 0x10001 (65537) so that exponentiation by the public is fast.
- Public key (e,n)
- Private key (d,n) or (d,p,q).

RSA Encryption

- Encryption: $m < n$, $C = m^e \bmod n$
- Decryption: $m = C^d \bmod n$
- Proof: $m^{ed} \bmod n = m^{k\phi(n)+1} \bmod n = (m^{\phi(n)} \bmod n)^k m \bmod n = (1)^k m = m$.
- Basis of the proof: Euler's theorem and the assumption that m is relatively prime to n.
- Slightly incomplete: what happens when m is not relatively prime to n?
    - Need the Chinese Remainder Theorem

The Chinese Remainder Theorem

- Suppose three receivers have public keys $(3, n_1)$, $(3, n_2)$, and $(3, n_3)$ and n_1, n_2, n_3 are relatively prime (highly likely).
- A encrypts message $m < n_i$ for the three and generates $c_i = m^3 \bmod n_i$ for $i = 1, 2, 3$.
- Anyone seeing the three ciphertexts c_i can use the Chinese Remainder Theorem (CRT) to compute $m^3 \bmod n_1n_2n_3 = m^3$ and then take cube root to recover the message m.

CRT: One-to-one Correspondence

- $n_1,\ldots,n_k$ relatively prime
- $n = \Pi_i n_i$
- A in Z_n corresponds to
- a_i = A mod n_i in Z_{ni} for i=1,…,k
- The correspondence is one-to-one (bijection) and the inverse is
- $A = \Sigma_i a_i p_i$ mod n,
- where ($m_i=n/n_i$) and $p_i = m_i(m_i^{-1}$ mod n_i)

RSA Correctness when m=cp

- Now we have $m^{ed} \bmod p = m \bmod p$
- and $m^{ed} \bmod q = m \bmod q$.
- Chinese Remainder Theorem says that the correspondence is one-to-one and thus $m^{ed} \bmod n$ is $m \bmod n$.
- The actual inverse map can be computed as follows.

Finding $(cp)^{ed}$ mod pq, c<q

- We know that when m=cp, c<q, m^{ed} =m mod q and m^{ed} = 0 mod p
- CRT formula for m^{ed} mod pq is
- $a_q M_q (M_q^{-1}$ mod q) mod n =
- (cp mod q)p(p^{-1} mod q) mod n =
- (cpp^{-1} mod q)p mod n = cp mod n
- = m mod n.

CRT for RSA Decryption

- The public key, e is often chosen as simple as possible, like 3 or 65537.
- $d=e^{-1} \bmod \phi(n)$ may be very large.
- Decryption $c^d \bmod pq$ can be accelerated using Chinese Remainder Theorem.
- Need to compute $c^d \bmod p$, $c^d \bmod q$, $p^{-1} \bmod q$, and $q^{-1} \bmod p$.

CRT for RSA Decryption

- Let d mod p-1 be r.
- $d = k(p-1) + r$
- $c^d \bmod p = (c^{p-1} \bmod p)^k c^r \bmod p$
- $= c^r \bmod p$.
- With precomputed r=d mod p-1, s=d mod q-1, pinv=p^{-1} mod q and qinv = q^{-1} mod p, the accelerated decryption will be the following.

CRT for RSA

- c is the ciphertext.
- $a_p = c^r \bmod p$, $a_q = c^s \bmod q$
- plaintext is computed as
- $a_p p(pinv) + a_q q(qinv) \bmod n$.

RSA Authentication

- RSA can be used to sign digests or certify public keys.
- Signer produces signature (certificate) $S = m^d \bmod n$ when m is the message digest (the public key of someone else).
- The digest or key is verified with signer's public key e, $m = S^e \bmod n$.
- Do not use the same key for both encryption and signature.
  - Chosen ciphertext attack: submit ciphertxt for signing

RSA Probabilistic Signature Scheme (RSA-PSS)

- Message → encoded message (EM) m
- Signing EM: $s = m^d \bmod n$

- Send Message and s.

- Verification: $m = s^e \bmod n$
- Message and m → H = H'?

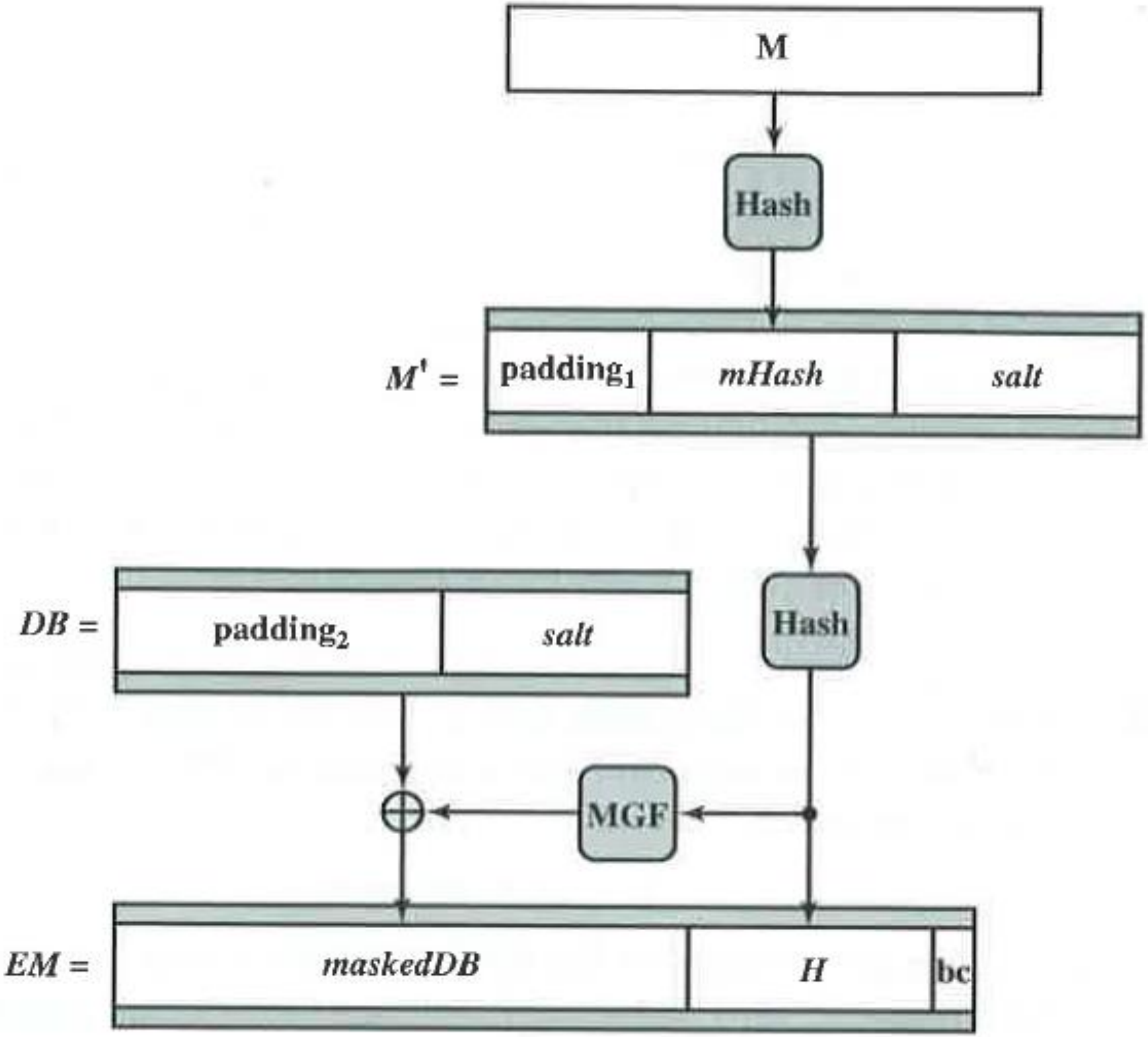

Figure 13.7 RSA-PSS Encoding

Options	Hash	hash function with output *hLen* octets. The current preferred alternative is SHA-1, which produces a 20-octet hash value.
	MGF	mask generation function. The current specification calls for MGF1.
	sLen	length in octets of the salt. Typically $sLen = hLen$, which for the current version is 20 octets.
Input	*M*	message to be encoded for signing.
	emBits	This value is one less than the length in bits of the RSA modulus *n*.
Output	*EM*	encoded message. This is the message digest that will be encrypted to form the digital signature.
Parameters	*emLen*	length of *EM* in octets $= \lceil emBits/8 \rceil$.
	padding_1	hexadecimal string 00 00 00 00 00 00 00 00; that is, a string of 64 zero bits.
	padding_2	hexadecimal string of 00 octets with a length $(emLen - sLen - hLen - 2)$ octets, followed by the hexadecimal octet with value 01.
	salt	a pseudorandom number.
	bc	the hexadecimal value BC.

The encoding process consists of the following steps.

1. Generate the hash value of M: $mHash = \text{Hash}(M)$
2. Generate a pseudorandom octet string $salt$ and form block $M' = \text{padding}_1 \| mHash \| salt$
3. Generate the hash value of M': $H = \text{Hash}(M')$
4. Form data block $DB = \text{padding}_2 \| salt$
5. Calculate the MGF value of H: $dbMask = \text{MGF}(H, emLen - hLen - 1)$
6. Calculate $maskedDB = DB \oplus dbMsk$
7. Set the leftmost $8emLen - emBits$ bits of the leftmost octet in $maskedDB$ to 0
8. $EM = maskedDB \| H \| \text{bc}$

Options	Hash	hash function with output *hLen* octets
Input	*X*	octet string to be masked
	maskLen	length in octets of the mask
Output	*mask*	an octet string of length *maskLen*

MGF1 is defined as follows:

1. **Initialize variables.**

   T = empty string

   $k = \lceil maskLen/hLen \rceil - 1$

2. **Calculate intermediate values.**

   for *counter* = 0 **to** *k*

   Represent *counter* as a 32-bit string *C*

   $T = T \,\|\, \mathrm{Hash}(X \,\|\, C)$

3. **Output results.**

   mask = the leading maskLen octets of T

Signature Verification

Options	Hash	hash function with output *hLen* octets.
	MGF	mask generation function.
	sLen	length in octets of the salt.
Input	*M*	message to be verified.
	EM	the octet string representing the decrypted signature, with length $emLen = \lceil emBits/8 \rceil$.
	emBits	This value is one less than the length in bits of the RSA modulus *n*.
Parameters	padding_1	hexadecimal string 00 00 00 00 00 00 00 00; that is, a string of 64 zero bits.
	padding_2	Hexadecimal string of 00 octets with a length $(emLen - sLen - hLen - 2)$ octets, followed by the hexadecimal octet with value 01.

1. Generate the hash value of M: $mHash = \text{Hash}(M)$
2. If $emLen < hLen + sLen + 2$, output "inconsistent" and stop
3. If the rightmost octet of EM does not have hexadecimal value BC, output "inconsistent" and stop
4. Let $maskedDB$ be the leftmost $emLen - hLen - 1$ octets of EM, and let H be the next $hLen$ octets
5. If the leftmost $8emLen - emBits$ bits of the leftmost octet in $maskedDB$ are not all equal to zero, output "inconsistent" and stop
6. Calculate $dbMask = \text{MGF}\,(H, emLen - hLen - 1)$
7. Calculate $DB = maskedDB \oplus dbMsk$
8. Set the leftmost $8emLen - emBits$ bits of the leftmost octet in DB to zero
9. If the leftmost $(emLen - hLen - sLen - 1)$ octets of DB are not equal to padding_2, output "inconsistent" and stop
10. Let $salt$ be the last $sLen$ octets of DB
11. Form block $M' = \text{padding}_1 \| mHash \| salt$
12. Generate the hash value of M': $\text{H}' = \text{Hash}(M')$
13. If $H = H'$, output "consistent." Otherwise, output "inconsistent"

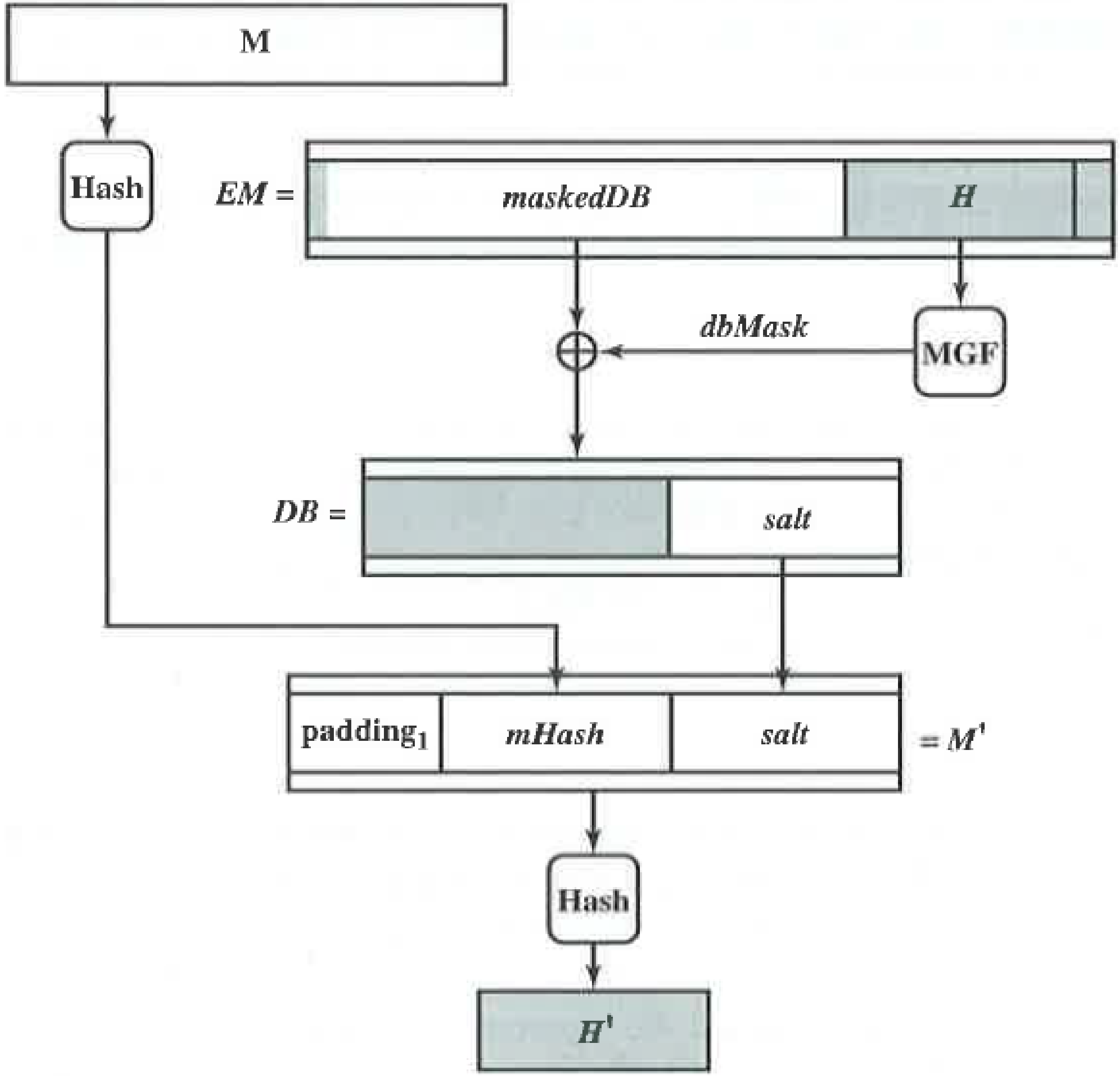

Figure 13.8 RSA-PSS EM Verification

Homework 17: due 3-30-15

- H17C.java generates an RSA key set.
  - RSAPublicKey.txt and RSAPrivateKey.txt have been generated with H17C
- H17A.java generates an RSA-PSS signature
  - java H17A RSAPrivateKey.txt < H17Message.txt > H17Signature.txt
- Complete decrypt() an steps 4-13 in H17B.java to verify an RSA-PSS signature.
  - java H17B RSAPublicKey.txt H17Signature.txt < H17Message.txt
- Submit your H17B.java and the verification of H17Signature.txt.