# Definitions and security

## Public Key Encryption

**Def**: A public-key encryption system is a triple of algorithms (G,E,D)

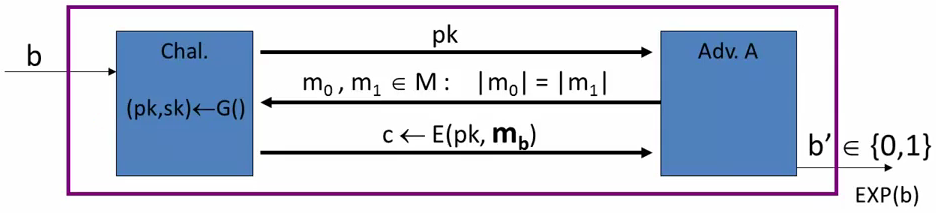
* G(): A randomized algorithm that outputs a key pair (pk, sk)
* E(pk,m): Randomized algorithm that takes m in M and outputs c in C
* D(sk,c): Deterministic algorithm that takes c in C and outputs m in M or bottom

**Consistency**: for all (pk,sk) output gy G():

for all m in M: D(sk, E(pk,m)) = m

## Security: against eavesdropping

For b=0,1 define experiments EXP(0) and EXP(1) as:



The attacker gets only one ciphertext and he must say which plain text corresponds to.

**Def**: EBIG = (G, E, D) is **semantically secure** (aka IND-CPA) if for all efficient A:

AdvSS[A,EBIG] = | Pr[EXP(0)=1] - Pr[EXP(1)=1] | < negligible

## Relation to symmetric cipher security

Recall: for symmetric cipher we had two security options:

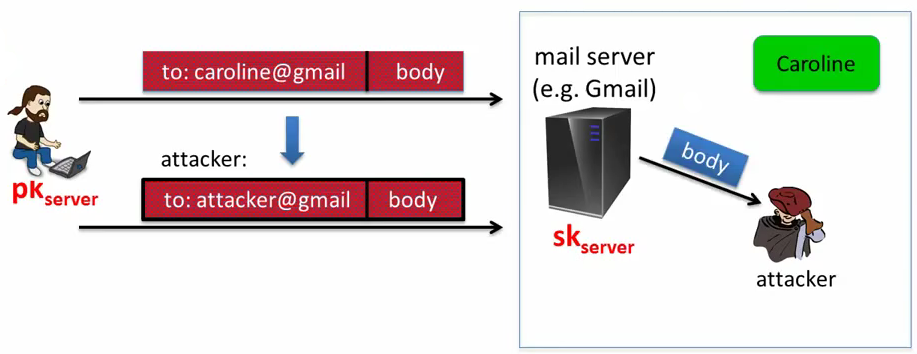
* One-time-security vs Many-time-security (CPA)
* We showed that one-time-security =/=> many-time-security

For public encryption:

* One-time security ⇒ many-time security (CPA)

This follows from the fact that the attacker can encrypt by himself

## Security againts active attacks

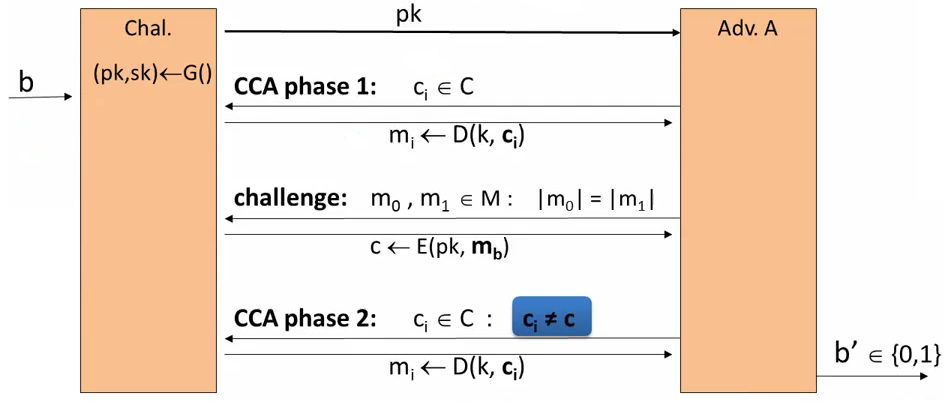


The attacker change the cipher text so the mails will be received by him and not by caroline. The server decrypts the message and gives the decrypted body to the attacker.

## (pub-key) Chosen Ciphertext Security

EBIG = (G,E,D) public-key encryption over (M,C). For b=0,1 define EXP(b):

The attacker is given the public key of the challenger.

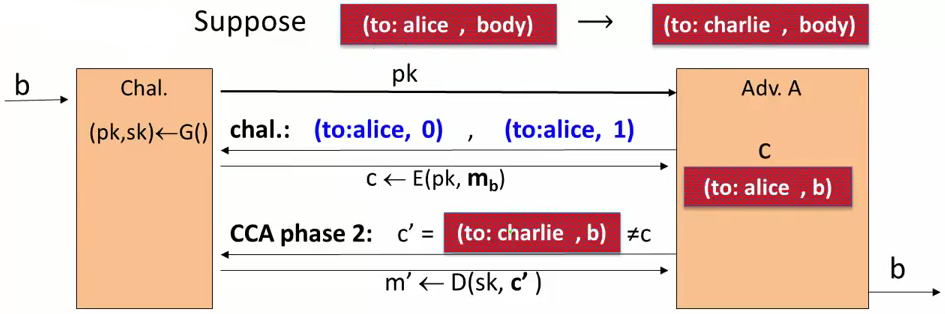


The attacker can request the decryption of any ciphered messages he wants except the cipher messages he obtains on the challenge phase.

**Def**: EBIG is CCA secure (aka IND-CCA) if for all efficient A:

AdvCCA[A,EBIG] = | Pr[EXP(0)=1] - Pr[EXP(1)=1) | is negligible

### Example:



# CONSTRUCTIONS

## Trapdoor functions (TDF)

**Def**: a trapdoor function X→ Y is a triple of efficient algorithms (G, F, F-1)

* G(): randomized algorithm outputs a key pair (pk, sk)
* F(pk,·): deterministic algorithm that defines a function X → Y
* F-1(sk,·): defines a function Y → X that inverts F(pk, ·)

More precisely:

for all (pk, sk) output by G

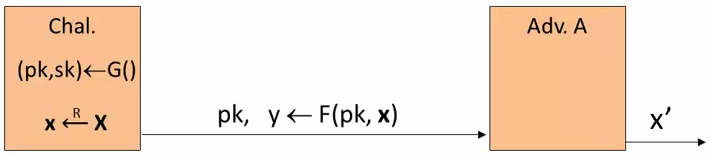
for all x in X: F-1(sk, F(pk,x)) = x

## Secure Trapdoor Functions (TDF)

(G, F, F-1) is secure if F(pk,·) is a “one-way” function:

can be evaluated, but cannot be inverted without sk

**Security**:



**Def**: (G, F, F-1) is a secure TDF if for all efficient A:

AdvOW[A,F] = Pr[x=x’] < negligible

## Public-key encryption from TDFs

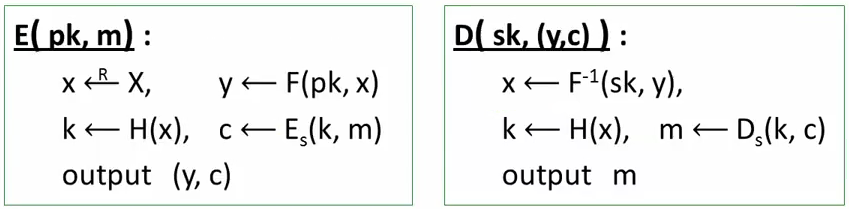
(G, F, F-1): secure TDF X → Y

(ES, DS): symetric authenticated encryption defined over (K, M, C)

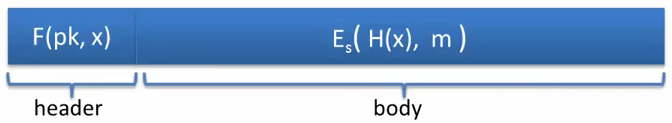
H: X → Y a hash function

We construct a pub-key encryption system (G, E, D):

Key generation G: same as G for TDF



In pictures:



**Security Theorem:**

If (G, F, F-1) is a secure TDF, (ES, DS) provides authenticated encryption and H: X→ K is a “random oracle” then (G, E, D) is CCAro secure.

## Incorrect use of a Trapdoor Function (TDF)

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

E(pk, m): output c← F(pk, m)

D(sk, c): output F-1(sk, c)

Problems:

* Deterministic: cannot be semantically secure!!
* Many attacks exist

## The RSA trapdoor permutation

Review: Three algorithms (G, F, F-1)

* G: outputs pk, sk. pk defines a function F(pk, ·): X→X
* F(pk, sk): evaluates the function at x
* F-1(sk, y): inverts the function at y using sk

**Secure** trapdoor permutation:

The function F(pk, ·) is **one-way** without the trapdoor sk

Review: Arithmetic mod composites

Let N=p·q where p,q are prime

ZN={0,1,2…,N-1}; (ZN)\* = {invertible elements in ZN}

**Facts**:

- x in ZN is invertible ⇐⇒ gcd(x,N) = 1

- Number of elements in (ZN)\* is

Euler’s Theorem:

## The RSA trapdoor permutation

First published: Scientific American, Aug. 1977

Very widely used:

- SSL/TLS: certificates and key-exchange

- Secure e-mail and file systems

- …

**G()**:

* Choose random primes p,q = 1024 bits. Set N=p·q
* Choose integers e,d such that
* Output pk = (N,e) , sk = (N,d)

F(pk, x): ZN\* → ZN\* ; RSA(x) = xe (in ZN)

The RSA assumption

## RSA is one-way permutation

For all efficient algorithms A:

Pr[A(N,e,y) = y1/e] < negligible

where p,q ← n-bit random primes, N← pq, y← (random) ZN\*

## RSA pub-key encryption (ISO standard)

(ES, DS): symmetric encryption scheme providing authenticated encryption

H: ZN→ K where K is key space of (ES, DS)

**G()**: generate RSA params: pk=(N,e), sk=(N,d)

**E(pk, m)**:

1. 1.- choose random x in ZN
2. 2.- y← RSA(x)=xe , k← H(x)
3. 3.- output (y, ES(k,m))

**D(sk, (y,c))**: output DS( H(RSA-1(y)), c) → m

## Textbook RSA is insecure

Textbook RSA encryption:

- public key: (N,e)

- secret key:(N,d)

Encrypt: c ← me (in ZN)

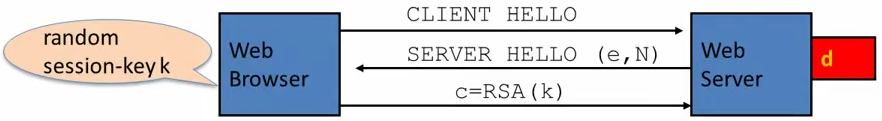
Decrypt: cd ← m

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,...,264}. Eavesdropper sees: c=ke in ZN

If k = k1·k2 where k1, k2 < 234 (prob aprox 20%) then c/k1e = k2e in ZN

Step 1: build table: c/1e, c/2e,...,c/234e. time: 234

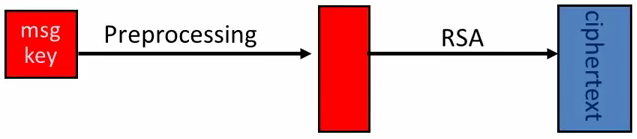
Step 2: for k2=0…, 234 test if k2e is in table. time: 234

Output matching (k1, k2). Total attack time: aprox = 240 << 264

## Public Key Cryptography Standard Number 1 (PKCS 1)

Review: Never use textbook RSA

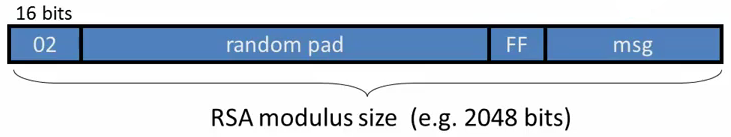
RSA in practice (since ISO standard is not often used):



Starting with the symmetric encryption key (e.g. AES 128 bits key) it is expanded into a full module size value (e.g. 2048 bits value) and it is this value the one used in RSA alg. RSA is asked to encrypt the generated key.

## PKCS1 v1.5

PKCS1 mode 2 (Encryption)

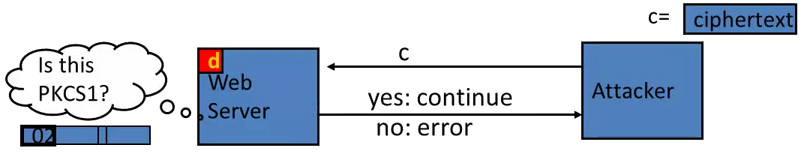


msg is the PT that we want to encrypt (on the example before this will be the generated symmetric key)

* Resulting value is RSA encrypted
* Widely deployed, e.g. in HTTPS

To recover the message the systems first sees a “02” and it knows the text corresponds to a PKCS1 mode 2 encrypted text. Removes these 16 bits moves all along the CT until it finds “FF”, it removes all bits until these “FF” and what stays are the bits that corresponds to the msg.

## Attack on PKCS1 v1.5 (Bleichenbacher 1998)



So attacker can test if 16 MSBs of PlainText is ‘02’, e.i., the decrypted message starts with ‘02’

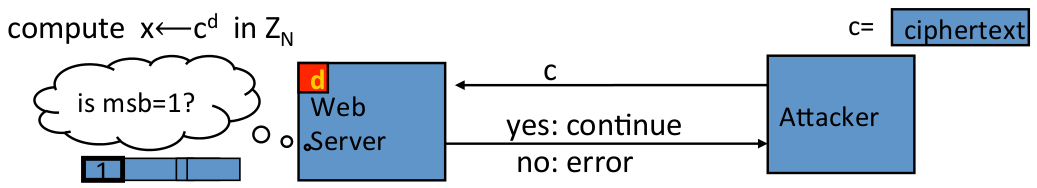
Chosen-ciphertext attack: to decrypt a given ciphertext c do:

- Choose r in ZN. Compute c’ ← re·c = (r · PKCS1(m))e

- Send c’ to web server and use response.

The attacker repeats this secuence many times (prob a million of times). With enough information recover from these attacks the attacker can deduce PCKS1(m).

## Baby Bleichenbacher



Suppose N is N=2n (an invalid RSA modulus)

Then:

* Sending c reveals msb(x)
* Sending 2e·c = (2x)e in ZN reveals msb(2x mod N) = msb2(x)
* Sending 4e·c = (4x)e in ZN reveals msb(4x mod N) = msb3(x)
* …

## HTTPS Defense (RFC 5246)

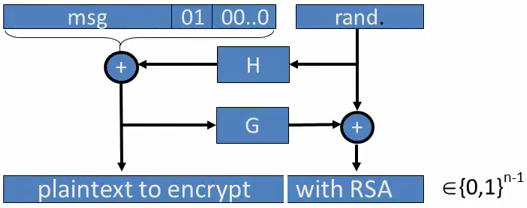
Attacks discovered by Bleichenbacher and Klima et al. can be avoided by treating incorrectly formatted message blocks… in a manner indistinguishable from correctly formatted RSA blocks.

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 = 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 ⇒ RSA-OAEP is CCA secure whel H,G are random oracles

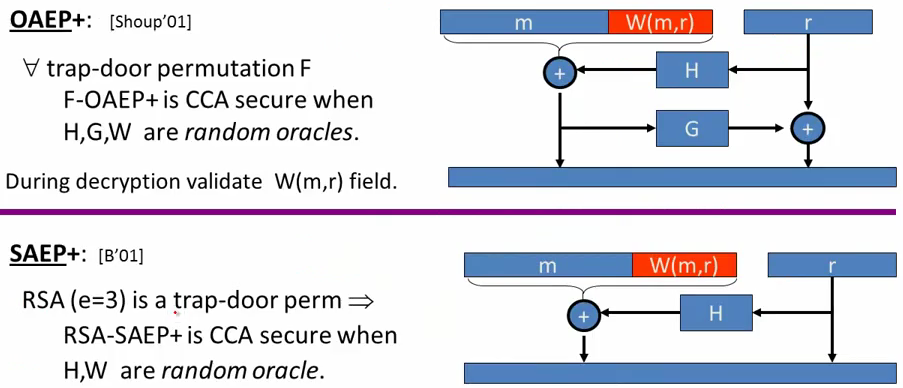
in practice: use SHA-256 for H and G

It is called optimal because the cipher text is as short as it can be. The message to encrypt is exactly the same length as one RSA output.

OAEP doesn’t work with a generic trap-door permutation, only with RSA.

## OAEP Improvements

To make OAEP function with Generic Trap-door permutation we have:



The decryption proccess with SAEP+ will be:

## Subtleties in implementing OAEP [M’00]

OAEP-decrypt(ct):

error = 0;

….

if (RSA-1(ct) > 2n-1)

{ error = 1; goto exit;}

…

if (pad(OAEP-1(RSA-1(ct))) != “010000”)

{ error = 1; goto exit; }

Problem: timing information leaks type of error ⇒ Attacker can decrypt any ciphertext

Lesson: Don’t implement RSA-OAEP yourself!!

# RSA as a one-way function

To invert the RSA ont-way function (without d) attacker must compute:

x from c=xe (mod N)

Best known algorithm:

- Step 1: factor N (hard)

- Step 2: compute e’th root modulo p and q (easy)

## Shortcuts to this Algorithms?

Must one factor N in roder to compute e’th roots?

To prove no shortcut exists show a reduction:

* Efficient algorithm for e’th roots mod N ⇒ efficient algorithm for factoring N
* Oldest problem in public key cryptography

Some evidence no reduction exists: (BV’98)

* “Algebraic” reduction ⇒ factoring is easy

How not to improve RSA’s performance

To speed up RSA decryption use small provate key d (d aprox 2128)

cd = m (mod N)

Wiener’87: if d<N0.25 then RSA is insecure (d < 2512)

BD’98: if d<N0.292 then RSA is insecure

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

## Wiener’s attack

Recall:

Continued fraction expansion of e/N gives k/d

e·d = 1 (mod k) ⇒ gcd(d,k)=1 ⇒ can find d from k/d

# RSA in practice

RSA with low public exponent

To speed up RSA encryptino use a small e: c=me (mod N)

Minimum value: e=3

Recommended value: e=65537=216+1

Encryption: 17 multiplications x65537 mod N

Asymmetry of RSA: fast enc. / slow dex.

- ElGamal: approx. same time for both.

## Key lengths

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

|  |  |
| --- | --- |
| **Cipher key-size** | **RSA Modulus size** |
| 80 bits | 1024 bits |
| 128 bits | 3072 bits (2048) |
| 256 bits (AES) | **15360** bits |

## Implementation attacks

**Timing attack**: (Kocher 97)

The time it takes to compute cd (mod N) can expose d.

**Power attacke**: (Kocher 99)

The power consumption of a smartcard while it is computing cd (mod N) can expose d.

**Faults attack**: (BDL 97)

A computer error during cd (mod 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=cd in ZN

decrypt mod p: xp=cd in Zp

decrypt mod q: xq=cd in Zq

combine to get x = cd in ZN

Suppose error occurs when computing xq, but no error in xp

Then: output is x’ where x’=cd in Zp

but x’!=cd in Zq ⇒ (x’)e = c in Zp

but (x’)e != c in Zq ⇒ 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
* N1, N2 : RSA keys from different devices ⇒ gcd(N1, N2) = p

Experiment: factors 0.4% of public HTTPS keys !!

Lesson:

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

## Further reading:

* Why chosen ciphertext security mattes, V. Shoup, 1998
* Twenty years of attacks on the RSA cryptosystem, D. Boneg, Notices of the AMS, 1999
* OAEP reconsidered, V. Shoup, Crypto 2001
* Key Lengths, A, Lenstra, 2004