Modern Cryptography Project

Implement a Scenario that uses a Blind Signature Scheme

Team Members:

HAMSINI G	**CB.EN.U4CYS22023**
VETTRIVEL G	**CB.EN.U4CYS22024**
GUNATEET DEV	**CB.EN.U4CYS22025**
DHARSHIKA S	**CB.EN.U4CYS22021**

SIGNATURE SCHEME:

Signature schemes are cryptographic techniques used to provide authentication, integrity, and non-repudiation for digital messages or documents. The primary purpose of a digital signature is to verify the origin and authenticity of a message, ensuring that it has not been altered during transmission and that it indeed came from the claimed sender.

Steps involved are

Key Pair Generation:

- Like other asymmetric cryptography techniques, digital signatures involve the use of key pairs: a private key and a public key.
- The private key is kept secret and is used to generate the signature.
- The public key is shared with others and is used to verify the signature.

Signing Process:

- To create a digital signature, the sender's private key is applied to a hash function, generating a unique value called a hash value or message digest.
- The sender's private key is then used to encrypt this hash value, creating the digital signature.
- The resulting signature is sent along with the original message.

Verification Process:

- The recipient uses the sender's public key to decrypt the digital signature, obtaining the hash value.
- The recipient independently computes the hash value of the received message using the same hash function.
- If the computed hash value matches the decrypted hash value from the digital signature, the signature is considered valid.

ELGAMAL DIGITAL SIGNATURE SCHEME

A digital signature is an electronic signature that demonstrate the authenticity of an electronic document or message in digital communication and uses encryption techniques to give confirmation of original and unmodified documentation . Two main properties are required in digital signature namely, authentication, data integrity. In the digital signature scheme, there are two participants, namely, the signer and the verifier. The signer uses his private key to sign a document and then sends this signature to the verifier. The verifier receives the signature, then he uses a public key to verify the validity of the signature . ElGamal Algorithm is one of the digital signature schemes. It was invented by Taher ElGamal in (1984). It is based on the difficulty of solving discrete logarithm problem (DLP). There were two participants, namely, the signer and the verifier and three phases, namely key generation,

signing, and verification. In key generation phase, parameters of the signer are initialized. In signing phase, the signer uses his private key to sign a document and then sends this signature to the verifier. In verification phase, the verifier uses signer's public key to verify the validity of the signature, which can be summarized as in scheme .

BLIND SIGNATURE SCHEME

Blind signature scheme is an extension of the digital signature scheme which allows a requester to get a signature from a signer without revealing any information about the message it signed .

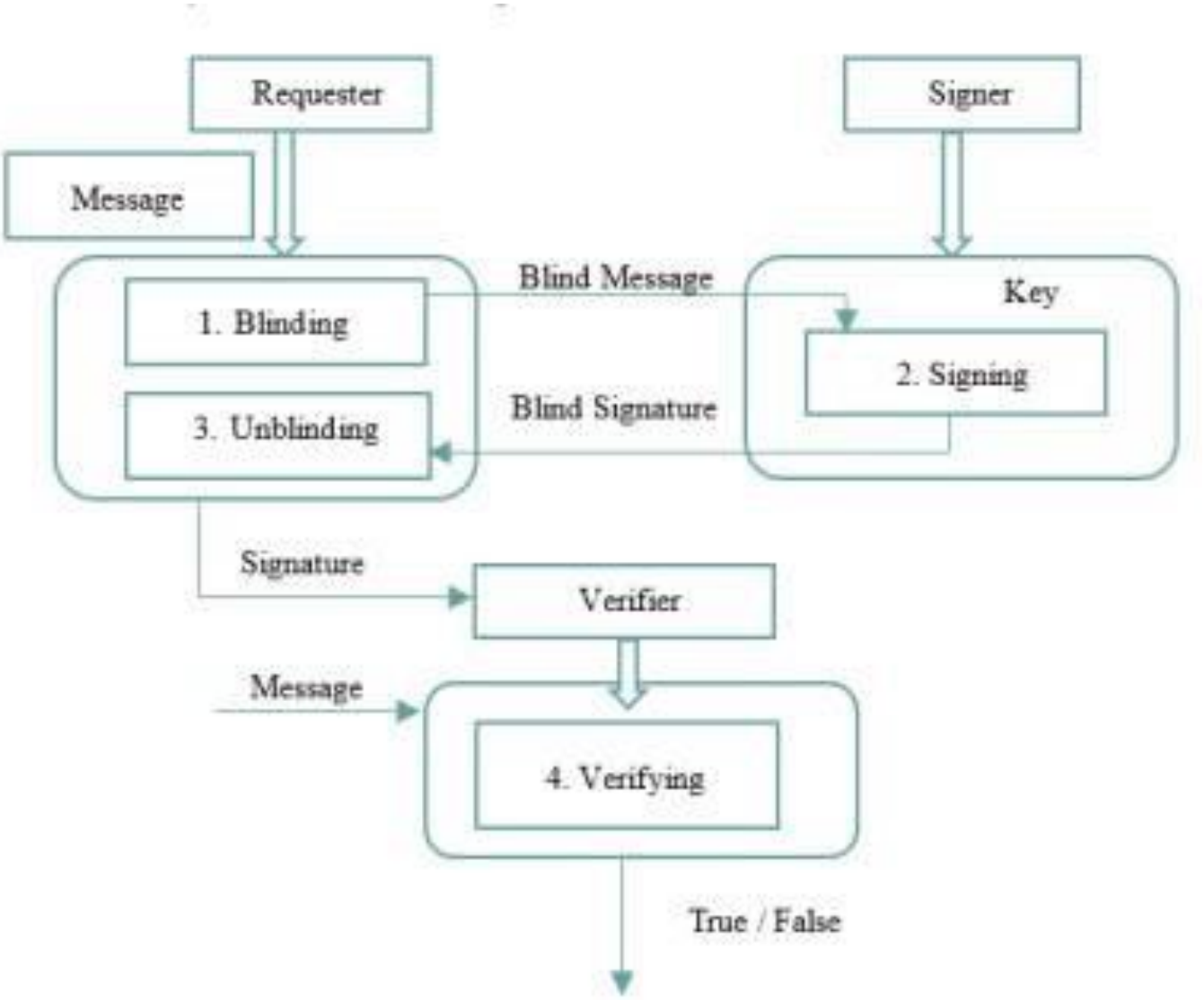
The blind signature scheme must achieve four requirements, namely, **Correctness, Blindness, Unforgeability and Anonymity.** Blind signature scheme has three participants, namely, the **signer, the requester and the verifier**.

Initialization (Key generation): All the system parameters of both requester and signer are initialized in this phase. □ Blinding: The requester blinds the message by selecting a blind factor and sends the blind message to the signer. □

Signing: Once the signer gets the blinded message, he use his private key to sign it and then sends back the blind signature to the requester. □ Unblinding: Once the requester receives it, he uses his blind factor to recover the signer's digital signature from the blinded message and sends it to the verifier.

Verifying: Verifier can use the signer's public key to verify whether the signature is authentic or not.

We have used a signature scheme under blind signature scheme



This is one of the blind signature schemes which we will be implementing further.

R. L. Shen, et al.'s Scheme :

A blind signature scheme based on the DLP and the modified ElGamal signature in .There are 18 modified ElGamal digital signature schemes shown by Dr. L. Harn . The signature equation for the modified ElGamal signature represented as

$$ax = bk + c \bmod p - 1$$

The verification equation could be:

$$y\,a = r\,bg\,c \bmod p$$

where (a, b, c) are the three parameters from the set of values(m, r, s), each parameter (a, b, c)could be a mathematical combination of (m, r, s), with certain criteria for the combination between these parameters.

The signature and verification equations that were the basis for R. L. Shen, et al.'s scheme were shown below.

Signature equation:

$$sx = k + m + r$$

Verification equation:

$$y^s = rg^{r+h(m)} \bmod p$$

In this scheme, there are three participants, namely, the requester, the signer, and the verifier; and five phases, namely,

(1) initialization phase

(2) blinding phase

(3) signing phase,

(4) unbinding phase

(5) verification phase

Initialization phase: The signer chooses a large prime p and g as a primitive root of p. Then randomly chooses a number x ($2 < x < (p-2)$) to computes $y = g^x \bmod p$. the signer publishes (y, g, p) as the public key, keeps x as the private key, and chooses a one-way hash function $h(.)$ such as SHA1 or MD5.

Blinding phase: First, the requester sends a request to the signer for signing message m. The signer choose random number k', such that $gcd(k', p-1) = 1$ and computes $r' = g^{k'} \bmod p$ then sends r' to the requester. The requester chooses the set of values(a, b, c) are relatively prime to $(p-1)$ and computes $r = r'^a y^{bg^c} \bmod p$ and $h(m)$ generated by the hash function $h(.)$. Next he blinds the value $h(m)$ with the blind equation $m' = a^{-1}c + hm + r - r'k' \bmod p - 1$,then he sends the value m' to the signer.

Signing phase: After the signer receives the value m', he computes $s' = x^{-1}k' + m' + r'k' \bmod (p-1)$. Then he sends the value s' to the requester. Unblinding phase: The requester computes $s = as' + b \bmod (p-1)$, after receiving s' from the signer, to obtain the message-signature(m, r, s).
Then he send the message signature pair (m, r, s) to the verifier. Verifying phase: When the verifier receives the message signature pair(m, r, s), he use the one-way hash function $h(.)$ and the public key (y, g, p) to verify the legitimacy of the signature by checking, $v1 = y^s \bmod p$ $v2 = rg^{r+h(m)} \bmod p$.

If $v1 = v2$, then the verification passes; else the verification fails

SECURITY ANALYSIS :

In this section we show that R. L. Shen, et al.'s scheme in satisfied all the requirements of blind scheme namely, Correctness, Blindness, Unforgeability, and Anonymity.

Correctness: The following steps confirm the verification equation

$y\ s = r\ g\ h\ m + r \bmod p$

$g\ xs \equiv g\ kg\ r + h\ m \bmod p$

$xs \equiv k + r + h\ m \bmod p - 1$

$x\ a\ s' + b \equiv a\ k' + bx + c + r + h\ m \bmod p - 1$

$xas' \equiv a\ k' + c + r + h\ m \bmod p - 1$

$a\ x\ s' - k' \equiv c + h\ m + r \bmod p - 1$ Multiplied

simultaneously by

$a - 1\ xs' - k' \equiv a - 1\ c + h\ m + r \bmod p - 1$

Subtracted simultaneously by

$x\ s' - k' - r' \equiv a - 1\ c + h\ m + r - r' \bmod p - 1$

$x\ s' - k' - r' \equiv m' \bmod p - 1$

$s'x \equiv k' + m' + r' \bmod p - 1.$

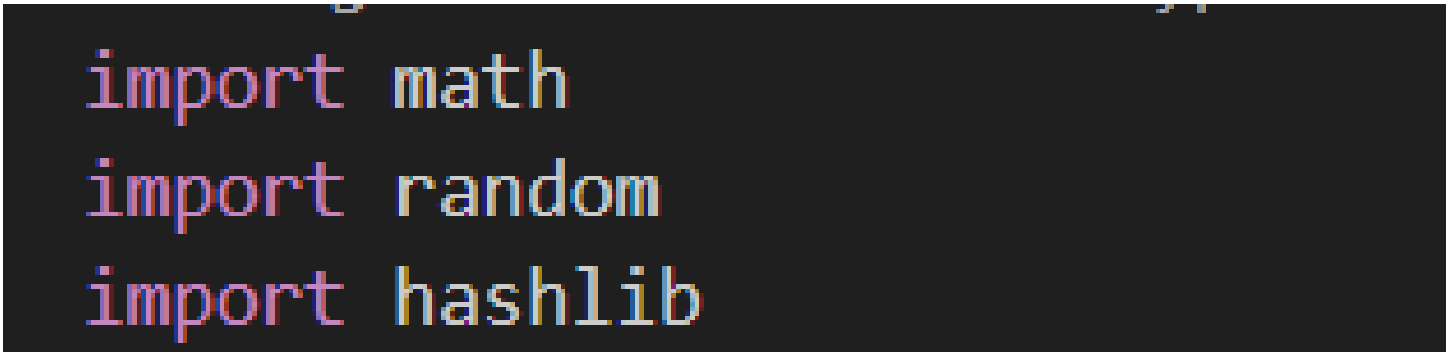
Blindness: The signer cannot obtain the message m from blinded equation $m' = a - 1$ c + $h\ m$ + r – $r'\ mod\ p$ — 1 , because the signer has three unknown parameters, namely, $a, c\ and\ r,$ so the signature scheme is blind.

Unforgeability: No one can forge a valid signature pair (r, s) on the message m to pass the verification, because it is very difficult to solve the discrete logarithm problem.

Anonymity: If the signer wants to trace the blind signature, he keeps the set of values(m', r', s'). When the requester publishes the message signature pair (m, r, s) in public, the signer unable to obtain any information from the set of values that he keeps. Because the signer does not know the values including $a, b, c\ and\ r$, he cannot link the relation between the message- signature pair and the blind signature.

Implementation of R. L. Shen, et al.'s Scheme :

We will be implementing this with the help of python.



- The **import math** line imports the math library, which provides a variety of mathematical functions.
- The **import random** line imports the random library, which provides a variety of random number generation functions.
- The **import hashlib** line imports the hashlib library, which provides a variety of cryptographic hash functions.

```
def gcd(a,b):
    return a if b==0 else gcd(b,a%b)
def ext_eu(a,b):
    if(a<b):
        a,b=b,a
    s=0;s1=1;t=1;t1=0;rem=b;divid=a
    while(rem>0):
        quot=divid//rem; rem1=rem
        divid,rem=rem,divid-quot*rem
        s1,s=s,s1-quot*s
        t1,t=t,t1-quot*t
    return [rem1,s1%a,t1%a]
def modinverse(a,b):
    return (ext_eu(a,b)[2])
```

gcd(a, b):

- This function calculates the greatest common divisor of two integers a and b using the Euclidean algorithm.
- The base case is when b becomes 0. In this case, the GCD is a.
- Otherwise, it recursively calls itself with arguments b and a % b.

ext_eu(a, b):

- This function calculates the extended Euclidean algorithm, which finds the GCD of a and b as well as the coefficients s and t such that as + bt = gcd(a, b).
- It initializes some variables like s, s1, t, t1, rem, and divide based on the input values a and b.
- It then enters a loop where it iteratively performs Euclidean algorithm steps and updates the coefficients s1, s, t1, and t.
- The loop continues until rem becomes 0, and the function returns an array [gcd, s1 % a, t1 % a], where gcd is the GCD of a and b, and s1 % a and t1 % a are the coefficients.

Mod inverse(a, b):

- This function calculates the modular inverse of a modulo b. It utilizes the extended Euclidean algorithm to find the coefficients s and t such that as + bt = gcd(a, b).
- The modular inverse exists only when gcd(a, b) is 1 (i.e., a and b are coprime).
- The function returns the modular inverse, which is the coefficient obtained from the extended Euclidean algorithm.

```
def initialising_phase(number):
    o = 1
    roots = []
    z = 0
    r = number - 1
    while z < 20:
        k = math.pow(r, o)
        k = k % number

        while k > 1:
            o += 1
            k *= r
            k = k % number
        if o == (number - 1):
            roots.append(r)
            z += 1
        o = 1
        r -= 1
    roots= list(reversed(roots))
    selected_root = random.choice(roots)
    count=0
    while count<100:
        x = random.randint(3, number - 3)
        if (gcd(x, number - 1) == 1):
            count+=1

            set_of_x_values.append(x)

    x=random.choice(set_of_x_values)
    print("secret number ",x)
    y = pow(selected_root, x) % number
    #t=()
    public_key=(y,selected_root,number)
    print(public_key)
    return public_key,x
```

Finding Primitive Roots:

- The function initialising_phase(number) initializes certain variables and aims to find primitive roots modulo number.
- It uses a loop that iterates until it finds 20 primitive roots. A primitive root is an integer whose powers, taken modulo number, generate all the non-zero residues coprime to number.
- The loop involves checking different values of r to see if they are primitive roots.

Selecting a Random Primitive Root:

- After finding the primitive roots, it randomly selects one from the list of primitive roots.

Generating Random Values:

- The code then generates 100 random values (x) such that 3 <= x <= number - 3 and gcd(x, number - 1) == 1. These values are collected in a list named set_of_x_values.

Selecting a Random x:

- From the generated set of random x values, it selects one random x and prints it as the "secret number."

Calculating Public Key:

- It calculates y as the result of raising the selected primitive root to the power of the selected random x, modulo number.
- The public key is then formed as a tuple (y, selected_root, number) and printed.

Returning Public Key and Secret x:

- The function returns the public key and the secret x value.

```
def random_coprime(n):
    while True:
        #k = random.randint(2, n-2)
        k=1067

        if math.gcd(k, n-1) == 1:
            print("Random chose prime, kdash",k)
            return k
```

Infinite Loop:

- The code is wrapped in a while True loop, which means it will keep running indefinitely until a specific condition is met.

Random Integer k:

- Inside the loop, a variable k is assigned the value 1067. Note that there is a commented-out line (# k = random.randint(2, n-2)) which suggests that the initial intention might have been to generate a random integer between 2 and n-2. However, in the current state, k is hardcoded to be 1067.

Check for Coprimality:

- The code then checks whether k is coprime to n-1 using the math.gcd(k, n-1) function.
- If the greatest common divisor (gcd) of k and n-1 is equal to 1, it means k and n-1 are coprime.

Print and Return:

- If a coprime k is found, it prints a message indicating that a random coprime was chosen (print("Random chose prime, kdash", k)).
- The function then returns the value of k.

```
def generate_relatively_prime_numbers(p):
        a=499
        b=501
        c=1021
        while True:
            a = random.randint(2, p-1)
            b = random.randint(2, p-1)
            c = random.randint(2, p-1)




            if a != b and b != c and a != c and gcd(a, p-1) == 1 and gcd(b, p-1) == 1 and gcd(c, p-1) == 1:
                print("value of a",a)
                print("value of b",b)
                print("value of c",c)
                return a, b, c
```

In this code we are generating the relatively prime numbers for algorithm also

we are using random numbers for security purposes,

here are some steps how we are achieving it

Initialization:

- Initially, three variables a, b, and c are assigned fixed values (a=499, b=501, c=1021). However, these values are later overwritten in the loop.

Infinite Loop:

- The code is wrapped in a while True loop, which means it will keep running indefinitely until a specific condition is met.

Random Integer Generation:

- Inside the loop, three random integers (a, b, and c) are generated using random.randint(2, p-1). This ensures that the values are between 2 and p-1.

Check for Relatively Prime:

- The code then checks whether a, b, and c are pairwise different and relatively prime to p-1. The condition a != b and b != c and a != c ensures that the values are different from each other.
- The conditions gcd(a, p-1) == 1, gcd(b, p-1) == 1, and gcd(c, p-1) == 1 ensure that each of the generated values is relatively prime to p-1.

Print and Return:

- If three suitable values are found, the code prints the values of a, b, and c.
- The function then returns the values of a, b, and c as a tuple.

```
def blinding_phase_signer(pkey):
    y=pkey[0]
    g=pkey[1]
    p=pkey[2]
    K=random_coprime(p)
    rdash=pow(g,K,p)%p
    print("value of rdash",rdash)
    return rdash,K
```

This is the blinding phase from a signer side .

Input Extraction:

- The function takes a public key pkey as input, which is expected to be a tuple of three values: y, g, and p.

Extracting Public Key Components:

- It extracts the components of the public key (y, g, and p) from the pkey tuple.

Generating a Random Co-Prime:

- It calls the function random_coprime(p) to generate a random integer K that is coprime to p-1. The coprimality condition ensures that the modular inverse of K

exists modulo p-1.

Blinding Operation:

- It then calculates rdash as the result of raising g to the power of K, modulo p. This operation is often referred to as blinding, and rdash is the blinded value.

Print and Return:

- The code prints the value of rdash.
- The function returns a tuple (rdash, K).

```
def blinding_phase_requester(m):
    a,b,c=generate_relatively_prime_numbers(p)
    print("This is the value of c ",c)

    rdashhed=blinding_phase_signer(pub_key)[0]
    r1 = (pow(rdashhed, a, p) * pow(y, b, p) * pow(g, c, p)) % p
    print("r value",r1)
    print("hello")

    h_m = int(hashlib.sha1(str(m).encode()).hexdigest(), 16)
    #h_m=50
    a_inv = modinverse(a, p-1)

    print("inverse of a",a_inv)
    addedhash=c+h_m+r1
    a_inverse_multiplied=a_inv*addedhash
    m_blinded=(a_inverse_multiplied-rdashhed)%(p-1)
    #m_blinded = ((a_inv*((c+h_m+r1)))-rdashhed)%(p-1)


    return m_blinded,a,b,r1,h_m
```

This part of the code deals with blinding phase from requester side.

- The function calls generate_relatively_prime_numbers(p) to obtain three random integers (a, b, and c), each of which is relatively prime to p-1.
- It prints the value of c.

It calls blinding_phase_signer(pub_key) to obtain the blinded value (rdashhed) and the blinding factor (K). pub_key is assumed to be a public key.

Calculate Blinded Value (r1):

- It calculates r1 using the blinded value rdashhed and the random values a, b, and c in a formula involving modular exponentiation (pow function).

Hash the Message:

- It calculates the hash value h_m of the message m using the SHA-1 hashing algorithm.

Calculate Modular Inverse:

- It calculates the modular inverse (a_inv) of a modulo p-1.

Blinding and Unblinding:

- It computes addedhash = c + h_m + r1.
- It then calculates a_inverse_multiplied = a_inv * addedhash.
- Finally, it computes m_blinded as (a_inverse_multiplied - rdashhed) % (p-1).

Return the Results:

- The function returns a tuple containing the blinded message m_blinded, along with the values of a, b, r1, and h_m.

```
def signing_phase(m_blinded):

    print("Secret key value",secret_key_x)
    rdashed,kay=blinding_phase_signer(pub_key)

    x_inv = modinverse(secret_key_x, p-1)
    #x_inv=1069
    print("x inverse",x_inv)
    temp=0
    temp=(temp+(m_blinded + rdashed))%(p-1)
    temp=(temp+kay)%(p-1)
    s1=(x_inv*temp)%(p-1)
   # s1 =( x_inv * (kay + ((m_blinded + rdashed)%(p-1))%(p-1)) % (p-1))%(p-1)
    return s1
```

Print Secret Key:

- The function prints the value of the secret key (secret_key_x).

Blinding Phase Signer:

- It calls blinding_phase_signer(pub_key) to obtain the blinded value (rdashed) and the blinding factor (kay). pub_key is assumed to be a public key.

Calculate Modular Inverse:

- It calculates the modular inverse (x_inv) of the secret key (secret_key_x) modulo p-1.

Calculate Intermediate Value (temp):

- It initializes a temporary variable temp to 0.
- It updates temp by adding the blinded message (m_blinded) and the blinded value from the blinding phase (rdashed).
- It then adds the blinding factor (kay) to temp.

Calculate Signature Component (s1):

- It calculates s1 as the result of multiplying the modular inverse of the secret key (x_inv) with the intermediate value (temp), modulo p-1.

Return Signature Component:

- The function returns the calculated value of s1.

```
def unblinding_phase(s1,a,b,m,rrr):

    s=(((a*s1)%(p-1))+b)%(p-1)
    print("value of s",s)

    message_signature_pair=(m,rrr,s)
    return message_signature_pair
```

This is the unblinding phase as we discussed above, the function takes the shown parameter and computes **s** which is further used to create a message signature pair which includes **s.**

After computing message_signature_pair ,the computed pair will be returned from this function.

```
def verifying_phase(message_signature_pair,h_m,sdashvalue):
    v1 = pow(y, message_signature_pair[2], p)
    print("s value from message signature pair",message_signature_pair[2])
    print("sdashed value from signer's value",sdashvalue)
    print("this is v1",v1)
    v2 = (mod_exp(g, message_signature_pair[1], p) * mod_exp(g, h_m, p) * (message_signature_pair[1]%p))%p
    print("this is v2",v2)
```

In this phase we compute v1 and v2 using the parameters as explained above. These values of v1 and v2 are used to verify the signature .while the verifier finds both the values equal ,the signature will be verified otherwise the verifier can reject the signature.

```
def mod_exp(base, exp, mod):
    result = 1
    base = base % mod

    while exp > 0:
        if exp % 2 == 1:
            result = (result * base) % mod

        exp = exp // 2
        base = (base * base) % mod

    return result
```

This is function we have used to calculate the modulus by taking required parameters to calculate and returning the result.

```
pub_key,secret_key_x=initialising_phase(1151)
print(pub_key)
y=pub_key[0]
g=pub_key[1]
p=1151
selected_choice = 50

message_blind,a,b,rrr,h_m=blinding_phase_requester(selected_choice)
print("Public key ",pub_key)
print("hash of m",h_m)

print("Selected Choice:",selected_choice)
print("message blinded: ",message_blind)


sdashed=signing_phase(message_blind)
print("sdashed from signing phase",sdashed)
m_s_pair=unblinding_phase(sdashed,a,b,selected_choice,rrr)
print("message signature pair",m_s_pair)
verifying_phase(m_s_pair,h_m,sdashed)
```

Here, we are calling the required functions as defined above to perform the task we need and printing the desired result.