THE UNIVERSITY OF MELBOURNE

# COMP90043 Cryptography and Security

## Assignment 2


**Student Name:** Lihua Wang

**Student ID:** 1164051


**Semester:** 2020/S2


School of Computing and Information Systems

# CONTENT

## Q1

I use python to implement the functions.

a) Code: function sign (*M, n, d*)

```
# b^p % n
def exp_mod(b, p, n):
    r = 1
    while n:
        if n & 1:
            r = (r * b) % p
        b = (b * b) % p
        n >>= 1
    return r

def sign(M, n, d):
    return exp_mod(M, n, d)
```

b) Code: function verify (*S, n, e, M*)

```
# S = sign (M, n. d)
def verify(S, n, e, M):
    verified_signature = exp_mod(S, n, e)
    return verified_signature == M
```

c) Code: function: BlindSign (M, n, d, e, x)

```
def blindSign(M, n, d, e, x):
    blind = M * (exp_mod(x, n, e))
    signature_of_blind = sign(blind, n, d)
    unblinded_signature_of_blind = divide(signature_of_blind, x, n)
    return signature_of_blind, unblinded_signature_of_blind

"""if x and n are coprime: return    S/x % n
    else: return 0 in this case"""
def divide(S, x, n):
    if x % n == 0:
        return 0, 1, n
    next_x = x
    next_n = n
    q = []
    while next_x % next_n != 0:
        q.append(-1 * (next_x // next_n))
        (next_x, next_n) = (next_n, next_x % next_n)
    (next_x, next_n, gcd) = (1, q.pop(), next_n)
    while q:
```

```
        (next_x, next_n) = (next_n, next_n * q.pop() + next_x)
    return (S * next_x) % n
```

d) The signature of message:

714735052714609508708447646211793036853468139790841608766403671513418708819109223941327741592728291368659269090199188493637570994083029695739886979323220492965395965694153442481059677924375066324035249015634065161797660195205226170

e) The signature of blind message:

728617448551059933645503667798818303626193266826340595101763896882618313836606534959902807979302596973046439853301472863850659635393488370618122104038003627669361062160974652186006467348827378278660302474490228843309271603189562130

The signature of original message:

714735052714609508708447646211793036853468139790841608766403671513418708819109223941327741592728291368659269090199188493637570994083029695739886979323220492965395965694153442481059677924375066324035249015634065161797660195205226170

**Q2**

a) The known plaintext attack will be against this encryption method. The attacker could try to calculate all the represents of 26 alphabets by the RSA encryption algorithm $C_i \equiv (i)^e \; mod \; n \; (0 \leq i < 26)$. Then decrypt the ciphertext $CT = (Z_1, Z_2, Z_3, \ldots, Z_l)$ by calculating the decryption algorithm $D(Z_j) = i : Z_j = C_i, 0 \leq i < 26 \; for \; each \; j = 1, \ldots, l$ and $Z_j$ could be obtained from the last step.

b) For this case, the countermeasure can be done by following: modifying the encryption algorithm to $C_i = (i)^{ie} \; mod \; n \; (0 \leq i < 26)$ for each alphabet.

**Q3**

a) In order to recover the message without knowing Jaiden's or Jiajia's private key , we can do the following strategy:

Since different e are relatively prime, then $gcd(e_1, e_2) = 1$. Hence, $\exists x, y \in \mathbb{Z}$ such that $xe_1 + ye_2 = 1$. We can use $XGCD$ to find $x, y$. Then, we can use $x, y$ to find $M$ as follows:

$$C_1^x C_2^y = (M^{e_1})^x (M^{e_2})^y = M^{xe_1+ye_2} = M^1 = M \ (mod\ n)$$

b) Jiajia can recover Jaiden's private key using the following strategy.

Firstly, since we know $d_2$, we can factor $n$. Then, we can use $XGCD$ to obtain $d_1$.

In order to factor $n$, let $k = d_2e - 1$. Since $ed - 1 = 0\ mod\ \phi(n)$, then $k$ is a multiple of $\phi(n)$. We also know that $k$ must be even. Hence, let $k = 2^t r$ with $r$ odd and $t \geq 1$. Now, we pick a random generator $g$, where $1 < g < N$, and compute the sequence $g^{k/2}, g^{k/4}, \ldots, g^{k/2^t}$. We determine the first sequence element $x \neq \pm 1$, where $gcd(x-1, n) > 1$.

If no such element exists, then choose another g and try again.

Otherwise, let $p = gcd(x-1, n)$ and $q = n/p$.

Now that we have $p$ and $q$, we can compute $\phi(n) = (p-1)(q-1)$. Using $XGCD$, we can now find $d_1 = e_1^{-1}\ mod\ \phi(n)$.

**Q4**

a) (i) A sends a request message for a new session key to B. This includes the ID of A and a unique nonce $N_a$ encrypted with their private key $K_a$.

(ii) In addition to A's original message, B also sends a mirror image of A's unique message to KDC, including B's ID and a a unique nonce $N_b$ encrypted with the private key $K_b$.

(iii) KDC sends a response message to B. The message contains two parts, one for A and one for B. Each component is encrypted with the private key of the desired recipient, including:

- The one-time session key, $K_s$, to be used for the session.
- The ID of the communication user on the other side (e.g. $ID_a$ for B).
- The original nonce, allow users to match responses with corresponding requests.

(iv) B forwards A's component, which they received from the KDC, to A. Now, both A and B have access to the session key.

b) A believes that $K_s$ is fresh since it is included in message (4) together with $N_A$ (and hence message (4) must have been constructed after message (1) was sent).

B believes $K_s$ is fresh since the reason is the same above. $K_s$ is included in message (3) together with $N_B$ (and hence message (3) must have been constructed after message (2) was sent).

c) Assume KDC is security and considered reliable.

The nonce is secured; since $N_a$ and $N_b$ are encrypted with their master key when the requests are sent over the network assure that the original request was not altered before reception by KDC. In addition, the information replied by KDC is also encrypted with the master key, which can protect from eavesdropping. A uses the master key $K_a$ to decrypt the message received from B. Since only A and KDC can access the master key $K_a$, A verifies that the received message is actually from KDC. The same anthentification process of B.

d) This scheme cannot guarantee the authenticity of both A and B. If B or attacker C sends a message $ID_c||E(K_c, N_c)$ (or any message containing a matching identity and encrypted random number pair) to KDC, the authenticity of B may be compromised. This means that when A receives a response, the responder's ID will not be what they believed in.

The authenticity of B can be compromised if either B or an attacker C sends the KDC the message $ID_c||E(K_c, N_c)$ (or any message consisting of a matching identity and encrypted nonce pair). This means that when A receives a response, the identity of the responder will not be who they believed it to be.

**Q5**

a) **Variable input size:** Satisfied. If the length $M_i$ of the last block is shorter than the fixed size, it can be filled to the specified fixed length.

b) **Fixed output size:** Satisfied. This is because a hash function is any function that can be used to map digital data of any size to digital data of a fixed size. In this case, the output is always mod $n$.

c) **Efficiency (easy to calculate):** On the one hand, since a message containing many blocks may require numerous expensive exponentiation computations. On the other hand, $XOR$ is a relatively simple operation, so $H(x)$ can be calculated quickly compared with other encryption operations of addition, subtraction, multiplication and division: it does not generate carry, and each part can be executed in parallel.

d) **Preimage resistant:** Satisfied, because generally hash function always based on the RSA problem. Given $H(x)$, we can construct $M_1||M_2||...||M_m$ in an arbitrary manner. Since we know $H(M)$, we have a $n$ bit string of $0s$ and $1s$. For each $0$ in $H(M)$, we can assign an even number of $1s$ (or all $0s$) to that position in the $l$ blocks. For each $1$ in $H(M)$, we can assign any odd number of $1s$ to that position in the $l$ blocks.

e) **Second preimage resistant:** Not Satisfied. Second preimage means finding $x' \neq x \rightarrow h(x') = h(x)$. Next, we can create a function $f: \{0,1\}^{2m} \rightarrow \{0,1\}^m$. As we proved in the above, different strings can have the same result $XOR$: $1010 \oplus 1111 = 0101$ and so does $0000 \oplus 0101$. Therefore, the messages $10101111$ and $00000101$ have the same $XOR$ and hence will get mapped to the same hash, and find two $x, x'$ with $x \neq x' \wedge H(x) = H(x')$ resulting in the second preimage.

f) **Collision resistant:** Not Satisfied. Looking for $M1 \neq M2$ but $H(M_1) = H(M_2)$. It need not be based on a given $H(M)$. Therefore, for some blocks $m$, $M_1||M_2||...||M_m$ can be constructed arbitrarily. We could change an even number of $1s$ from a given position to $0$, from $0$ to $1$, etc. This will change the message, but not the hash value.

**Reference**

1. What are preimage resistance and collision resistance, and how can the lack thereof be exploited? [online] Available at: https://crypto.stackexchange.com/questions/25308/why-might-xors-lead-to-hash-functions-lacking-2nd-pre-image-resistance [Accessed 03 Oct. 2020].

2. Why Might XOR's Lead to Hash Functions Lacking 2nd Pre-image Resistance? [online] Available at: https://crypto.stackexchange.com/questions/1173/what-are-preimage-resistance-and-collision-resistance-and-how-can-the-lack-ther [Accessed 03 Oct. 2020].

## Appendix

### *RSA.py*

```
# b^p % n
def exp_mod(b, p, n):
    r = 1
    while n:
        if n & 1:
            r = (r * b) % p
        b = (b * b) % p
        n >>= 1
    return r


"""if x and n are coprime: return    S/x % n
    else: return 0 in this case"""
def divide(S, x, n):
    if x % n == 0:
        return 0, 1, n
    next_x = x
    next_n = n
    q = []
    while next_x % next_n != 0:
        q.append(-1 * (next_x // next_n))
        (next_x, next_n) = (next_n, next_x % next_n)
    (next_x, next_n, gcd) = (1, q.pop(), next_n)
    while q:
        (next_x, next_n) = (next_n, next_n * q.pop() + next_x)
    return (S * next_x) % n


def sign(M, n, d):
    return exp_mod(M, n, d)


# S = sign(M, n. d)
def verify(S, n, e, M):
    verified_signature = exp_mod(S, n, e)
    return verified_signature == M


def blindSign(M, n, d, e, x):
    blind = M * (exp_mod(x, n, e))
```

```
    signature_of_blind = sign(blind, n, d)
    unblinded_signature_of_blind = divide(signature_of_blind, x, n)
    return signature_of_blind, unblinded_signature_of_blind


def main():
    # SID: 1164051
    x = 4051
    M = 3141592656405193
    n =
1139631134290681913324518075250462509444792614577115360833700594253534083115108212461164873379591734542309312064780949257819665132832661342154198437454459926525649486600336464897081397167045104842672493488133506984881500857942197501
    e = 65537
    d =
2072957680681022794565143350330464253031321659272440333933281166989087050798053771266543548767583665330861850424073864444696973004489931710794150224779958495944479817291689146397299649575294462296501865902209905922547000385620 58305
    print("The signature is\n", sign(M, n, d), "\n")
    print("The signature is valid? ", verify(sign(M, n, d), n, e, M), "\n")
    print("The blind signature is\n", blindSign(M, n, d, e, x)[0], "\n")
    print("The unblind signature is\n", blindSign(M, n, d, e, x)[1], "\n")
    print("The unblind signature is equal to signature of original message? ",
          blindSign(M, n, d, e, x)[1] == sign(M, n, d))


if __name__ == '__main__':
    main()
```

***Result:***

The signature is

714735052714609508708447646211793036853468139790841608766403671513418708819
109223941327741592728291368659269090199188493637570994083029695739886979323
220492965395965694153442481059677924375066324035249015634065161797660195205
226170

The signature is valid?    True

The blind signature is

728617448551059933645503667798818303626193266826340595101763896882618313836

6065349599028079793025969730464398533014728638506596353934883706181221040380036276693610621609746521860064673488273782786603024744902288433092716031895621 30

The unblind signature is

714735052714609508708447646211793036853468139790841608766403671513418708819109223941327741592728291368659269090199188493637570994083029695739886979323220492965395965694153442481059677924375066324035249015634065161797660195205226170

The unblind signature is equal to signature of original message? True