Public-Key Encryption and Digital Signatures

PRNGs: Summary

- True randomness requires sampling a physical process
  - Slow, expensive, and biased (low entropy)
- PRNG: An algorithm that uses a little bit of true randomness to generate a lot of random-looking output
  - Seed(entropy): Initialize internal state
  - Reseed(entropy): Add additional entropy to the internal state
  - Generate(n): Generate n bits of pseudorandom output
  - Security: Computationally indistinguishable from truly random bits
- HMAC-DRBG: Use repeated applications of HMAC to generate pseudorandom bits
- Application: UUIDs

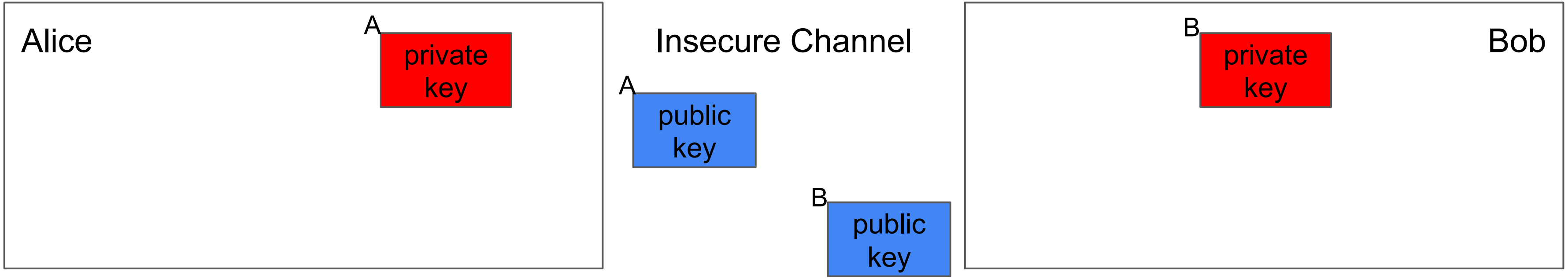
Summary: Diffie-Hellman Key Exchange

- Algorithm:
  - Alice chooses a and sends g^a mod p to Bob
  - Bob chooses b and sends g^b mod p to Alice
  - Their shared secret is $(g^a)^b = (g^b)^a = g^{ab}$ mod p
- Diffie-Hellman provides forwards secrecy: Nothing is saved or can be recorded that can ever recover the key
- Issues
  - *Not* secure against MITM
  - Both parties must be online
  - Does not provide authenticity

Public-Key Cryptography (Asymmetric Key Cryptography)

Public-Key Cryptography

- A cryptography scheme that both parties in the communication use ***different*** keys
- In public-key schemes, each person has two keys
  - **Public key**: Known to everybody
  - **Private key**: Only known by that person
  - Keys come in pairs: every public key corresponds to one private key (mathematically related)

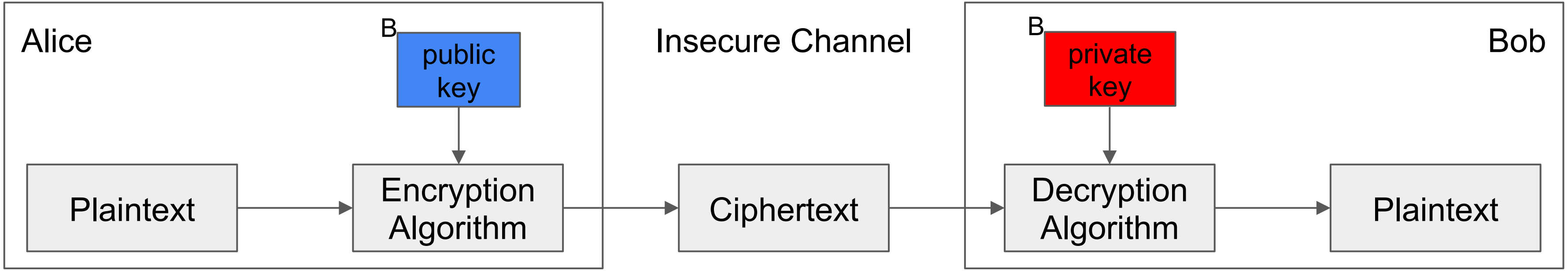


Public-Key Cryptography

- Uses number theory
  - Examples: Modular arithmetic, factoring, discrete logarithm problem
  - Contrast with symmetric-key cryptography (uses XORs and bit-shifts)
- Messages are numbers
  - Contrast with symmetric-key cryptography (messages are bit strings)
- Benefit: No longer need to assume that Alice and Bob already share a secret
- Drawback: Much slower than symmetric-key cryptography
  - Number theory calculations are much slower than XORs and bit-shifts

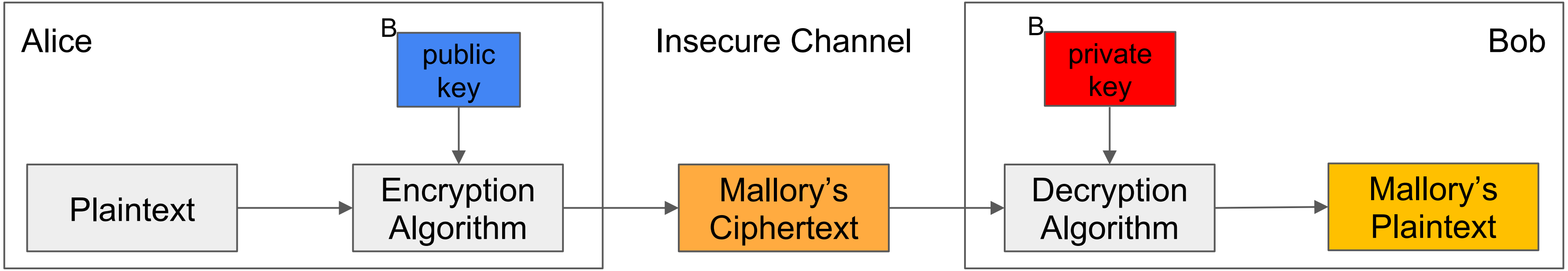
Public-Key Cryptography for Confidentiality

- Scenario
  - Alice wants to send a message to Bob
  - Alice uses Bob's public key to encrypt the message
  - Bob decrypt the message with his private key
- Who can perform the encryption? i.e., send messages to Bob
  - Anyone, because Bob's public key is public
- Who can perform the decryption? i.e., see the message for Bob
  - Only Bob, with his private key



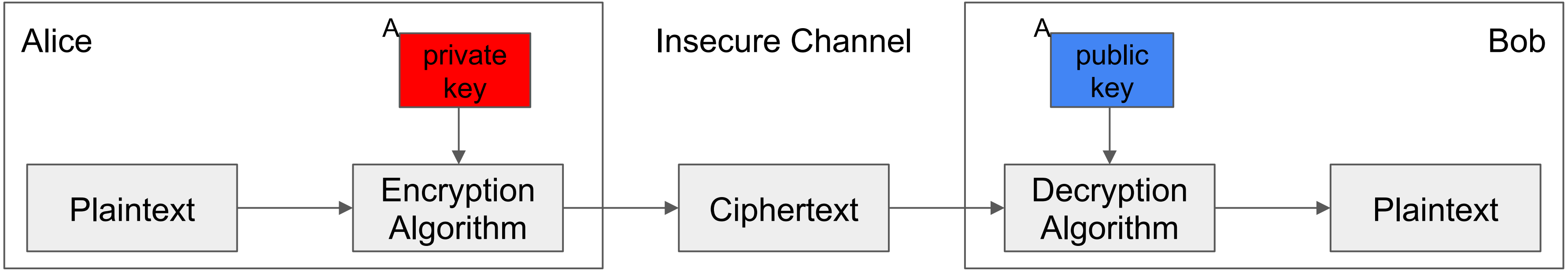
Public-Key Cryptography (MITM)

- Scenario
  - Alice wants to send a message to Bob
  - Alice uses Bob’s public key to encrypt the message
  - Mallory intercepts the message, changes it into another message encrypted with Bob’s public
  - Bob decrypts the message with his private key, cannot tell if it’s from Alice



Public-Key Cryptography for Integrity

- Scenario
  - Alice wants to send a message to Bob
  - Alice uses her private key to encrypt the message
  - Bob decrypts the message with Alice public key
- Who can perform the encryption? i.e. who can produce the message
  - Only Alice, with her private key
- Who can perform the decryption? i.e., who can verify the message
  - Anyone, because Alice's public key is public



Public-Key Cryptography

- Encryption with the public key, e.g., send message to Alice
  - C = Enc(pub-alice, M)
  - M = Dec(priv-alice, C)

- Encryption with the private key, e.g., Alice signs the message
  - C = Enc(priv-alice, M)
  - M = Dec(pub-alice, C)

- Two common encryption/decryption pairs, other pairs do not work

Public-Key Encryption

Public-Key Encryption: Definition

- Three parts:
  - KeyGen() → *PK*, *SK*: Generate a public/private keypair, where *PK* is the public key, and *SK* is the private (secret) key
  - Enc(*PK*, *M*) → *C*: Encrypt a plaintext *M* using public key *PK* to produce ciphertext *C*
  - Dec(*SK*, *C*) → *M*: Decrypt a ciphertext *C* using secret key *SK*
- Properties
  - **Correctness**: Decrypting a ciphertext should result in the message that was originally encrypted
    - Dec(*SK*, Enc(*PK*, *M*)) = *M* for all *PK*, *SK* ← KeyGen() and *M*
  - **Efficiency**: Encryption/decryption should be fast
  - **Security**: Similar to IND-CPA, but Alice (the challenger) just gives Eve (the adversary) the public key, and Eve doesn't request encryptions, except for the pair M_0, M_1
    - You don't need to worry about this game (it's called "semantic security")

RSA Encryption

Cryptography Roadmap

	Symmetric-key	Asymmetric-key
Confidentiality	• One-time pads • Block ciphers with chaining modes (e.g. AES-CBC)	• RSA encryption
Integrity, Authentication	• MACs (e.g. HMAC)	• Digital signatures (e.g. RSA signatures)

- Hash functions
- Pseudorandom number generators
- Public key exchange (e.g. Diffie-Hellman)

- Key management (certificates)
- Password management

RSA Encryption

- The first public key cryptosystem
- Invented by Rivest, Shamir, and Adleman
- Any bit size is OK
  - Bit size of the prime numbers used to create public and private keys
    - Different from symmetric key encryption
  - 512 was standard when it was released
  - 2048 or 4096 is standard now

- Based on prime numbers and factoring

RSA Encryption: Definition

- KeyGen():
  - Randomly pick two large primes, p and q
    - Done by picking random numbers and then using a test to see if the number is (probably) prime
  - Compute $N = pq$
    - N is usually between 2048 bits and 4096 bits long
  - Choose e
    - Requirement: e is not a factor of $(p - 1)(q - 1)$
    - Requirement: $2 < e < (p - 1)(q - 1)$
  - Compute $d = e^{-1} \bmod (p - 1)(q - 1)$
    - d is the modular multiplicative inverse of e
    - $1 = (d * e) \bmod (p - 1)(q - 1)$
    - Algorithm: Extended Euclid's algorithm
  - **Public key**: N and e
  - **Private key:** d

- Example
  - Randomly pick two (large) primes, p and q
    - $p = 3$, $q = 7$
  - Compute $N = pq$
    - $N = p*q = 21$
  - Choose e
    - $e = 5$ (not a factor of $2 * 6 = 12$)
  - Compute $d = e^{-1} \bmod (p - 1)(q - 1)$
    - $d = 5$ since $(d * e) \bmod (p - 1)(q - 1) = 5 * 5 \bmod (2 * 6) =1$
  - **Public key**: $N=21$ and $e=5$
  - **Private key:** $d = 5$

RSA Encryption: Definition

- Enc(e, N, M):
  - Output: $M^e \bmod N$
- Dec(d, C):
  - Output: $C^d \bmod N$
  - $C^d \bmod N = (M^e)^d \bmod N$

Example

- Enc(e, N, M):
  - Output $M^e \bmod N$
  - $M = 12, e = 5, N = 21$
  - $C = 12^5 \bmod 21 = 3$
- Dec(d, C):
  - Output $C^d \bmod N$
  - $C = 3, d = 5, N = 21$
  - $M = 3^5 \bmod 21 = 243 \bmod 21 = 12$

RSA Encryption: Correctness

1. **Theorem: M^{ed} mod $N \equiv M$ mod N**
2. Euler's theorem: $a^{\varphi(N)} \equiv 1$ mod N
   - $\varphi(N)$ is the totient function of N
   - If N is prime, $\varphi(N) = N - 1$ (Fermat's little theorem)
   - For a semi-prime pq, where p and q are prime, $\varphi(pq) = (p - 1)(q - 1)$
3. Notice: $e*d \equiv 1$ mod $(p - 1)(q - 1)$ so $ed \equiv 1$ mod $\varphi(N)$
   - This means that $ed = k\varphi(n) + 1$ for some integer k
4. (1) can be written as $M^{k\varphi(N) + 1} \equiv M$ mod N
5. $M^{k\varphi(N)}M^1 \equiv M$ mod N
6. $1M^1 \equiv M$ mod N by Euler's theorem
7. $M \equiv M$ mod N

RSA Encryption: Security

- **RSA problem**: Given N and $C = M^e \bmod N$, it is hard to find M
  - No harder than the factoring problem
  - If you can factor N, you can recover d), because $1 = (d * e) \bmod (p - 1)(q - 1)$, and $N = p*q$
- A brute-force attack is basically trying to factor the public key into two prime numbers
- Current best solution is to factor N, but unknown whether there is an easier way
  - If the RSA problem is as hard as the factoring problem, then the scheme is secure as long as the factoring problem is hard
  - Factoring problem is assumed to be hard, but we have no proof

RSA Encryption: Issues

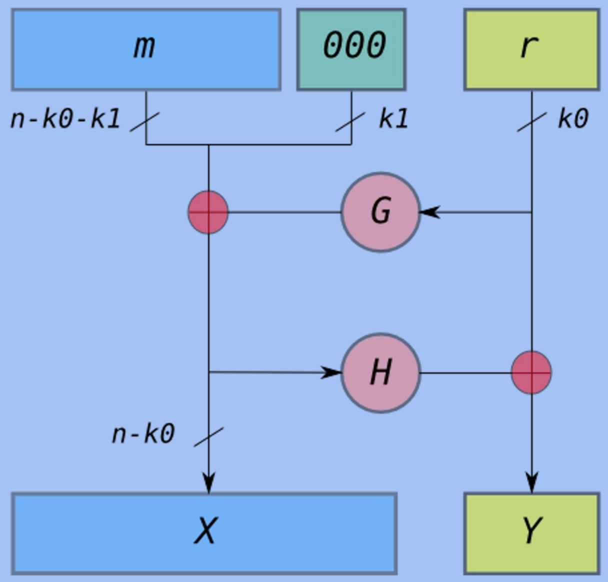
- Is RSA encryption IND-CPA secure?
  - No. It's deterministic. No randomness was used at any point!
- Sending the same message encrypted with different public keys also leaks information
  - $m^{e_a} \bmod N_a$, $m^{e_b} \bmod N_b$, $m^{e_c} \bmod N_c$
  - Small *m* and *e* leaks information
    - *e* is usually small (~16 bits) and often constant (3, 17, 65537)
- Side channel: A poor implementation leaks information
  - The time it takes to decrypt a message depends on the message and the private key
  - This attack has been successfully used to break RSA encryption in OpenSSL
- Result: We need a probabilistic padding scheme

OAEP

- **Optimal asymmetric encryption padding** (**OAEP**): A variation of RSA that introduces randomness
    - Different from “padding” used for symmetric encryption, used to add randomness instead of dummy bytes
- Idea: RSA can only encrypt “random-looking” numbers, so encrypt the message with a random key

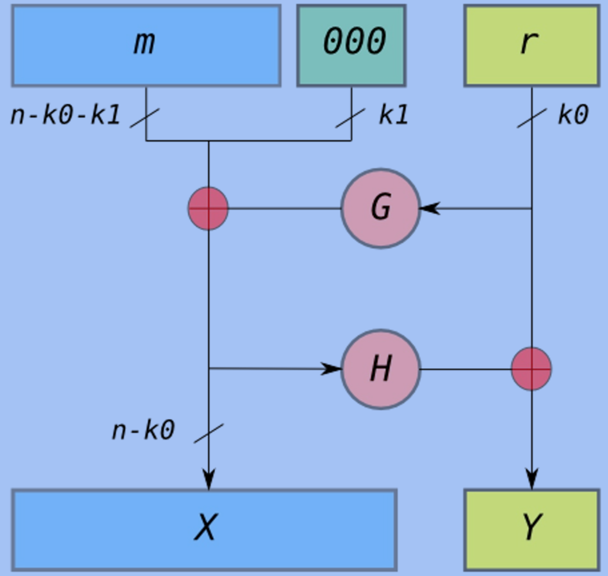
OAEP: Padding

1. k_0 and k_1 constants defined in the standard, and G and H are hash functions
   - M can only be n - k_0 - k_1 bits long
   - G produces a (n - k_0)-bit hash, and H produces a k_0-bit hash
2. Pad M with k_1 0's
   - Idea: We should see 0's here when unpadding, or else someone tampered with the message
3. Generate a random, k_1-bit string r
4. Compute $X = M \,||\, 00...0 \oplus G(r)$
5. Compute $Y = r \oplus H(X)$
6. Result: $X \,||\, Y$



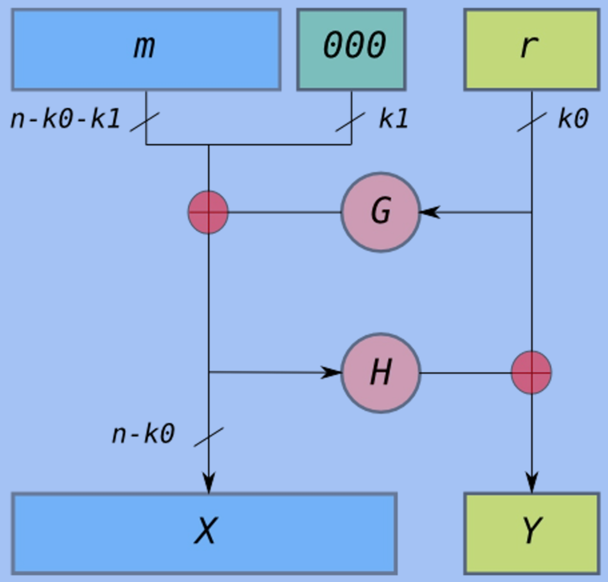
OAEP: Unpadding

1. Compute $r = Y \oplus H(X)$
2. Compute $M \,||\, 00\ldots0 = X \oplus G(r)$
3. Verify that $M \,||\, 00\ldots0$ actually ends in k_1 0's
    - Error if not



OAEP

- Even though *G* and *H* are irreversible, we can recover their inputs using XOR and work backwards
- This structure is called a **Feistel network**
    - Can be used for encryption algorithms if *G* and *H* depend on a key
        - Example: DES (out of scope)
- **Takeaway**: To fix the problems with RSA (it's only secure encrypting random numbers and isn't IND-CPA), use RSA with OAEP, abbreviated as RSA-OAEP



Hybrid Encryption

- Issues with public-key encryption
  - Notice: We can only encrypt small messages because of the modulo operator
  - Notice: There is a lot of math, and computers are slow at math
  - Result: Asymmetric doesn't work for large messages
- **Hybrid encryption**: Encrypt data under a randomly generated key *K* using symmetric encryption, and encrypt *K* using asymmetric encryption
  - Benefit: Now we can encrypt large amounts of data quickly using symmetric encryption, and we still have the security of asymmetric encryption
- Almost all cryptographic systems use hybrid encryption
  - Scenario:
    - Alice wants to send a message to Bob
    - Alice chooses / generates a random symmetric key K
    - Alice computes C1 = Enc(K, M) and sends it to Bob (Symmetric encryption)
    - Alice computes C2 = Enc(pub_bob, K) and sends it to Bob (Asymmetric encryption)
    - Bob receives both messages
      - uses his private key to decrypt C2 and get K, and then
      - use K to decrypt C1 and get M

Digital Signatures

Cryptography Roadmap

	Symmetric-key	Asymmetric-key
Confidentiality	• One-time pads • Block ciphers with chaining modes (e.g. AES-CBC)	• RSA encryption • ElGamal encryption
Integrity, Authentication	• MACs (e.g. HMAC)	• Digital signatures (e.g. RSA signatures)

- Hash functions
- Pseudorandom number generators
- Public key exchange (e.g. Diffie-Hellman)

- Key management (certificates)
- Password management

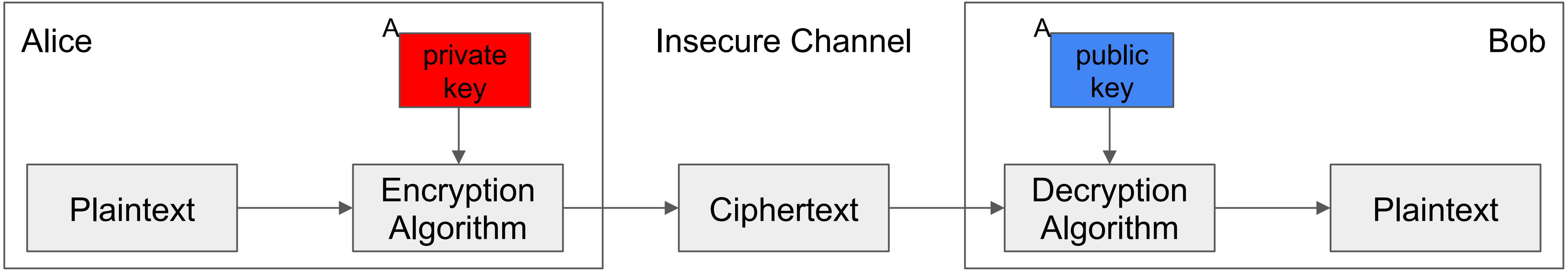
Digital Signatures

- Asymmetric cryptography is good because we don’t need to share a secret key
- Digital signatures are the asymmetric way of providing integrity/authenticity to data
- Assume that Alice and Bob can communicate public keys without Mallory interfering
    - We will see how to fix this limitation later

Public-key Signatures

ITIS 6200 / 8200

- Only the owner of the private key can sign messages with the private key
- Everybody can verify the signature with the public key



Digital Signatures: Definition

- Three parts:
  - KeyGen() → *PK*, *SK*: Generate a public/private keypair, where *PK* is the verify (public) key, and *SK* is the signing (secret) key
  - Sign(*SK*, *M*) → *sig*: Sign the message *M* using the signing key *SK* to produce the signature *sig*
  - Verify(*PK*, *M*, *sig*) → {0, 1}: Verify the signature *sig* on message *M* using the verify key *PK* and output 1 if valid and 0 if invalid
- Properties
  - **Correctness**: Verification should be successful for a signature generated over any message
    - Verify(*PK*, *M*, Sign(*SK*, *M*)) = 1 for all *PK*, *SK* ← KeyGen() and *M*
  - **Efficiency**: Signing/verifying should be fast
  - **Security**: EU-CPA, same as for MACs

Digital Signatures in Practice

- If you want to sign message *M*:
  - First hash *M*
  - Then sign H(*M*)
- Why do digital signatures use a hash?
  - Allows signing arbitrarily long messages
- Digital signatures provide integrity *and authenticity* for *M*
  - The digital signature acts as proof that the private key holder signed H(*M*), so you know that *M* is authentically endorsed by the private key holder

RSA Signatures

RSA Signatures

- Recall RSA encryption: $M^{ed} \equiv M \bmod N$
  - There is nothing special about using *e* first or using *d* first!
  - If we encrypt using *d*, then anyone can “decrypt” using *e*
    - Given x and $x^d \bmod N$, can’t recover *d* because of discrete-log problem, so *d* is safe

RSA Signatures: Definition

- KeyGen():
  - Same as RSA encryption:
    - **Public key**: N and e
    - **Private key:** d
- Sign(d, M):
  - Compute $H(M)^d \bmod N$
- Verify(e, N, M, sig)
  - Verify that $H(M) \equiv sig^e \bmod N = (H(M))^{d*e} \bmod N$

Summary: Public-Key Cryptography

- Public-key cryptography: Two keys; one undoes the other
- Public-key encryption: One key encrypts, the other decrypts
  - Security properties similar to symmetric encryption
  - RSA: Produce a pair *e* and *d* such that $M^{ed} = M \bmod N$
    - Not IND-CPA secure on its own
- Hybrid encryption: Encrypt a symmetric key, and use the symmetric key to encrypt the message
- Digital signatures: Integrity and authenticity for asymmetric schemes
  - RSA: Same as RSA encryption, but encrypt the hash with the *private* key