CS5285
Tutorial 5

Question 1

RSA Assume that we use RSA with the prime numbers $p = 3$ and $q = 11$.

(a) Calculate n and $\phi(n)$.

(b) Given the public exponent $e = 7$, calculate d.

(c) Encrypt the message $M = 9$

(d) Sign the message $M = 15$.

(e) Verify the Signature $S = 9$.

Solution (1)

(a) We have $n = pq = 33$. We know that p and q are prime and therefore $\phi(n) = \phi(pq) = (p-1) \cdot (q-1) = 20$.

(b) We use the extended Euclidean algorithm to obtain $d = 3$, such that $e \cdot d = 1 \mod 20$.

Looking to solve for x and y in form $x.20 + y.7 = 1$

$20 = 2 * 7 + 6$

$7 = 1.6 + 1$

Start working back $1 = 7 - 6$

$1 = 7 - (20 - 2 * 7) = -20 + 3.7$

So $-20 + 3.7 \; mod \; 20 = 3.7 \; mod \; 20 = 21 \; mod \; 20 = 1 \; mod \; 20$

Therefore 3 is modular inverse of 7 mode 20

(c) The ciphertext is calculated as $C = M^e \mod n = 9^7 \mod 33 = 15$.

(d) The signature is calculated as $S = M^d \mod n = 15^3 \mod 33 = 9$.

(e) The verification is calculated as $M = S^e \mod n = 9^7 \mod 33 = 15$.

Question 2

(a) Does the hash function $h(x) = g^x \mod p$ satisfy one-wayness? x in this case is the data being hashed.

(b) From Fermat's little theorem we can see that if $y = x + k \cdot (p-1)$, and k is an integer/ p is prime and not divisible by a, then we should have $g^x \mod p = g^y \mod p$.

Why? $g^{x+k\cdot(p-1)} \mod p \equiv g^x \cdot g^{k\cdot(p-1)} \mod p \equiv (g^x \mod p) \cdot (g^{k\cdot(p-1)} \mod p) \equiv g^x \mod p \cdot (1 \mod p) \equiv g^x \mod p$

Using this fact can you find another message that will result in the same hash as for message $x = 1$ when $p = 19$ and $g = 7$?

(c) Do you think $h(x) = g^x \mod p$ is a good hash function?

What is public key crypto based on?

- Public key crypto is based on mathematical one way functions
  - Easy to compute output given the inputs
  - Difficult to compute input given the output

- Factorisation problem
  - Multiplying two prime numbers
  - Given prime x and y it is easy to compute x.y = z
  - Given z it is not easy to compute x and y

- Discrete logarithm problem
  - Exponentiation of a number
  - Given a, b and prime n is it easy to calculate $z= a^b \bmod n$
  - Given z, a and n it is not easy to compute b

- 'Not easy' means it is currently not computationally feasible…

Question 2a

(a) Does the hash function $h(x) = g^x \mod p$ satisfy one-wayness?

Yes, under the assumption that the discrete logarithm problem is hard, $h(x) = g^x \mod p$ satisfies one-wayness.

Question 2b

(b) From Fermat's little theorem we can see that if $y = x + k \cdot (p-1)$, and k is an integer/ p is prime and is not divisor of a, then we should have $g^x \mod p = g^y \mod p$.

Why? $g^{x+k\cdot(p-1)} \mod p \equiv g^x \cdot g^{k\cdot(p-1)} \mod p \equiv (g^x \mod p) \cdot (g^{k\cdot(p-1)} \mod p) \equiv g^x \mod p \cdot (1 \mod p) \equiv g^x \mod p$

Using this fact can you find another message that will result in the same hash as for message $x = 1$ when $p = 19$ and $g = 7$?

For example if $p = 19$, $g = 7$ and $x = 1$ then we can set y to $y = x + 1 \cdot 18 = 19$ and then easily verify that

$$g^x \mod p = 7^1 \mod 19 = 7$$
$$g^y \mod p = 7^{19} \mod 19 = 7$$

Hash Function

- A cryptographic hash function h(x) should provide
  - Two functional properties
    - Compression – arbitrary length input to output of small, fixed length
    - Easy to compute – expected to run fast
  - Three security properties
    - One-way – given a hash value y it is infeasible to find an x such that h(x) = y (also called pre-image resistance)
    - Second pre-image resistance – given y and h(y), cannot find x where h(x)=h(y)
    - Collision resistance – infeasible to find x and y, with $x \neq y$ such that h(x) = h(y)
- Note: As h is a compression algorithm, there should theoretically be collisions. Collision resistance require that it is hard to find any collision

Question 2c

Do you think $h(x) = g^x \mod p$ is a good hash function?

- One-way function – given hash y it is infeasible to find an x such that h(x) = y?
- Second pre-image resistance – given y and h(y), cannot find x where h(x)=h(y)?
- Collision resistance – infeasible to find x and y, with $x \neq y$ such that h(x) = h(y)?

From question 1(b) we have seen that it is trivial, given an input x, to create an input y such that $h(x) = h(y)$, so $h(\cdot)$ is not second pre-image resistant or collision resistant in general. This is therefore not a good hash function.

Question 3

Suppose we use a hash function $H(x)$ of output length 128 bits, which accepts input in blocks of size 16 characters (16 bytes/128 bits), meaning that a message is always split into blocks of 16 characters and input into the hash function.

(a) We want to hash the message `TheMessage` (10 characters). Because we can only hash 16 characters at a time, we pad the input message with `0`, so the input becomes `TheMessage000000`. As a rule, padding always has to be applied, meaning that if a message is already 16 characters long, we add a whole block of 16 characters of padding. Show that using this padding scheme, it is trivial to come up with a different message which will produce the same hash value.

(b) Show a minor modification of the padding scheme which will resist the attack above.

(c) Assuming that padding scheme is secure, what is the estimated computational effort needed to find a collision on this hash?

Question 3a

- Is it possible to make any other messages that will be the same as:

  TheMessage padded with 000000
- Remember it can be any message…it does not need to make sense…

In fact, any message which consists of the original message (`TheMessage`) plus one or more `0` at the end will be hashed to the same value. So the following messages will be hashed to the same value:

```
TheMessage0
TheMessage00
TheMessage000
TheMessage0000
TheMessage00000
```

- All these message pad to TheMessage000000! So hash result equal

Question 3b

- How could we change the padding to prevent this issue?

Instead of simply padding with `0`, we first add a `1` and then fill the rest of the block with `0`, like this:

```
TheMessage100000
```

Now, even if we try to hash a message such as `TheMessage1`, the padding will transform this to `TheMessage110000` which will result in a different hash value. This padding scheme is thus no longer trivially breakable.

Note that here also we always apply padding, even if it means adding one more block of input. If we did not always add padding, then the two following messages

```
M1 = abcdefghijklmn
M2 = abcdefghijklmn10
```

would be hashed to the same value. If we require that padding is always applied, `M2` would become

```
M2 = abcdefghijklmn10 1000000000000000
```

after padding, resulting in a different hash than `M1`. Modern padding schemes are still more complicated than this to make them even harder to break.

Question 3c

Suppose we use a hash function $H(x)$ of output length 128 bits, which accepts input in blocks of size 16 characters (16 bytes/128 bits), meaning that a message is always split into blocks of 16 characters and input into the hash function.

Assuming that padding scheme is secure, what is the estimated computational effort needed to find a collision on this hash?

- What should we be thinking about here?
- What probability 'problem' did we discuss related to collision resistance?

The effort needed is linked to the Birthday Problem - need average $2^{n/2}$ attempts. So the effort needed is $2^{128/2} = 2^{64}$.

Question 4

Suppose you are using a MAC based on a block cipher in CBC mode (Ci = E(K; Pi XOR Ci-1), IV=0 for C0), and you know the following two messages:

M'= M0 M1

M'' = M2 M3 M4

together with their corresponding MAC tags T' and T''.

(a) Show that you can create a new message M''' = M0||M1||X||M3||M4 and the correct MAC tag T''' without knowing the key K. You can choose any value for X to make this work. For the purposes of calculating the MAC IV is always 0.

A MAC Algorithm

- MAC can be constructed from a block cipher operated in CBC mode (with IV=0).
- Suppose a plaintext has 4 plaintext blocks $P=P_0$, P_1, P_2, P_3
- Suppose K is the secret key shared between sender and receiver.

$C_0 = E(K, P_0)$,

$C_1 = E(K, C_0 \oplus P_1)$,

$C_2 = E(K, C_1 \oplus P_2)$,…

$C_{N-1} = E(K, C_{N-2} \oplus P_{N-1}) = \text{MAC tag}$

Question 4

A CBC MAC is calculated exactly the same as when we are encrypting a message. We just discard all the ciphertext except for the final ciphertext block, which is our MAC

value.

First we explicitly write down the MAC tags for both M‘ and M’’:

$T' = C'1 = E_K(C'0 \text{ XOR } M1) = E_K(E_K(IV \text{ XOR } M0) \text{ XOR } M1) = E_K(E_K(M0) \text{ XOR } M1)$

$T'' = C''2 = E_K(C1 \text{ XOR } M4) = E_K(E_K(C0 \text{ XOR } M3) \text{ XOR } M4) = E_K(E_K(E_K(M2) \text{ XOR } M3) \text{ XOR } M4)$

Remember that for MACs based on a block cipher, the IV is always set to zero. So we can essentially leave out the IV (given the anything XOR with) stays the same.

Question 4

We can now create a new message by setting:

M = M0, M1, M2 XOR T’, M3, M4

Calculating the MAC tag for M we get:

C0 = E_K(M0 XOR IV) = E_K(M0)

C1 = E_K(M1 XOR E_K(M0)) = T’

C2 = E_K(C1 XOR M2 XOR T’) = E_K(M2)

C3 = E_K(EK(M2) XOR M3)

T’’’ = C4 = E_K(E_K(E_K(M2) XOR M3) XOR M4) = T’’

We can thus create a new message with link X = M2 XOR T’ out of two messages M’ and M’’ (and the corresponding tags T’ and T’’), and the MAC tag for M will be T = T’’.

The end!

Any questions…