CS489/698 Privacy, Cryptography, Network and Data Security

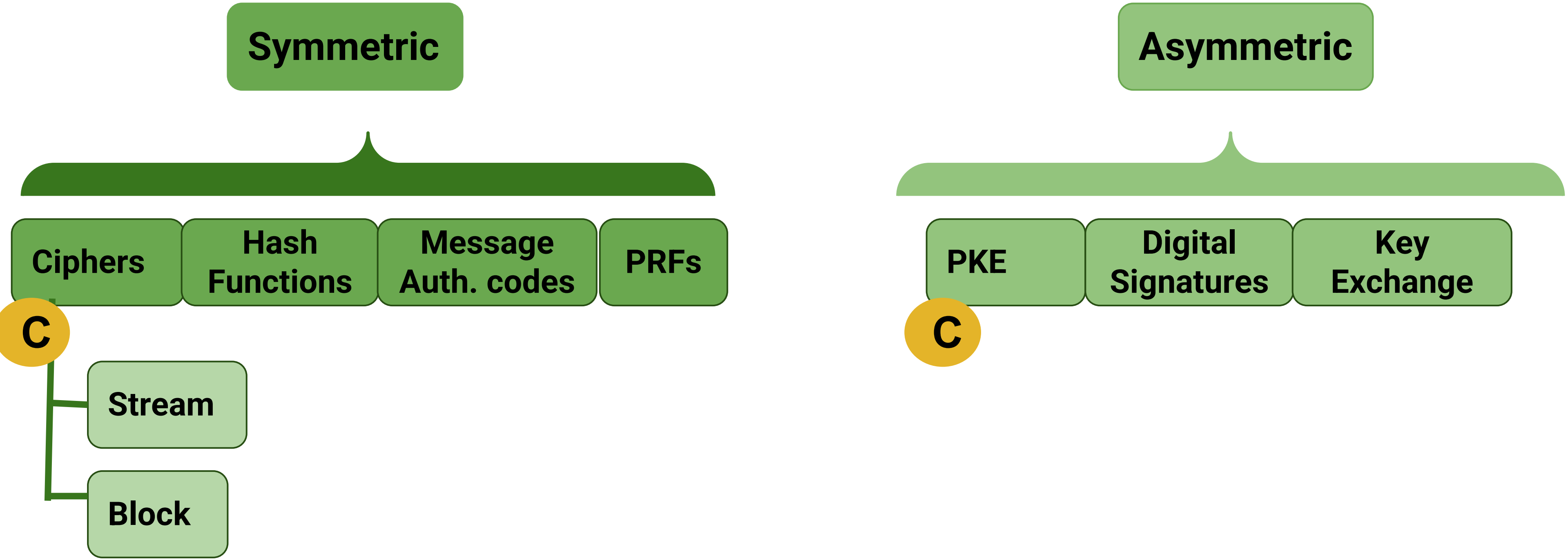
Public Key Cryptography (RSA)

Spring 2024, Monday/Wednesday 11:30am-12:50pm

Assignment One

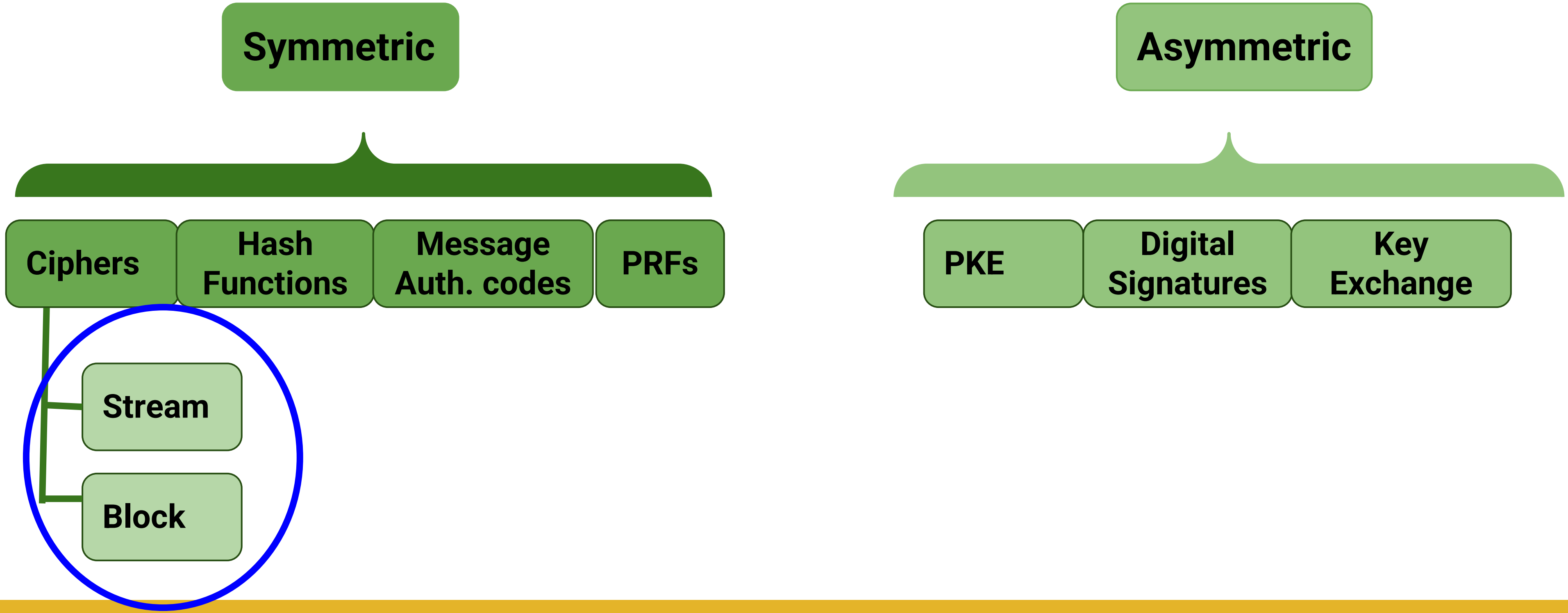
- Available on Learn today at 3pm
- Due **May 29th, 3pm**
- Written and programming

Cryptography Organization

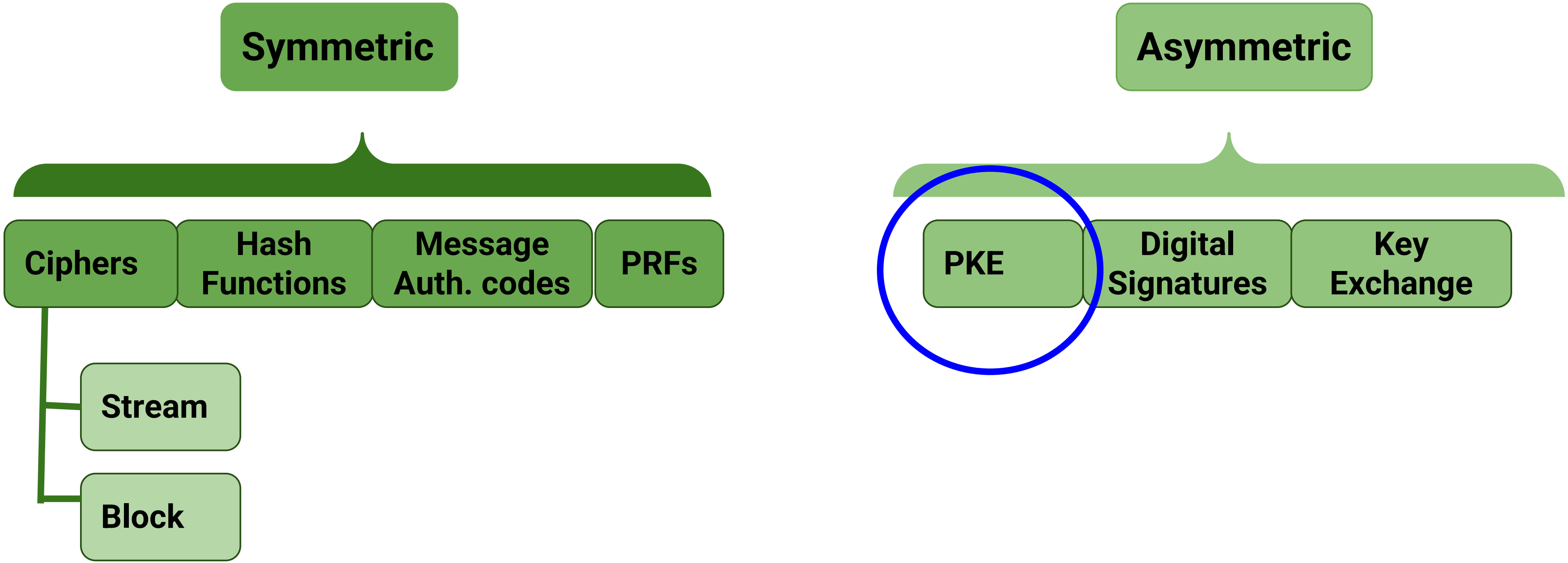


Organization display source: Doug Stebila

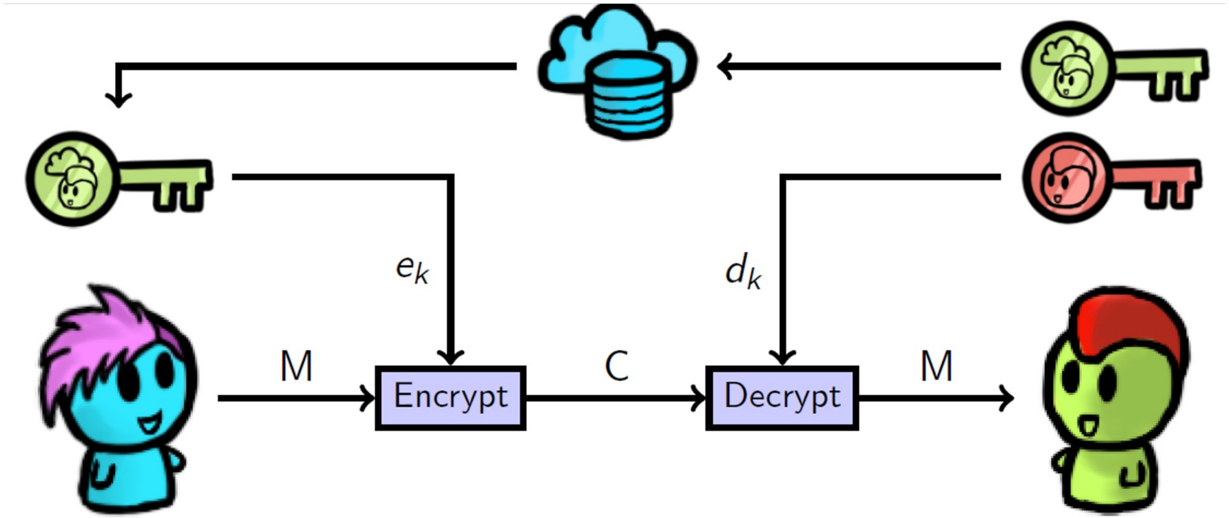
Cryptography Organization



Cryptography Organization



Public Key Cryptography, "1970s"



Examples:

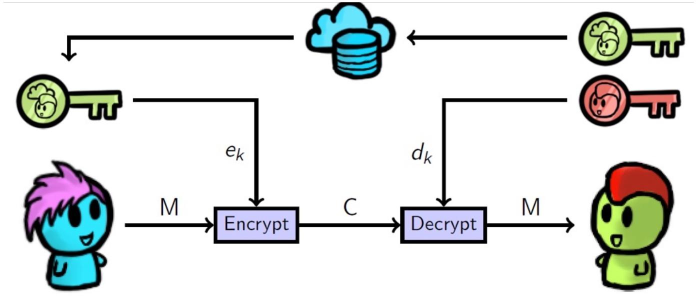
- RSA, ElGamal, ECC, NTRU

Steps for Public Key Cryptography?

1. Bob generates pair

2. Bob gives everyone the public key

3. Alice encrypts m and sends it



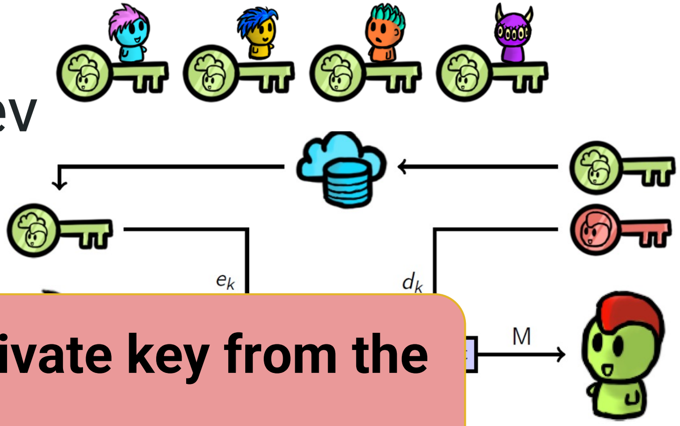
4. Bob decrypts using private key

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

Steps for Public Key Cryptography?

1. Bob generates pair 

2. Bob gives everyone the public kev



3. Alice encrypts m and sends it

4. Bob d

It must be hard to derive the private key from the public 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

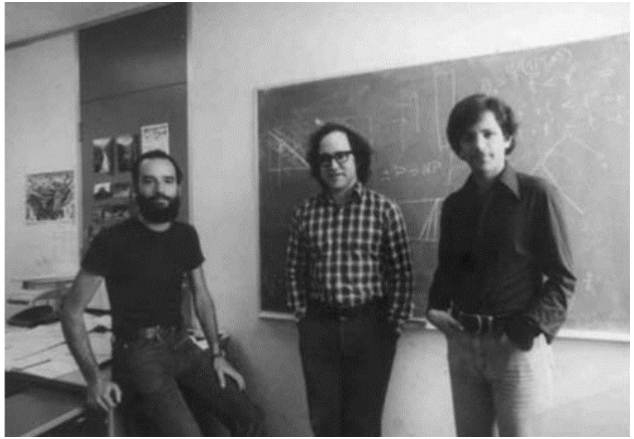
- The encryption function? Must be easy to compute 
- The inverse, decryption? Must be hard for anyone without the key  vs. 

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

But because of decryption,
we need a “trapdoor”

Time for Textbook RSA

- A clever arithmetic trick based on a “trapdoor permutation”
- 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.)

Time for Textbook RSA

- Overview:

$$(x^e)^d \equiv x \bmod N$$

- 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 can’t find **d**!!!

Time for Textbook RSA

- Encryption:

$$y = x^e \bmod N$$

The ciphertext is equal to **x** multiplied by itself **e** times modulo **N**.

Public key is given by **PubK** = (e, N)

Time for Textbook RSA

- Decryption:

$$x = y^d \bmod N = (x^e)^d \bmod N = x^{ed} \bmod N$$

Decryption relies on number **d** such that **e**.**d** = 1 mod **N**, and where $x^{ed} \bmod N = x^1 \bmod N = x$

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

Private key is given by **PrivK** = (d)

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

- Pick two random primes **p** and **q**, such that **p**.**q** = **N**
- Generate **φ(N)** = (p-1).(q-1)
  - all relative primes to (p−1)(q−1) form a group with respect to multiplication and are invertible
- 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** + <u>5</u>

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

- 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 $(x^e)^d \equiv x \bmod n$
4. **Public key**: (e, n)
5. **Private key**: d
6. Encryption: $y = x^e \bmod n$
7. Decryption: $y^d \bmod n$

Example (Tiny RSA)

Parameters:

- p=53, q=101, N=5353
- φ(N) = (53-1).(101-1) = 5200
- e=139 (random pick)
- d=1459 (extended Euclidean)

- Message:
- x=20

Encryption: $y = x^e \bmod N$

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

Decryption: $x = y^d \bmod N$

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

Nice!

Example (Tiny RSA)

Parameters:

- p=53, q=101, N=5353
- φ(N) = (53-1).(101-1) = 5200
- e=139 (random pick)
- d=1459 (extended Euclidean)

- Message:
- x=<u>20</u>

Encryption: $y = x^e \bmod N$

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

Decryption: $x = y^d \bmod N$

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

Attacking RSA

I know **e** and **N**…
What can I do to find **d**?

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
- φ(N) = (53-1).(101-1) = 5200
- e=139
- d=1459

- y = 5274

Attacking RSA

W... 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
- φ(N) = (53-1).(101-1) = 5200
- e=139
- ...=1459
- y ... 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

Strawman Approach at Factoring

- 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

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

Can we use this 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 this 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 this 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:
      $gcd(a^{k(p-1)}-1, N) =$
      $= gcd(\underline{p}.r, \underline{p}.q) =$
      $= p$

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

Inputs: Odd integer N and a "bound" b

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

1. a = 2
2. for j =2 to b
   a. Do $a \equiv a^j \bmod N$
3. d = gcd(a-1,N)
4. if 1 < d < N
   a. Then return (d)
   b. Else return ("failure")

The Pollard p-1 Factoring Algorithm

Let's factor N = 713:

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 $a \equiv a^j \bmod N$
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

  $\mathbf{p-1} = p_1^{e1} \,.\, p_2^{e2} \ldots, \forall\, p_i^{ei},\ p_i^{ei} \leq B$

  - Pollard p-1 algorithm is useful when **p** is smooth
  - Its iterative approach is more likely to include **p −1** sooner rather than later

So far so good, but…

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

Why not "Textbook RSA"? Example

Example: (Tiny RSA), $p=53$, $q=101$, $e=139$, $d=1459$

Encryption: $y \equiv x^e \pmod{N}$, **Decryption:** $x = y^d \pmod{N}$

- Compute N
- Compute $Y_1 = E_e(1011)$. Verify the decryption works
- Compute $Y_2 = E_e(4)$. Verify the decryption works
- Compute $D_d(Y_1 * Y_2)$. What is happening…and why?

Note::

- The * here indicates multiplication/compute a product

Why not "Textbook RSA"? Example

Example: (Tiny RSA), p=53, q=101, e=139, d=1459

Encryption: $y \equiv x^e \pmod N$, **Decryption:** $x = y^d \pmod N$

- Compute N
- Compute $Y_1 = E_e(1011)$. Verify the decryption works
- Compute $Y_2 = E_e(4)$. Verify the decryption works
- Compute $D_d(Y_1 * Y_2)$. What is happening…and why?

A: The decryption is the product of the original plaintexts!!!

Note::

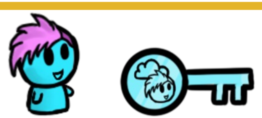
- The * here indicates multiplication, compute a product

Malleability

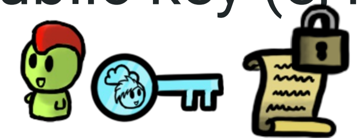
A: $y_1 * y_2 = (x_1)^e * (x_2)^e = (x_1 * x_2)^e$

- It is possible to transform a ciphertext into another ciphertext that decrypts to a related plaintext
- Undesirable (most of the time)

RSA and a Chosen Ciphertext Attack



- Alice is using RSA, public key (e, n)



- Bob sends $y = E_e(x)$

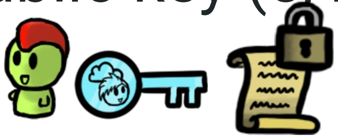
- We are Eve! We snag y.
- Alice…is confident about textbook RSA, will decrypt any ciphertext except y for us

Goal: Ask Alice to decrypt something (other than y) that helps us learn x

Executing CCA on Textbook RSA

- Alice is using RSA, public key (e, n)

- Bob sends $y = E_e(x)$



I am so clever mwahaha

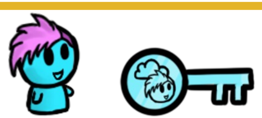
- We-Eve ask Alice to decrypt $y_2 = 2^e * y_1$

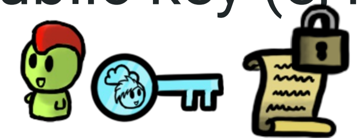
Q: Decrypts to?

Q: Decrypts to?

Executing CCA on Textbook RSA

- Alice is using RSA, public key (e, n)





- Bob sends $y = E_e(x)$

I am so clever mwahaha

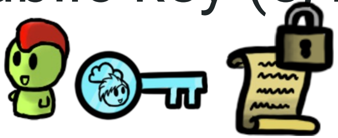
- We-Eve ask Alice to decrypt $y_2 = 2^e * y_1$

A: decryption gives $(2^e * y_1)^d \equiv 2x$

Q: Decrypts to?

Executing CCA on Textbook RSA

- Alice is using RSA, public key (e, n)



- Bob sends $y = E_e(x)$

I am so clever mwahaha

- We-Eve ask Alice to decrypt $y_2 = 2^e * y_1$

Q: Decrypts to?

A: decryption gives $(2^e * y_1)^d \equiv 2x$

Textbook RSA: vulnerable to CCA
Note: Can be addressed with padding techniques

Show Naive RSA Encryption is not IND-CPA Secure

1. Eve produces two plaintexts, x_0 and x_1

Show Naive RSA Encryption is not IND-CPA Secure

1. Eve produces two plaintexts, x_0 and x_1

2. “Challenger” encrypts an x as $y^* <- x_b^e$ (mod N), secret b

Show Naive RSA Encryption is not IND-CPA Secure

1. Eve produces two plaintexts, x_0 and x_1

2. "Challenger" encrypts an x as y^* <- x_b^e (mod 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, x_0 and x_1

2. “Challenger” encrypts an x as $y^* <- x_b^e$ (mod N), secret b

3. Eve’s goal? Determine *b* ∈ {0,1}

4. Sooo, Eve computes $y <- x_1^e$ (mod N)

   If $y^* = y$ then Eve knows $x_b = x_1$
   If $y^* <> y$ then Eve knows $x_b = x_0$

Show Naive RSA Encryption is not IND-CPA Secure

1. Eve produces two plaintexts, x_0 and x_1

2. "Challenger" encrypts an x as $y^* \leftarrow x_b^e \pmod{N}$, secret b

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

4. Sooo, Eve computes $y \leftarrow x_1^e \pmod{N}$

   If $y^* = y$ then Eve knows $x_b = x_1$
   If $y^* \neq y$ then Eve knows $x_b = x_0$

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 secret key k or
- Win Bob's private key k_b
- Can decrypt any y_i *for:*

$$y_i = E_k(x) \text{ or } y_i = E_{kb}(x)$$

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

Goal 2: Partial Break

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

**Need to occur with non-negligible probability.

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

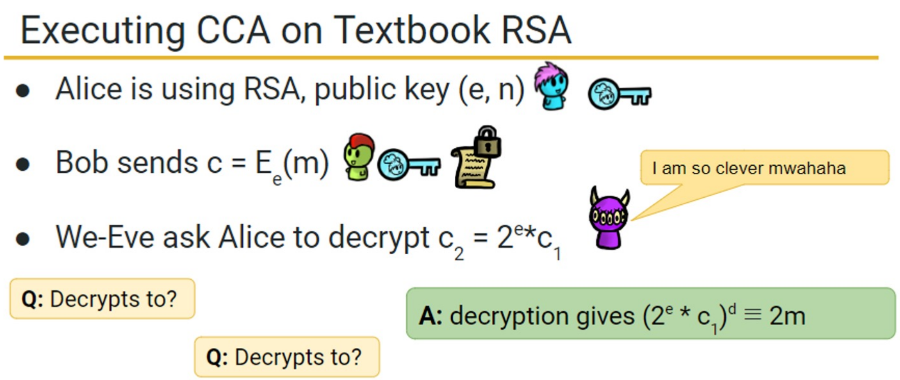
Goal 3: Distinguishable Ciphertexts

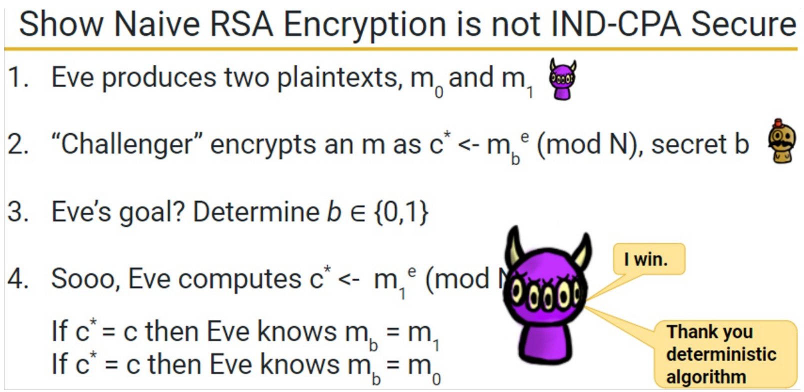
- *P{learn b* ∈ {0,1}} exceeds ½
- Distinguish between $E(x_1)$ and $E(x_2)$ or between E(x) and E(random string)

- The ciphertexts are leaking small/some information…

Semantic Security of RSA

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





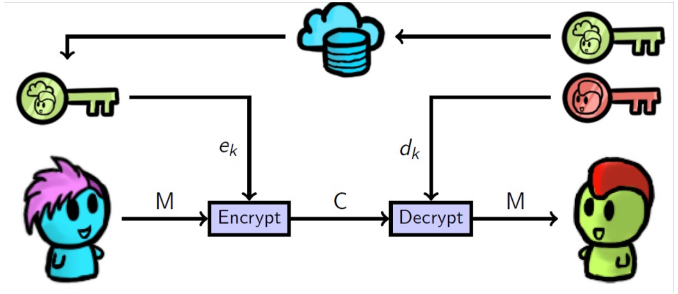
Fix it? Ciphertext Distinguishability

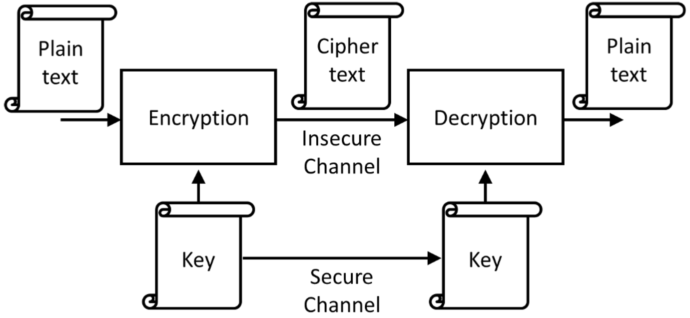
Goal: prove (given comp. assumptions) no information regarding *x* is revealed in polynomial time by examining $y = E(x)$

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

RSA-OAEP: Optimal Asymmetric Encryption Padding

Practicality of Public-Key vs. Symmetric-Key

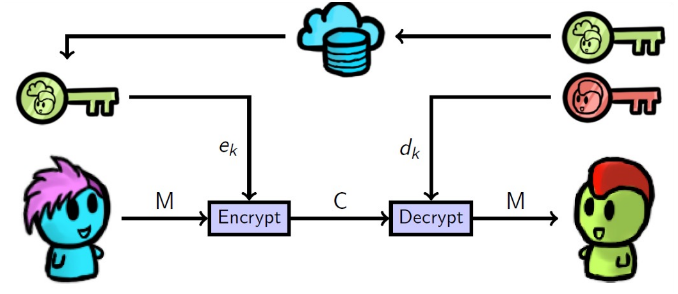




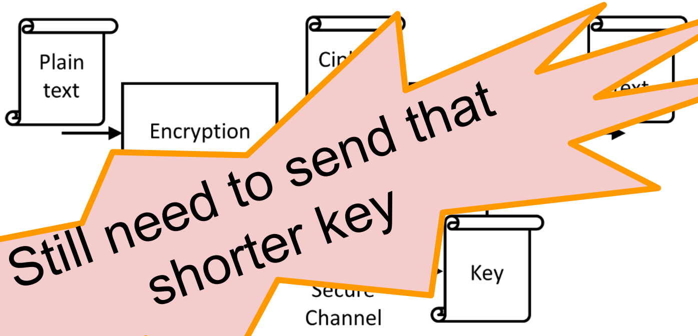
1. Longer keys
2. Slower
3. Different keys for E(x) and D(y)

1. Shorter keys
2. Faster
3. Same key for E(x) and D(y)

Practicality of Public-Key vs. Symmetric-Key



1. Longer keys
2. Slower
3. Different keys for E(x) and D(y)



Still need to send that shorter key

1. Shorter keys
2. Faster
3. Same key for E(x) and D(y)

Hybrid Cryptography

- Combine the two!!!!!!!
- Pick a random "128-bit" key K for a symmetric-key system
- Encrypt the large message with the key K (e.g., using AES)

And then…

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

Hybrid Cryptography

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

And then…

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

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

Just Checking…

Public: (e_A, d_A)

Secret: K

Public: (e_B, d_B)

Secret: ?

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

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

Just Checking…

Public: (e_A, d_A)

Secret: K

Public: (e_B, d_B)

Secret: ?

- Enc/Dec functions: $E_{key}(*)$, $D_{key}(*)$
- 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 $y_1 = E_{eB}(K)$, $y_2 = E_K(x)$ and sends $<y_1||y_2>$
Bob recovers $K = D_{dB}(y_1)$ and then $x = D_K(y_2)$

Up next: More Cryptography…