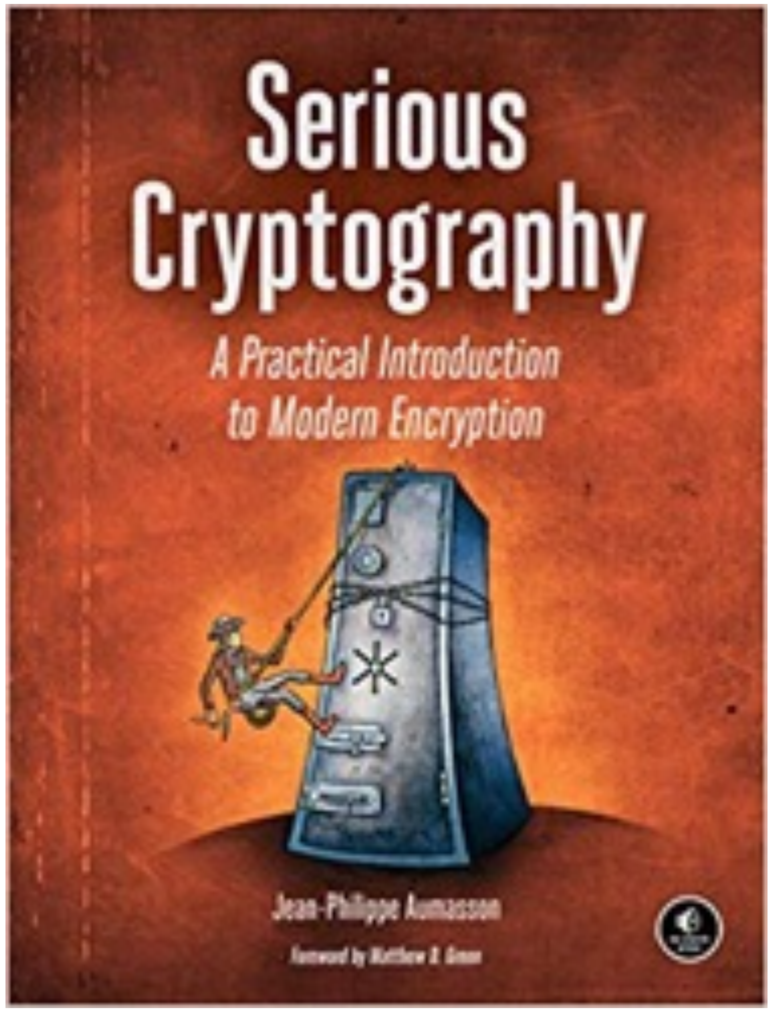
CNIT 141

Cryptography for Computer Networks



10.RSA

Updated 11-6-23

Topics

- The Math Behind RSA
- The RSA Trapdoor Permutation
- RSA Key Generation and Security
- Encrypting with RSA
- Signing with RSA
- RSA Implementations
- How Things Can Go Wrong

The Math Behind RSA

The Group $\mathbf{Z}_N^*$

- Integers modulo N
  - Must contain an identity element: 1
  - Each member must have an inverse
    - So zero is excluded
- But $\mathbf{Z}_4^*$ contains {1,3}
  - 2 has no inverse
    - Because 4 is a multiple of 2

The Group $\mathbf{Z}_{N}^{*}$

- If N is prime, $\mathbf{Z}_{p}^{*}$ contains {1,2, 3, ... p-1}
  - So $\mathbf{Z}_{5}^{*}$ contains {1,2,3,4}
  - 0 is excluded because it has no inverse

The Group $\mathbf{Z}_{10}^*$

- $\mathbf{Z}_{10}^*$ contains {1, 3, 7, 9}
  - 0, 2, 4, 5, 6, 8 are excluded
  - Because they have no inverse
  - Because they have a common factor with 10
  - They aren't ***co-prime*** with 10

Euler's Totient

- How many elements are in the group Z_n*?
  - When ***n*** is not prime, but the product of several prime numbers
  - $\boldsymbol{n = p_1 \times p_2 \times \ldots \times p_m}$
- $\phi(\boldsymbol{n}) = (\boldsymbol{p_1} - 1) \times (\boldsymbol{p_2} - 1) \times \ldots \times (\boldsymbol{p_m} - 1)$
- For $\mathbf{Z_{10}^*}$
  - $\boldsymbol{n} = 10 = 2 \times 5$
  - $\phi(10) = 1 \times 4$: 4 elements in $\mathbf{Z_{10}^*}$

The RSA Trapdoor Permutation

RSA Parameters

- ***n*** is the ***modulus***
  - Product of two primes ***p*** and ***q***
- ***e*** is the ***public exponent***
  - In practice, usually 65537
- Public key: (***n***, ***e***)
- Private key: ***p*** and other values easily derived from it

Trapdoor Permutation

- ***x*** is the plaintext message
- ***y*** is the ciphertext

Encryption: $\boldsymbol{y = x^e \bmod n}$

Decryption: $\boldsymbol{x = y^d \bmod n}$

- ***d*** is the decryption key
  - calculated from ***p*** and ***q***

Calculating *d*

- $ed = 1 \bmod \phi(n)$
- Decryption: $\mathbf{x} = \mathbf{y^d \bmod n}$

$$\begin{aligned} \mathbf{x} &= \mathbf{y^d \bmod n} \\ &= \mathbf{(x^e)^d \bmod n} \\ &= \mathbf{x^{ed} \bmod n} \\ &= \mathbf{x \bmod n} \\ &= \mathbf{x} \end{aligned}$$

Why mod ϕ(*n*) ?

$$a^{\varphi(n)} \equiv 1 \pmod{n}.$$

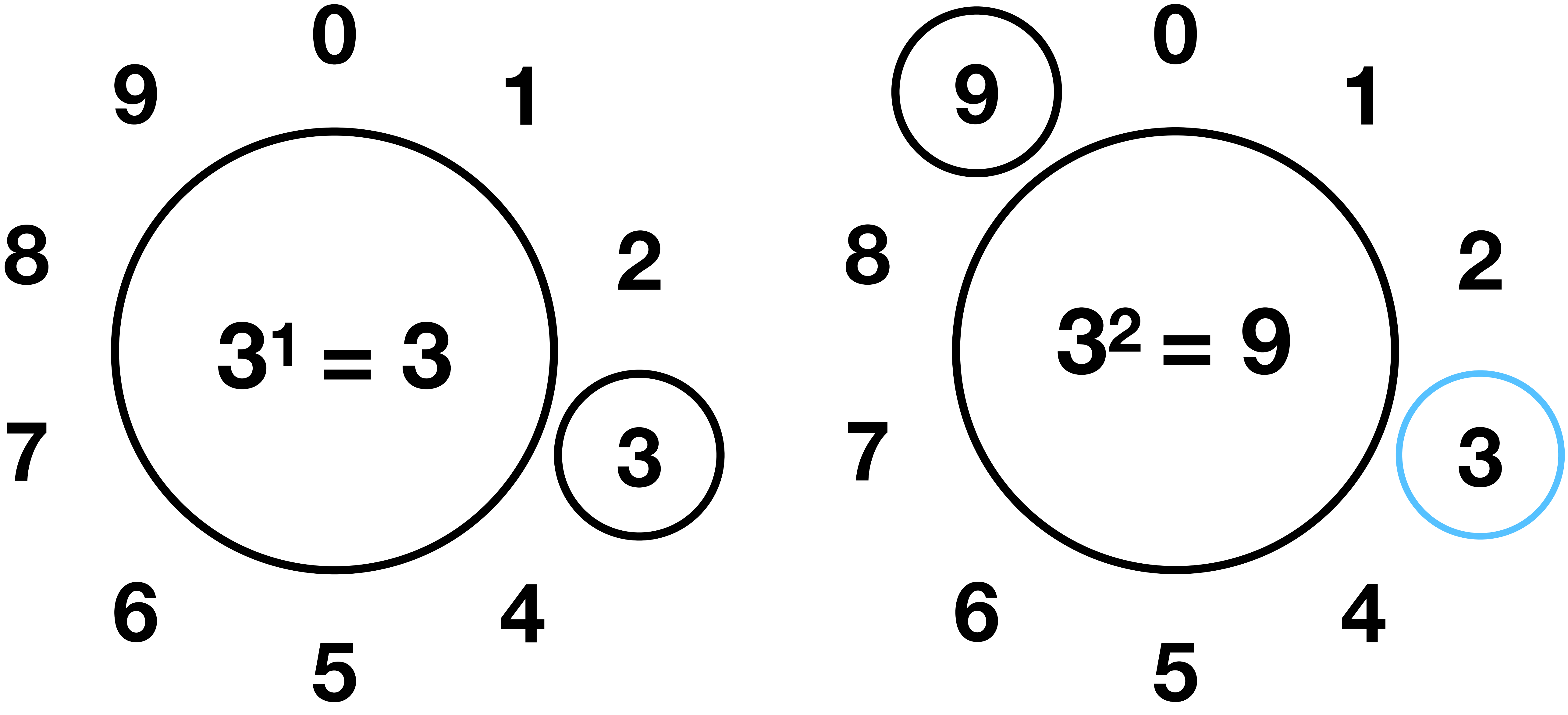
- Fermat's Little Theorem (aka Euler's Theorem)
- Stated by Fermat in 1640 without proof
- Proven by Euler in 1736

Example: n=10

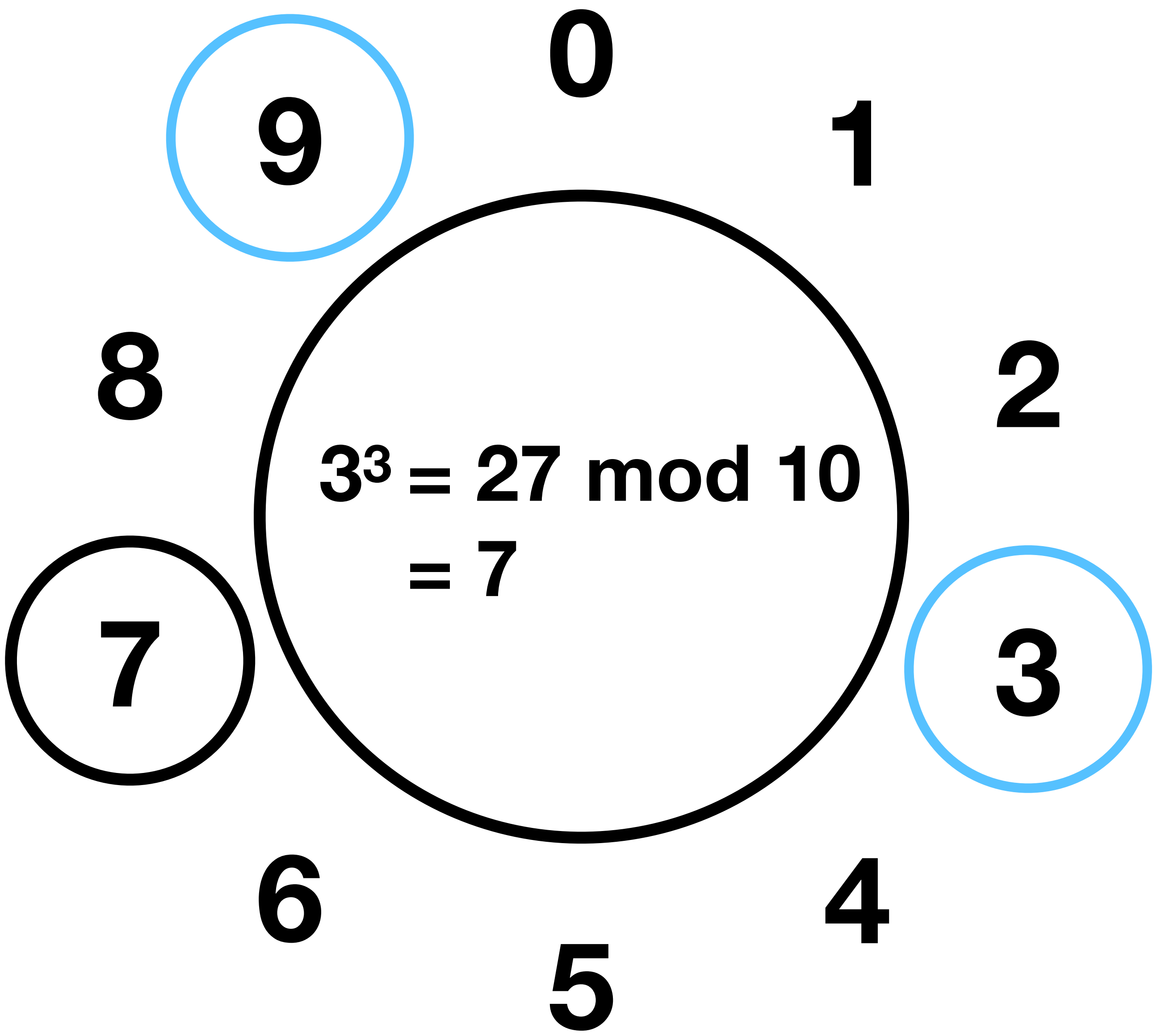
Example: n = 10

- 10 = 2 * 5; p = 2, q = 5
- ϕ = (***p*** - 1) (***q*** - 1) = 1 * 4 = 4 elements in group
- 3 is a ***generator*** of the group (see next slides)

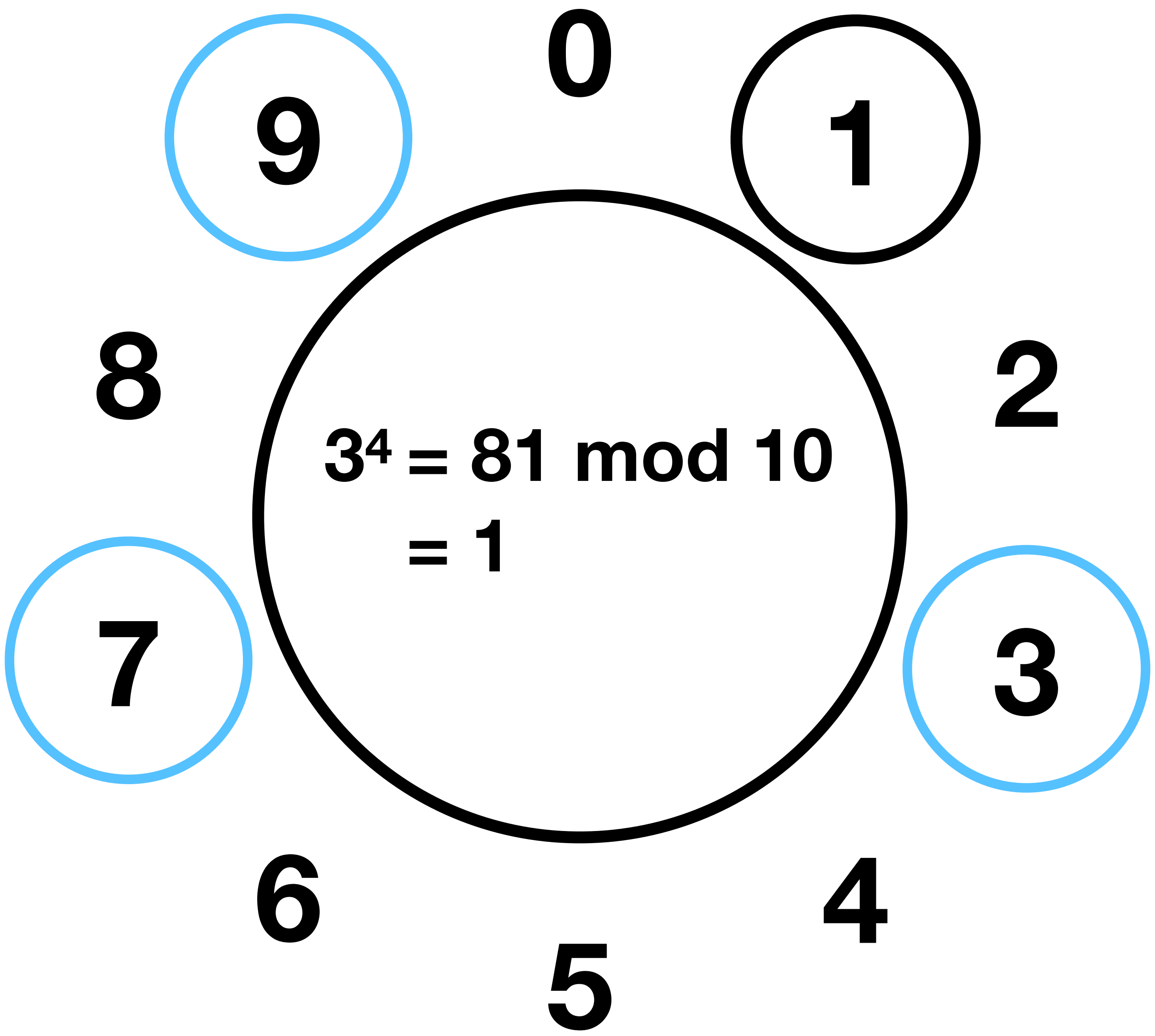
$n=10; x=3$



n=10; x=3



3^3 = 27 mod 10
= 7



3^4 = 81 mod 10
= 1

Powers of 3

$3^1 \bmod 10 = 3$
$3^2 \bmod 10 = 9$
$3^3 \bmod 10 = 7$
$3^4 \bmod 10 = 1$
$3^5 \bmod 10 = 3$

- $\mathbf{Z_{10}^*}$ contains {1, 3, 7, 9}
  - 4 elements
- 3 is a ***generator*** of the group
- Although ***n*** is 10, the powers of 3 repeat with a cycle of 4 (**ϕ**(***n***))
- Encrypt by raising ***x*** to power ***e***, forming ***y***
- Decrypt by raising ***y*** to power ***d***, returning ***x***

Finding ***d***

n=10; ***e***=3

- **Z_{10}*** contains {1, 3, 7, 9}
  - 4 elements
- ***p* = 2** and ***q* = 5**
- **ϕ(n) = (p-1)(q-1) = 1**x**4 = 4**
- ***ed*** = 1 mod **ϕ(n)**
- For ***e* = 3**, ***d* = 3**

3x1 mod 4 = 3
3x2 mod 4 = 2
3x**3 mod 4 = 1**

n=10; x=3; e=3; d=7

- x is the plaintext message (3)
- y is the ciphertext

Encryption:

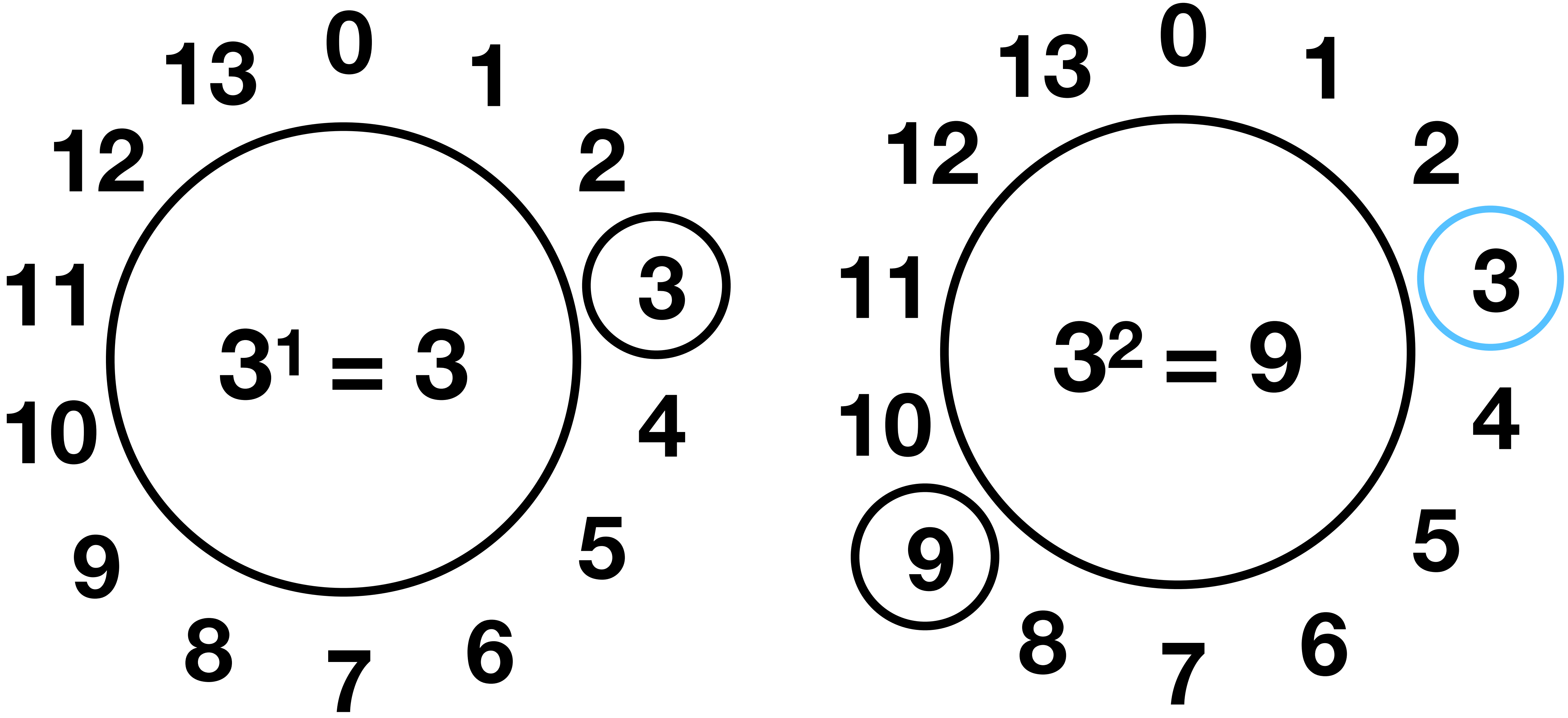
$$\begin{aligned} y &= x^e \bmod n \\ &= 3^3 \bmod 10 \\ &= 27 \bmod 10 \\ &= 7 \end{aligned}$$

Decryption:

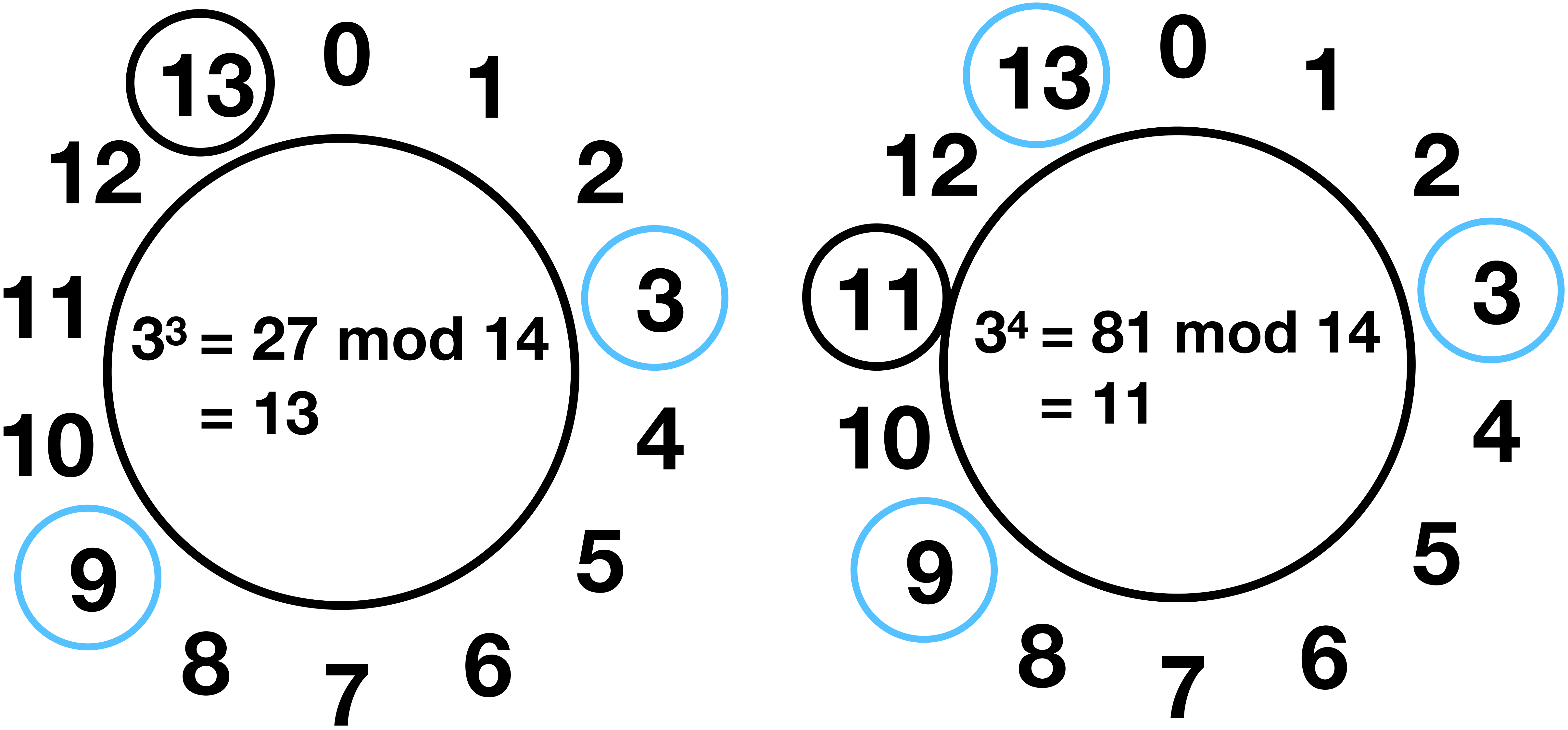
$$\begin{aligned} x &= y^d \bmod n \\ &= 7^3 \bmod 10 \\ &= 343 \bmod 10 \\ &= 3 \end{aligned}$$

Example: $n=14$

n=14; x=3



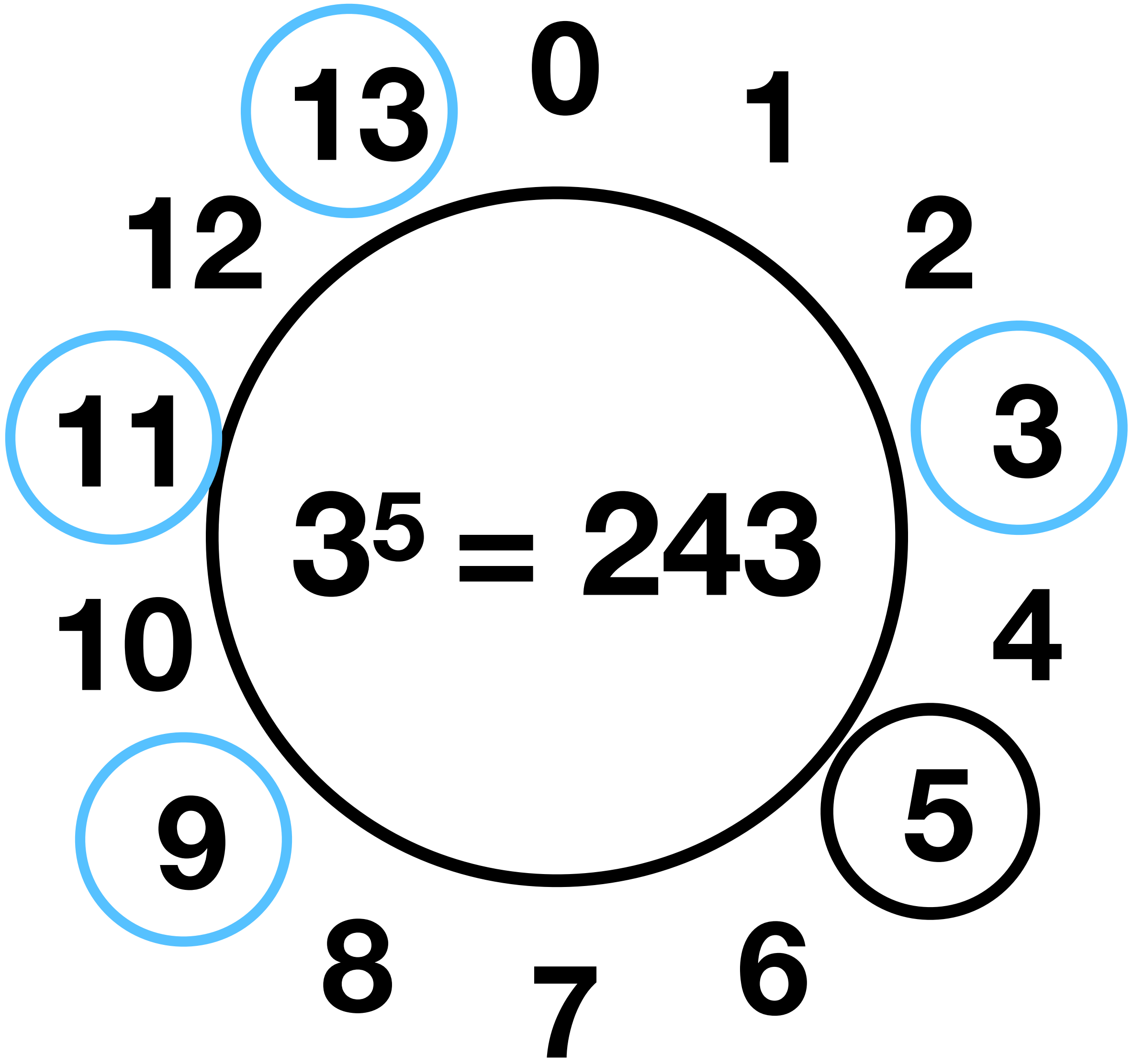
$n=14; x=3$

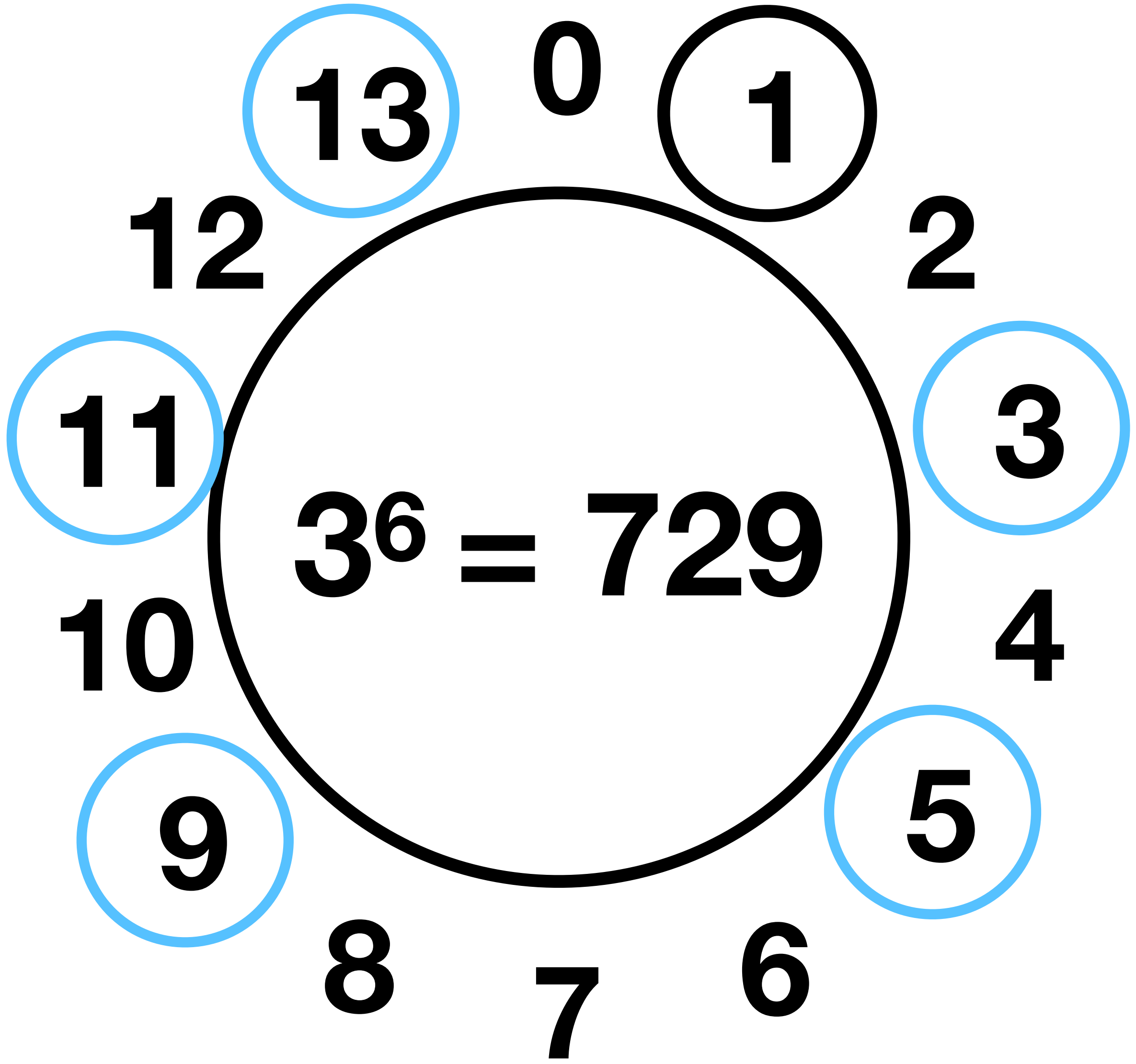


$3^3 = 27 \bmod 14$
$= 13$

$3^4 = 81 \bmod 14$
$= 11$

$n=14; x=3$





Powers of 3

3^1 mod 14 = 3

3^2 mod 14 = 9

3^3 mod 14 = 13

3^4 mod 14 = 11

3^5 mod 14 = 5

3^6 mod 14 = 1

- $\mathbf{Z_{14}}$***** contains {1, 3, 5, 9, 11, 13}
  - 6 elements ($\boldsymbol{\phi}(\boldsymbol{n})$)
- 3 is a ***generator*** of the group

Finding $\boldsymbol{d}$

$\boldsymbol{n}=14$; $\boldsymbol{x}=3$; $\boldsymbol{e}=5$

- $\mathbf{Z}_{14}^*$ contains {1, 3, 5, 9, 11, 13}
  - 6 elements
- $\boldsymbol{p} = \mathbf{2}$ and $\boldsymbol{q} = \mathbf{7}$
- $\boldsymbol{\phi(n) = (p-1)(q-1) = 1 \times 6 = 6}$
- $\boldsymbol{ed} = 1 \bmod \boldsymbol{\phi(n)}$
- For $\boldsymbol{e} = \mathbf{5}$, $\boldsymbol{d} = \mathbf{5}$

$\mathbf{5 \times 1 \bmod 6 = 5}$

$\mathbf{5 \times 2 \bmod 6 = 4}$

$\mathbf{5 \times 3 \bmod 6 = 3}$

$\mathbf{5 \times 4 \bmod 6 = 2}$

$\mathbf{5 \times 5 \bmod 6 = 1}$

n=14; x=3; e=5; d=5

- x is the plaintext message (3)
- y is the ciphertext

Encryption:

$$\begin{aligned} y &= x^e \bmod n \\ &= 3^5 \bmod 14 \\ &= 243 \bmod 14 \\ &= 5 \end{aligned}$$

Decryption:

$$\begin{aligned} x &= y^d \bmod n \\ &= 5^5 \bmod 14 \\ &= 3125 \bmod 14 \\ &= 3 \end{aligned}$$

RSA Key Generation and Security

Key Generation

- Pick random primes $\boldsymbol{p}$ and $\boldsymbol{q}$
- Calculate $\boldsymbol{\phi(n)}$ from $\boldsymbol{p}$ and $\boldsymbol{q}$
- Pick $\boldsymbol{e}$
- Calculate $\boldsymbol{d}$ (inverse of $\boldsymbol{e}$)

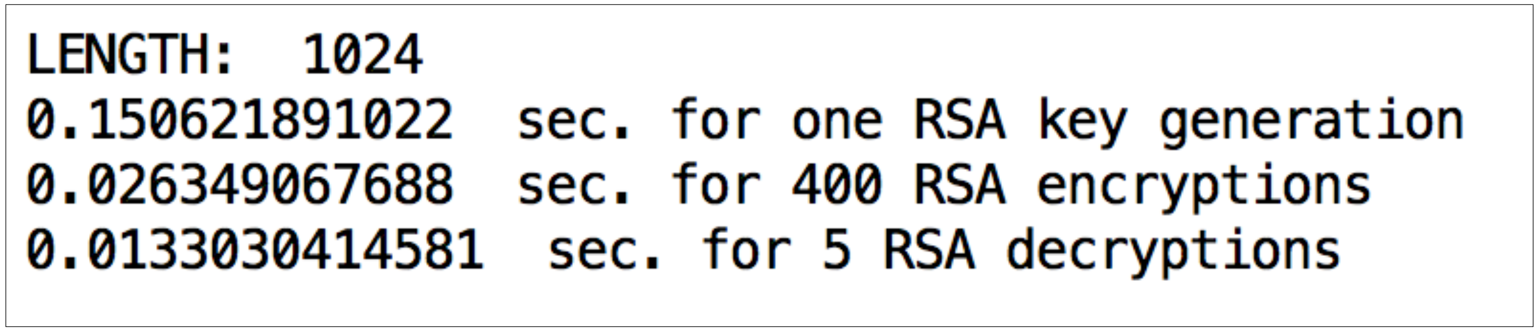
RSA in Python 3 Commands

```
python3 -m pip install pycryptodome

python3

from Crypto.PublicKey import RSA

from Crypto.Cipher import PKCS1_OAEP

key = RSA.generate(2048)

plaintext = b"encrypt this message"

cipher_rsa = PKCS1_OAEP.new(key)

ciphertext = cipher_rsa.encrypt(plaintext)

decrypted = cipher_rsa.decrypt(ciphertext)

print("Ciphertext:", ciphertext)

print("Decrypted:", decrypted)
```

RSA Encryption and Decryption in Python 3

```
>>> from Crypto.PublicKey import RSA
>>> from Crypto.Cipher import PKCS1_OAEP
>>>
>>> key = RSA.generate(2048)
>>>
>>> plaintext = b"encrypt this message"
>>> cipher_rsa = PKCS1_OAEP.new(key)
>>>
>>> ciphertext = cipher_rsa.encrypt(plaintext)
>>> decrypted = cipher_rsa.decrypt(ciphertext)
>>>
>>> print("Ciphertext:", ciphertext)
Ciphertext: b'[\xe5\xb2\x1fYR\xa5\x05\xfa\x04\xf8\x08\x8e\x0e}\x1e\x98\xc8w\xc5\
xadX\xff\xa9\xda\xbeP6\x8e7]GU\xf0\x83\xfb\xd3.D\x88\xa1\xc3\xcc\xd2\xd6\x97|\xa
3gX\xc1t\x10\xbf>P\x1b\x95\xcf\x0cr\xfd\x8f\xea\xc8\xc3\xd2\x93\xf4R\x94\xd2\x9e
\x0c\xaf\x15\xfe$\x8d\x93\xd2\x7f\xc4aK\xe1(+\xdc\x0c\x8a\xd3\n\xcb\n#\x1aC\xe1"
\xc6\xcb\'U\'\x9e\xdb|\xd7\xdd\\m"\x05\x1c\xac\xb2\xd8R\xad\x15\n\x00\x8a\xfcQ\x
c0\xa47\xcd\x1aD{\xa65\n2\x9e\x8a\x9f/\xba\xc9\x109\xf9b\xd9E\x11\x87%v\xec\xfe\
xa3\x8d\x00\x91X\xb2\xf6{\x15a\xac\xeb\xb7\x88\xe6RM\xa4\xfd\x9f2\xb0\xeb3H\x1b&
\xec\t-c2\xd4\xf8\x9b\x19<\xad0\x90\xccd\x9b\xa7RBf\x14\x9b\xc8\xdd\xc0\x00@\xb1
-\x1cu\xa2b\xa8\x8eT9c\xf8\x06\x9b\xee\xaa\x93\x9e\r\x05;\x9cS\x8c\xcb\xb8\xa3\x
ba|\x8a\x1f\x92\x93a.\xc9\xa7\x8f2(;\x8a\x86\x03\xdf\x0b\x01'
>>> print("Decrypted:", decrypted)
Decrypted: b'encrypt this message'
```

Speed of Calculations



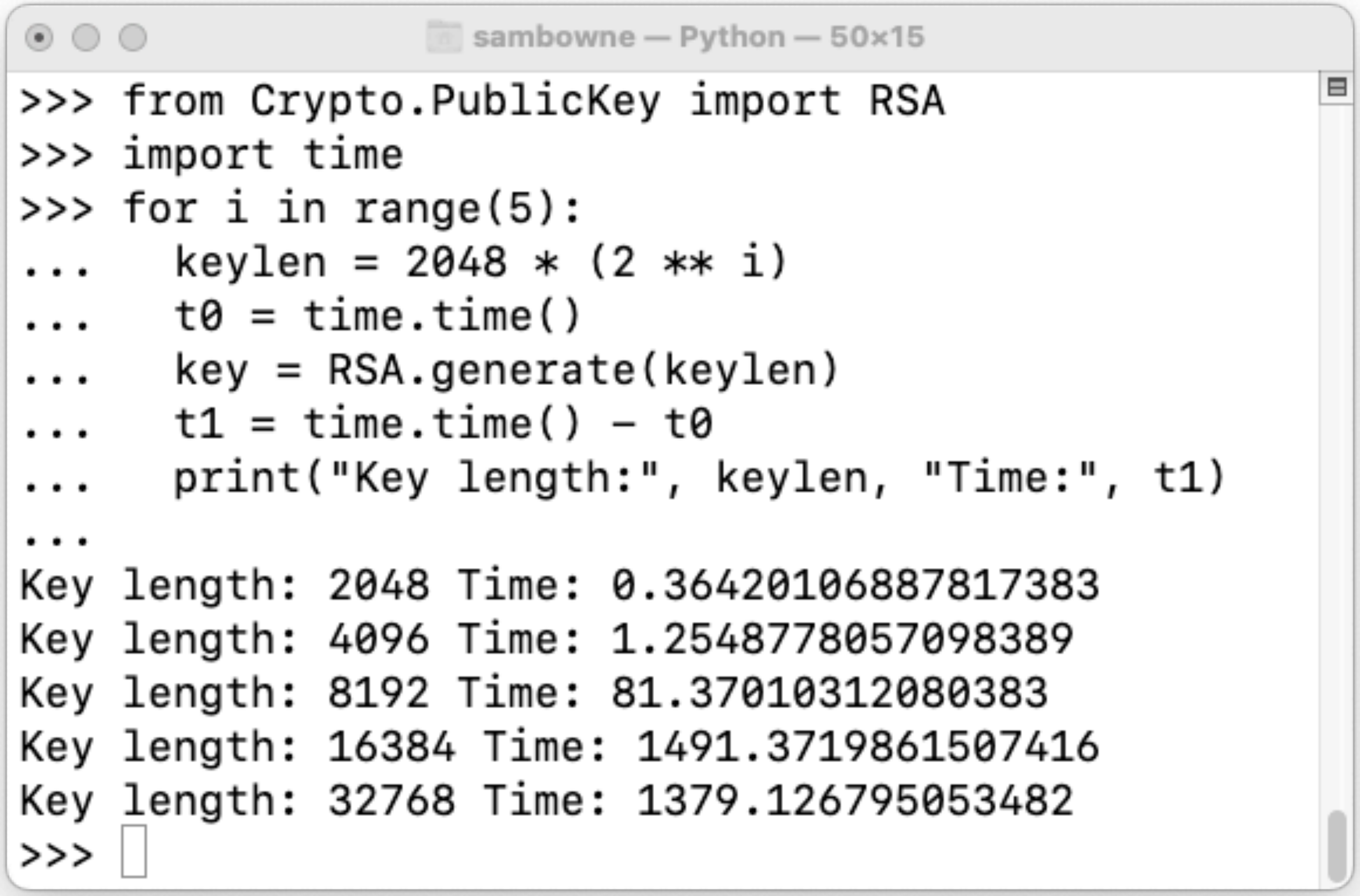
```
LENGTH:  1024
0.150621891022  sec. for one RSA key generation
0.026349067688  sec. for 400 RSA encryptions
0.0133030414581  sec. for 5 RSA decryptions
```

- Encryption is **fastest**
- Decryption is **much slower**
- Key generation is **slowest**

RSA in Python 3 Commands

```
from Crypto.PublicKey import RSA

import time

for i in range(5):

  keylen = 2048 * (2 ** i)

  t0 = time.time()

  key = RSA.generate(keylen)

  t1 = time.time() - t0

  print("Key length:", keylen, "Time:", t1)
```

Key Generation Times



```
>>> from Crypto.PublicKey import RSA
>>> import time
>>> for i in range(5):
...   keylen = 2048 * (2 ** i)
...   t0 = time.time()
...   key = RSA.generate(keylen)
...   t1 = time.time() - t0
...   print("Key length:", keylen, "Time:", t1)
...
Key length: 2048 Time: 0.36420106887817383
Key length: 4096 Time: 1.2548778057098389
Key length: 8192 Time: 81.37010312080383
Key length: 16384 Time: 1491.3719861507416
Key length: 32768 Time: 1379.126795053482
>>> 
```

Encrypting with RSA

Used with AES

- RSA typically not used to encrypt plaintext directly
- RSA is used to encrypt an AES private key

Textbook RSA

- Plaintext converted to ASCII bytes
- Placed in $\boldsymbol{x}$
- RSA used to compute $\boldsymbol{y} = \boldsymbol{x}^{e} \bmod \boldsymbol{n}$

Malleability

- Encrypt two plaintext messages x_1 and x_2
  - $y_1 = x_1^e \bmod n$
  - $y_2 = x_2^e \bmod n$
- Consider the plaintext $(x_1)(x_2)$
  - Multiplying x_1 and x_2 together
    - $y = (x_1^e \bmod n)(x_2^e \bmod n) = (y_1)(y_2)$
- An attacker can create valid ciphertext without the key

Strong RSA Encryption: OAEP

- Optimal Asymmetric Encryption Padding
- Padded plaintext is as long as ***n***
- Includes extra data and randomness

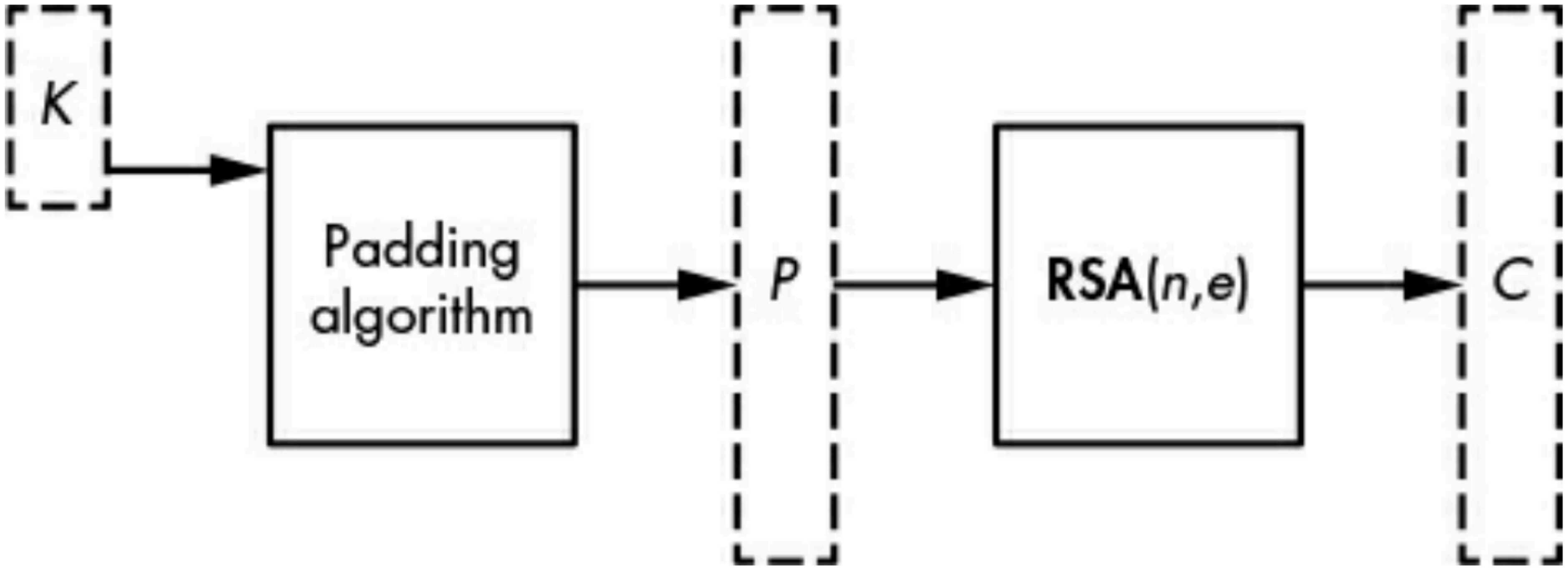
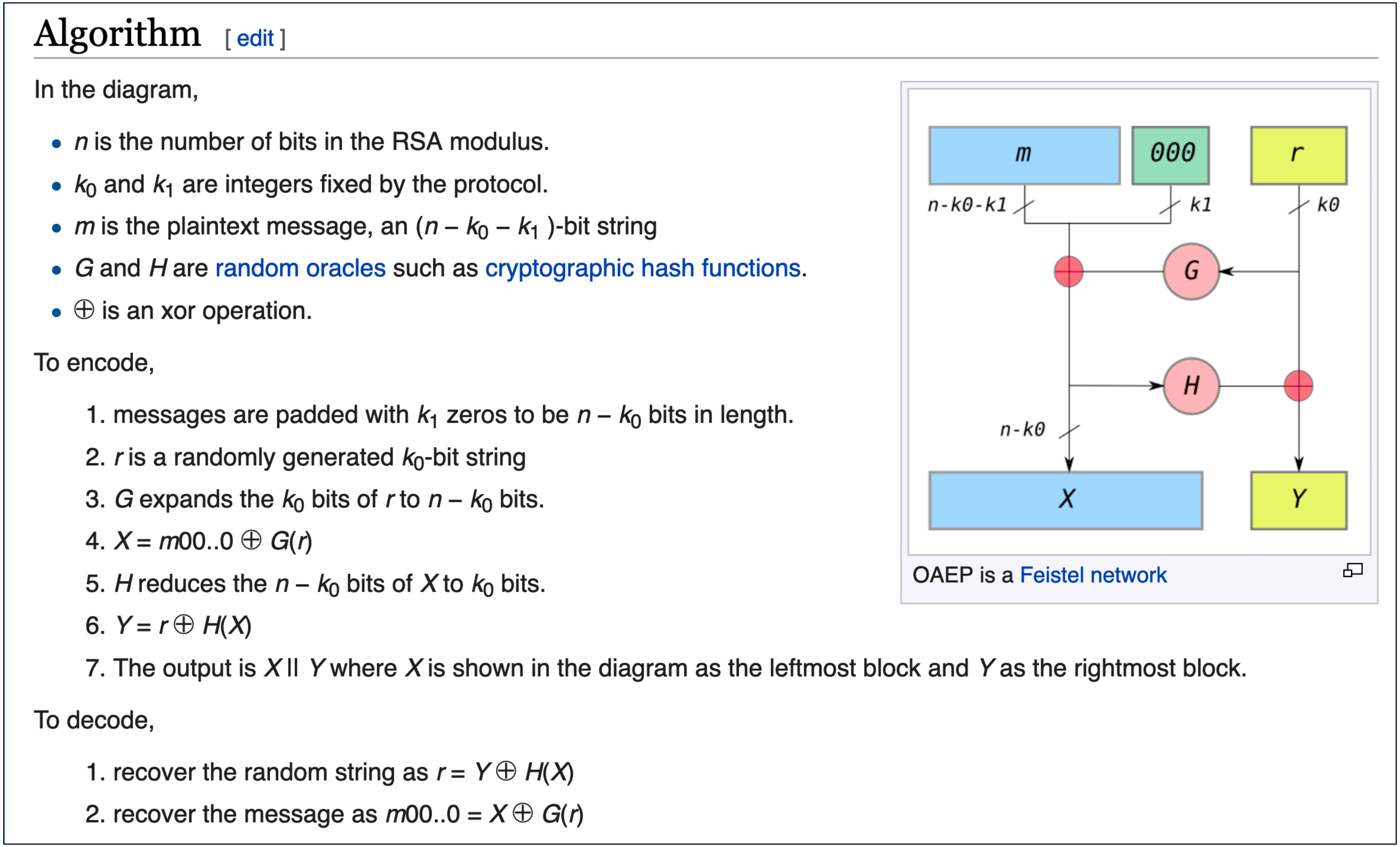


Figure 10-1: Encrypting a symmetric key, K, *with RSA using* (n, e) *as a public key*



Algorithm [edit]

In the diagram,

- n is the number of bits in the RSA modulus.
- k_0 and k_1 are integers fixed by the protocol.
- m is the plaintext message, an $(n - k_0 - k_1)$-bit string
- G and H are random oracles such as cryptographic hash functions.
- $\oplus$ is an xor operation.

To encode,

1. messages are padded with k_1 zeros to be $n - k_0$ bits in length.
2. r is a randomly generated k_0-bit string
3. G expands the k_0 bits of r to $n - k_0$ bits.
4. $X = m00..0 \oplus G(r)$
5. H reduces the $n - k_0$ bits of X to k_0 bits.
6. $Y = r \oplus H(X)$
7. The output is $X \| Y$ where X is shown in the diagram as the leftmost block and Y as the rightmost block.

To decode,

1. recover the random string as $r = Y \oplus H(X)$
2. recover the message as $m00..0 = X \oplus G(r)$

OAEP is a Feistel network

PKCS#1 v1.5

- An old method created by RSA
- Message ***m*** is padded to
  - 0x00||0x02||***r***||0x00||***m***
  - Where ***r*** is a random string
- Much less secure than OAEP
- Used in many systems
  - https://crypto.stackexchange.com/questions/66521/why-does-adding-pkcs1-v1-5-padding-make-rsa-encryption-non-deterministic

Signing with RSA

Digital Signatures

- Sign a message x with $y = x^d \bmod n$
- No one can forge the signature because d is secret
- Everyone can verify the signature using e
- $x = y^e \bmod n$
- Notice that x is not secret
- Signatures prevent forgeries, they don't provide confidentiality

Signing a Hash

- Signing long messages is slow and uncommon
- Typically the message is hashed
- Then the hash is signed

Textbook RSA Signatures

- Sign a message $\boldsymbol{x}$ with $\boldsymbol{y} = \boldsymbol{x}^d \bmod \boldsymbol{n}$
- Attacker can forge signatures
    - For $\boldsymbol{x} = 0$, 1, or $\boldsymbol{n} - 1$

Blinding Attack

- You want to get the signature for message M
  - Which the targeted user would never willingly sign
- Find a value R such that R^eM mod n
  - Is a message the user will sign
  - That signature S is $(R^eM)^d$ mod n
    - = $R^{ed}M^d$ mod n = RM^d mod n
  - The signature we want is M^d mod n = S/R

The PSS Signature Standard

- Probabilistic Signature Standard
- Makes signatures more secure, the way OAEP makes encryption more secure
- Combines the message with random and fixed bits

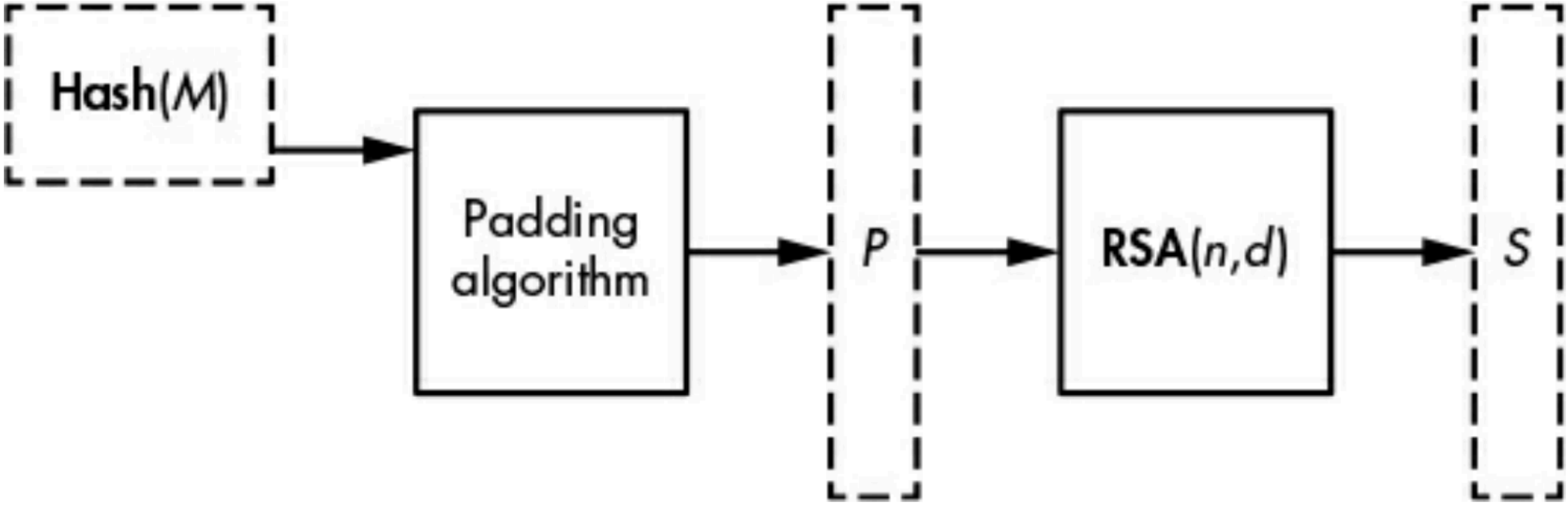


Figure 10-3: Signing a message, M, *with RSA and with the PSS standard, where* (n, d) *is the private key*

Full Domain Hash Signatures (FDH)

- The simplest scheme
- x = Hash(message)
- Signature $y = x^e \bmod n$

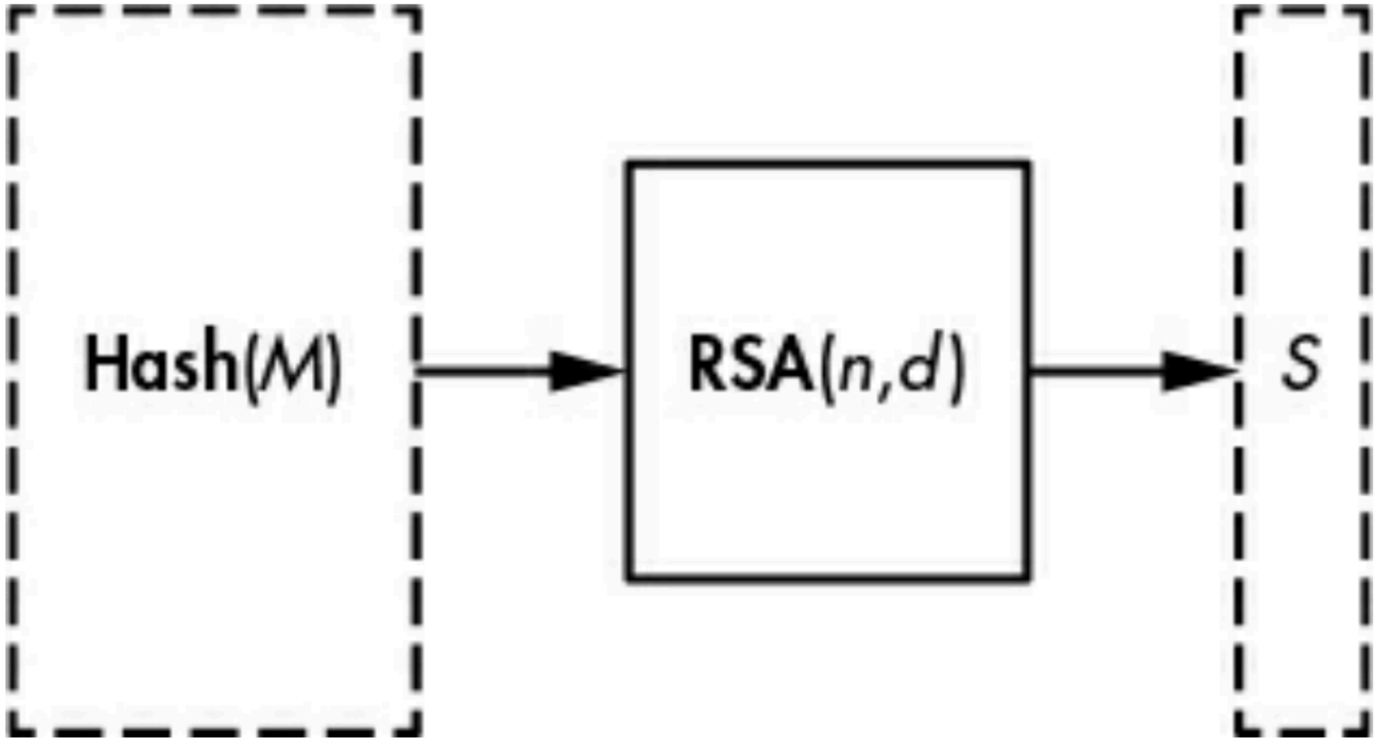


Figure 10-4: Signing a message with RSA using the Full Domain Hash technique

FDH v. PSS

- PSS came later
    - More proof of theoretical security
    - Because of added randomness
- But in practice they are similar in security
- But PSS is safer against ***fault attacks***
    - Explained later

RSA Implementations

Just Use Libraries

- Much easier and safer than writing your own implementation

Square-and-Multiply

$$y = e_{k_{pub}}(x) \equiv x^e \bmod n \quad \text{(encryption)}$$

$$x = d_{k_{pr}}(y) \equiv y^d \bmod n \quad \text{(decryption)}$$

- Consider RSA with a 1024-bit key
- We need to calculate x^e where e is 1024 bits long
- x * x * x * x .... 2^{1024} multiplications
- Competely impossible -- we can't even crack a 72-bit key yet (2^{72} calculations)

Square-and-Multiply

- Use memory to save time
- Do these ten multiplications
  - x2 = x * x
  - x4 = x2 * x2
  - x8 = x4 * x4
  - x16 = x8 * x8
  - ...
  - x1024 = x512 * x512
  - ...
- Combine the results to make any exponent

Square-and-Multiply

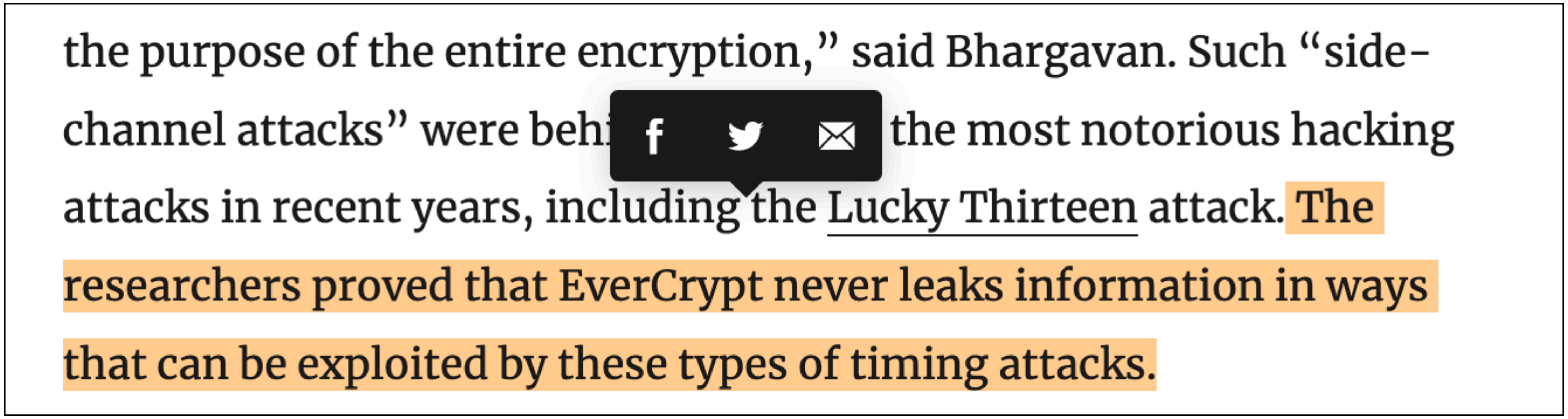
- With this trick, a 1024-bit exponent can be calculated with only 1536 multiplications
- But each number being multiplied is 1024 bits long, so it still takes a lot of CPU

Side-Channel Attacks

- The speedup from square-and-multiply means that exponent bits of 1 take more time than exponent bits of 0
- Measuring power consumption or timing can leak out information about the key
- Few libraries are protected from such attacks

Cryptography That Can't Be Hacked

- EverCrypt -- a library immune to timing attacks
  - From Microsoft research
  - Link Ch 10b



the purpose of the entire encryption," said Bhargavan. Such "side-channel attacks" were behi the most notorious hacking attacks in recent years, including the Lucky Thirteen attack. The researchers proved that EverCrypt never leaks information in ways that can be exploited by these types of timing attacks.

Small ***e*** for Faster Encryption

  - Encryption: $\boldsymbol{y} = \boldsymbol{x}^{e} \bmod \mathbf{n}$
  - Decryption: $\boldsymbol{x} = \boldsymbol{y}^{d} \bmod \mathbf{n}$
- Choosing small ***e*** makes encryption faster, but decryption slower

Chinese Remainder Theorem

- Replaces one operation mod ***n***
  - Encryption: ***y*** = x^e mod **n**
- With two operations mod ***p*** and ***q***
  - Making RSA four times faster

$$x = x_p \times q \times (1/q \bmod p) + x_q \times p \times (1/p \bmod q) \bmod n$$

How Things Can Go Wrong

Bellecore Attack on RSA-CRT

- ***Fault injection*** causes the CPU to make errors
  - Alter the power supply
  - or hit chips with a laser pulse
- Can break deterministic schemes like CRT
  - But not ones including randomness like PSS

Kahoot!