C) Public Key Cryptography

C.a) Fundamentals

C.b) RSA with Applications

C.c) DSA and Diffie Hellman

C.a) Fundamentals

C.1 Introducing Remark

- Public key cryptosystems are widely spread. They are used for various purposes, in particular to ensure secrecy and to provide authenticity and data integrity.
- In any case there exist two keys, a *secret* (*private*) *key* to which only its legitimate owner should have access to and a *public key* which is publicly known (as its name indicates).
- It *shall be practically infeasible to determine the secret key from the public key* although this is principally possible (with unlimited computational power).

C.1 (continuation)

- In *public key encryption schemes* the legitimate receiver of a message uses his secret key to decrypt the ciphertext that has been encrypted with his public key.
- In *public key signature schemes* the public key is used to verify signatures that have been generated with the secret key.
- The security of a public key cryptosystem usually depends on a number theoretic problem that is assumed to be *practically* infeasible (e.g., the factorization of large numbers $\rightarrow$ RSA, Section C.b).

C.2 Remark

- Many proposals for public key cryptosystems have turned out to be insecure (e.g. knapsack cryptosystems).
- Before we consider concrete examples of public key cryptosystems we provide fundamental facts that will be needed in the later sections.

C.3 Definition

The *Euler phi function* (*Euler totient function*) is defined by

$\varphi: N \to N, \quad \varphi(n) := |\{ k \leq n : \gcd(k,n)=1 \}|,$

i.e. it assigns n the number of coprime positive integers that are $\leq n$.

<u>Example:</u> $\varphi(1) = 1$, $\varphi(6) = 2$, $\varphi(101) = 100$

C.4 Some Useful Facts

(i) $\varphi(p) = p-1$ for p prime

(ii) $\varphi(p^s) = (p-1)\, p^{s-1}$ for p prime and $s \geq 1$

(iii) $\varphi(ab)= \varphi(a)\varphi(b)$ for any coprime a,b

(iv) Assume that $n = p_1^{s_1} p_2^{s_2} \ldots p_m^{s_m}$ where $p_1, \ldots, p_m$ are different primes and $s_1,\ldots,s_m \geq 1$. By (ii) and (iii) we have

$$\varphi(n)= \varphi(p_1^{s_1})\ldots\varphi(p_m^{s_m})$$
$$= (p_1-1)\, p_1^{s_1-1}\ldots (p_m-1)p_m^{s_m-1}$$

<u>Details:</u> Blackboard + Exercises

C.5 Remark

- *If the factorization of n is known* the computation of $\varphi(n)$ is easy even for large n.

Note: *If the factorization of n is unknown* the computation of $\varphi(n)$ may become practically infeasible for large n.

C.6 Square & Multiply Exponentiation Algorithm

- A typical task in public key cryptography is the computation of y^d (mod n) for large integers y, d, n.
- The 'natural' attempt, namely to compute y^d first and then to compute its remainder modulo n is not practically feasible because the intermediate value y^d is gigantic. For typical RSA parameters that are used today y^d had up to about 10^{310} decimal digits.
- Instead, a modular exponentiation algorithm has to be applied that processes the exponent in small portions.

C.6 (continued)

computes $y \rightarrow y^d \pmod{n}$ with $d = (d_{w-1},\ldots,d_0)_2$

```
temp := y

for i=w-2 down to 0 do {

    temp := temp^2 (mod n)

    if (d_i = 1) then  temp := temp * y (mod n)

    }
return temp   (= y^d (mod n) )
```

C.7 Remark

- The square & multiply exponentiation algorithm (s&m) is the most elementary modular exponentiation algorithm.
- To compute y^d (mod n) the s&m algorithm requires $\approx \log_2(d)$ modular squarings and about $0.5*\log_2(d)$ modular multiplications with the basis y. If d denotes a secret RSA key then d is usually in the same order of magnitude as the modulus n.
- At cost of additional memory the number of multiplications can be reduced by applying a table-based modular exponentiation algorithm (cf. “Handbook of Applied Cryptography”, for instance).

C.8 Fermat's Little Theorem

Theorem:

Let p denote a prime. Then

$$a^{p-1} \equiv 1 \pmod{p} \qquad \text{if} \quad \gcd(a,p)=1.$$

C.9 Remark

- Fermat's formula usually fails for composite moduli.

<u>Counterexample:</u>

$14^{14} \equiv 1 \pmod{15}$ but

$2^{14} \equiv 4 \pmod{15}$

- Euler's Theorem (next slide) generalizes Fermat's Little Theorem.

C.10 Euler's Theorem

Theorem:

For any positive integer n

$a^{\varphi(n)} \equiv 1 \pmod{n}$ if $\gcd(a,n)=1$.

C.11 Primality Testing

Task: Verify whether an integer is prime

Straight-forward approach (trial division):

Divide n by all primes $\leq \sqrt{n}$.

- The straight-forward approach is appropriate for small n *but practically infeasible for large n*. (It costs too much time.)
- In practice, *probabilistic* primality tests are applied.
- Fermat's little Theorem suggests the following primality test (next slide).

C.12 Fermat's Primality Test

Goal: verify whether n is prime

Input: n (odd integer), t (security parameter)

```
flag:=0; i=1;
while ((i ≤ t) && (flag=0)) do {
   choose a random integer a ∈ {2,…,n-2};
   if  a^(n-1) ≢ 1 (mod n) then flag:=1;
}
if (flag=1) return 'n is composite'
else return 'n is (probably) prime'.
```

C.12 (continued)

- If gcd(a,n)=1 and $a^{n-1} \not\equiv 1 \pmod{n}$ then n cannot be a prime, I.e. it is composite.
- Even if $a^{n-1} \equiv 1 \pmod{n}$ for all t trials *n need not necessarily be a prime*! (Recall that $14^{14} \equiv 1 \pmod{15}$, for instance, although 15 is not prime.)
- Therefore Fermat's and other primality tests are called 'probabilistic'.
- Alternatively, before exponentiation it may be checked whether gcd(a,n)>1, which proved compositeness without exponentiation. This has little practical meaning since it is very unlikely to find such integers by chance.

C.13 Definition

- For $a \in \{1,\ldots,n-1\}$ let $a^{n-1} \not\equiv 1 \pmod{n}$. Then a is called a *witness* (to compositeness) for n.
- If n is composite and $a \in \{1,\ldots,n-1\}$ fulfils $a^{n-1} \equiv 1 \pmod{n}$ then a is called a *Fermat liar* for n, and n is called a *pseudoprime* to the base a.

<u>Example</u> (cf. C.9):

(i) 2 is a witness for 15.

(ii) 14 is a Fermat liar for 15, and 15 is a pseudoprime to the base 14.

C.14 Efficiency

- Assume that n is composite

Fact: If there exists one integer $a \in Z_n^*$ with $a^{n-1} \not\equiv 1 \pmod{n}$ then there are at least (n / 2) many integers in {1,…,n-1} with this property.

Consequence: In this case the probability that n is erroneously assumed to be prime (since n passes all t trials of Fermat's primality test) is $\leq 0.5^t$.

For t=40, for instance, the right-hand-side $\approx 10^{-12}$.

C.14 (continued)

Attention: There exist composite integers n with $a^{n-1} \equiv 1 \pmod{n}$ for all coprime a (i.e. for all $a \in Z_n^*$).

Such integers are called *Carmichael numbers*.

Consequence: For Carmichael numbers Fermat's primality test only outputs 'n is composite' if gcd(a,n)>1. It is yet very unlikely to find such a base a by chance.

Note: Although there exist infinitely many Carmichael numbers they are relatively rare.

Details: Blackboard + Exercises

C.14 (continued)

Note: There exist other probabilistic primality tests that are more efficient than Fermat’s primality test. In practice, usually the Miller-Rabin primality test ($\rightarrow$ Exercises) is applied.

C.15 Factoring Large Integers

Goal: Factorize a composite integer n

Straight-forward approach (trial division):

Divide n successively by the primes $\leq \sqrt{n}$.)

- The straight-forward approach is appropriate for small n *but practically infeasible for large n*.
- For large n more efficient factorization algorithms are needed.
- Fermat's little Theorem suggests the following factorization algorithm.

C.16 Pollard's p-1 method

Input:

n (odd integer with unknown factorization $p_1p_2\ldots p_m$ where $p_1,\ldots,p_m$ denote distinct primes; RSA: m=2)

B (integer, 'smoothness bound')

Goal: Find the prime factors $p_1,\ldots,p_m$

- First step: Find any non-trivial factor d of n (i.e., $1<d<n$).
- If the non-trivial factors are still composite apply the factorization algorithm the these integers.

C.16 (continued)

$$r := \prod_{q \le B} q^w$$ where q is prime and w the largest exponent with $q^w \le n$

Choose a random integer $a \in \{2,\ldots,n-1\}$

If d:=gcd(a,n)>1 return d

Compute a^r (mod n)

d:= gcd(a^r –1 (mod n),n)

if (d=1) or (d=n) return ‘failure’

else return d

C.16 (continued)

Note:

If $1 < d < n$ then d and (n/d) are non-trivial factors of n.

There exist different variants to construct r. In any case it is a product of many small primes.

C.17 Justification

- If $\gcd(a, p_j)>1$ a nontrivial factor of n is found. For large n this is very unlikely.
- Assume that p_j is a prime factor of n such that *all prime factors of (p_j-1)* are ≤ B. Then r is a multiple of p_j-1. If $\gcd(a,p_j)=1$ Fermat's Little Theorem then implies $a^r - 1 \equiv 0 \pmod{p_j}$, i.e. $a^r - 1$ is a multiple of p_j and hence $d:=\gcd(a^r - 1 \pmod{n}, n) \geq p_j$.
- If d=1 the algorithm may be run again with a larger smoothness bound B.
- Note that if $p_i - 1$ divides r for *each* prime p_i then d=n. If d=n the algorithm should be run again with a smaller smoothness bound B.

C.18 Efficiency

- Pollard's p-1 algorithm is much more efficient than trial divisions since one run of the algorithm checks *all primes p simultaneously* for which all prime factors of p-1 are ≤ B.
- It is yet very likely that p-1 itself has at least one prime factor which is non-negligibly large (compared to the size of p). Unless n is relatively small (or p-1 falls into unusually small primes) Pollard's p-1 algorithm requires a gigantic smoothness bound B.
- Consequently, for large integers n more efficient factorization algorithms are needed.

C.18 (continued)

- For ‘medium sized’ integers n elliptic curve factorization methods are appropriate.
- For ‘large’ integers n (e.g., RSA moduli) usually the quadratic sieve or the number field sieve are applied. These algorithms are continuously improved.
- Presently, the number field sieve is the most efficient factorization algorithm.

Note: In 2005 a 667 bit integer (RSA challenge) was factored with the number field sieve.

C.18 (continued)

Basic idea of sieving algorithms:

- Find integers x and y with $x^2 \equiv y^2 \pmod{n}$.
- Justification: This equation is equivalent to $0 \equiv x^2 - y^2 \equiv (x+y)(x-y) \pmod{n}$.
- If $x \not\equiv \pm y \pmod{n}$ then gcd(x+y,n) gives a non-trivial divisor of n.

C.19 Discrete Logarithm

- We already know that the computation of $y^d \pmod{n}$ is easy even for large integers
- Now consider the inverse problem:

  Given the triple (y,b,n) find an integer (often, the smallest non-negative integer) with

  $y^x \equiv b \pmod{n}$

  (if there is such a number x!).

C.19 (continued)

Definition: Let G denote a finite group and $g \in G$. The order of g, denoted by ord(g), equals the smallest exponent r for which $g^r = 1$ in G.

Note: The equation $y^x \equiv b \pmod{n}$ has a solution for each $b \in Z_n^*$ if and only if $y \in Z_n^*$ *generates* Z_n^*, i.e., if $\langle y \rangle := \{y, y^2 \pmod{n}, \ldots, y^{ord(y)} \pmod{n} = 1\} = Z_n^*$.

C.20 Definition

In analogy to the real numbers the value x is called the *discrete logarithm* of b (to base y).

The problem of finding the integer x in the equation $y^x \equiv b \pmod{n}$ is called a *discrete log problem*.

C.21 Remark

- The discrete log problem can be formulated in any finite group G. Some authors called it the *generalized discrete log problem*.
- Several public key cryptosystems rely on discrete log problems that are assumed to be practically intractable.
- The hardness of the discrete log problem depends on the group G.

C.22 Example

- Let G denote the additive group Z_n. In Z_n the discrete log problem is very easy. In fact, if gcd(y,n)=1 solving the equation

  $y+\ldots+y = y \cdot x \equiv b \pmod{n}$ (additive group!)

  merely demands the computation of the multiplicative inverse $y^{-1} \pmod{n}$.
- Let <y> = Z_p^* for a large prime p (let’s say 1024 bit). The discrete log problem

  $y^x \equiv b \pmod{p}$

  in Z_p^* is practically intractable.

C.23 Remark

- Over the reals the logarithm function is easy to compute since $x_1 < x_2$ implies $\log(x_1) < \log(x_2)$.
- This is not true in Z_p^*, for instance.

Example:

For p=5 and y=2 we have $2^2 \equiv 4 > 2^3 \equiv 3 \pmod{5}$.

Note: Simplified speaking, this is the reason for the hardness of the discrete log problem in Z_p^*.

C.24 Solving the Discrete Log Problem

- For small n one may simply compute y, y^2 (mod n), y^3 (mod n), … until the first term equals b.
- For large n more efficient algorithms are needed.
- We discuss the baby step – giant step algorithm, an elementary algorithm which is applicable in any group G since it does exploit any peculiarities of G.

C.25 Baby-Step Giant-Step Algorithm

Goal: Given a finite group G, a generator y of G and an element $b \in G$, solve the equation

$y^x = b$ (e.g., $y^x \equiv b \pmod p$ for $G = Z_p^*$)

- Let m denote the smallest integer that is

$$\geq \sqrt{ord(y)} = \sqrt{|G|}$$

- Then $x = vm+w$ with unknown integers $0 \leq v,w < m$.

Observation: The above equation can simply be transformed into $(y^m)^v = b(y^w)^{-1}$

C.25 (continued)

- For $w = 0,1,\dots,m-1$ compute and store the pairs $(w, b(y^w)^{-1})$ in a Table T (*baby steps*).
- Order the entries of T with respect to their second components.
- Compute $r := y^m$
- For $i=0$ to $m-1$ do {

  compute r^i (*giant step*) and check whether r^i is contained in T

  if yes: return $x := im+$(first component of that T-entry)

  }

C.26 Efficiency

- The baby-step giant-step algorithm needs at most $2*|G|^{0.5}$ group operations (compared to $0.5*|G|$ group operations (average value) for exhaustive search). Additionally, the storage and the ordering of $|G|^{0.5}$ data pairs are necessary.

- Example: For $G = Z_p^*$, $p = 999983$, the baby-step giant-step algorithm needs the computation of at most 2*1000 modular multiplications modulo p, and the storage and ordering of 1000 data pairs. The exhaustive search needs 500000 modular multiplications in average.

C.26 Efficiency

- However, large groups G demand gigantic tables. (Example: A 200 bit prime requires 2^{100} table entries.)
- There exist more efficient algorithms to solve the discrete log problem.
- This is yet beyond the scope of this course. We just mention that the *index calculus method* and a new algorithm that uses the number field sieve are most efficient.
- In 2006 the discrete log problem in Z_p^* for a 448 bit prime p was solved.

C.27 The Chinese Remainder Theorem (CRT)

<u>Theorem:</u> Let $n_1,\ldots,n_t$ denote pairwise relatively prime integers (i.e. $\gcd(n_i,n_j) = 1$ for $i \neq j$) and $n:=n_1\ldots n_t$.

(i) To any set of congruences

$y_1 \equiv a_1 \pmod{n_1}$

…

$y_t \equiv a_t \pmod{n_t}$

there exists an integer y with $y \equiv a_j \pmod{n_j}$ for all $j \leq t$.

(ii) In Z_n this solution is unique, and any two solutions $y_{[1]}$ and $y_{[2]}$ in Z differ by a multiple of n.

C.27 (continued)

(iii) There exist integers $N_1,\ldots,N_t$ with the following property:

$N_i \equiv 1 \pmod{n_i}$ but $N_i \equiv 0 \pmod{n_j}$ for all $j \neq i$.

(iv) $y \equiv a_1 N_1 + \ldots + a_t N_t \pmod{n}$

<u>Proof:</u> see literature

<u>More Details:</u> Blackboard

C.28 Hash Functions

- Hash functions map bit strings of arbitrary length to bit strings of fixed length m.

Examples:

- MD5 (m=128)
- SHA-1, RIPEMD160 (m=160)
- SHA-2 family (m $\geq$ 224)
- Whirlpool (m=512)
- …

C.28 (continued)

A hash function H should meet several conditions. In particular:

- (one-way property) Given $h \in \{0,1\}^m$ *it shall not be practically feasible* to find a pre-image x with $H(x)=h$ with non-negligible probability.

Note: Of course, for each $h \in \{0,1\}^m$ infinitely many pre-images should exist. The difficulty is to find them.

C.28 (continued)

- (second pre-image resistance) Given H(x)=h *it shall not be practically feasible* to find a second pre-image x'$\neq$x with H(x')=h with non-negligible probability.
- (collision resistance) *It shall not be practically feasible* to find two values x $\neq$ y with H(x)=H(y) with non-negligible probability.

C.29 Security

(i) Usually the collision resistance is the condition that is hardest to achieve. (Note that the so-called *birthday paradox* limits the necessary number of operations to $2^{m/2}$.)

(ii) Nearly all known successful attacks on hash functions violate the collision resistance.

(iii) MD5 is no longer collision-resistant. Collisions can be generated within about a minute. The needed number of operations is by far smaller than $2^{128/2}=2^{64}$.

(iv) Today no SHA-1 collisions are known. However, the SHA-1 algorithm is doubtlessly not as strong as it was believed some years ago.

C.30 Fields of Application and Efficiency

- Hash functions are used in different areas of cryptography, e.g. for
  - digital signatures ($\rightarrow$ C.b)
  - MACs ($\rightarrow$ B.c, C.b (HMAC))
  - random number generators ($\rightarrow$ B.e)
  - …
- The widespread dedicated hash functions are tailored to 32 bit architectures. Hence they run very fast on computers but are usually slow on smart cards.