Cryptology part 2 Keränen/ Teeriaho Ramk 2006

6. Key agreement

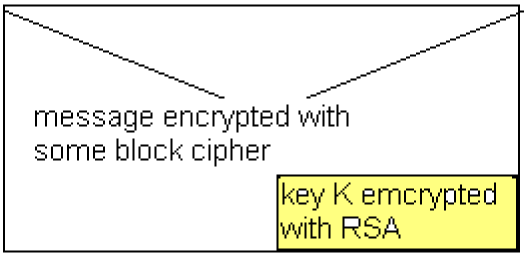
6.1 Digital envelope

Perhaps the most common method of key agreement is that the sender of the message generates a symmetric key, uses the key in encryption with a block cipher and send the key encyrpted with a public key algorithm as an attachment of the message . This is used in so called hybrid cryptosystems, which utilise both symmteric and public key algorithms.

Below is a descritpion of key agreement, which uses RSA.

Digital envelope

1. Alice generates a session key k for a block cipher
2. Alice encrypts the message with the block cipher
3. Alice sends Bob
    a) the ciphertetx
    b) the key k encrypted with Bobs public RSA - keys
4. Bob decrypts the encrypted k using his private RSA key
5. Bob decrypts the message using the key k

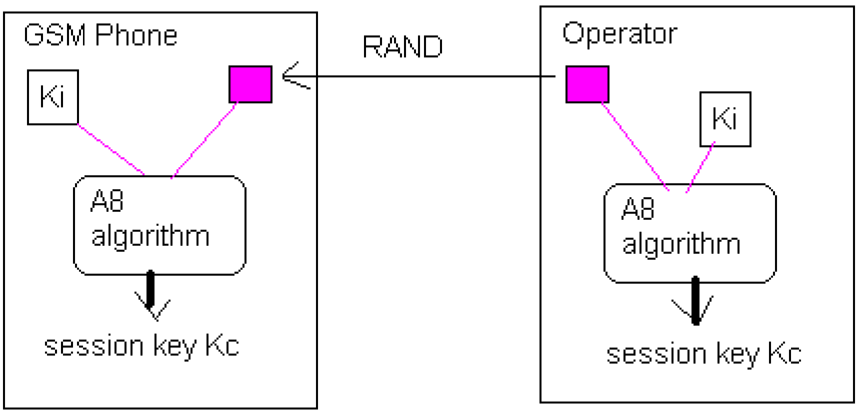


6.2 A8 - key agreement in GSM network

Key agreement in GSM -network (A8 algorithm)

In GSM mobile phones SIM -card and the operator both have the customers SIM - key Ki. This is however not the encryption key in A5. Encryption key Kc is generated for every phone call in the following way:

1. Operator generates a random integer RAND an send it to the SIM - card
2. SIM - card calculate the encryption key Kc from Ki and RAND with A8 algorithm
3. Operator calculates Kc in the same way.



6.3 Diffie Hellman protocol

In the famous speech in 1977 Diffie and Hellman presented the idea of public key ciphers.

Diffie and Hellman also presented a secure and effective way of agreeing on a symmetric key in a network. This method is called Diffie- Hellman key agreement protocol , DH.

Diffie Hellman key agreement protocol

1) A large prime p (1024 bits) is chosen as the basis of the system

Also a generator g of Z_p^* is needed .
(A generator or a primitive element is an element of Z_p^* , the powers of which give all numbers 1 ... (p-1).

2) Users A and B generate themselves two random integers a and b

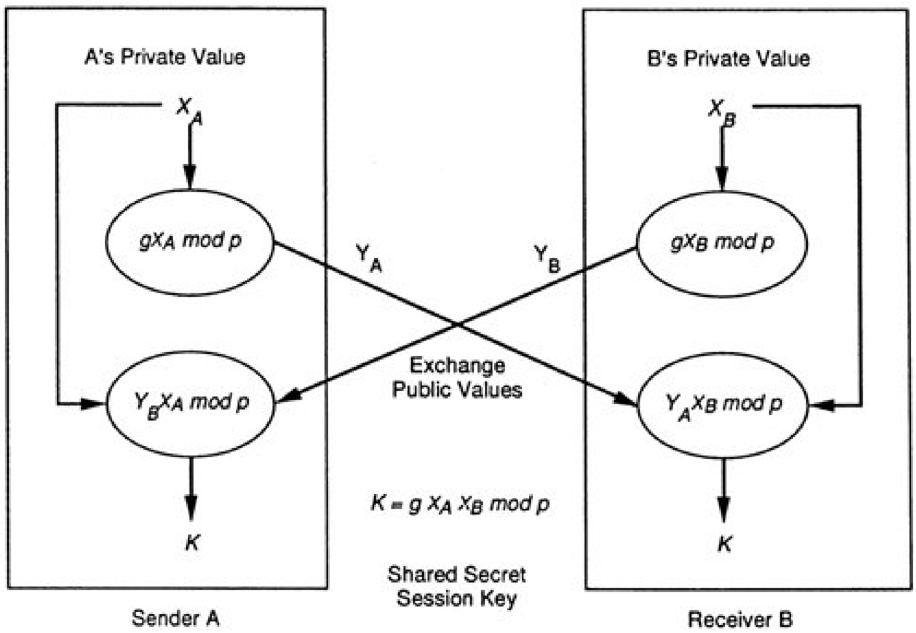
3) A sends B the power jA = g^a mod p
Similarly B sends A the power jB = g^b mod p

4) The symmetric key k (session key) is calculated in the following way :

Alice calculates: k = jB^a mod p
Bob calculates: k = jA^b mod p

Both get the same key k = $g^{a b}$ mod p

DH scheme



■ 6.4 Example of DH

Let p = 983 and g = 511 form the basis of DH protocol

1) Alice and Bob generate random keys

```
p = 983; g = 511;
```

```
a = Random[Integer, {2, p - 1}]
b = Random[Integer, {2, p - 1}]

632
```

```
563
```

2) Alice and Bob calculate their public keys, which they send to each other.

```
jA = PowerMod[g, a, p]
jB = PowerMod[g, b, p]

953
```

```
933
```

3) Both calculate the session key

```
{PowerMod[jB, a, p], PowerMod[jA, b, p]}

{85, 85}
```

=> k = 85

6.5 Discrete logarithm problem DLP

The enemy knows the algorithm, the prime p and the generator g. He can also listen to the channel and find out about the publc keys jA and jB.

To be able to calculate the session key k, he should be able to solve **a** from

$$\mathbf{jA} = g^{a} \ \mathbf{mod\ p}$$

This problem is called DLP : discrete logarith problem. It is one of the hard problems of mathematics with no fast solution. If i p is 1024 - bit integer, it takes years to solve DLP

Definition: Discrete logarithm problem (DLP)

means solving exponent x from

$$\mathbf{y = a^x \bmod p}$$

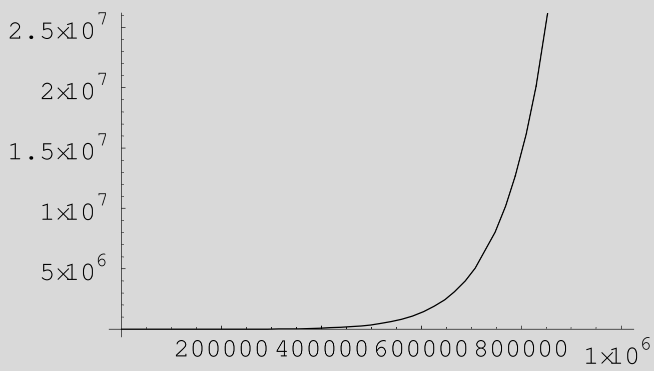
when p , a and $y \in Z_p^*$ are given

According to Addleman (1979) time solving DLP is $t(p) = e^{c\sqrt{p \ln(p)}}$, where c is some constant depending on the speed of the computer.

Later (Coppersmith 1984) faster algorithms have been found, but even today 1024 bit is considered a safe value for p.

Picture of the Addleman's t(p) - function

```
c = 0.005;
Plot[e^(c*Sqrt[p Log[p]]), {p, 1, 1000000}];
```



■ 6.6 Algorithms needed for implementation of DH

A) Prime generation

Prime generation algorithm

1) Generate an odd integer k with required bit length

2) Apply some primality test on k

3) If k is not prime go to step 1

4) Return k

Implementation with Mathematica

```
k = 4;
n = 1000;       (* required bit length *)
While[PrimeQ[k] == False,  (*  test primality *)
 k = Random[Integer, {2^(n-1), 2^n}];
]
k
```

```
907274717969019571126831858481520167726243092200289758355935458071\
 34929518348700290016326724814604685319330438482422853926017980834\
 65422871150895874206956666877381336288814287394332142955015653278\
 71097525361140662000926133535900137174381564811345582619663561050\
 0130600757953042572190940432874242059971
```

B) Primality tests

As we saw, we need a reliable primality test to test whether the integer is prime or not

There exists two types of primality tests

1. **Deterministic tests** give 100% right answer on the question about primality. For very large integers these tests are too slow.

2. **Probabilistic** tests give 100% right answer only when the test result is NOT PRIME.
If the test result is PRIME, there is a non zero possibility that the result is wrong

Rabin-Miller test is the most used primality test

* Test is probabilistic
* If the test result is PRIME, the answer is true with probability $1 - 4^{-K}$, where k is a security parameter you can choose freely.
* If the test result is NOT PRIME, this is 100% true.

Rabin-Miller algorithm

Assume n is the integer to be tested

1. Write n–1 in the form $2^s r$, where r is odd.

2. Choose random integer a between $1 \le a \le$ n-1 .

3. If a^r=1 (mod n) of $a^{2^j r} = -1$ (mod n) for some j between $1 \le j \le$ s-1, the n passes the test

A prime passes the test for all a.

Mathematica - code

```
millerrabin[n_, k_] := Module[ {r, s, j, i, y, a, cond1, cond2, prim},
  s = 0; r = n - 1; cond1 = cond2 = False; prim = True;
  If[EvenQ[n] ∧ n ≥ 3, prim = False];
  (* even numbers >2 are not primes *)

  If[OddQ[n] ∧ n ≥ 3,
   While[EvenQ[r], r = r / 2; s++;];
   (* n-1 presented as 2^s r , where s is prime *)
   i = k;
   While[i ≥ 1,
     a = Random[Integer, {1, n - 1}];
     y = PowerMod[a, r, n];
    cond1 = (y ≠ 1);
     j = 1;
     While[j ≤ s - 1 ∧ y ≠ n - 1,
       y = Mod[y^2, n];
         j++;
      ];
     cond2 = (y ≠ n - 1);
    If[cond1 ∧ cond2, prim = False];
    i--;
   ];
  ];
  prim
 ]
```

Example: Test the primality of 1351 Use k = 5.

```
millerrabin[1351, 5]

False
```

1351 is not prime.
(Because 1351 is small, we can factorize it)

```
FactorInteger[1351]

{{7, 1}, {193, 1}}
```

C) How to find a generator g of the multiplicative group $Z_p{}^*$

According to the group theory the powers of elements of $Z_p{}^*$:{a , a^2, a^3,..., $a^{p-1} = 1$} form cyclic groups of sizes, which are divisors of p–1 . It the size of the cyclic group is maximal p–1 , the element a is called **a generator** of **primitive element.**

Examples of power tables in $Z_{19}{}^*$

```
Table[PowerMod[11, x, 19], {x, 1, 18}]

{11, 7, 1, 11, 7, 1, 11, 7, 1, 11, 7, 1, 11, 7, 1, 11, 7, 1}
```

```
Table[PowerMod[8, x, 19], {x, 1, 18}]

{8, 7, 18, 11, 12, 1, 8, 7, 18, 11, 12, 1, 8, 7, 18, 11, 12, 1}
```

```
Table[PowerMod[3, x, 19], {x, 1, 18}]

{3, 9, 8, 5, 15, 7, 2, 6, 18, 16, 10, 11, 14, 4, 12, 17, 13, 1}
```

Only number 3 is a generator

Number 11 generates a cyclic subgroup with 3 elements.
Number 8 generates a cyclic subgroup with 6 elements.

The sizes of cyclic subgroups are divisors of p– 1.

In the example p–1 = 18 with divisors 1, 2, 3, 6, 9 and 18.
An element a is a generator, if none of the powers a^1, a^2 , a^3 , a^6, a^9 equals 1 mod 19.

The more p–1 has divisors, the more difficult it is to decide whether an element is a generator or not.

Algorithm for finding the generator of $Z_p{}^*$

1) Find all divisors of p – 1: 1, 2, d1, d2, d3 ,... , $\frac{p-1}{2}$, p – 1

2) Choose random a between 2 ... p – 2

3) Calculate powers a^2 mod p, a^{d1} mod p , ... , $a^{\frac{p-1}{2}}$ mod p

4) If for some of those powers is 1, a is not a generator.
Go back to step 2.

Example: Find generator for $Z_{29}{}^*$.

```
p = 29;

d = Divisors[p - 1]

{1, 2, 4, 7, 14, 28}
```

Examine two candidates: 12 and 11.

```
a = 12;
(* Calculate all powers with divisors of p-1 as exponents *)
PowerMod[a, d, p]

{12, 28, 1, 17, 28, 1}
```

Number 12 is not a generator because a^4 (mod 29) = 1 .

```
a = 11;
PowerMod[a, d, p]

{11, 5, 25, 12, 28, 1}
```

Number 11 is a generator , becaus e only a^{p-1} mod 29 = 1.

This algorithm requires the knowledge of divisors of p– 1. In cryptography p –1 can be very large (1024 bit), whence we may not know the divisors.

D) Strong primes

Finding a group generator g is easiest if p– 1 has minimum number of divisors.
That is why we introduce a new concept.

Definition:
Integers p is called a strong prime, if (p – 1)/2 is also a prime.

(Some text books define the concept in more mild way : To be a strong prime it is enough that (p–1)/2 has a large prime factor).

There are benefits in using strong primes as p:
1. It is easier to find generator.
2. DLP is harder to break, if p is strong prime

If p is strong prime, then p–1 has only two factors: 2 and (p–1)/2

If p is a strong prime,
then a $\in Z_p$ is a generator <=> $a^2 \neq 1$ mod p and $a^{\frac{p-1}{2}} \neq 1$ mod p.

Example: Finding a strong prime p and a generator g

1) Find a 400 -bit strong prime p

```
n = 400;
p = Random[Integer, {2^(n-1), 2^n}];
If[Mod[p, 2] == 0, p = p + 1];
While[PrimeQ[p] ∧ PrimeQ[(p - 1)/2] == False,
  p = p + 2;];
p
```

```
141792155503875117383250515405098821236339966240027151230075945632\
 9033343022245413120466475528202950107931780552421712781
```

2) Examine if g = 57395385751 is a generator of Z_p^*

```
g = 57395385751;
PowerMod[g, 2, p] ≠ 1 ∧ PowerMod[g, (p - 1)/2, p] ≠ 1
```

```
True
```

Hence g is a generator and Diffie Hellman protocol can be based on p and g