IT-Security 1

Chapter 4: Asymmetric Cryptography

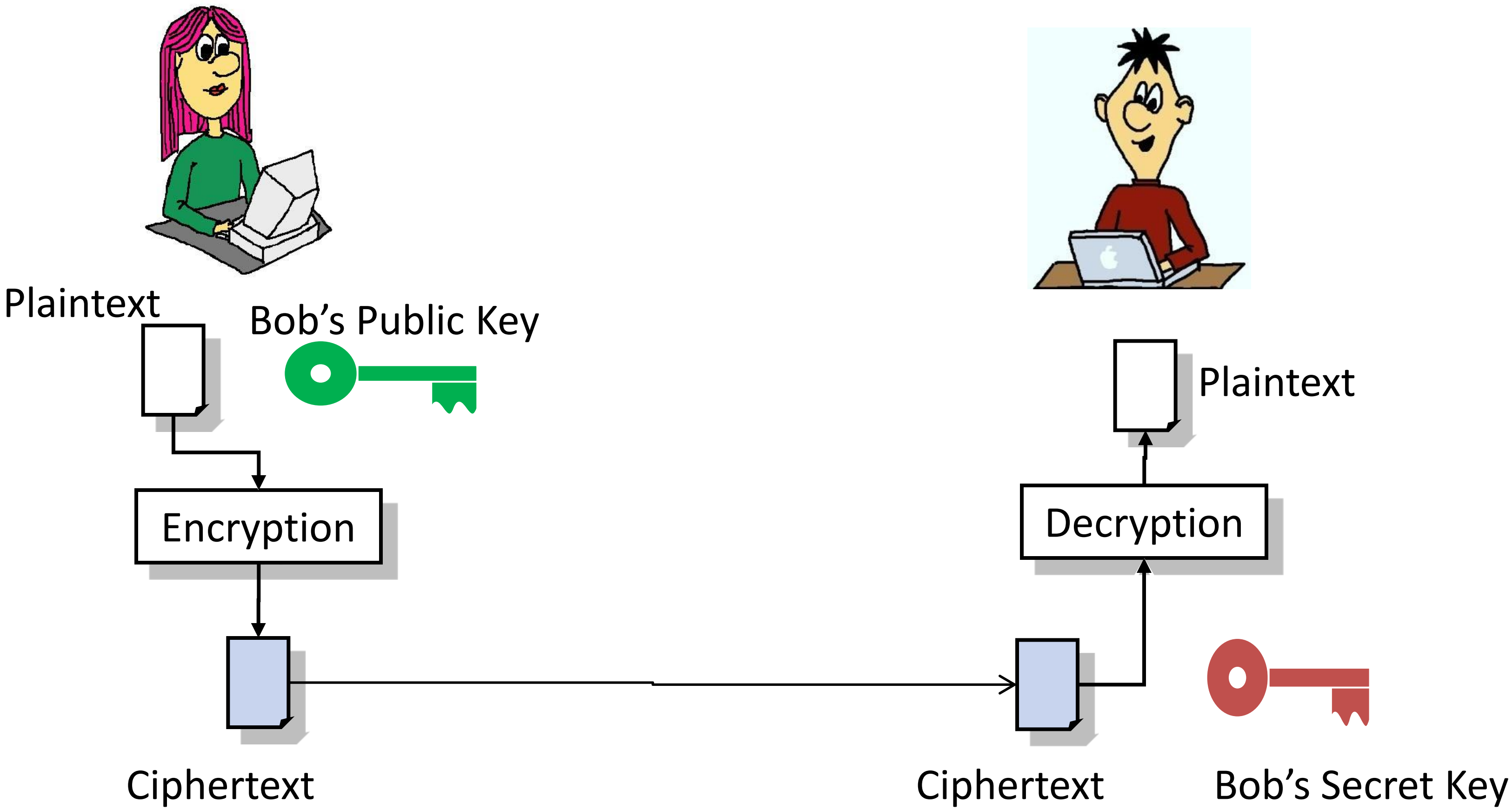
Prof. Dr.-Ing. Ulrike Meyer

WS 15/16

Chapter Overview

- Asymmetric Encryption
  - General Idea of Asymmetric Encryption
  - Modular Arithmetic
  - RSA
- Digital Signatures
  - General Idea of Digital Signatures
  - RSA-based signatures
  - Digital Signature Standard
- Diffie-Hellman Key-Agreement
- RSA Backdoors

General Idea of Asymmetric Encryption



Note: There needs to be a mechanisms that ensures that Alice gets into possession of Bob's public key

Modular Arithmetic (1)

- Let n be a positive integer $\neq 0$, then for any integer **k**
  - **k** mod **n** is defined as the remainder of **k** divided by **n**
    - Example: 10 mod 7 = 3
- We define addition and multiplication of the numbers $\mathbf{Z}_n = \{0,\ldots, n-1\}$ by
  - **a** + **b** := (**a**+**b**) mod **n**
  - **ab** := (**ab**) mod **n**
- Then
  - For all **a**, **b**, **c** $\in \mathbf{Z}_n$: (**a**+**b**) + **c** = **a** + (**b** + **c**)
  - For all **a**, **b**, **c** $\in \mathbf{Z}_n$: **a** + **b** = **b** + **a**
  - For all **a** $\in \mathbf{Z}_n$: **a** + **0** = **a**
  - For all **a** $\in \mathbf{Z}_n$ there is an element **x** $\in \mathbf{Z}_n$ with **a** + **x** = **0**. This element **x** is called the additive inverse of **a** mod **n** and is denoted as "**-a**"
- I.e. the set $\mathbf{Z}_n$ together with the + operation forms a commutative group

Modular Arithmetic (2)

- And
  - For all **a**, **b** Є {1,…,n-1}: **ab** = **ba**
  - For all **a**, **b**, **c** Є {1,…,n-1}: (**ab**)**c** = **a**(**bc**)
  - For all **a** Є {1,…,n-1}: **a1** = **a**
- If for an **a** Є {1,…,n-1} there is an **x** Є {1,…,n-1} with
  - **ax** = **1** mod **n**
- Then **a** is called invertible mod n and **x** is called the inverse of **a** and **x** is denoted as $\mathbf{a}^{-1}$
- The set {0,…, n-1} together with the + and • operations forms a commutative ring with 1

Example: Addition and Multiplication mod 6

+	0	1	2	3	4	5
0	0	1	2	3	4	5
1	1	2	3	4	5	0
2	2	3	4	5	0	1
3	3	4	5	0	1	2
4	4	5	0	1	2	3
5	5	0	1	2	3	4

*	1	2	3	4	5
1	1	2	3	4	5
2	2	4	0	2	4
3	3	0	3	0	3
4	4	2	0	4	2
5	5	4	3	2	1

1 and 5 are invertible mod 6
2, 3, and 4 are not

Example: Addition and Multiplication mod 5

+	0	1	2	3	4
0	0	1	2	3	4
1	1	2	3	4	0
2	2	3	4	0	1
3	3	4	0	1	2
4	4	0	1	2	3

*	1	2	3	4
1	1	2	3	4
2	2	4	1	3
3	3	1	4	2
4	4	3	2	1

$2^{-1} = 3$, $3^{-1} = 2$,
$1^{-1} = 1$, $4^{-1} = 4$

Modular Exponentiation

- For all **a** Є {0,…,n-1} $\mathbf{a}^k$ mod n is defined as

$$\mathbf{a}^k \bmod \mathbf{n} = \underbrace{\mathbf{a} \cdots \mathbf{a}}_{\text{k times}} \bmod \mathbf{n}$$

Multiplicative Inverses in General

- An integer **k** has a multiplicative inverse mod **n** iff **k** and **n** are relatively prime
  - E.g. in mod 10: 1, 3, 7, and 9 are invertible while 2, 4, 5, 6, and 8 are not
- Relatively prime means that **k** and **n** do not share any prime divisors
- The number of integers relatively prime to **n** is called $\varphi(\mathbf{n})$
- If **n** is prime, then all integers $\neq 0$ are invertible mod **n**
  - $\varphi(\mathbf{n}) = \mathbf{n}-1$
- If **n** is the product of two different prime numbers **p**, **q** ($\mathbf{n} = \mathbf{pq}$)
  - Then: $\varphi(\mathbf{n}) = (\mathbf{p}-1)(\mathbf{q}-1)$
- The set of integers that are invertible mod **n** are denoted $\mathbf{Z_n}^*$
  - E.g. $\mathbf{Z_5}^* = \{1, 2, 3, 4\}$, $\mathbf{Z_4}^* = \{1, 3\}$, $\mathbf{Z_{10}}^* = \{1, 3, 7, 9\}$

What Euclid's Algorithm Does and How

- Allows to compute the greatest common divisor (gcd) of two integers
- Two integers are relatively prime iff their gcd = 1
- With Euclid's algorithm one can therefore check if an integer **k** has an inverse mod **n** by checking if gcd(**k**,**n**) = 1
  - Example how Euclid's algorithm works for 408 and 595:
    - 595:408 = 1 remainder 187
    - 408:187 = 2 remainder 34
    - 187:34 = 5 remainder **17**
    - 34:17 = 2 remainder 0
  - ➢ gcd(408,595) = 17

Why does Euclid's Algorithm Work?

- gcd(**k**,**n**) = gcd(**k**-**n**,**n**)
  - Proof:
    - If **d** divides **k** and **n**, then **k** = **jd**, **n** = **ld** for some **j**, **l**, therefore **k** - **n** = (**j** - **l**)**d**, so any divisor of **n** and **k** divides **n**-**k** as well
    - Vice versa if **d** divides **n** and **k** - **n**, then **n** = **yd** and **k** - **n** = **xd** for some **y**, **x**, therefore **k** = **k** - **n** + **n** = (**x** + **y**)**d**, so **d** divides **k** as well
    - So, any divisor of **k** and **n** divides **k**-**n** and **n** as well and vice versa, in particular the greatest common devisers are therefore the same
- Repeating this several times gives us
  - gcd(**k**,**n**) = gcd(**k** mod **n**, **n**)
- Repeating this with **k** mod **n** instead of **n** and **n** instead of **k** now gives us
  - gcd(**k** mod **n**, **n**) = gcd(**k** mod **n**, **n** mod (**k** mod **n**))
- We continue doing this until the remainder is 0
- The remainder before the 0 is then the gcd

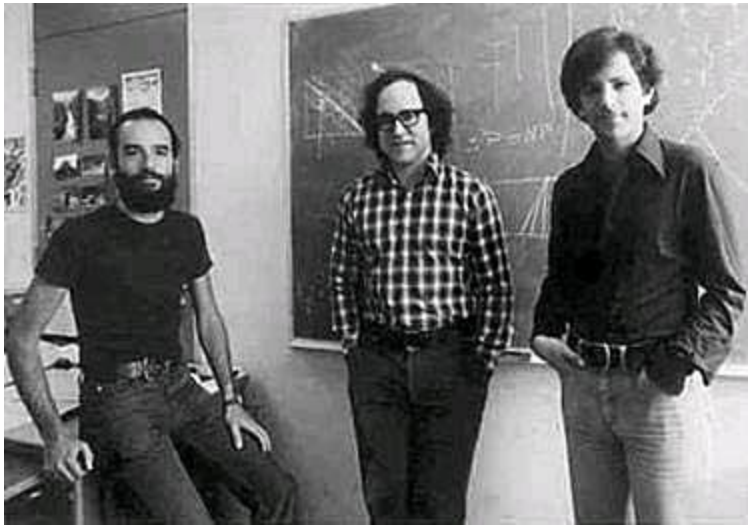
Euclid's Algorithm in Formulas

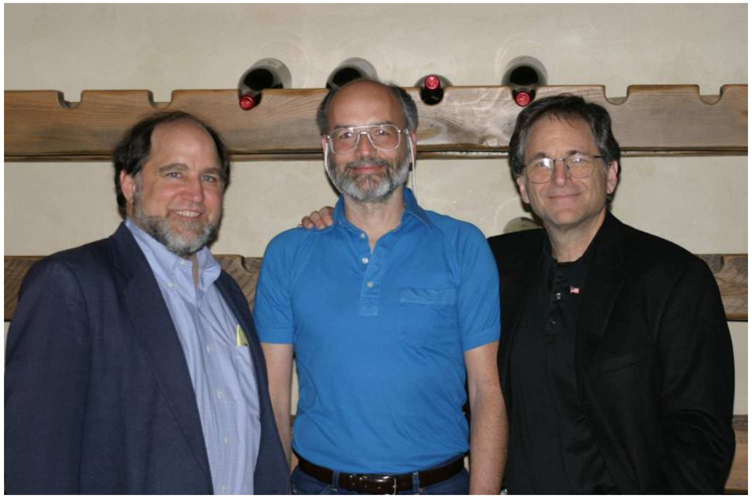
- Goal: compute gcd(**k**,**n**)
- Initial Setup
  - Let r_0 = **n**, r_1 = **k**, i = 0
- Step i
  - If $r_{i+1} = 0$, then gcd(**k**,**n**) = r_i
  - Else, divide r_i by r_{i+1} to get quotient q_{i+1} and remainder r_{i+2}
  - Set i = i+1 and repeat
- Note: if we additionally set
  - $u_0 = 1$, $u_1 = 0$, $v_0 = 0$, $v_1 = 1$ in the initial step
  - and then $u_{i+1} = u_{i-1} – q_i u_i$ and $v_{i+1} = v_{i-1} - q_i v_i$ for $i > 0$
- Then: if $r_{s+1} = 0$, then r_s = gcd(k,n) and
  - gcd(k,n) = $u_s n + v_s k$

Compute the Inverse with Euclid

- If **k** and **n** are relatively prime, then Euclid's algorithm gives us **u**, **v** with
  - 1 = **uk** + **vn**
- As a consequence: **u** is the inverse of **k** mod **n**

RSA

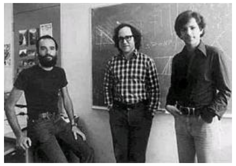




- Invented by Rivest, Shamir, and Adleman
- In 1977 at MIT
- Was patented from 1983 to 2000
- First published public key cryptosystem
- Original idea goes back to Diffie and Hellman
  - Theory of how it could work
  - No mathematical function

RSA Cryptosystem: Some Facts

- Is an asymmetric encryption / decryption mechanism
- Can be defined for different key lengths
  - Today typically 1024 bit, some applications use 2048 bit modulus as default already
- Variable block size but smaller or equal to the key
- Cipher blocks are always as long as the key
- Rarely used for encryption of longer messages
- Often used to
  - Encrypt/decrypt symmetric keys when they are distributed
  - Encrypt authentication challenges
  - Construct digital signatures



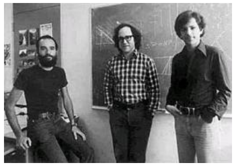
RSA Key Generation

- Public key:
  - Choose two prime numbers **p**, **q**
  - Compute **n** = **pq**
  - Choose **e** invertible mod φ(**n**)
  - Public key: (**n**,**e**)

Check if **e** is invertible with the help of Euclid's Algorithm

- Private key:
  - Find **d** with **ed** = 1 mod φ(**n**)
  - Select **d** as the private key
  - Note: **de** = 1 + φ(**n**)k for some integer k $\neq$ 0

Find **d** with the help of Euclid's Algorithm

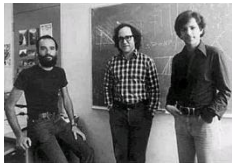


RSA Encryption / Decryption

- Encryption
  - For each plaintext $\mathbf{m} \in Z_n$: $\mathbf{c} = \mathbf{m}^{\mathbf{e}} \bmod \mathbf{n}$
- Decryption
  - For each ciphertext $\mathbf{c} \in Z_n$: $\mathbf{m} = \mathbf{c}^{\mathbf{d}} \bmod \mathbf{n}$
- Why does this work?
  - For any $\mathbf{c} \in Z_n$: $\mathbf{c}^{\mathbf{d}} \bmod \mathbf{n} = (\mathbf{m}^{\mathbf{e}})^{\mathbf{d}} \bmod \mathbf{n} = (\mathbf{m}^{\mathbf{ed}}) \bmod \mathbf{n}$

$= \mathbf{m}^{\varphi(\mathbf{n})k+1} \bmod \mathbf{n} = \mathbf{m} \bmod \mathbf{n}$

Euler's Theorem for $\mathbf{n}=\mathbf{pq}$



Euler's Theorem

Euler's Theorem:

For any $\mathbf{a} \in \mathbf{Z}^*_n$: $\mathbf{a}^{\varphi(n)} = 1 \bmod \mathbf{n}$

Reformulation:

For any $\mathbf{a} \in \mathbf{Z}^*_n$, and any integer $\mathbf{s}$: $\mathbf{a}^{\varphi(n)s+1} = \mathbf{a} \bmod \mathbf{n}$

Proof:

Note: If $\mathbf{a}, \mathbf{b} \in \mathbf{Z}_n^*$, then $\mathbf{ab} \in \mathbf{Z}_n^*$

Also note: Multiplying all elements of $\mathbf{Z}_n^*$ with some $\mathbf{a}$ in $\mathbf{Z}_n^*$ just reorders them:

Assume $\mathbf{x}$ is the product of all different $\mathbf{x}_1,\ldots,\mathbf{x}_{\varphi(n)} \in \mathbf{Z}_n^*$. Then, for any $\mathbf{a} \in \mathbf{Z}_n^*$: $\mathbf{ax}_1\mathbf{ax}_2\cdots\,\mathbf{ax}_{\varphi(n)} = \mathbf{a}^{\varphi(n)}\mathbf{x} = \mathbf{x}$ (otherwise $\mathbf{ax}_i = \mathbf{ax}_j$ for some $i \neq j$)

Multiplying the above equation with $\mathbf{x}^{-1}$ on both sides yields $\mathbf{a}^{\varphi(n)} = 1 \bmod \mathbf{n}$.

Generalization to Arbitrary a for n=pq

Generalization to arbitrary all $\mathbf{a} \in Z_n$ in case $\mathbf{n}=\mathbf{pq}$:

In case $\mathbf{n} = \mathbf{pq}$, $\mathbf{a}^{\varphi(n)+1} = \mathbf{a} \bmod \mathbf{n}$ holds for all $\mathbf{a} \in Z_n$ and not only for the ones relatively prime to $\mathbf{n}$.

Proof:

- For $\mathbf{a} = 0$ the equation clearly holds.
- For $\mathbf{a}$ invertible mod $\mathbf{n}$ see proof on last slide.
- If $\mathbf{a}$ is not invertible, then $\mathbf{a}$ is not relatively prime to

$\mathbf{n}=\mathbf{pq}$. Therefore $\mathbf{a}$ is divisible by either $\mathbf{p}$ or $\mathbf{q}$ (if by both, then $\mathbf{a} = 0$).

  - So assume $\mathbf{a}$ is divisible by $\mathbf{q}$ but not by $\mathbf{p}$, then $\mathbf{a}^{p-1} = 1 \bmod \mathbf{p}$, and $\mathbf{a}^{q-1} = 0 \bmod \mathbf{q}$.
  - As a consequence $\mathbf{a}^{\varphi(n)+1} = \mathbf{a}^{(p-1)(q-1)+1} = \mathbf{a} \bmod \mathbf{p}$ and $\mathbf{a}^{\varphi(n)+1} = \mathbf{a}^{(p-1)(q-1)+1} = \mathbf{a} \bmod \mathbf{q}$.
  - So there are integers k,s with $a^{\varphi(n)+1} = \mathbf{a}+k\mathbf{p}$ and $\mathbf{a}^{\varphi(n)+1} = \mathbf{a}+s\mathbf{q}$.
  - So $s\mathbf{q} = k\mathbf{p}$, such that $\mathbf{q}$ divides k.
  - So there is an integer l with $k = l\mathbf{q}$ and we have $\mathbf{a}^{\varphi(n)+1} = \mathbf{a} + k\mathbf{p} = \mathbf{a} + l\mathbf{qp} = \mathbf{a} \bmod \mathbf{n}$

Trivial RSA Example

Setup

Let $\mathbf{p} = 3$, $\mathbf{q} = 5$, then $\mathbf{n} = \mathbf{pq} = 15$

$\varphi(\mathbf{n}) = (\mathbf{p} - 1)(\mathbf{q} - 1) = 2 \cdot 4 = 8$

Choose $\mathbf{e} = 3$ (prime to $\varphi(\mathbf{n})$)

Then $\mathbf{d} = 3$ ($\mathbf{ed} = 1 \bmod 8$)

Encryption of $\mathbf{m} = 7$:

$\mathbf{m}^{\mathbf{e}} \bmod \mathbf{n} = 7^3 \bmod 15 = 343 \bmod 15 = 13$

Decryption of $\mathbf{c} = 13$:

$\mathbf{c}^{\mathbf{d}} \bmod \mathbf{n} = 13^3 \bmod 15 = 2197 \bmod 15 = 7$

How Secure is RSA? (1)

Theorem: Let **p**, **q** prime numbers and **n** = **pq**. Then **n** can efficiently be factorized iff φ(**n**) can be efficiently computed.

Proof:

" ⇒": If **n** can be efficiently factorized then **p** and **q** can efficiently be computed from **n** and therefore φ(**n**) = (**p**-1)(**q**-1) is efficiently computable

"⇐": If φ(**n**) is known, then one can compute **p** and **q** from the two equations **n** = **pq** und φ(**n**) = (**p** − 1)(**q** − 1)

So factorizing **n** is equivalent to computing φ(**n**)

How Secure is RSA? (2)

Theorem: Let **p**, **q** prime, **n** = **pq** and (**e**,**n**) a public RSA key and **d** the corresponding private RSA key. Then **d** can be efficiently computed from (**n**,**e**) iff **n** can be factorized efficiently.

Proof Outline:

"$\Leftarrow$" clear!

"$\Rightarrow$" There is a probabilistic polynomial-time algorithm that computes **p** and **q** from **d**, **e**, and **n**

So, computing **d** is equivalent to factorizing **n**

How Secure is RSA? (3)

- Putting it together we have
  - Being able to compute a private RSA key **d** from (**e**, **n**) is equivalent to being able to factorize **n**
  - Which in turn is equivalent to being able to efficiently compute φ(**n**)

- However: it is unclear if there is a way to decrypt RSA-encrypted messages without knowledge of the private key **d**.

How Hard is Factorization?

- There is currently no polynomial time algorithm for factorization
- There are algorithms with sub-exponential run time.
  - Pollard's Rho Method
  - Quadratic Sieve
  - Number Sieve
  - …
- Currently, RSA modules of 1024 bit still often in use but some applications moved to 2048 bit as default already
- The prime numbers **p** and **q** are required to be of 512 bit or 1024 bit length respectively

Modular Exponentiation

- RSA Encryption and Decryption are based on modular exponentiation: $\mathbf{x}^k \bmod \mathbf{n}$
- "Naïve" modular exponentiation requires k modular multiplications
- Problem: the size of the exponent is of the same order as the size of the modulus **n**
  - E.g. 1024 bit or 2048 bit
- ➢ "Naïve" modular exponentiation is not efficient

Efficient Modular Exponentiation

- Idea: Use the binary representation of the exponent

$$k = \sum k_i 2^i = k_0 + 2(k_1 + 2(k_2 + \cdots) \cdots),$$

where $k_0, k_1, k_2 \in \{0,1\}$

- Then we get:

$$\mathbf{x}^k = \prod \mathbf{x}^{k_i 2^i}$$

- So all we have to do is multiply by **x** or square

Example for efficient Exponentiation

$$37 = 1 * 2^0 + 0 * 2^1 + 1 * 2^2 + 0 * 2^3 + 0 * 2^4 + 1 * 2^5$$

$$\mathbf{x}^{37} = \mathbf{x} * \mathbf{x}^{2^2} * \mathbf{x}^{2^5} = (((((\mathbf{x}^2)^2)^2)\mathbf{x})^2)^2\,\mathbf{x}$$

PKCS#1v.2.1

- PKCS = Public Key Cryptography Standard
  - By RSA Laboratories
- Defines two encoding methods for RSA Encryption
  - RSAES-PKCS1-v1_5
  - RSAES-OAEP
- For new applications RSAES-OAEP should be used
  - RSAES-PKCS1-v1_5 is vulnerable to chosen ciphertext attacks
- Both methods describe how to pad bit string messages and how to convert them to large integers
- OAEP = Optimal Asymmetric Encryption Padding
- RSAES = RSA Encryption Scheme

RSAES-OAEP Encryption

RSAES-OAEP-ENCRYPT ((**n**, **e**), **M**, **L**)

Options:

- Hash function (hLen: length of hash in octets)
- MGF: mask generation function
  - May be based on the hash function
  - Main difference: can generate output of arbitrary length

Input:

- (**n**, **e**) recipient's RSA public key (k: length of **n** in octets)
- **M** message to be encrypted, an octet string of length mLen, where mLen = k – 2hLen – 2
- **L** optional label to be associated with the message; the default value for **L**, if **L** is not provided, is the empty string

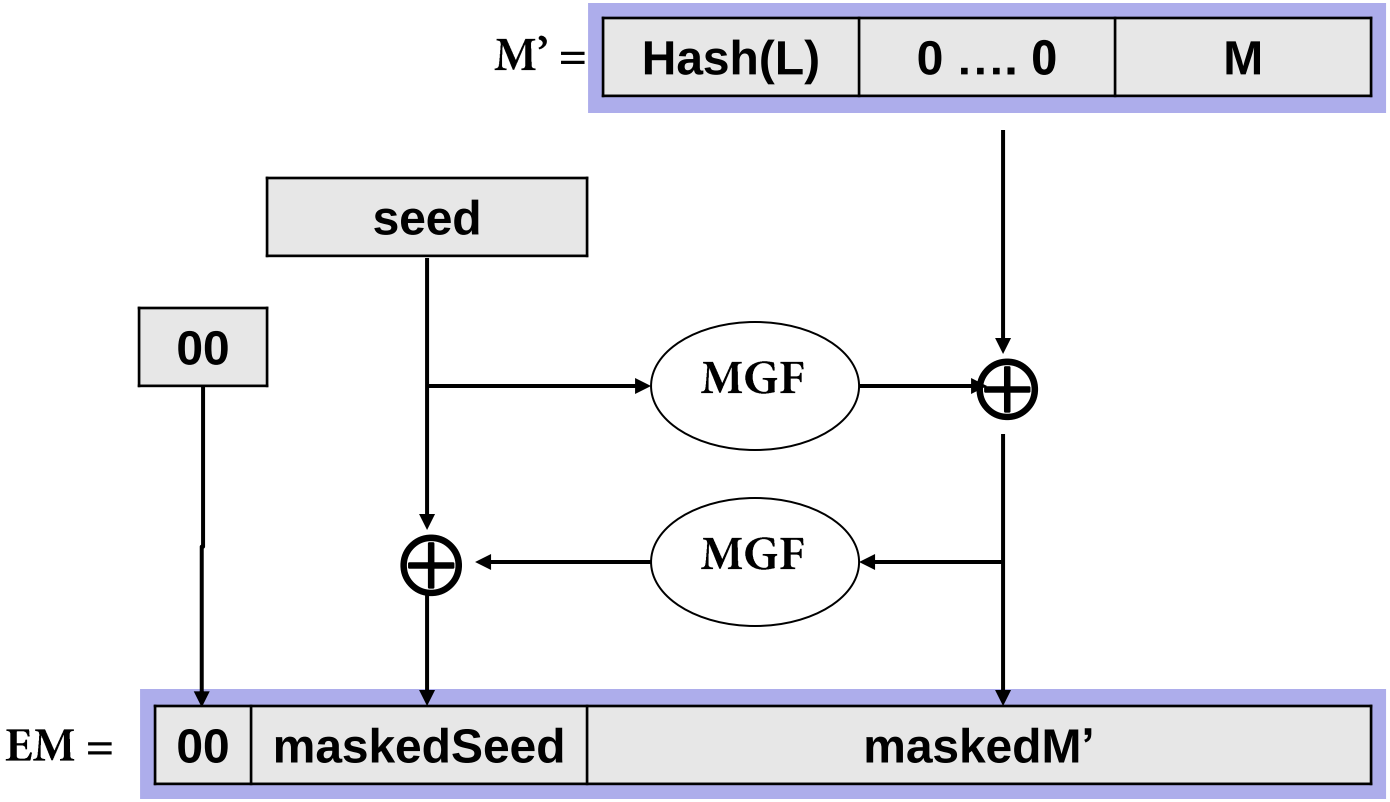
Output:

- **C** ciphertext, an octet string of length k

RSAES-OAEP Overview

- Octet string message **M** is padded to an encoded message **EM** of k octets
- **EM** is converted to an integer
- The integer representation of **EM** is encrypted with (**n**,**e**) to an integer **c**
- The integer **c** is converted to the **k**-octet string ciphertext **C**

Encoding of M to EM



- Padding with zeros: k – mLen – 2hLen – 2 zeros
- M message
- Seed = random octet string of length hLen
- The leading octet of zeros in EM is used to detect decryption errors
- **How does decoding work?**

Intuitively: Why PKCS#1?

- The encoding methods in PKCS1 help to detect if a message M is a valid plaintext
  - "Recognizable" plaintext
- In particular the encodings protect against someone exchanging a cipher text **c** = **m**^{e} mod **n** with a cipher text **cs**^{e}
  - Without the encoding: **cs**^{e} mod **n** is the cipher text of **ms**
  - With the encoding, **ms** will not have the correct structure

This property is called multiplicative homomorphic

Why OAEP?

- Turns a deterministic public key encryption system such as RSA into a probabilistic one
- I.e. if the same plaintext is encrypted twice using OAEP, the resulting ciphertexts will be different due to the random seed

Symmetric vs. Asymmetric Encryption

- Advantage of asymmetric encryption schemes
  - Do not require a secret (but still an authentic) channel for key exchange
  - Less keys required to be kept secret
- Advantages of symmetric encryption schemes
  - Can be implemented more efficiently
- Goal: Leverage on the advantages of both
  - Use hybrid schemes in which the secret key for symmetric encryption is exchanged using asymmetric primitives

Backdoors in the RSA Key Generation

- RSA is quite secure due to the hardness of the factorization problem
- It is, nevertheless, possible to attack RSA by manipulating the key generation function
- Whenever RSA is used, keys have to be generated with the help of the key generation operations
- Whoever implements these operations can try to integrate a backdoor into the generated public/private key pair
- This backdoor can then later on allow him to retrieve the private key corresponding to a public key

Entities Involved

- Manufacturer (Attacker)
  - Designer of the backdoor
  - Integrates the backdoor in the key generation code
- User (Victim)
  - In possession of a device or piece of code for key generation e.g. for RSA manipulated by the manufacturer
  - Can observe public and private keys generated by his device
- External attacker
  - Can observe public keys used by the user

Naïve RSA Backdoor

- Change the RSA key generation process, such that one fixed prime is used always
  - Fix a prime **p**
  - Choose a second prime **q** at random
  - Set n = **pq**
  - Select **e** relatively prime to φ(**n**) and **d** such that **ed** = 1 mod φ(**n**)

- Now the manufacturer of the backdoor can use **p** to find **d**:
  - Compute $q = p^{-1}n$
  - Compute **d** as the inverse of **e** mod φ(**n**) = (**p**-1)(**q**-1)

Disadvantages of Naïve RSA

- Assume an external attacker knows that the manufacturer has integrated the naïve backdoor in the key generation code
- Assume the external attacker is able to obtain two keys (**n**,**e**), (**n’**,**e’**) generated with the naïve backdoor
- Then the attacker can compute
  - gcd (**n**,**n’**) = **p**
- Now he can reconstruct the secret keys **d** and **d’** just in the same way as the manufacturer can

- I.e. anyone can break RSA keys generated with this backdoor

- Can the manufacturer do better?

Better RSA Backdoor

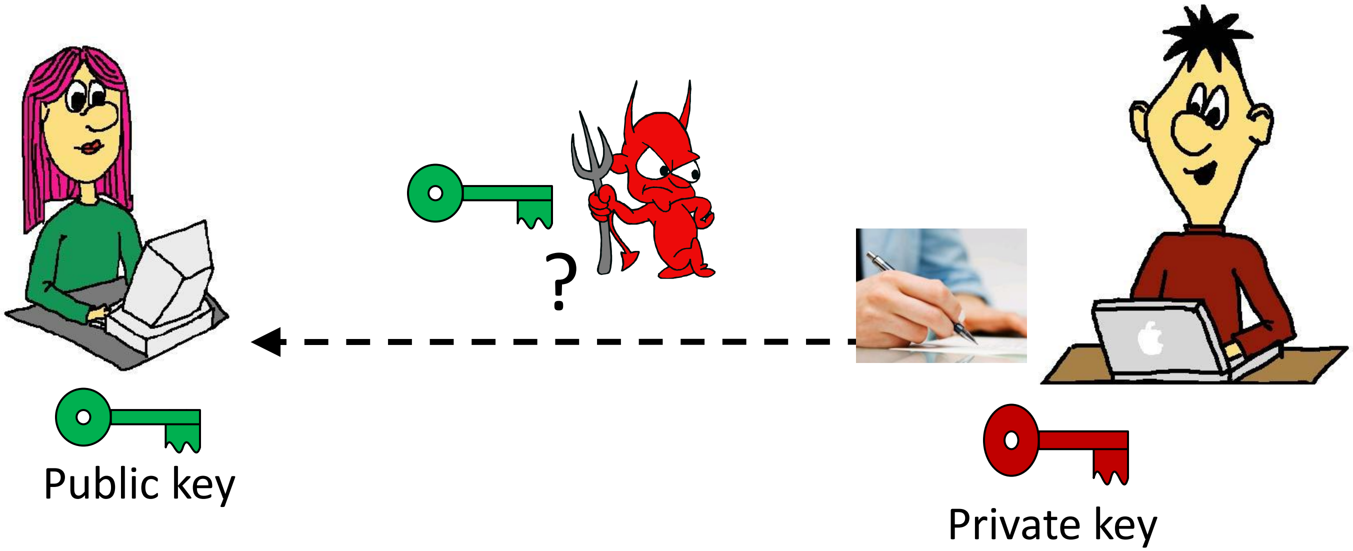
- Assume the manufacturer is in possession of a public RSA key (**E**,**N**) and a private key **D**
- Manipulate the key generation process as follows
  - Pick prime numbers **p** and **q** at random and set **n** = **pq**
  - Compute **e** = p^E mod **N**
  - Check if **e** is invertible mod φ(**n**)
  - If yes, compute the inverse **d** and output (**e**,**n**), **d**
  - If no, pick a new prime number **p** and start again

- If the attacker now observes a public key (**e**,**n**), he can
  - Compute e^D mod **N** = **p** and can use this to compute **d**

Advantage of the “Better RSA Backdoor”

- Using the backdoor requires knowledge of the private key of the manufacturer

Basic Idea of Digital Signatures

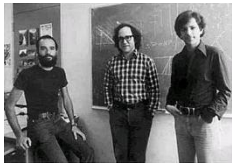


- Given:
  - Everybody knows Bob's public key
  - Only Bob knows the corresponding private key
- Goal: Bob can digitally sign a message such that anyone can verify his signature
  - To compute a signature, one must know the private key
  - To verify a signature it is enough to know the public key

Definition of a Digital Signature Scheme

- A digital signatures scheme consists of a
  - Key generation algorithm
  - A signature generation algorithm
  - A signature verification algorithm
- Key generation algorithm specifies
  - Generation of the public key for signature verification
  - Generation of the private key for signature generation
- Signature generation algorithm
  - Takes a message **M** and the private key as input and outputs the signature
- Signature verification algorithm
  - Takes the message **M**, the public key, and the signature as input and returns success or failure of signature verification

Example: Naïve RSA Signatures (Insecure Version)

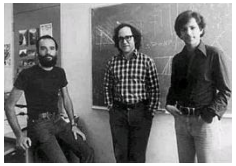


- Key generation (as in RSA Encryption)
  - Choose two large primes **p**,**q** randomly
  - Choose an exponent **e** Є $Z^*_{\varphi(n)}$
  - Compute **n** = **pq** and **d** Є $Z^*_{\varphi(n)}$ with **ed** = 1 mod φ(**n**)
  - The public key is then (**n**,**e**), the private key is **d**

- Signing a message **m**: uses private key **d**
  - Signature **s** on **m** is **s** = $\mathbf{m}^{\mathbf{d}}$ mod **n**

- Signature verification: uses public key **e**
  - On receipt of (**m**,**s**) the verifier computes **m'** = $\mathbf{s}^{\mathbf{e}}$ mod **n** and verifies that **m** = **m'**

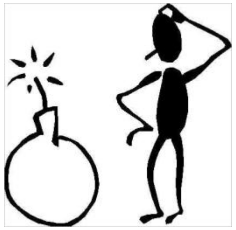
Example for RSA Signature



- Assume Alice chooses **p** = 11, **q** = 23, **e** = 147
- Then **n** = 253, **d** = 3
- The public key is (147, 253), the private key is 3
- If Alice wants to sign the message **m** = 111, she computes
  - **s** = 111^{3} mod 253 = 166
- On receipt of (**m**,**s**) Bob verifies the signature by computing
  - **m’** = 166^{147} mod 253 = 111
- As **m’** = **m**, Bob knows that Alice has generated the signature **s**

Why is the Naïve RSA Signature Scheme Insecure?

- Naïve RSA is vulnerable to existential forgery:
  - An attacker can choose $\mathbf{s} \in Z_n$ randomly and claim that **s** is a signature generated by Alice on the message $\mathbf{m} := \mathbf{s}^{\mathbf{e}} \bmod \mathbf{n}$
  - I.e. it is easy to generate pairs of (**m**,**s**) such that s is a valid signature on **m**
  - Note: it is not easy to generate a valid **s** for a given meaningful **m**
- RSA is multiplicative homomorphic:
  - If $\mathbf{m}_1$, $\mathbf{m}_2$ are two messages and $\mathbf{s}_1$, $\mathbf{s}_2$ the corresponding signatures of $\mathbf{m}_1$ and $\mathbf{m}_2$, then
  - $\mathbf{s} := \mathbf{s}_1\mathbf{s}_2 = \mathbf{m}_1^{\mathbf{d}}\mathbf{m}_2^{\mathbf{d}} = (\mathbf{m}_1\mathbf{m}_2)^{\mathbf{d}} \bmod \mathbf{n}$
  - I.e. **s** is a valid signature on $\mathbf{m}_1\mathbf{m}_2$



RSA Signature Scheme (Secure Version)

- RSA scheme can be protected against existential forgery and the multiplicatively problem with the help of a hash function

- Key generation as in Naïve RSA Signature Scheme

- Signing a message **m**: uses private key **d** and a publically known cryptographic hash function **h**
  - Signature **s** on **m** is $\mathbf{s} = \mathbf{h(m)}^{\mathbf{d}} \bmod \mathbf{n}$

- Signature verification: uses public key **e**
  - On receipt of (**m**,**s**) the verifier computes **h(m**) and $\mathbf{h'} = \mathbf{s}^{\mathbf{e}} \bmod \mathbf{n}$ and verifies that **h(m) = h’**

Applying the Hash Function Helps

- Signing the hash instead of the message directly
  - Signature **s** on **m** is $\mathbf{s} = \mathbf{h(m)}^d \bmod \mathbf{n}$

- Protects against
  - Existential forgery
    - Existential forgery would now require Eve to chose some s and find a message m such that $\mathbf{s}^e = h(\mathbf{m})$. If h is pre-image resistant, this is hard
  - Forging signatures due to RSAs multiplicativity
    - The fact that RSA is multiplicative is not a problem any more as it is hard to find a message m such that $h(\mathbf{m}) = h(\mathbf{m}_1)h(\mathbf{m}_2)$ due to the pre-image resistance of h

Signing the Hash Instead of the Message

- Hashing messages before signing is generally applied, not only in case of RSA
  - I.e., instead of signing a message **m** of arbitrary length a hash function $h: \{0,1\}^* \rightarrow \{0,1\}^n$ is used and h(**m**) is signed
- The advantages are
  - Signatures are computed on smaller values
  - Security advantages for some signature schemes
    - See e.g. above for RSA
  - Larger messages do not have to be split in blocks small enough for the signature scheme
    - If messages are split receiver of the signed blocks is not able to recognize if all the blocks are present and in the appropriate order

PKCS#1v.2.1

- Defines two encoding schemes for RSA signature schemes
  - RSASSA-PSS
    - Recommended for new applications
  - RSASSA-PKCS1-v1_5

- Both encoding schemes specify how to
  - hash an octet string message
  - encode the hash to a padded octet string
  - convert the encoded hash to an integer
  - generate the signature

Use of Digital Signatures

- Authentication of messages
  - Can also be provided by a MAC
  - However, not to the general public
- Message integrity
  - Can also be provided by a MAC
- Non-repudiation
  - Can not be provided by a MAC: a MAC can be generated by the “signer” and the “verifier”
  - A signature can only be computed by the entity that is in possession of the private key. The signer can therefore not repudiate having signed the message
- Authentication Protocols

Types of Attacks on Digital Signatures

- Key-Only Attack: The attacker tries to generate valid signatures while only in possession of the public verification key
- Known-Message Attack: The attacker knows some message/signature pairs and tries to generate another valid signature on some message
- Chosen-Message Attack: The attacker can choose messages, can make the signer sign them and tries to generate another valid signature on some other message

- Power of attacker highest in chosen-message attack

Hierarchy of Attack Results

- Total break: attack results in recovery of the signature key
- Universal forgery: attack results in the ability to forge signatures for any message
- Selective forgery: attack results in a signature on a message of the adversary's choice
- Existential forgery: merely results in some valid message/signature pair not already known to the adversary

- The strongest notion of security for a signature scheme is to be secure against existential forgery in a chosen message attack

DSS

- The Digital Signature Standard (DSS) was adopted by NIST in 1994
- Is standardized in FIPS 186
  - Updated in FIPS 186-2, 186-3
- Uses the Digital Signature Algorithm (DSA)
- Uses SHA-1 (160 bit) as hash function h
- Is based on the discrete logarithm problem
  - Given a prime number **p** and **g**, **x** $\in Z^*_p$, and **Y** = g^x mod **p** then it is hard to compute the "discrete logarithm" **x** of **Y** from **Y**, **g**, and **p**
- Is closely related to a Signature Scheme by Schnorr

Key Generation for DSS

- Signer generates two prime numbers p, q with
  - p of 512 bit
  - q of 160 bit
  - $q | (p-1)$
- Signer chooses $\mathbf{x} \in Z^*_p$ such that $\mathbf{x}^{(p-1)/q} \bmod p \neq 1$ and sets $g := \mathbf{x}^{(p-1)/q} \bmod p$
  - Then the smallest integer i for which $g^i = 1 \bmod p$ is $i = q$
- Signer chooses $a \in \{1,\ldots,q-1\}$ and computes
  - $A = g^a \bmod p$
- The public key of the signer is then (p,q,g,A) and the private key is a

Signature Generation in DSS

Signature Generation on message m:

- Signer randomly chooses **k** Є {1,…,q-1} and computes
  - $\mathbf{r} = (\mathbf{g}^{k} \bmod \mathbf{p}) \bmod \mathbf{q}$
  - $\mathbf{s} = (\mathbf{k}^{-1}(h(\mathbf{m}) + \mathbf{a}\mathbf{r})) \bmod \mathbf{q}$, where $\mathbf{k}^{-1}$ is the inverse of **k** mod **q**
- The signature on **m** then is the pair **(r,s)**

- Note: all exponents used in the signature generation are 160 bit max -> efficiency
- Note: a different **k** has to be chosen for each message

Why different "k" for different m?

- Assume **k** is used to sign two known messages $\mathbf{m_1}$ and once for $\mathbf{m_2}$, then
  - $\mathbf{r} = (g^k \bmod p) \bmod q$
  - $\mathbf{s_1} = (\mathbf{k}^{-1}(\, h(\, \mathbf{m_1}\,) + a\mathbf{r})) \bmod q$
- Then $\mathbf{s_1 - s_2 = k^{-1}}\ (\, h(\mathbf{m_1}) - h(\mathbf{m_2})\,) \bmod q$
- So $\mathbf{k} = (\mathbf{s_1 - s_2})^{-1}\ (\, h(\mathbf{m_1}) - h(\mathbf{m_2})\,) \bmod q$
- And thus: $a = \mathbf{r}^{-1}(\, \mathbf{s_1 k} - h(\mathbf{m_1})\,) \bmod q$, i.e. the private key can be computed

Signature Verification in DSS

- Upon receipt of **m**, **s**, **r** the verifier
  - Checks if **r** Є {1,…,q-1} and **s** Є {1,…,q-1}
  - Computes $\mathbf{u}_1 = h(\mathbf{m})\mathbf{s}^{-1} \bmod \mathbf{q}$, $\mathbf{u}_2 = \mathbf{rs}^{-1} \bmod \mathbf{q}$
  - Computes $\mathbf{v} = (\mathbf{g}^{\mathbf{u}_1}\mathbf{A}^{\mathbf{u}_2} \bmod \mathbf{p}) \bmod \mathbf{q}$
  - Accepts the message if **v** = **r** and rejects the message otherwise

Why the Verification Works

- If **m**, **s**, and **r** were received correctly by the verifier, then:

$$\mathbf{v} = (g^{u_1} A^{u_2} \bmod p) \bmod q$$

$$= (g^{h(\mathbf{m})\mathbf{s}^{-1}} g^{a\mathbf{rs}^{-1}} \bmod p) \bmod q$$

$$= (g^{(h(\mathbf{m})+a\mathbf{r})\mathbf{s}^{-1}} \bmod p) \bmod q$$

$$= (g^{\mathbf{kss}^{-1}} \bmod p) \bmod q$$

$$= (g^{\mathbf{k}} \bmod p) \bmod q$$

Remember: $g^q = 1 \bmod p$ such that we can compute all exponents mod q without changing the result mod p: $g^k \bmod p = g^{\mathbf{k} \bmod q} \bmod p$

Trivial Example for DSS

Key generation

- Let **p** = 11, **q** = 5, (5 divides 10)
- Choose e.g. **x** = 2 and therefore **g** = 4
- Choose **a** = 3 and **A** = g^a mod **p** = 4^3 = 9 mod 11

Signature generation

- Signing h(**m**) = 4 works as follows:
  - Choose **k** Є {1,..4}, e.g. **k** = 3
  - Compute **r** = g^3 mod **p** mod **q** = 4^3 mod 11 mod 5 = 9 mod 5 = 4
  - Compute **s** = k^{-1} (h(**m**) + **ar**) mod **q** = 2 (4 + 3 • 4) mod 5 = 2
- I.e. the signature on h(**m**) = 4 is (4,2)

Note

- Note that if **r** = 0 mod **p** mod **q** or if **s** = 0 mod **q** the signature verification will not work
  - Intuitively this is due to the fact that the private key is not used in the generation of **s** any more if **r** = 0
- The DSS standard states on this problem
  - “As an option, one may wish to check if **r**=0 or **s**=0, a new value of **k** should be generated and the signature should be recalculated”

Security of DSS

- The same **k** should not be used twice
- If no hash function is used or the conditions for the length of the **r** and **s** are not checked by the signer, existential forgery is possible
- If the discrete logarithm problem is efficiently solvable, then an attacker can totally break DSS signatures (as he can compute the secret key **a** from the public key (**p**,**q**,**g**,**A**))

Diffie-Hellman Key Exchange(1)

- Oldest public key mechanism
  - Invented in 1976
- Is a key establishment protocol by which two parties can
  - Establish a symmetric secret key **K**
  - Based on publicly exchanged values
- The security of the key exchange is based on the discrete logarithm problem
  - As already discussed in context of DSS

Diffie-Hellman Key Exchange(2)

Public information known to both parties

- Prime number **p** and a generator **g** of Z^*_p

Private / Public Diffie-Hellman values

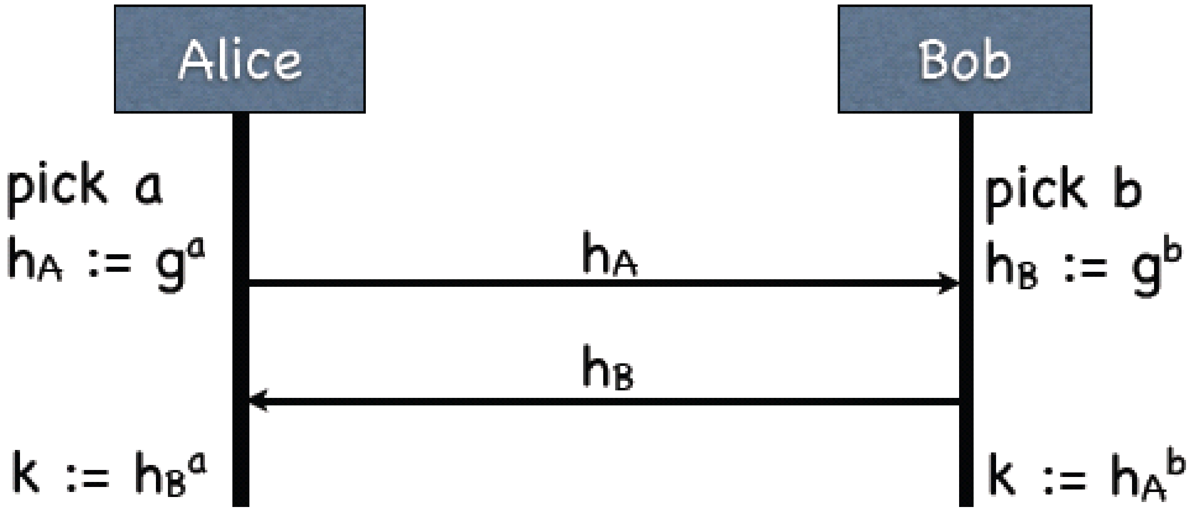
- Alice
  - Selects a private DH value **a** Є {1,…,p-2} randomly
  - Computes the corresponding public DH value $h_A = g^a \bmod p$
- Bob
  - Selects a private DH value **b** Є {1,…,p-2} randomly
  - Computes its public DH value $h_B = g^b \bmod p$

Key exchange

- Alice sends h_A to Bob, Bob sends h_B to Alice

Diffie-Hellman Key Exchange(2)

- With Alice's public value and his private value Bob can compute $K = h_A{}^b = g^{ab} \mod p$
- With Bob's public value and his private value Alice can compute $K = h_B{}^a = g^{ba} \mod p$

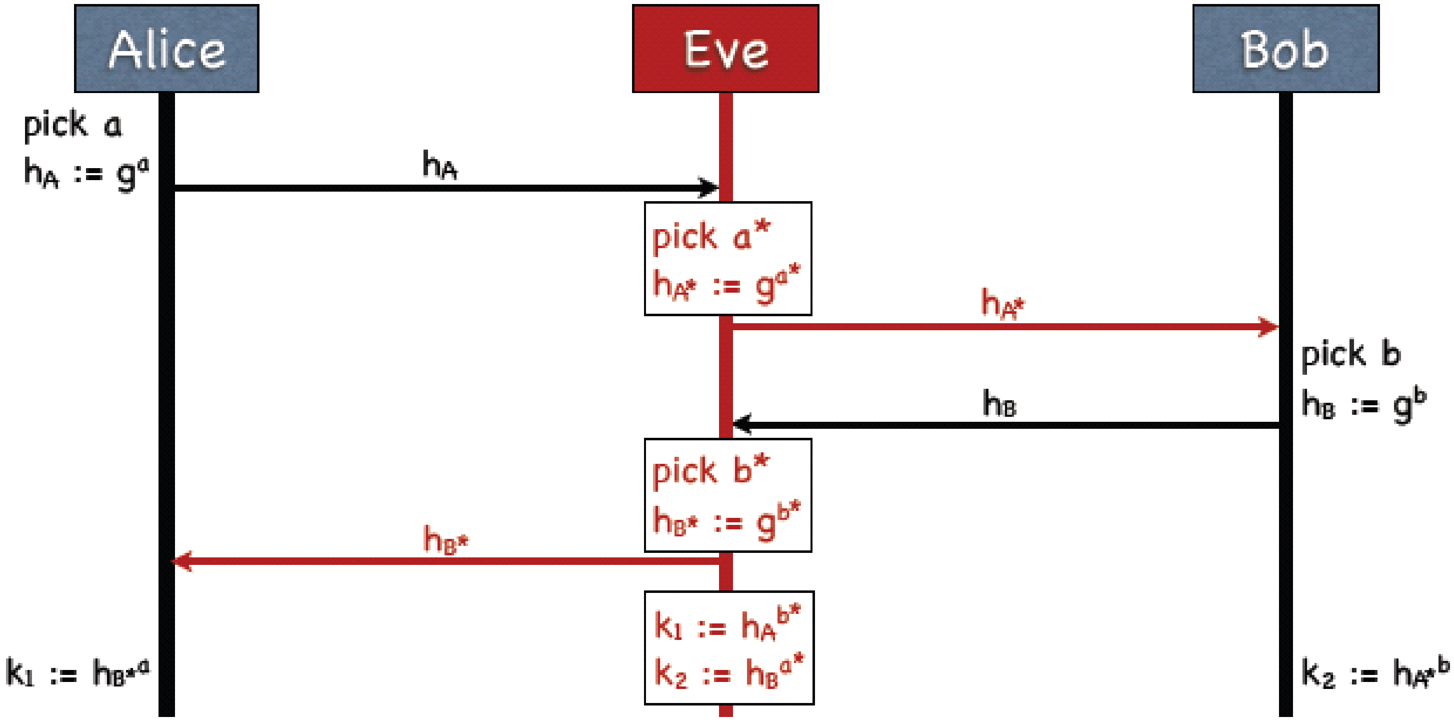


Example

- Let $p = 17$, $g = 3$
- Assume Alice chooses $a = 7$, then $h_A = 3^7 = 11 \mod 17$
- Assume Bob chooses $b = 4$, then $h_B = 3^4 = 13$
- Alice receives B = 13 from bob and computes $K = 13^7 \mod 17 = 4$
- Bob receives A = 11 from Alice and computes $K = 11^4 \mod 17 = 4$

➢ Alice and Bob share the key **K** = 4

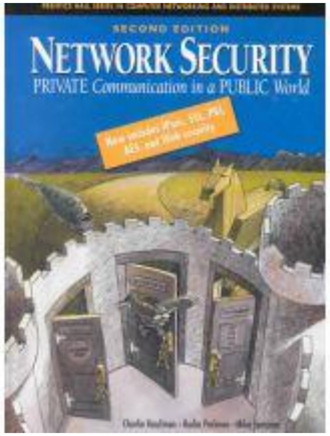
Man-in-the-Middle Attack



- Eve has a key with each Alice and Bob
- Alice and Bob do not share a key
- Works because Bob and Alice have no proof that the public values are authentic -> see Chapter 5 on certificates

References and Further Reading

- Kaufmann et al., *Network Security*
  - Chapter 6 and 7



- PKCS#1, v.2.1
  - ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-1/pkcs-1v2-1.pdf
- DSS specification: FIPS 186-2
  - http://csrc.nist.gov/publications/fips/archive/fips186-2/fips186-2.pdf