# Public-Key Encryption and Digital Signatures

Adapted From CS 161 Fall 2022 - Lecture 9

# 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
- CTR-DRBG: Use a block cipher in CTR mode to generate pseudorandom 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
- Diffie-Hellman can be performed over other mathematical groups, such as elliptic-curve Diffie-Hellman (ECDH)
- Issues
  - *Not* secure against MITM
  - Both parties must be online
  - Does not provide authenticity

# Public-Key Cryptography

# Public-Key Cryptography



- 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
- 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

# Modular Arithmetic



- Given an integer n > 1, called a *modulus*, two integers a and b are said to be *congruent modulo n*, if n is a divisor of their difference (that is, if there is an integer k such that a − b = kn). a和b在模n下同余
- Congruence modulo n is a congruence relation, meaning that it is an equivalence relation that is compatible with the operations of addition, subtraction, and multiplication.
- Congruence modulo n is denoted as:

  a ≡ b (mod n)

Suppose we have
a ≡ b (mod n)，p ≡ q (mod n),
and c is a positive integer,
then we have:

$$a + c \equiv b + c \pmod{n}$$
$$a - c \equiv b - c \pmod{n}$$
$$ac \equiv bc \pmod{n}$$
$$a^c \equiv b^c \pmod{n}$$
$$a + p \equiv b + q \pmod{n}$$
$$ap \equiv bq \pmod{n}$$

# Public-Key Encryption

Textbook Chapter 11

# Public-Key Encryption



- Everybody can encrypt with the public key
- Only the recipient can decrypt with the private key

# 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")

# ElGamal Encryption

Textbook Chapter 11.4

# Cryptography Roadmap



| | Symmetric-key | Asymmetric-key |
| --- | --- | --- |
| Confidentiality | • One-time pads<br>• Block ciphers with chaining modes (e.g. AES-CBC) | • RSA encryption<br>• 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

# ElGamal Encryption



- Diffie-Hellman key exchange is great: It lets Alice and Bob share a secret over an insecure channel
- Problem: Diffie-Hellman by itself can't send messages. The secret $g^{ab} \bmod p$ is random.
- Idea: Let's modify Diffie-Hellman so it supports encrypting and decrypting messages directly

# ElGamal Encryption: Protocol



- KeyGen():
  - Bob generates private key $b$ and public key $B = g^b \bmod p$ *(Note: cannot derive b from B easily)*
    - Intuition: Bob is completing his half of the Diffie-Hellman exchange
- Enc($B$, $M$):
  - Alice generates a random $r$ and computes $R = g^r \bmod p$
    - Intuition: Alice is completing her half of the Diffie-Hellman exchange
  - Alice computes M × $\underline{B^r \bmod p}$
    - Intuition: Alice derives the shared secret and multiples her message by the secret
  - Alice sends $C_1 = R$, $C_2 = M \times B^r \bmod p$
- Dec($b$, $C_1$, $C_2$)
  - Bob computes $C_2 \times C_1^{-b} = M \times B^r \times R^{-b} = M \times g^{br} \times g^{-br} = M \bmod p$
    - Intuition: Bob derives the (inverse) shared secret and multiples the ciphertext by the inverse shared secret

# ElGamal Encryption: Protocol Example



- KeyGen(): *$p$ = 37 (大素数p，且要求p-1有大素数因子), $g$ = 2 (模p的本原元)*
  - Bob generates private key $b$ and public key $B = g^b \bmod p$ *Let $b = 5$, then $B = 32$*
    - Intuition: Bob is completing his half of the Diffie-Hellman exchange
- Enc($B$, $M$):
  - Alice generates a random $r$ and computes $R = g^r \bmod p$ *Let $r = 7$, then $R = 17$*
    - Intuition: Alice is completing her half of the Diffie-Hellman exchange
  - Alice computes $M \times B^r \bmod p$ *Let $M = 29$, then $C_2 = 29*32\text{^}7 \bmod 37 = 29*(-5)\text{^}7 \bmod 37 = 33$*
    - Intuition: Alice derives the shared secret and multiples her message by the secret
  - Alice sends $C_1 = R$, $C_2 = M \times B^r \bmod p$ *$C_1 = 17$, $C_2 = 33$*
- Dec($b$, $C_1$, $C_2$)
  - Bob computes $C_2 \times C_1^{-b} = M \times B^r \times R^{-b} = M \times g^{br} \times g^{-br} = M \bmod p$
    - Intuition: Bob derives the (inverse) shared secret and multiples the ciphertext by the inverse shared secret *$C_2 \times C_1^{-b} = 33*(17\text{^}5)\text{^}(-1) \bmod 37 = 33*2 \bmod 37 = 29$*

*$C_1^{-b} = 17\text{^}(-5) \bmod 37 = (17\text{^}5)\text{^}(-1) \bmod 37 = 2$; because $17\text{^}5 \bmod 37 = 19$ and $19*2 \bmod 37 = 1$.*

# ElGamal Encryption: Security



- Recall Diffie-Hellman problem: Given $g^a$ mod $p$ and $g^b$ mod $p$, hard to recover $g^{ab}$ mod $p$
- ElGamal sends these values over the insecure channel
  - Bob's public key: $B$
  - Ciphertext: $R$, $M \times B^r$ mod $p$
- Eve can't derive $g^{br}$, so she can't recover $M$

# ElGamal Encryption: Issues



- Is ElGamal encryption IND-CPA secure?
  - No. The adversary can send $M_0 = 0$, $M_1 \neq 0$
  - Additional padding and other modifications are needed to make it semantically secure
- Malleability: The adversary can tamper with the message
  - The adversary can manipulate $C_1' = C_1$, $C_2' = 2 \times C_2 = 2 \times M \times g^{br}$ to make it look like $2 \times M$ was encrypted

# RSA Encryption

Textbook Chapter 11.3

# Cryptography Roadmap



| | Symmetric-key | Asymmetric-key |
|---|---|---|
| Confidentiality | • One-time pads<br>• Block ciphers with chaining modes (e.g. AES-CBC) | • RSA encryption<br>• 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

# 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 relatively prime to $z = (p - 1)(q - 1)$ e, z互质
    - Requirement: $2 < e < z$
  - Compute $d = e^{-1} \bmod z$ (**actually e*d mod z = 1**)
    - Algorithm: Extended Euclid's algorithm (CS 70, but out of scope)
  - **Public key**: $N$ and $e$
  - **Private key:** $d$

# RSA Encryption: Definition



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

0、给定公钥(n,e)和私钥(n,d)

1、加密报文m(m<n)时，计算 $c = m^e mod\, n$ 即 $f_{公钥}(I_{原消息}) = M_{密文}$

2、解密密文c时，计算 $m = c^d mod\, n$ 即 $f_{私钥}(M_{密文}) = I_{原消息}$

$$m = (m^e mod\, n)^d mod\, n$$

$$f_{私钥}(f_{公钥}(I_{原消息})) = I_{原消息}$$

# RSA Encryption: Correctness (out of scope)



1. Theorem: $M^{ed} \equiv M \bmod N$
2. Euler's theorem: $a^{\varphi(N)} \equiv 1 \bmod N$
   - $\varphi(N)$ 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, $\varphi(pq) = (p - 1)(q - 1)$
   - This is all out-of-scope CS 70 knowledge
3. Notice: $ed \equiv 1 \bmod (p - 1)(q - 1)$ so $ed \equiv 1 \bmod \varphi(N)$
   - This means that $ed = k\varphi(n) + 1$ for some integer $k$
4. (1) can be written as $M^{k\varphi(N) + 1} \equiv M \bmod N$
5. $M^{k\varphi(N)}M^1 \equiv M \bmod N$
6. $1M^1 \equiv M \bmod N$ by Euler's theorem
7. $M \equiv M \bmod N$

# 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 $d$)
- 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 (if you don't have a massive quantum computer, that is)

# 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 (previously padding is about adding 0s and 1s, here we mean injecting some randomness)

# OAEP

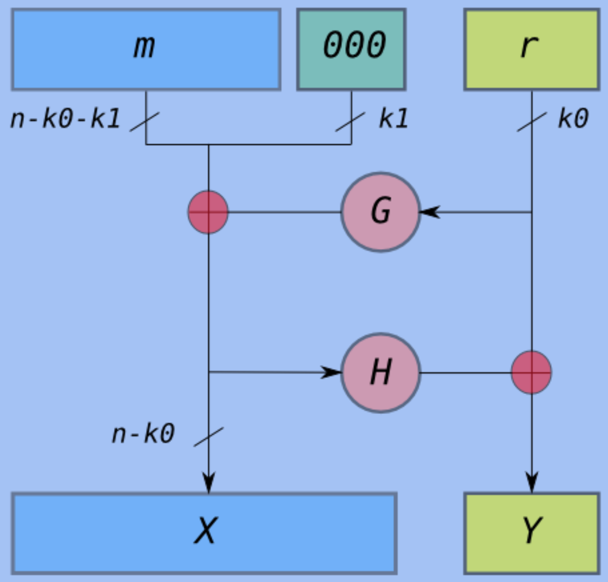


- **Optimal asymmetric encryption padding** (**OAEP**): A variation of RSA that introduces randomness
  - Different from "padding" used for symmetric encryption, used to add randomness instead of dummy bytes
- Idea: RSA can only encrypt "random-looking" numbers, so encrypt the message with a random key
- Where can you get Randomness? Hash functions!

# OAEP: Padding



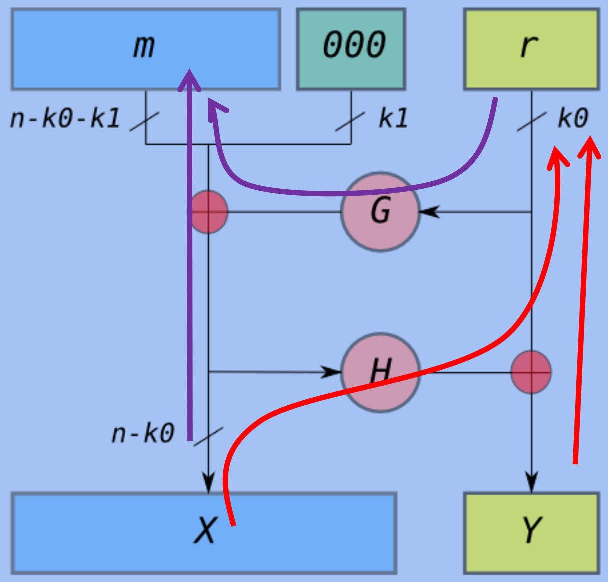
1. $k_0$ and $k_1$ constants defined in the standard, and $G$ and $H$ are hash functions
   - $M$ can only be $n$ - $k_0$ - $k_1$ bits long
   - $G$ produces a ($n$ - $k_0$)-bit hash, and $H$ produces a $k_0$-bit hash
2. Pad $M$ with $k_1$ 0's
   - Idea: We should see 0's here when unpadding, or else someone tampered with the message
3. Generate a random, $k_0$-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



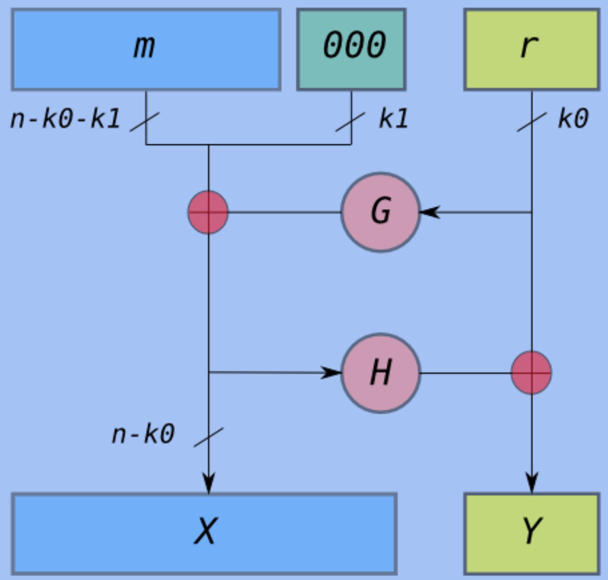
1. Compute $r = Y \oplus H(X)$
2. Compute $M \,||\, 00...0 = X \oplus G(r)$
3. Verify that $M \,||\, 00...0$ actually ends in $k_1$ 0's
   - Error if not



# 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

# Digital Signatures

Textbook Chapter 12

# Cryptography Roadmap



| | Symmetric-key | Asymmetric-key |
|---|---|---|
| Confidentiality | • One-time pads<br>• Block ciphers with chaining modes (e.g. AES-CBC) | • RSA encryption<br>• 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



- 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



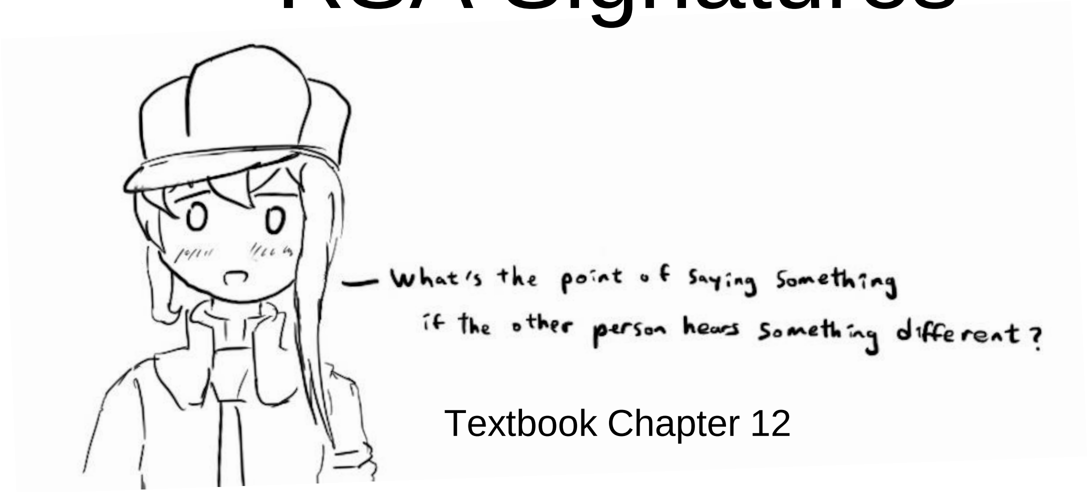
- Three parts:
  - KeyGen() → *PK*, *SK*: Generate a public/private keypair, where *PK* is the verify (public) key, and *SK* is the signing (secret) key
  - Sign(*SK*, *M*) → *sig*: Sign the message *M* using the signing key *SK* to produce the signature *sig*
  - Verify(*PK*, *M*, *sig*) → {0, 1}: Verify 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




Textbook Chapter 12

# 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 $H(M) \equiv sig^e \bmod N$

# RSA Signatures: Definition



- 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

# DSA Signatures

# DSA Signatures



- A signature scheme based on Diffie-Hellman
  - The details of the algorithm are out of scope
- Usage
  - Alice generates a public-private key pair and publishes her public key
  - To sign a message, Alice generates a random, secret value *k* and does some computation
  - Note: *k* is not Alice's private key
  - Note: *k* is sometimes called a nonce but it is not: it must be *random* and never reused
  - The signature itself does not include *k*!
- *k* must be random and secret for each message
  - An attacker who learns *k* can also learn Alice's private key
  - If Alice reuses *k* on two signatures, an attacker can learn *k* (and use *k* to learn her private key)

# DSA Signatures: Attacks



- Sony PlayStation 3 (PS3)
- Digital rights management (DRM)
  - Prevent unauthorized code (e.g. pirated software) from running
  - The PS3 was designed to only run signed code
  - Signature algorithm: Elliptic-curve DSA
- Running alternate operating systems
  - The PS3 had an option to run alternate operating systems (Linux) that was later removed
  - This was catnip to reverse engineers (“The best way to get people interested is *removing* Linux from a device”)
- One of the authentication keys used to sign the firmware reused *k* for multiple signatures → security lost!

# DSA Signatures: Attacks



- Android OS vulnerability (2013)
  - The "SecureRandom" function in its random number generator (RNG) wasn't actually secure!
  - Not only was it low entropy, it would sometimes return the same value multiple times
- Multiple Bitcoin wallet apps on Android were affected
  - Bitcoin payments are signed with elliptic-curve DSA and published publicly
  - Insecure RNG caused multiple payments to be signed with the same *k*
- Attack: Someone scanned for all Bitcoin transactions signed insecurely
  - Recall: When multiple signatures use the same *k*, the attacker can learn *k* and the private key
  - In Bitcoin, your private key unlocks access to all your money

# DSA Signatures: Attacks



- Chromebooks have a built-in U2F (universal second factor) security key
  - Uses signatures to let the user log in to particular websites
  - Signature algorithm: 256-bit elliptic-curve DSA
- There was a bug in the secure hardware!
  - Instead of using 256-bit $k$, a bug caused $k$ to be 32 bits long!
  - An attacker with a signature could simply try all possible values of $k$
- Fortunately the damage was slight
  - Each signature is only valid for logging into a single website
  - Each website used its own private key
- **Takeaway**: DSA (or ECDSA) is particularly vulnerable to incorrect implementations, compared with RSA signatures

# 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
    - ElGamal: Based on Diffie-Hellman
        - The public key is $g^b$, and $C_1$ is $g^r$.
        - Not IND-CPA secure on its own
    - 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