CS459/698 Privacy, Cryptography, Network and Data Security

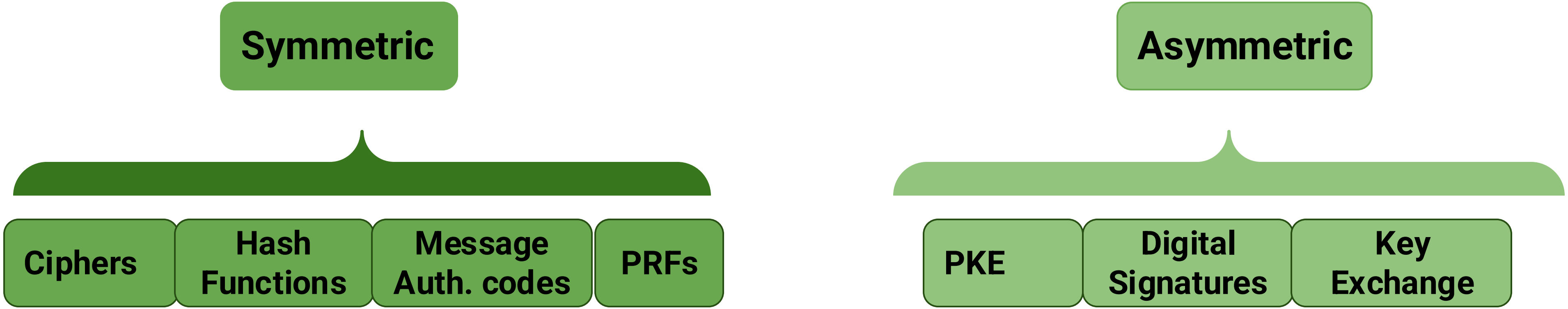
Public Key Cryptography (RSA)

Fall 2024, Tuesday/Thursday 02:30pm-03:50pm

Assignment One

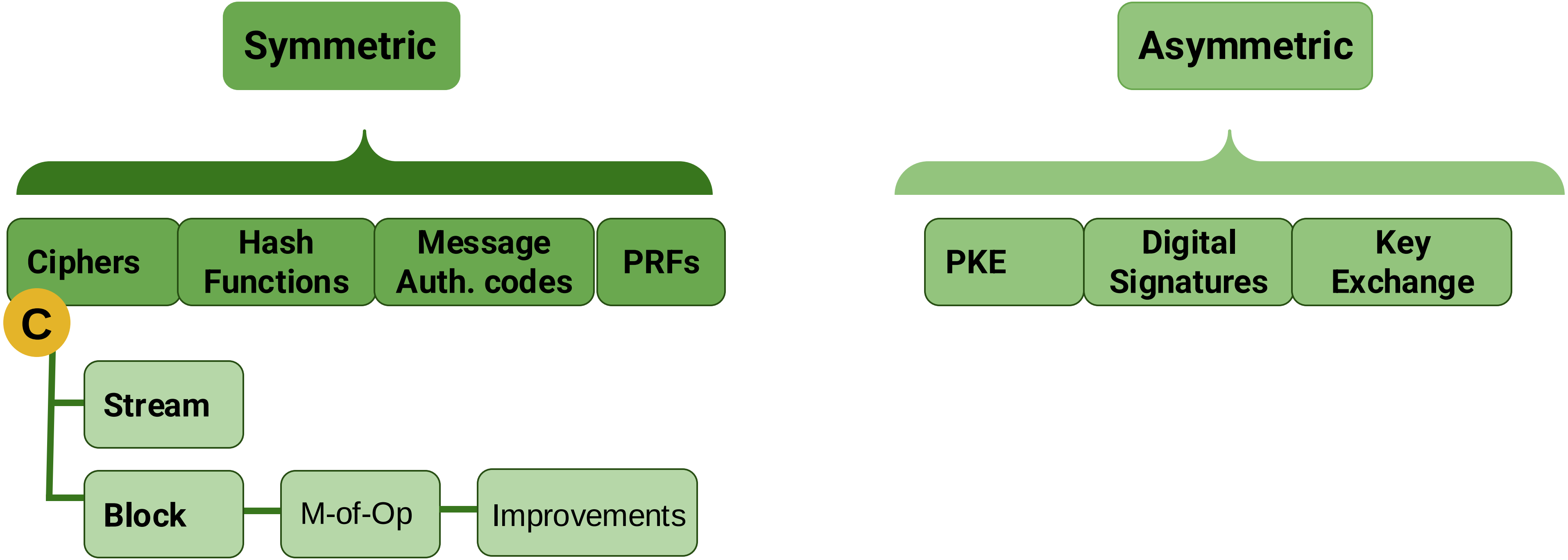
- Available on Learn today at 4pm
- Due **October 3rd, 4pm**
- Written and programming

Cryptography Organization

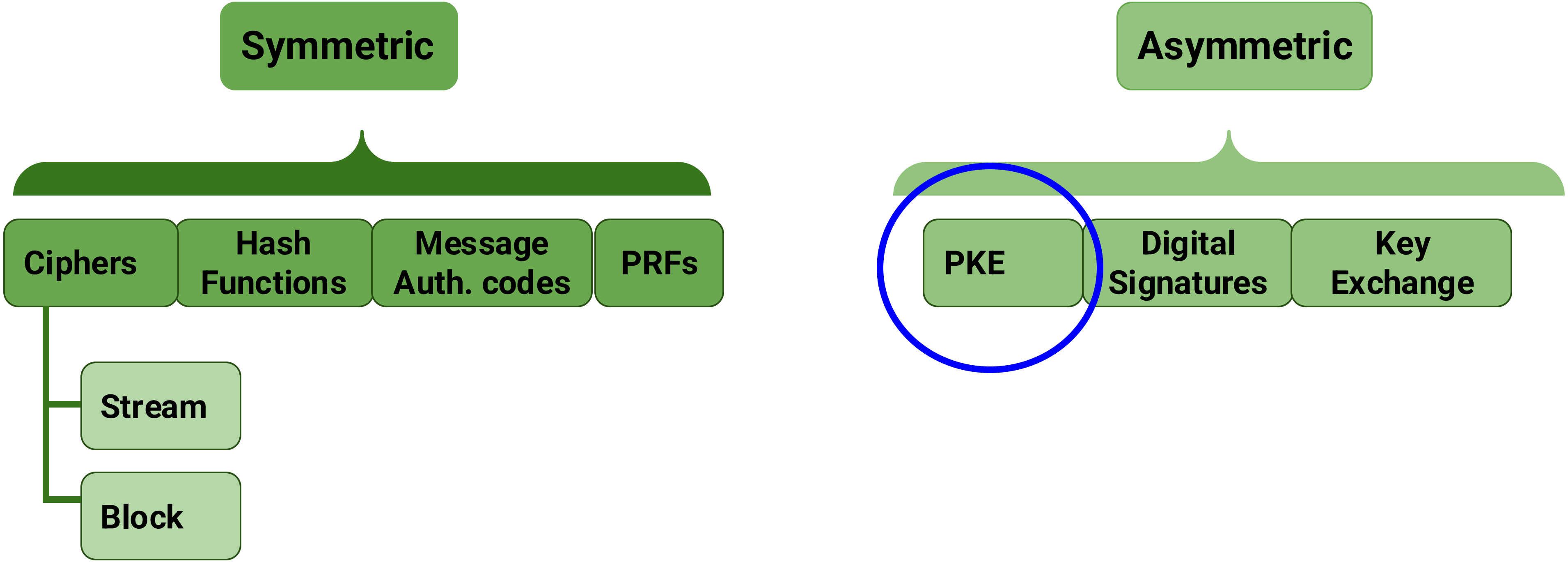


Organization display source: Doug Stebila

Cryptography Organization



Cryptography Organization



Public-key Cryptography

- Invented (in public) in the 1970's
- Also called Asymmetric Cryptography
  - Allows Alice to send a secret message to Bob without any prearranged shared secret!
  - In secret-key cryptography, the same (or a very similar) key encrypts the message and also decrypts it 
  - In public-key cryptography, there's one key for encryption, and a different key for decryption! 
- Some common examples:
  - RSA, ElGamal, ECC, NTRU, McEliece

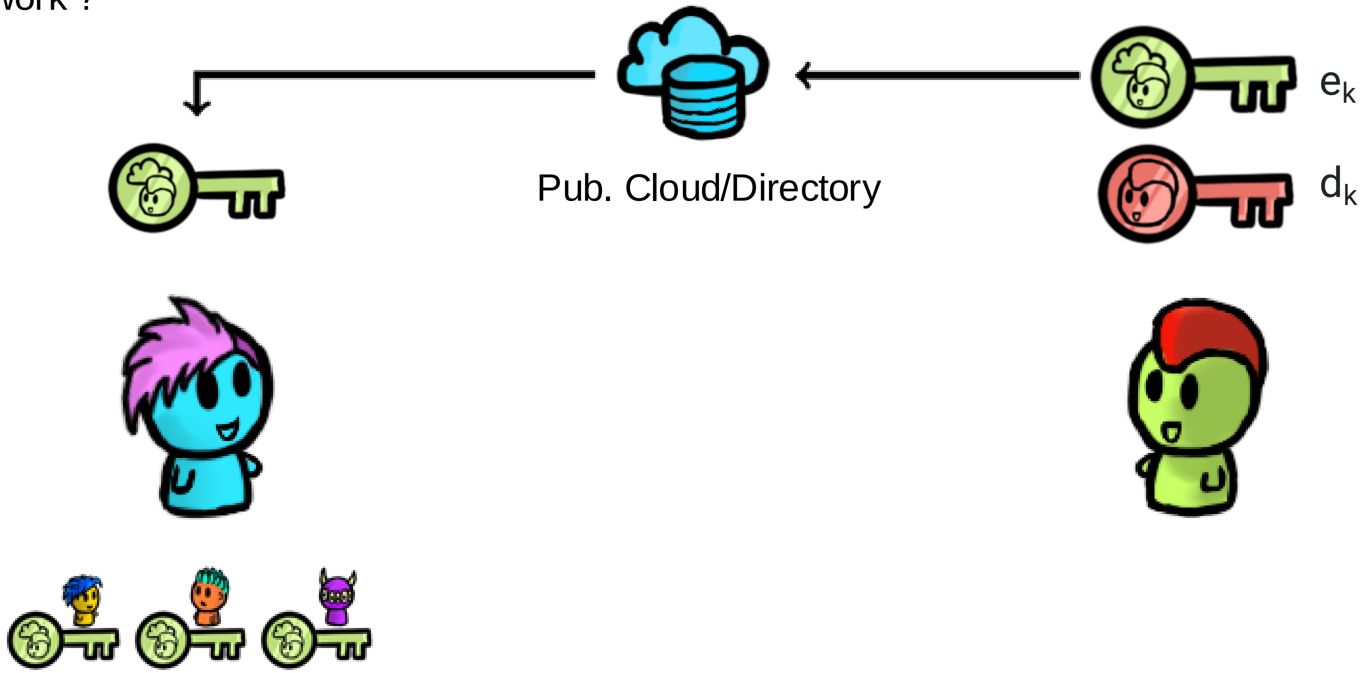
Public-key Cryptography

How does it work ?

key pair (e_k , d_k)

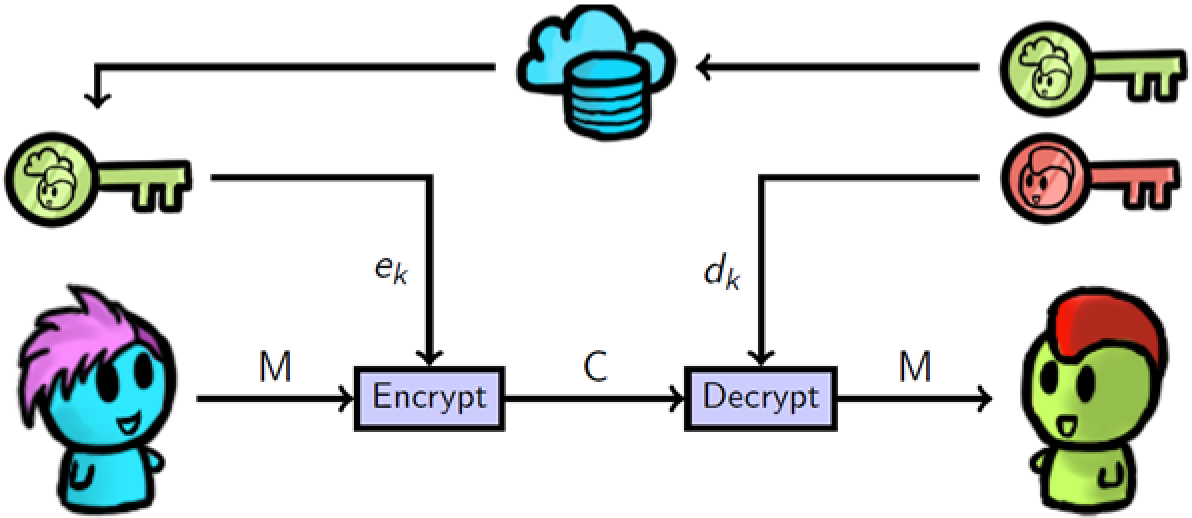
Public-key Cryptography

How does it work ?



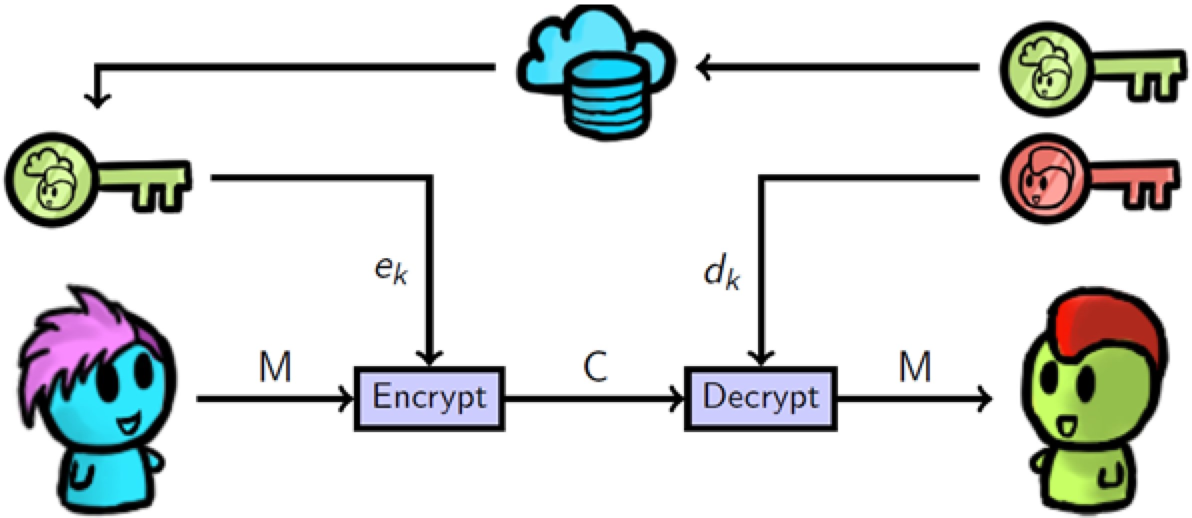
Public-key Cryptography

How does it work ?



Public-key Cryptography

How does it work ?



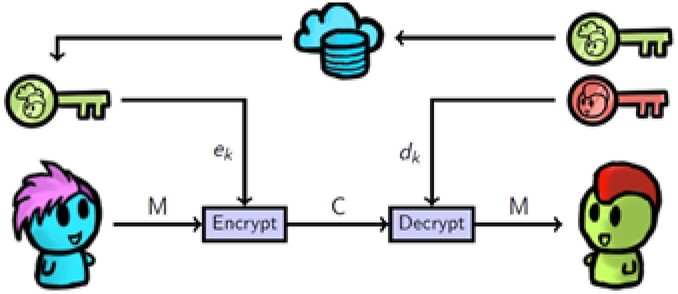
- ✓ Eve can't decrypt; she only has the encryption key e_k
- ✓ Neither can Alice!
- ✓ It must be HARD to derive d_k from e_k

Steps for PKE?

1. Bob creates a key pair

2. Bob gives everyone the public key

3. Alice encrypts m and sends it
4. Bob decrypts using private key



5. Eve and Alice can’t decrypt, only have encryption key

Requirements for PKE

- The encryption function?
  - Must be easy to compute 
- The inverse, decryption?
  - Must be hard for anyone without the key  vs. 

Thus, we require so called “one-way” functions for this.

Requirements for PKE

- The encryption function?
  - Must be easy to compute 
- The inverse, decryption?
  - Must be hard for anyone without the key  vs. 

Thus, we require so called “one-way” functions for this.

But because of decryption, we need a “Trapdoor”

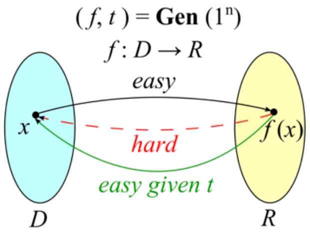
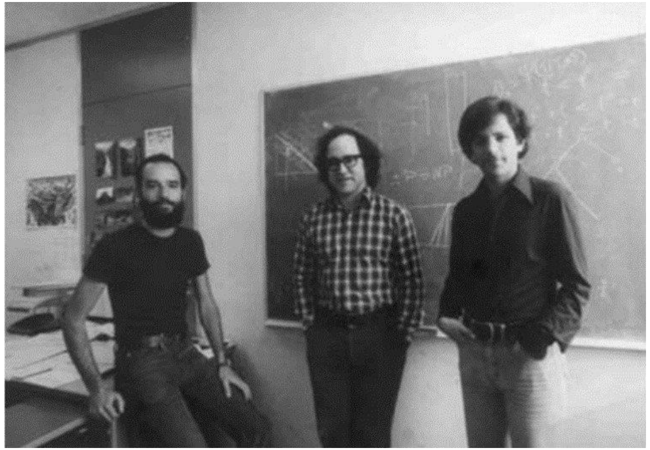


Image Credit: https://en.wikipedia.org/wiki/Trapdoor_function

Textbook RSA

- Relies on the practical difficulty of the "Factoring problem"
- Modular arithmetic: integer numbers that "wrap around"



Left to right: Ron Rivest, Adi Shamir, and Leonard Adleman.

Fun (?) Facts:

- RSA was the first popular public-key encryption method, published in 1977

Prime Numbers

- **Prime:** a natural number that can only be divided by 1 or itself

- **Primes and factorization:** An integer number can be written as a unique product of prime numbers
  - E.g., 1234567 = 127 * 9721

How to know if a number is prime?

Run a primality test algorithm (Solovay-Strassen, Miller-Rabin, etc.)

How to discover a number's factors?

Run a factorization algorithm (Pollard p-1, etc.)

Textbook RSA

- High-level idea
  - It is easy to find large integers e, d, and n (=p.q), that satisfies:

$$(m^e)^d \equiv m \pmod{n}$$

- Computational difficulty of the **factoring problem**
  - Given two large primes p.q = n, it is very hard to factor n.

Easy for me to pick **e**, **d**, and **n** that satisfy that equation

Ugh. I know **e** and **n** and (even **m**) extremely hard to find **d**!!!

Textbook RSA

- Encryption:

$$C = m^e \pmod{n}$$

The ciphertext is equal to **m** multiplied by itself **e** times modulo **n**.

Public key: $Pub_{Key} = (e, n)$

Textbook RSA

- Decryption:

$$m = C^d \pmod{n} = (m^e)^d \pmod{n} = m^{ed} \pmod{n}$$

Decryption relies on number **d** satisfying **e**.**d** = 1 (mod **n**),

$$\text{s.t. } m^{ed} \pmod{n} = m^1 \pmod{n} = m$$

  - In other words, **d** is the multiplicative inverse of **e** mod **n**

Private key: $Priv_{Key}$ = d (other numbers can be discarded)

Key Generation (how to choose **e** and find **d**)

- Pick two random primes **p** and **q**, such that **p**.**q** = **n**
- Generate φ(n) = (p-1).(q-1)
  - We know all relative primes to (p−1)(q−1) form a group with respect to multiplication and are invertible
  - φ(n) is the order of the multiplicative group of units mudulo n
- Pick **e** as a random prime smaller than φ(n)
  - **e** chosen as relative prime to (p−1)(q−1) to ensure it has a multiplicative inverse mod (p−1)(q−1)
- Generate **d** (the inverse of e mod φ(n))
  - **e**.**d** = 1 mod φ(n)
  - Can be obtained via the extended Euclidean algorithm

*If gcd(a,b) = 1, then we say that a and b are **relatively prime** (or coprime).

Extended Euclidean Algorithm

- Given two integers **a** and **b**, the algorithm finds integers **r** and **s** such that **r.a + s.b = gcd(a, b)**. When **a** and **b** are coprime, gcd(a, b) = 1, and r is the modular multiplicative inverse of **a** modulo **b**.
- **Idea:** start with the GCD and recursively work your way backwards.

Say n = 40, e = 7

e.**d** = 1 mod φ(n)

7d = 1 mod 40

Extended Euclidean Algorithm

- Given two integers **a** and **b**, the algorithm finds integers **r** and **s** such that **r.a + s.b = gcd(a, b)**. When **a** and **b** are coprime, gcd(a, b) = 1, and r is the modular multiplicative inverse of **a** modulo **b**.
- **Idea:** start with the GCD and recursively work your way backwards.

Say n = 40, e = 7

e.**d** = 1 mod φ(n)

7d = 1 mod 40

Euclidean Algorithm:

40 = 5 * **7** + 5

Extended Euclidean Algorithm

- Given two integers **a** and **b**, the algorithm finds integers **r** and **s** such that **r.a + s.b = gcd(a, b)**. When **a** and **b** are coprime, gcd(a, b) = 1, and r is the modular multiplicative inverse of **a** modulo **b**.
- **Idea:** start with the GCD and recursively work your way backwards.

Say n = 40, e = 7

e.**d** = 1 mod φ(n)

7d = 1 mod 40

Euclidean Algorithm:

40 = 5 * **7** + 5
7 = 1 * **5** + 2

Extended Euclidean Algorithm

- Given two integers **a** and **b**, the algorithm finds integers **r** and **s** such that **r.a + s.b = gcd(a, b)**. When **a** and **b** are coprime, gcd(a, b) = 1, and r is the modular multiplicative inverse of **a** modulo **b**.
- **Idea:** start with the GCD and recursively work your way backwards.

Say n = 40, e = 7

e.**d** = 1 mod φ(n)

7d = 1 mod 40

Euclidean Algorithm:

40 = 5 * **7** + 5
7 = 1 * **5** + 2
5 = 2 * **2** + 1

Stop at last non-zero remainder
gcd(7, 40) = 1

Extended Euclidean Algorithm

- Given two integers **a** and **b**, the algorithm finds integers **r** and **s** such that **r.a + s.b = gcd(a, b)**. When **a** and **b** are coprime, gcd(a, b) = 1, and r is the modular multiplicative inverse of **a** modulo **b**.
- **Idea:** start with the GCD and recursively work your way backwards.

Say n = 40, e = 7

e.**d** = 1 mod φ(n)

7d = 1 mod 40

Euclidean Algorithm:

40 = 5 * **7** + 5
7 = 1 * **5** + 2
5 = 2 * **2** + 1   1 = 5 – 2 * 2

Stop at last non-zero remainder
gcd(7, 40) = 1

Extended Euclidean (backtrack):

1 = 5 - 2 * 2

Extended Euclidean Algorithm

- Given two integers **a** and **b**, the algorithm finds integers **r** and **s** such that **r.a + s.b = gcd(a, b)**. When **a** and **b** are coprime, gcd(a, b) = 1, and r is the modular multiplicative inverse of **a** modulo **b**.
- **Idea:** start with the GCD and recursively work your way backwards.

Say n = 40, e = 7

e.**d** = 1 mod φ(n)

7d = 1 mod 40

Euclidean Algorithm:

40 = 5 * **7** + 5
7 = 1 * **5** + 2   2 = 7 – 1 * 5
5 = 2 * **2** + 1

Stop at last non-zero remainder
gcd(7, 40) = 1

Extended Euclidean (backtrack):

1 = 5 - 2 * **2**
1 = 5 - 2 (7 - 1 * 5)
1 = 5 - 2 * 7 + 2 * 5
1 = 3 * 5 - 2 * 7

Extended Euclidean Algorithm

- Given two integers **a** and **b**, the algorithm finds integers **r** and **s** such that **r.a + s.b = gcd(a, b)**. When **a** and **b** are coprime, gcd(a, b) = 1, and r is the modular multiplicative inverse of **a** modulo **b**.
- **Idea:** start with the GCD and recursively work your way backwards.

Say n = 40, e = 7

e.**d** = 1 mod φ(n)

7d = 1 mod 40

Euclidean Algorithm:

40 = 5 * **7** + 5   5 = 40 – 5 * 7
7 = 1 * **5** + 2
5 = 2 * **2** + 1

Stop at last non-zero remainder
gcd(7, 40) = 1

Extended Euclidean (backtrack):

1 = 5 - 2 * 2
1 = 5 - 2 (7 - 1 * 5)
1 = 5 - 2 * 7 + 2 * 5
1 = 3 * **5** - 2 * 7
1 = 3 (40 - 5 * 7) - 2 * 7
1 = 3 * 40 - 17 * 7

Extended Euclidean Algorithm (find **d**)

- Given two integers **a** and **b**, the algorithm finds integers **r** and **s** such that **r.a + s.b = gcd(a, b)**. When **a** and **b** are coprime, gcd(a, b) = 1, and r is the modular multiplicative inverse of **a** modulo **b**.
- **Idea:** start with the GCD and recursively work your way backwards.

Say n = 40, e = 7

e.**d** = 1 mod φ(n)

7d = 1 mod 40

Euclidean Algorithm:

40 = 5 * **7** + 5
7 = 1 * **5** + 2
5 = 2 * **2** + 1

Stop at last non-zero remainder
gcd(7, 40) = 1

Extended Euclidean (backtrack):

1 = 5 - 2 * 2
1 = 5 - 2 (7 - 1 * 5)
1 = 5 - 2 * 7 + 2 * 5
1 = 3 * 5 - 2 * 7
1 = 3 (40 - 5 * 7) - 2 * 7
1 = ~~3 * 40~~ **- 17** * **7**
d = -17 = 23 mod 40

Textbook RSA (summary)

1. Choose two **“large primes”** *p* and *q* (secretly)
2. Compute n = p*q
3. “Choose” value *e* and find *d* such that
   - $(m^e)^d \equiv m \bmod n$
4. **Public key**: (e, n)
5. **Private key**: d
6. Encryption: $C = m^e \bmod n$
7. Decryption: $m = C^d \bmod n$

Textbook RSA (summary)

1. Choose two “**large primes**” *p* and *q* (secretly)
2. Compute n = p*q
3. “Choose” value *e* and find *d* such that
    - $(m^e)^d \equiv m \bmod n$
4. **Public key**: (e, n)
5. **Private key**: d
6. Encryption: $C = m^e \bmod n$
7. Decryption: $m = C^d \bmod n$

- ✓ Note that the decryption works.
- ✓ This is textbook RSA, never do this!! (we’ll see one of the reasons next)

Example (Tiny RSA)

Parameters:

- p=53, q=101, n=5353
- $\varphi(n) = (53-1).(101-1) = 5200$
- e=139 (random pick)
- d=1459 (extended Euclidean)

- Message:

  m=<u>20</u>

Encryption: $c = m^e \bmod n$

$C = 20^{139} \bmod 5353 = 5274$

Decryption: $m = c^d \bmod N$

$m = 5274^{1459} \bmod 5353 = \underline{20}$

Example (Tiny RSA)

Parameters:

- p=53, q=101, n=5353
- $\varphi(n) = (53-1).(101-1) = 5200$
- e=139 (random pick)
- d=1459 (extended Euclidean)

- Message:

  m=<u>20</u>

Encryption: $c = m^e \bmod n$

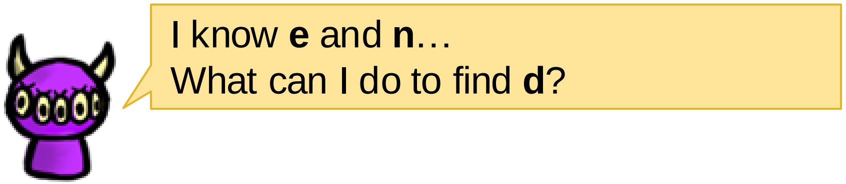
$$C = 20^{139} \bmod 5353 = 5274$$

Decryption: $m = c^d \bmod N$

$$m = 5274^{1459} \bmod 5353 = \underline{20}$$

Applying **e** or **d** to encrypt does not really matter from a functionality perspective

Attacking RSA(Bad primes)



Attack idea:

- Factor **n** to obtain **p** and **q**
- Obtain φ**(n)**
- From φ**(n)** and **e**, generate **d** just like Alice would

Parameters:

- p=53, q=101, n=5353
- $\varphi(n) = (53-1).(101-1) = 5200$
- e=139
- d=1459

- c = 5274

Attacking RSA(Bad primes)

... e a... N...
Wh... find **d**?

WARNING: Factoring is hard, But...

Attack idea:

- Factor **n** to obta... **p** and ...
- Obtain φ**(n)**
- From φ**(n)** and **e**, generate **d** just like Alice would

Parameters:

- p=53, q=101, n=5353
- $\varphi(n) = (53-1).(101-1) = 5200$
- e=139
- ...=1459
- c...5274

Factoring and RSA

You want to factor the public modulus?

- Good news, abundant literature on factoring algorithms
- Bad news, “appropriate” primes will not be defeated

Factoring and RSA

You want to factor the public modulus?

- Good news, abundant literature on factoring algorithms
- Bad news, “appropriate” primes will not be defeated

Bad primes: easily factored

Approach at Factoring

Strawman approach:

- Try to divide a number by all numbers smaller than it until you find a number **a** that divides n
- Then, carry on to divide n with **a+1** and so on…
- We end up with a list of factors of n

Way too computationally expensive.

A Smarter Approach at Factoring

- We only need to test prime numbers (not every $a < n$)

- We only need to test those smaller than $\sqrt{n}$
    - If both p and q are larger than n, then $p.q > n$, which is impossible

A Smarter Approach at Factoring

- We only need to test prime numbers (not every a < n)

- We only need to test those smaller than $\sqrt{n}$
  - If both p and q are larger than n, then p.q > n, which is impossible

Still too computationally expensive for large n.

n = 4096 bits requires about 2^{128} operations

AMD's EPYC or Intel's Xeon series, 3 teraflops/sec
≈ 13.8 billion years

Attacking ”bad primes”

- Some primes are not suited to be used for RSA, as they make n easier to factor

- Examples:
  - Either **p** or **q** are small numbers
  - **p** and **q** are too close together
  - **p** and **q** are both close to 2^b, where b is a given bound
  - $n = \mathbf{p}^r.\mathbf{q}^s$ and $r > \log p$
  - …

Let’s dive into an example…

Fermat's Little Theorem

- The theorem states:
  - $a^p \equiv a \bmod p$, for prime **p** and integer **a**

  - Special case when **p** is co-prime with integer **a**
    → $gcd(p,a) = 1$, $a^{p-1} \equiv 1 \bmod p$

  - This is also true for any multiple of p-1 (you keep wrapping around):
    → $a^{k(p-1)} \equiv 1 \bmod p$

  - We can rewrite this as:
    $a^{k(p-1)} - 1 = \mathbf{p}.\mathbf{r}$

Can we use F.L.T to find factors of N?

- Consider we have **n** = **p**.**q**
  - Recall:
    $a^{k(p-1)} - 1 = \mathbf{p.r}$
  - Putting this together, we have:
    $\gcd(a^{k(p-1)}-1, n) =$
    $= \gcd(\underline{p}.r, \underline{p}.q) =$
    $= p$

Can we use F.L.T to find factors of N?

- Consider we have **n** = **p**.**q**
    - Recall:
      $a^{k(p-1)} - 1 = \mathbf{p}.\mathbf{r}$
    - Putting this together, we have:
      $\gcd(a^{k(p-1)}-1, n) =$
      $= \gcd(\underline{p}.r, \underline{p}.q) =$
      $= p$

This allow us to find a factor of **n**

Can we use F.L.T to find factors of N?

- Consider we have **n** = **p**.**q**
  - Recall:
    $a^{k(p-1)} - 1 = \mathbf{p}.\mathbf{r}$
  - Putting this together, we have:
    $gcd(a^{k(p-1)}-1, n) =$
    $= gcd(\underline{p}.r, \underline{p}.q) =$
    $= p$

This allow us to find a factor of **n**

But how does this help us? We don't know **p**, nor do we have a way of calculating **k**.

The Pollard p-1 Factoring Algorithm

- We guess **k(p-1)** by brute-force
- Place **a** to the power of integers with a lot of prime factors. Likely that the factors of p−1 are there.
  → Calculate $a^{k!} \bmod n$
- Calculate $\gcd(a^{k(p-1)}-1, n)$
- If it is not equal to one, we found a factor

Inputs: Odd integer n and a “bound” b*

1. a = 2
2. for j =2 to b
   a. Do a ← a^j mod n
3. d = gcd(a-1,n)
4. if 1 < d < n
   a. Then return (d)
   b. Else return (“failure”)

* Usually, a large prime

The Pollard p-1 Factoring Algorithm

Let's factor n = 713:

Calculate a, a^2, $(a^2)^3$, $((a^2)^3)^4$, … and each GCD

a	**d**
$2^1 \equiv 2 \bmod 713$,	gcd(1,713)==1
$2^2 \equiv 4 \bmod 713$,	gcd(3,713)==1
$4^3 \equiv 64 \bmod 713$,	gcd(63,713)==1
$64^4 \equiv 326 \bmod 713$,	gcd(325,713)==1
$326^5 \equiv 311 \bmod 713$,	gcd(310,713)==**31**

1. a = 2
2. for j =2 to b
   a. Do a ← a^j mod n
3. d = gcd(a-1,n)
4. if 1 < d < N
   a. Then return (d)
   b. Else return ("failure")

713/31 = 23

23 * 31 = 713

The case of “smooth” factors

- A prime is deemed smooth if it has multiple small factors
  - **p-1** = $p_1^{e1} . p_2^{e2} \ldots$, $\forall p_i^{ei}$ s.t. $p_i^{ei} \leq B$

- Pollard p-1 algorithm is useful when **p** is smooth
  - Its iterative approach is more likely to include **p −1** sooner rather than later
  - i.e., if p is smooth, k! will includes small prime factors, making the exponentiation $a^{k!}$ mod n reduce to 1 simplifying the calculation of the GCD.

So far so good, but…

WARNING: this was textbook RSA, do **not** use!!!

Why not "Textbook RSA"?

Example: Given the following parameters: p=53, q=101, e=139, d=1459.

Encryption: $c \equiv m^e \pmod{n}$, **Decryption:** $m = c^d \pmod{n}$

- Compute n.
- Compute $C_1 = Enc_e(m_1)$. Verify the decryption works.
- Compute $C_2 = Enc_e(m_2)$. Verify the decryption works.
- Compute $m = Dec_d(C_1 . C_2)$. What is happening? Why ?

A: The decryption would yield the product of the original plaintexts.

$(m_1)^e . (m_1)^e \equiv (m_1 . m_1)^e$

Malleability: it is possible to transform a ciphertext into another ciphertext that decrypts to a transformation of the original plaintext.

This is typically (but not always!) undesirable.

Attacking RSA (CCA)

Chosen Ciphertext Attack (CCA)

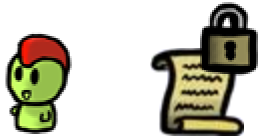
- We are Eve. Alice is using RSA and her public key is (e, n). 
- Bob sends secret message m, encrypted as c = $Enc_e(m)$. 
- We intercept c. 
- Alice is convinced her textbook RSA is very secure, so she is willing to decrypt any ciphertext we send her (except for c).

Attacking RSA (CCA)

Chosen Ciphertext Attack (CCA)

- We are Eve. Alice is using RSA and her public key is (e, n).

- Bob sends secret message m, encrypted as $c = Enc_e(m)$.



- We intercept c.

- Alice is convinced her textbook RSA is very secure, so she is willing to decrypt any ciphertext we send her (except for c).

Goal: Ask Alice to decrypt something (other than c) that helps us guess m

Attacking RSA (CCA)

Chosen Ciphertext Attack (CCA): Solution

- Alice's public key is (e, n).
- Bob sends $c_1 = Enc_e(m)$. We intercept c_1.

Q: Ask Alice to decrypt, e.g., $c_2 = 2^e \cdot c_1$.

Attacking RSA (CCA)

Chosen Ciphertext Attack (CCA): Solution

- Alice's public key is (e, n).
- Bob sends $c_1 = Enc_e(m)$. We intercept c_1.

Q: Ask Alice to decrypt, e.g., $c_2 = 2^e \cdot c_1$.

A: This decryption yields $(2^e \cdot c_1)^d \equiv 2m$.
We divide the result by 2, and we get m.

Example: given m=5, e=3, and n=33 → $c_1 = 26$, $c_2 = 208$ → $m_2 = 10$

Attacking RSA (CCA)

Chosen Ciphertext Attack (CCA): Solution

- Alice's public key is (e, n).
- Bob sends $c_1 = Enc_e(m)$. We intercept c_1.

Q: Ask Alice to decrypt, e.g., $c_2 = 2^e \cdot c_1$.

A: This decryption yields $(2^e \cdot c_1)^d \equiv 2m$.
We divide the result by 2, and we get m.

- ✓ Textbook RSA is vulnerable against chosen ciphertext attacks (among other things)
- ✓ We can fix this with padding techniques (RSA-OAEP).

Show Naive RSA Encryption is not IND-CPA Secure

1. Eve produces two plaintexts, m_0 and m_1 

Show Naive RSA Encryption is not IND-CPA Secure

1. Eve produces two plaintexts, m_0 and m_1 

2. “Challenger” encrypts an m as $c^* = m_b^e \pmod{n}$, secret b 

Show Naive RSA Encryption is not IND-CPA Secure

1. Eve produces two plaintexts, m_0 and m_1

2. "Challenger" encrypts an m as $c^* = m_b^e \pmod{n}$, secret b

3. Eve's goal? Determine $b \in \{0,1\}$

Show Naive RSA Encryption is not IND-CPA Secure

1. Eve produces two plaintexts, m_0 and m_1 

2. "Challenger" encrypts an m as $c^* = m_b^e \pmod n$, secret b 

3. Eve's goal? Determine $b \in \{0,1\}$

4. Sooo, Eve computes $c = m_1^e \pmod n$

   If $c^* = c$ then Eve knows $m_b = m_1$
   If $c^* \neq c$ then Eve knows $m_b = m_0$

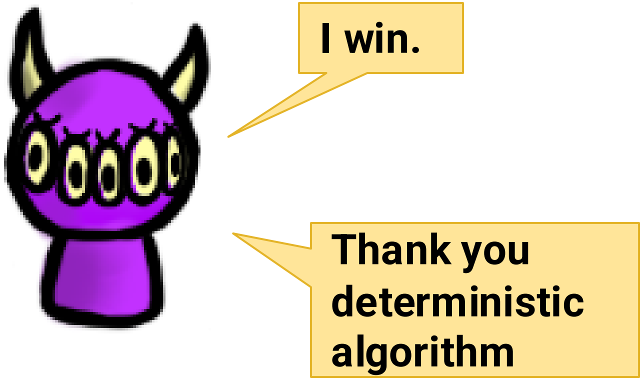
Show Naive RSA Encryption is not IND-CPA Secure

1. Eve produces two plaintexts, m_0 and m_1

2. "Challenger" encrypts an m as $c^* = m_b^e \pmod{n}$, secret b

3. Eve's goal? Determine $b \in \{0,1\}$
4. Sooo, Eve computes $c = m_1^e \pmod{n}$

   If $c^* = c$ then Eve knows $m_b = m_1$
   If $c^* \neq c$ then Eve knows $m_b = m_0$



Adversaries and their Goals

You’ve assumed my goal is the secret/private key…

Adversaries and their Goals

You've assumed my goal is the secret/private key…

…but less ambitious goals can be very effective…

Adversaries and their Goals

Goal 1: Total Break

- Win the Symmetric key K
- Win Bob's private key k_b
- ()Can decrypt any c_i *for:*

$$c_i = Enc_K(m)$$

or

$$c_i = Enc_{kb}(m)$$

- All messages using compromised k revealed
- Unless **detected** game over

Goal 2: Partial Break

- Decrypt **a ciphertext c** (without the key)
- Learn **some** specific information about a message *m* from c

**Need to occur with non-negligible probability.

- **Some (or a)** message revealed

Goal 3: Distinguishable Ciphertexts

- *Pr {learn b* ∈ {0,1}} exceeds ½
- Distinguish between $\text{Enc}(m_1)$ and $\text{Enc}(m_2)$

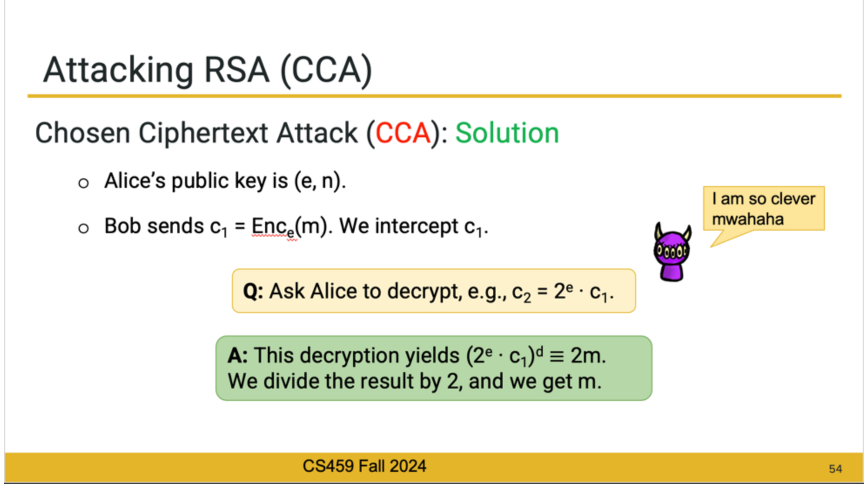
  or

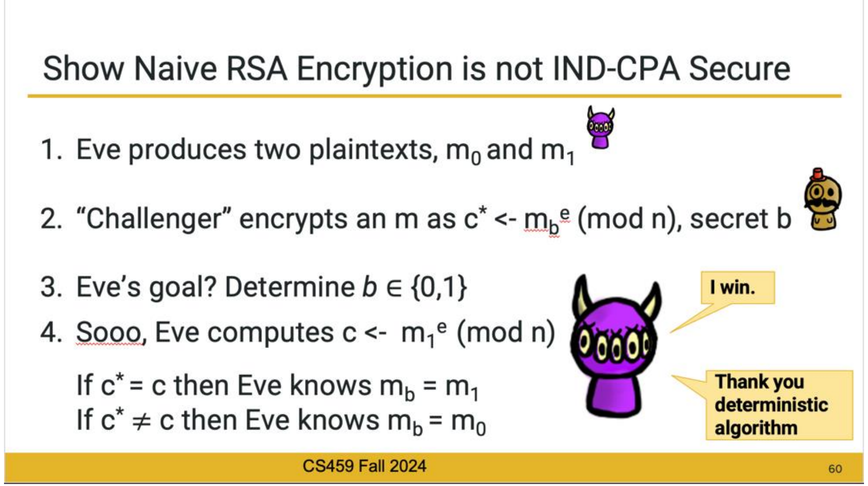
  between Enc(m) and Enc(random string)

- The ciphertexts are leaking small/some information…

Semantic Security of RSA

- We saw CCA against Naive RSA
- We showed IND-CPA on Naive RSA





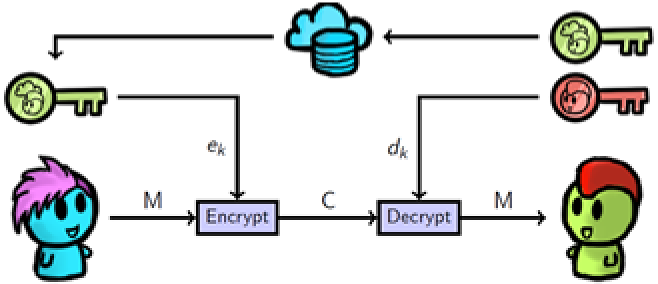
Fix it? Ciphertext Distinguishability

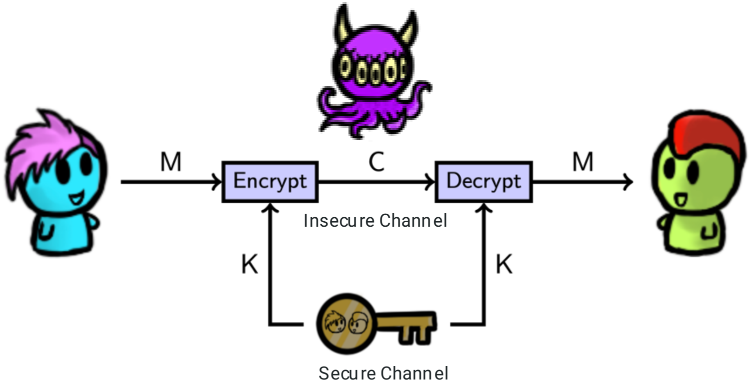
Goal: prove (given comp. assumptions) that no information regarding *m* is revealed in polynomial time by examining *c* = *Enc(m)*

- If Enc( ) is deterministic, fail
- Thus, require some randomization

RSA-OAEP: Optimal Asymmetric Encryption Padding

Practicality of Public-Key vs. Symmetric-Key

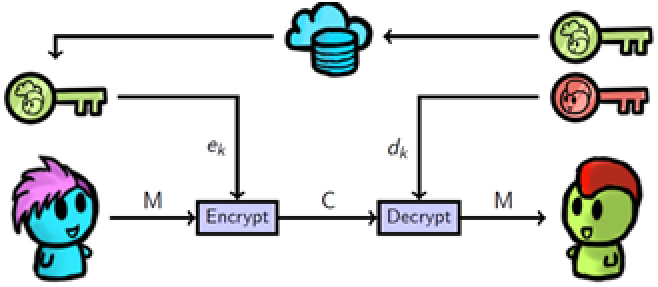




1. Longer keys
2. Slower
3. Different keys for Enc(m) and Dec(c)

1. Shorter keys
2. Faster
3. Same key for Enc(m) and Dec(c)

Practicality of Public-Key vs. Symmetric-Key



1. Longer keys
2. Slower
3. Different keys for Enc(m) and Dec(c)

1. Shorter keys
2. Faster
3. Same key for Enc(m) and Dec(c)

Public-Key sizes

- Recall that if there are no shortcuts, Eve would have to try 2^{128} iterations in order to read a message encrypted with a 128-bit key

- Unfortunately, all of the public-key methods we know do have shortcuts
  - Eve could read a message encrypted with a 128-bit RSA key with just 2^{33} work, which is easy!
  - Comparison of key sizes for roughly equal strength

AES	RSA	ECC
80	1024	160
116	2048	232
128	2600	256
160	4500	320
256	14000	512

What cab be done? (Hybrid Cryptography)

We can get the best of both worlds:

- Pick a random “128-bit” key K for a symmetric-key cryptosystem
- Encrypt the large message with the key K (e.g., using AES)

And then…

- Encrypt the key K using a public-key cryptosystem
- Send the encrypted message and the encrypted key to Bob

Hybrid cryptography is used in (many) applications on the internet today

Knowledge Check!

Public: (e_A, d_A)

Secret: K

Public: (e_B, d_B)

Secret: ?

- Enc/Dec functions: $Enc_{key}(*)$, $Dec_{key}(*)$
- Alice wants to send a **large** message *m* to Bob.

Q: How should Alice build the message efficiently? How does Bob recover m?

Knowledge Check!

Public: (e_A, d_A)

Secret: K

Public: (e_B, d_B)

Secret: ?

- Enc/Dec functions: $Enc_{key}(*)$, $Dec_{key}(*)$
- Alice wants to send a **large** message *m* to Bob.

Q: How should Alice build the message efficiently? How does Bob recover m?

FYI: PKE is slow!! We don't want to use it on m.

Knowledge Check!

Public: (e_A, d_A)

Secret: K

Public: (e_B, d_B)

Secret: ?

- Enc/Dec functions: $Enc_{key}(*)$, $Dec_{key}(*)$
- Alice wants to send a **large** message *m* to Bob.

Q: How should Alice build the message efficiently? How does Bob recover m?

A: Alice computes $c_1 = Enc_{eB}(K)$, $c_2 = E_K(m)$ and sends $<c_1||c_2>$.
Bob recovers $K = Dec_{dB}(c_1)$ and then $m = Dec_K(c_2)$

Knowledge Check!

We know how to "send secret messages", and Eve cannot do anything about it. What else is there to do?

- Mallory can modify our encrypted messages in transit!
- Mallory won't necessarily know what the message says, but can still change it in an undetectable way
  - e.g. bit-flipping attack on stream ciphers
- This is counterintuitive, and often forgotten

Q: How do we make sure that Bob gets the same message Alice sent?

Up next: More Cryptography…