

# The RSA Trapdoor Permutation

# Recap

Public key encryption: (Gen, E, D)

$$\text{Gen}() \rightarrow (\text{pk}, \text{sk}), \quad \text{E}(\text{pk}, m) \rightarrow c, \quad \text{D}(\text{sk}, c) \rightarrow m$$



Constructions: (1) ElGamal encryption, (2) today: RSA

Security from last lecture:

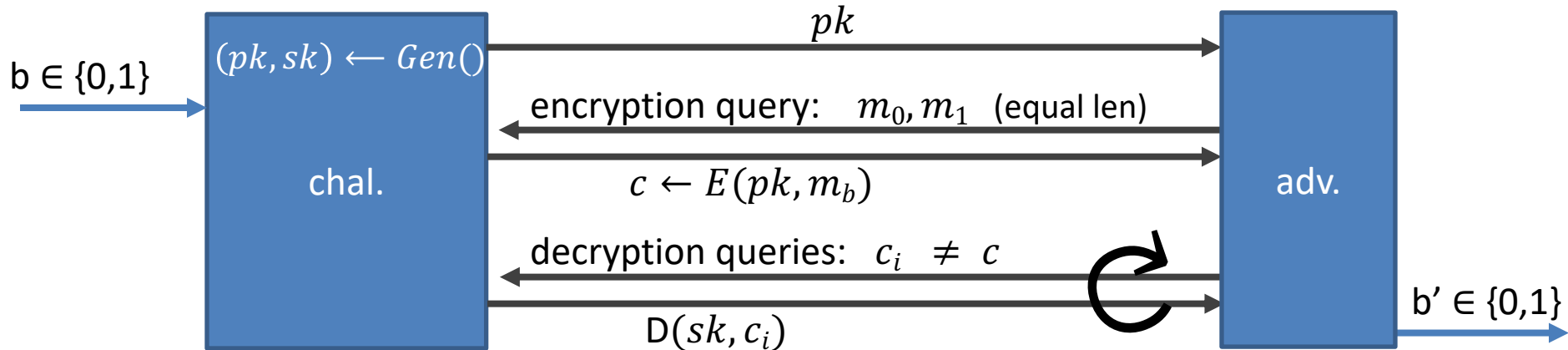
**semantic security against an eavesdropper**

In practice security against eavesdropping is insufficient:

**adversary can make up ciphertexts  
and see how recipient reacts**

# Security against chosen ciphertext attacks (CCA)

A PKE  $(\text{Gen}, E, D)$  is chosen-ciphertext secure if no "efficient" adversary can win the following game with non-negl. advantage:



**Thm:** ElGamal encryption from last lecture is CCA secure assuming interactive-CDH in  $G$  holds, and  $H$  is modeled as a random oracle

# Recap

Public key encryption: (Gen, E, D)

$$\text{Gen}() \rightarrow (\text{pk}, \text{sk}), \quad \text{E}(\text{pk}, m) \rightarrow c, \quad \text{D}(\text{sk}, c) \rightarrow m$$


Security: **semantic security against a chosen-ciphertext attack**

- Semantic security against adv. that can issue decryption queries

Constructions: (1) ElGamal encryption, (2) today: RSA

... but first: **trapdoor functions**

# Trapdoor functions (TDF)

**Def:** a trapdoor func.  $X \rightarrow Y$  is a triple of efficient algs.  $(\text{Gen}, F, F^{-1})$

- $\text{Gen}()$ : randomized alg. outputs a key pair  $(pk, sk)$
- $F(pk, \cdot)$ : det. alg. that defines a function  $X \rightarrow Y$
- $F^{-1}(sk, \cdot)$ : defines a function  $Y \rightarrow X$  that inverts  $F(pk, \cdot)$

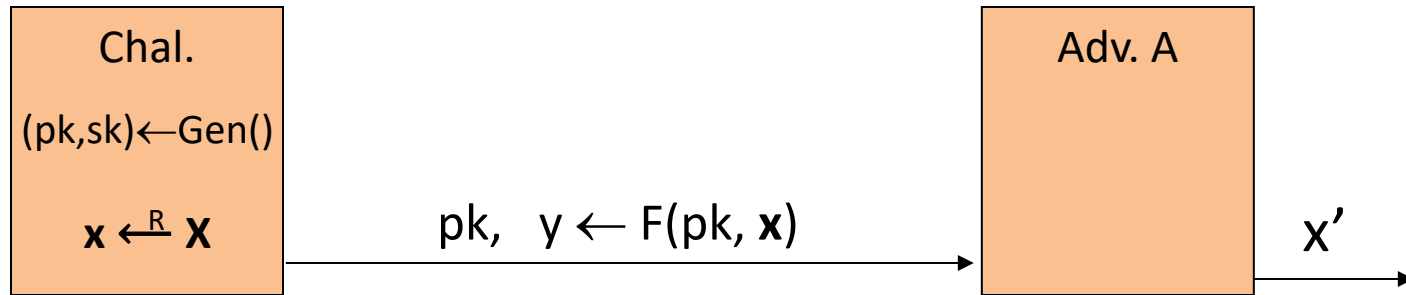
More precisely:  $\forall (pk, sk)$  output by  $\text{Gen}$

$$\forall x \in X: F^{-1}(sk, F(pk, x)) = x$$

# Secure Trapdoor Functions (TDFs)

$(\text{Gen}, F, F^{-1})$  is secure if  $F(\text{pk}, \cdot)$  is a “one-way” function:

can be evaluated, but cannot be inverted without  $\text{sk}$



**Def:**  $(\text{Gen}, F, F^{-1})$  is a secure TDF if for all efficient  $A$ :

$$\text{Adv}_{\text{OW}}[A, F] = \Pr[\mathbf{x} = \mathbf{x}'] < \text{negligible}$$

# Public-key encryption from TDFs

- $(\text{Gen}, F, F^{-1})$ : secure TDF  $X \rightarrow Y$
- $(E_s, D_s)$ : symmetric auth. encryption defined over  $(K, M, C)$
- $H: X \rightarrow K$  a hash function

We construct a pub-key enc. system  $(\text{Gen}, E, D)$ :

Key generation  $\text{Gen}$ : same as  $\text{Gen}$  for TDF

# Public-key encryption from TDFs

- $(\text{Gen}, F, F^{-1})$ : secure TDF  $X \rightarrow Y$
- $(E_s, D_s)$ : symmetric auth. encryption defined over  $(K, M, C)$
- $H: X \rightarrow K$  a hash function

$E(pk, m)$  :

$$x \xleftarrow{R} X, \quad y \leftarrow F(pk, x)$$

$$k \leftarrow H(x), \quad c \leftarrow E_s(k, m)$$

output  $(y, c)$

$D(sk, (y, c))$  :

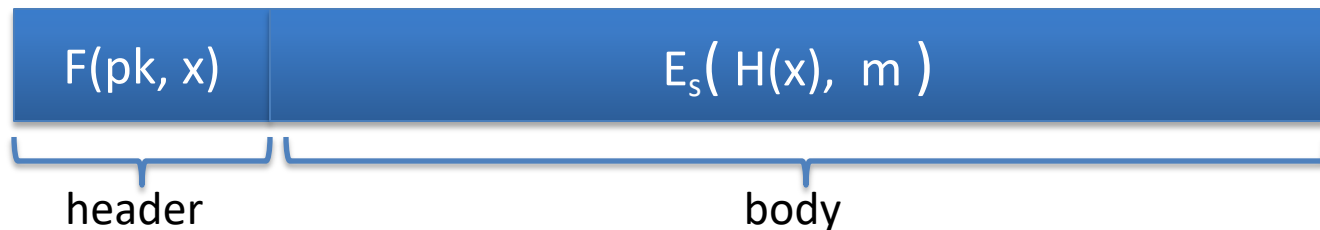
$$x \leftarrow F^{-1}(sk, y),$$

$$k \leftarrow H(x), \quad m \leftarrow D_s(k, c)$$

output  $m$



In pictures:



### Security Theorem:

If  $(\text{Gen}, F, F^{-1})$  is a secure TDF,  $(E_s, D_s)$  provides auth. enc.  
and  $H: X \rightarrow K$  is a “random oracle”  
then  $(\text{Gen}, E, D)$  is  $\text{CCA}^{\text{ro}}$  secure.

# Incorrect use of a Trapdoor Function (TDF)

**Never** encrypt by applying  $F$  directly to plaintext:

$E(pk, m)$  :

output  $c \leftarrow F(pk, m)$

$D(sk, c)$  :

output  $F^{-1}(sk, c)$

Problems:

- Deterministic: cannot be semantically secure !!
- Many attacks exist (coming)

# The RSA trapdoor permutation

# Review: arithmetic mod composites

Let  $N = p \cdot q$  where  $p, q$  are prime

$$\mathbb{Z}_N = \{0, 1, 2, \dots, N-1\} \quad ; \quad (\mathbb{Z}_N)^* = \{\text{invertible elements in } \mathbb{Z}_N\}$$

Facts:  $x \in \mathbb{Z}_N$  is invertible  $\iff \gcd(x, N) = 1$

– Number of elements in  $(\mathbb{Z}_N)^*$  is  $\varphi(N) = (p-1)(q-1) = N - p - q + 1$

Euler's thm:

$$\forall x \in (\mathbb{Z}_N)^* : x^{\varphi(N)} = 1$$

# The RSA trapdoor permutation

First published: Scientific American, Aug. 1977.

Applications:

- HTTPS: web certificates (... but less and less so)
- deprecated for key exchange in TLS 1.3

# The RSA trapdoor permutation

**Gen()**: choose random distinct primes  $p, q \approx 1024$  bits.

Set  $\mathbf{N} = \mathbf{pq}$ .

choose integers  $\mathbf{e}, \mathbf{d}$  s.t.  $\mathbf{e} \cdot \mathbf{d} = \mathbf{1} \pmod{\varphi(\mathbf{N})}$

output  $\mathbf{pk} = (\mathbf{N}, \mathbf{e})$  ,  $\mathbf{sk} = (\mathbf{N}, \mathbf{d})$

---

$\mathbf{F}(\mathbf{pk}, \mathbf{x}) : \mathbb{Z}_N^* \rightarrow \mathbb{Z}_N^*$  ;  $\mathbf{RSA}(\mathbf{x}) = \mathbf{x}^e$  (in  $\mathbb{Z}_N$ )

---

$\mathbf{F}^{-1}(\mathbf{sk}, \mathbf{y}) = \mathbf{y}^d$  ;  $\mathbf{y}^d = \mathbf{RSA}(\mathbf{x})^d = \mathbf{x}^{ed} = \mathbf{x}^{k\varphi(\mathbf{N})+1} = \left(\mathbf{x}^{\varphi(\mathbf{N})}\right)^k \cdot \mathbf{x} = \mathbf{x}$

# The RSA assumption

$\text{RSA}_e$  assumption: RSA with exp.  $e$  is a one-way permutation

For all efficient algs.  $A$ :

$$\Pr \left[ A(N, e, \mathbf{y}) = \mathbf{y}^{1/e} \right] < \text{negligible}$$

where  $p, q \xleftarrow{R} n\text{-bit primes}$ ,  $N \leftarrow pq$ ,  $\mathbf{y} \xleftarrow{R} \mathbb{Z}_N^*$

# RSA pub-key encryption (ISO std)

$(E_s, D_s)$ : symmetric enc. scheme providing auth. encryption.

$H: \mathbb{Z}_N \rightarrow K$  where  $K$  is key space of  $(E_s, D_s)$

- **Gen()**: generate RSA params:  $pk = (N, e)$ ,  $sk = (N, d)$
- **E(pk, m)**:
  - (1) choose random  $x$  in  $\mathbb{Z}_N^*$
  - (2)  $y \leftarrow \text{RSA}(x) = x^e$ ,  $k \leftarrow H(x)$
  - (3) output  $(y, E_s(k, m))$
- **D(sk, (y, c))**: output  $D_s(H(\text{RSA}^{-1}(y)), c)$



# Textbook RSA is insecure

Textbook RSA encryption:

– public key:  $(N, e)$

Encrypt:  $c \leftarrow m^e$  (in  $\mathbb{Z}_N$ )

– secret key:  $(N, d)$

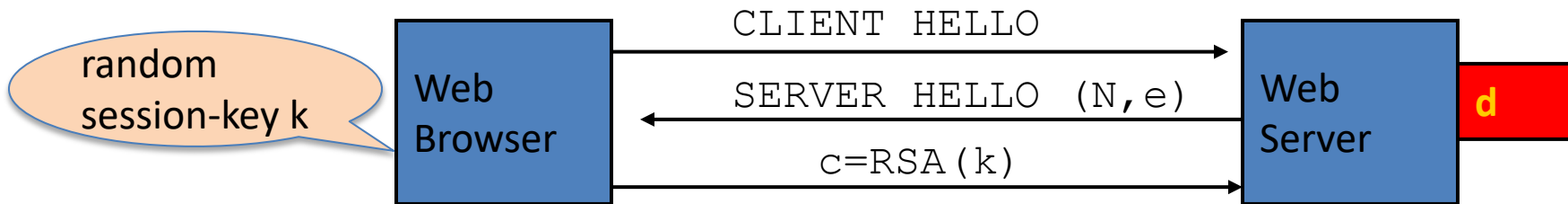
Decrypt:  $c^d \rightarrow m$

Insecure cryptosystem !!

– Is not semantically secure and many attacks exist

⇒ The RSA trapdoor permutation is not an encryption scheme !

# A simple attack on textbook RSA



Suppose  $k$  is 64 bits:  $k \in \{0, \dots, 2^{64}\}$ . Eve sees:  $c = k^e$  in  $\mathbb{Z}_N$

If  $k = k_1 \cdot k_2$  where  $k_1, k_2 < 2^{34}$  (prob.  $\approx 20\%$ ) then  $c/k_1^e = k_2^e$  in  $\mathbb{Z}_N$

Step 1: build table:  $c/1^e, c/2^e, c/3^e, \dots, c/2^{34e}$ . time:  $2^{34}$

Step 2: for  $k_2 = 0, \dots, 2^{34}$  test if  $k_2^e$  is in table. time:  $2^{34}$

Output matching  $(k_1, k_2)$ .

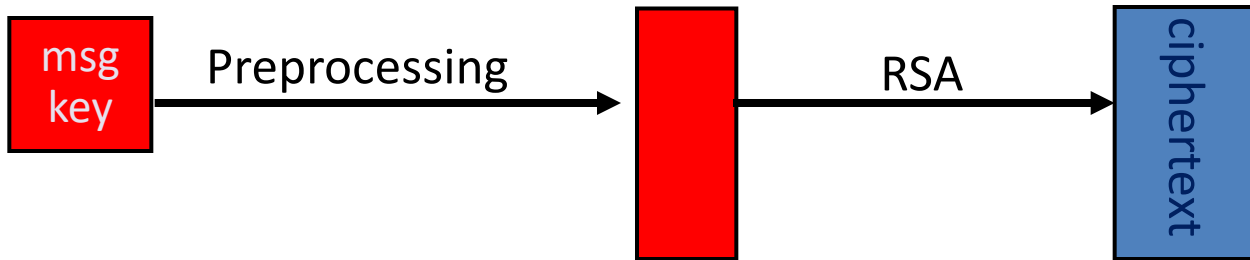
Total attack time:  $\approx 2^{34} \ll 2^{64}$

# RSA in practice

# RSA encryption in practice

Never use textbook RSA.

RSA in practice (since ISO standard is not often used) :

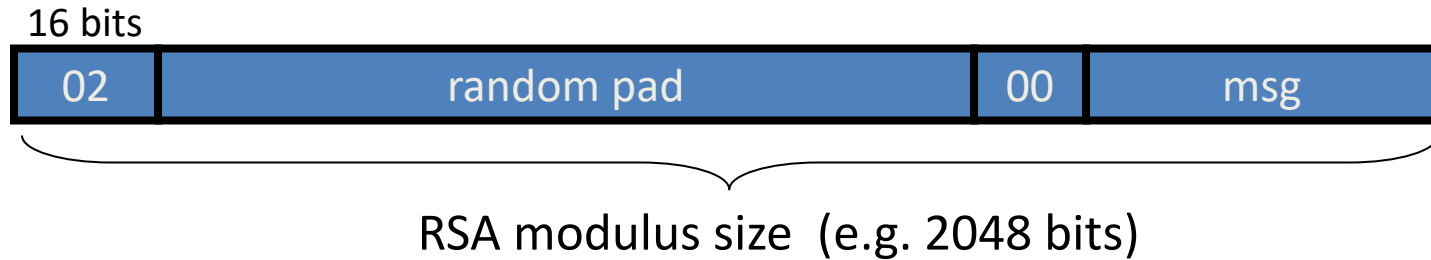


Main questions:

- How should the preprocessing be done?
- Can we argue about security of resulting system?

# PKCS1 v1.5

PKCS1 mode 2: (encryption)

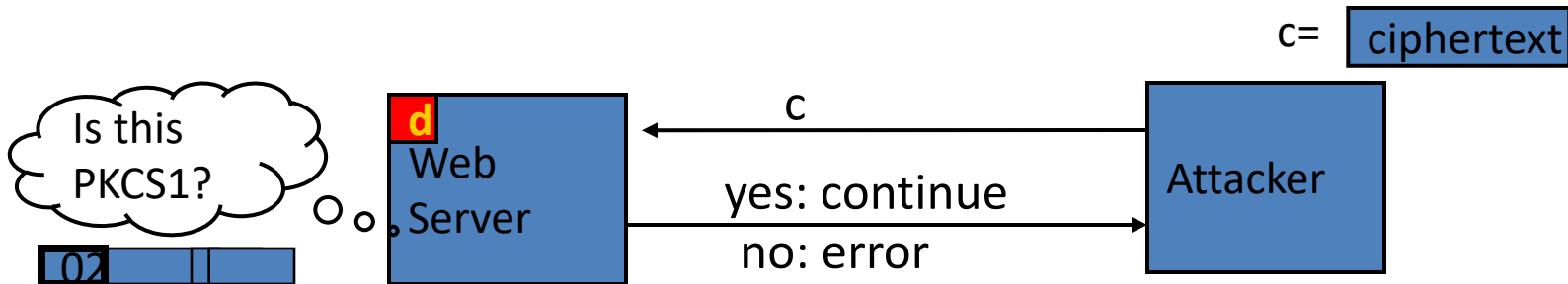


- Resulting value is RSA encrypted
- Widely deployed, e.g. in HTTPS (TLS 1.2) -- now deprecated

# Attack on PKCS1 v1.5

(Bleichenbacher 1998)

PKCS1 used in HTTPS:



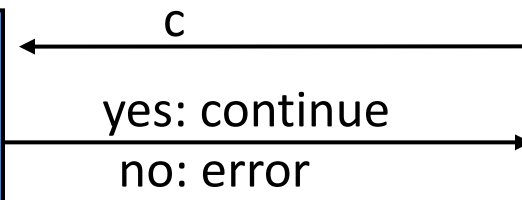
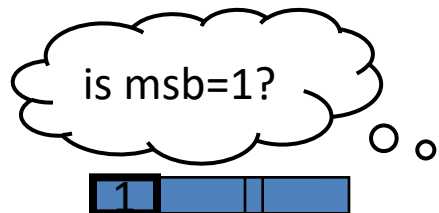
⇒ attacker can test if 16 MSBs of plaintext = '02'

Chosen-ciphertext attack: to decrypt a given ciphertext  $c$  do:

- Choose  $r \in \mathbb{Z}_N$ . Compute  $c' \leftarrow r^e \cdot c = (r \cdot \text{PKCS1}(m))^e$
- Send  $c'$  to web server and use response

# Baby Bleichenbacher

compute  $x \leftarrow c^d$  in  $\mathbb{Z}_N$



$c =$  ciphertext

Suppose  $N$  is  $N = 2^n$  (an invalid RSA modulus). Then:

- Sending  $c$  reveals  $\text{msb}(x)$
  - Sending  $2^e \cdot c = (2x)^e$  in  $\mathbb{Z}_N$  reveals  $\text{msb}(2x \bmod N) = \text{msb}_2(x)$
  - Sending  $4^e \cdot c = (4x)^e$  in  $\mathbb{Z}_N$  reveals  $\text{msb}(4x \bmod N) = \text{msb}_3(x)$
- ... and so on to reveal all of  $x$

# HTTPS Defense (RFC 5246)

*Attacks discovered by Bleichenbacher and Klima et al. ... can be avoided by treating incorrectly formatted message blocks ... in a manner indistinguishable from correctly formatted RSA blocks.  
In other words:*

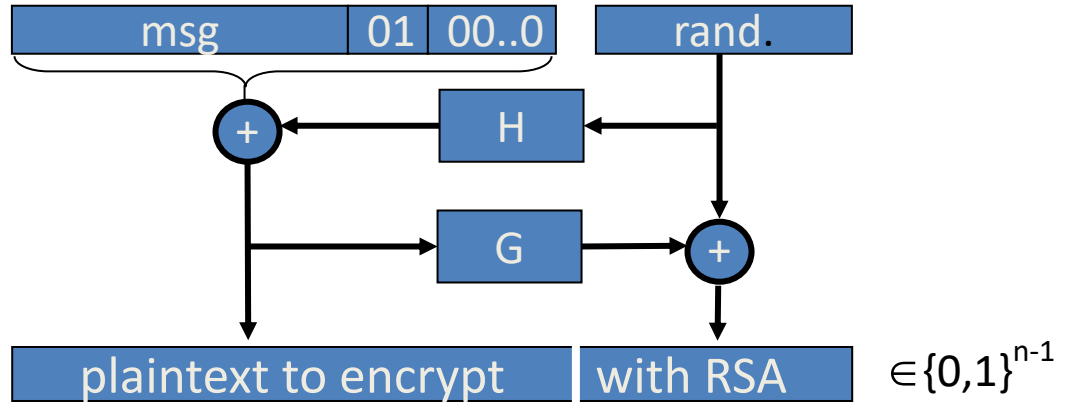
- 1. Generate a string **R** of 46 random bytes*
- 2. Decrypt the message to recover the plaintext **M***
- 3. If the PKCS#1 padding is not correct*  
$$pre\_master\_secret = \mathbf{R}$$



# PKCS1 v2.0: OAEP

New preprocessing function: OAEP [BR94]

check pad  
on decryption.  
reject CT if invalid.



**Thm** [FOPS'01]: RSA is a trap-door permutation  $\Rightarrow$   
RSA-OAEP is CCA secure when `H,G` are *random oracles*

in practice: use SHA-256 for `H` and `G`

# Subtleties in implementing OAEP

[M '00]

```
OAEP-decrypt(ct):  
    error = 0;  
    .....  
    if (  $\text{RSA}^{-1}(\text{ct}) > 2^{n-1}$  )  
        { error = 1; goto exit; }  
    .....  
    if (  $\text{pad}(\text{OAEP}^{-1}(\text{RSA}^{-1}(\text{ct}))) \neq \text{"01000"}$ )  
        { error = 1; goto exit; }
```

Problem: timing information leaks type of error

⇒ Attacker can decrypt any ciphertext

Lesson: Don't implement RSA-OAEP yourself !

Is RSA a one-way function?

---

# Is RSA a one-way permutation?

To invert the RSA one-way func. (without  $d$ ) attacker must compute:

$$x \text{ from } c = x^e \pmod{N}.$$

How hard is computing  $e$ 'th roots modulo  $N$  ??

Best known algorithm:

- Step 1: factor  $N$  (hard)
- Step 2: compute  $e$ 'th roots modulo  $p$  and  $q$  (easy)

# Shortcuts?

Must one factor  $N$  in order to compute  $e$ 'th roots?

To prove no shortcut exists show a reduction:

- Efficient algorithm for  $e$ 'th roots mod  $N$   
 $\Rightarrow$  efficient algorithm for factoring  $N$ .
- Oldest open problem in public key cryptography.

Some evidence no reduction exists: (BV'98)

- “Algebraic” reduction  $\Rightarrow$  factoring is easy.

# How **not** to improve RSA's performance

To speed up RSA decryption use small private key  $d$  ( $d \approx 2^{128}$ )

$$c^d = m \pmod{N}$$

Wiener'87: if  $d < N^{0.25}$  then RSA is insecure.

BD'98: if  $d < N^{0.292}$  then RSA is insecure (open:  $d < N^{0.5}$ )

Insecure: priv. key  $d$  can be found from  $(N,e)$

# Wiener's attack

Recall:  $e \cdot d = 1 \pmod{\varphi(N)} \Rightarrow \exists k \in \mathbb{Z} : e \cdot d = k \cdot \varphi(N) + 1$

$$\left| \frac{e}{\varphi(N)} - \frac{k}{d} \right| = \frac{1}{d \cdot \varphi(N)} \leq \frac{1}{\sqrt{N}}$$

# Wiener's attack

Recall:  $e \cdot d = 1 \pmod{\varphi(N)} \Rightarrow \exists k \in \mathbb{Z}: e \cdot d = k \cdot \varphi(N) + 1$

$$\left| \frac{e}{\varphi(N)} - \frac{k}{d} \right| = \frac{1}{d \cdot \varphi(N)} \leq \frac{1}{\sqrt{N}}$$

---

$$\varphi(N) = N - p - q + 1 \Rightarrow |N - \varphi(N)| \leq p + q \leq 3\sqrt{N}$$



# Wiener's attack

Recall:  $e \cdot d = 1 \pmod{\varphi(N)} \Rightarrow \exists k \in \mathbb{Z}: e \cdot d = k \cdot \varphi(N) + 1$

$$\left| \frac{e}{\varphi(N)} - \frac{k}{d} \right| = \frac{1}{d \cdot \varphi(N)} \leq \frac{1}{\sqrt{N}}$$

---

$$\varphi(N) = N - p - q + 1 \Rightarrow |N - \varphi(N)| \leq p + q \leq 3\sqrt{N} \leq \sqrt{N}$$

$$d \leq N^{0.25}/3 \Rightarrow \left| \frac{e}{N} - \frac{k}{d} \right| \leq \left| \frac{e}{N} - \frac{e}{\varphi(N)} \right| + \left| \frac{e}{\varphi(N)} - \frac{k}{d} \right| \leq \frac{1}{2d^2}$$

# Wiener's attack

Recall:  $e \cdot d = 1 \pmod{\varphi(N)} \Rightarrow \exists k \in \mathbb{Z}: e \cdot d = k \cdot \varphi(N) + 1$

$$\left| \frac{e}{\varphi(N)} - \frac{k}{d} \right| = \frac{1}{d \cdot \varphi(N)} \leq \frac{1}{\sqrt{N}}$$

---


$$\varphi(N) = N - p - q + 1 \Rightarrow |N - \varphi(N)| \leq p + q \leq 3\sqrt{N} \leq \sqrt{N}$$

$$d \leq N^{0.25}/3 \Rightarrow \left| \frac{e}{N} - \frac{k}{d} \right| \leq \underbrace{\left| \frac{e}{N} - \frac{e}{\varphi(N)} \right|}_{\leq \frac{1}{\sqrt{N}}} + \left| \frac{e}{\varphi(N)} - \frac{k}{d} \right| \leq \frac{1}{2d^2}$$

$$\frac{1}{2d^2} - \frac{1}{\sqrt{N}} \geq \frac{1}{2 \cdot 10^3}$$

$$\leq \frac{3\sqrt{N}}{N} \cdot \frac{e}{\varphi(N)} \leq \frac{3}{\sqrt{N}} \leq \frac{1}{2d^2} - \frac{1}{\sqrt{N}}$$

→ Continued fraction expansion of  $e/N$  gives  $k/d$ .

$e \cdot d = 1 \pmod{k} \Rightarrow \gcd(d, k) = 1 \Rightarrow$  can find  $d$  from  $k/d$

# RSA With Low public exponent

To speed up RSA encryption use a small  $e$ :  $c = m^e \pmod{N}$

- Minimum value:  **$e=3$**  ( $\gcd(e, \phi(N)) = 1$ )
- Recommended value:  **$e=65537=2^{16}+1$**

Encryption: 17 multiplications

Asymmetry of RSA: fast enc. / slow dec.

– ElGamal: approx. same time for both.

# Key lengths

Security of public key system should be comparable to security of symmetric cipher:

<u>Cipher key-size</u>	<u>RSA Modulus size</u>	<u>Elliptic Curve Modulus size</u>
80 bits	1024 bits	160 bits
128 bits	3072 bits	256 bits
256 bits (AES)	<b><u>15360</u> bits</b>	512 bits

Best factoring algorithm (GNFS): n-bits integer, time  $\approx \exp(n^{1/3})$

# Implementation attacks

**Timing attack:** [Kocher et al. 1997] , [BB'04]

The time it takes to compute  $c^d \pmod{N}$  can expose  $d$

**Power attack:** [Kocher et al. 1999]

The power consumption of a smartcard while it is computing  $c^d \pmod{N}$  can expose  $d$ .

**Faults attack:** [BDL'97]

A computer error during  $c^d \pmod{N}$  can expose  $d$ .

A common defense: check output. 10% slowdown.

# An Example Fault Attack on RSA (CRT)

A common implementation of RSA decryption:  $x = c^d$  in  $\mathbb{Z}_N$

decrypt mod  $p$ :  $x_p = c^d$  in  $\mathbb{Z}_p$   
decrypt mod  $q$ :  $x_q = c^d$  in  $\mathbb{Z}_q$  } combine to get  $x = c^d$  in  $\mathbb{Z}_N$

Suppose error occurs when computing  $x_q$ , but no error in  $x_p$

Then: output is  $x'$  where  $x' = c^d$  in  $\mathbb{Z}_p$  but  $x' \neq c^d$  in  $\mathbb{Z}_q$

$\Rightarrow (x')^e = c$  in  $\mathbb{Z}_p$  but  $(x')^e \neq c$  in  $\mathbb{Z}_q \Rightarrow \gcd((x')^e - c, N) = p$

# RSA Key Generation Trouble [Heninger et al./Lenstra et al.]

OpenSSL RSA key generation (abstract):

```
prng.seed(seed)
p = prng.generate_random_prime()
prng.add_randomness(bits)
q = prng.generate_random_prime()
N = p*q
```

Suppose poor entropy at startup:

- Same  $p$  will be generated by multiple devices, but different  $q$
- $N_1, N_2$  : RSA keys from different devices  $\Rightarrow \gcd(N_1, N_2) = p$

# RSA Key Generation Trouble [Heninger et al./Lenstra et al.]

Experiment: factors 0.4% of public HTTPS keys !!

Lesson:

- Make sure random number generator is properly seeded when generating keys



THE END