**CNS LAB EXPERIMENTS**

Name :- Shreya Khambe

PRN :- 21510025

**7.** **Implementation of RSA Algorithm**

The **RSA algorithm** (Rivest–Shamir–Adleman) is a public-key cryptographic algorithm that is widely used for secure data transmission. RSA encryption and decryption algorithm, which is a widely used public-key cryptosystem. RSA involves key generation, encryption using a public key, and decryption using a private key.

RSA (Rivest-Shamir-Adleman) encryption working :-

**1. Key Generation:**

* Two large prime numbers, p and q, are selected.
* Calculate n = p \* q. This n forms part of both the public and private keys.
* Calculate the Euler's totient function, ϕ(n) = (p - 1) \* (q - 1).
* Choose a public exponent e, typically a small prime number (common choice is 65537) that is coprime with ϕ(n).
* Calculate the private key exponent d, which is the modular inverse of e modulo ϕ(n), i.e., d ≡ e⁻¹ (mod ϕ(n)).
* The **public key** is (n, e).
* The **private key** is (n, d).

**2. Encryption:**

* The sender converts the plaintext message into a number M using a predefined encoding scheme (e.g., UTF-8).
* The sender encrypts the message using the receiver’s public key (n, e) with the formula: C= M^e mod n .where C is the ciphertext.

**3. Decryption:**

* The receiver uses their private key (n, d) to decrypt the ciphertext C using the formula: M=C^d mod n.This retrieves the original message M (in number form).
* The receiver then converts M back into readable text using the reverse of the encoding scheme.

**4. Security:**

* RSA security relies on the difficulty of **factoring** the large number n into its prime factors p and q. If n is large enough (e.g., 2048 bits), factoring it is computationally infeasible.

**Summary:**

* **Public Key (n, e)**: Used for encryption, shared with everyone.
* **Private Key (n, d)**: Used for decryption, kept secret.
* **Encryption**: Convert plaintext to a number, raise it to the power of e modulo n.
* **Decryption**: Raise the ciphertext to the power of d modulo n to retrieve the original plaintext.

import random

from sympy import mod\_inverse, isprime

# step 1. Key generation

# Function to generate a large prime number

def generate\_prime\_number():

    while True:

        num = random.randint(100, 999)  # Generate random numbers in a range

        if isprime(num):

            return num

# Function to compute GCD

def gcd(a, b):

    while b != 0:

        a, b = b, a % b

    return a

# Generate RSA Key

def generate\_rsa\_keys():

    # Step 1: Choose two large prime numbers p and q

    p = generate\_prime\_number()

    q = generate\_prime\_number()

    # Step 2: Compute n = p \* q

    n = p \* q

    # Step 3: Compute Euler’s Totient function: phi(n) = (p-1)\*(q-1)

    phi\_n = (p - 1) \* (q - 1)

    # Step 4: Choose an integer e such that 1 < e < phi\_n and gcd(e, phi\_n) = 1

    while True:

        e = random.randint(2, phi\_n - 1)

        if gcd(e, phi\_n) == 1:

            break

    # Step 5: Compute the modular inverse of e mod phi\_n

    d = mod\_inverse(e, phi\_n)

    # Public key (e, n) and private key (d, n)

    return ((e, n), (d, n))

# Step 2. RSA Encryption (Using public key)

def encrypt\_rsa(public\_key, plaintext):

    e, n = public\_key

    ciphertext = [pow(ord(char), e, n) for char in plaintext]

    return ciphertext

# Step 2. RSA Decryption (Using private key)

def decrypt\_rsa(private\_key, ciphertext):

    d, n = private\_key

    plaintext = ''.join([chr(pow(char, d, n)) for char in ciphertext])

    return plaintext

# Generate and Display RSA Keys

public\_key, private\_key = generate\_rsa\_keys()

# Display the keys

print("Public Key:", public\_key)

print("Private Key:", private\_key)

message = input("Enter Plaintext : ")

print(f"Plaintext : {message}")

# Encrypt the message

encrypted\_message = encrypt\_rsa(public\_key, message)

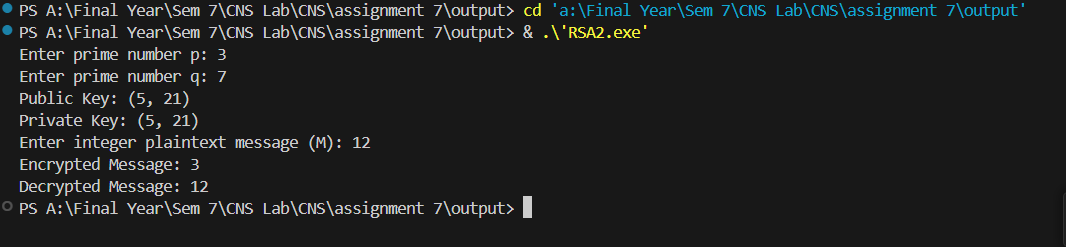
print("Encrypted Message:", encrypted\_message)

# Decrypt the message

decrypted\_message = decrypt\_rsa(private\_key, encrypted\_message)

print("Decrypted Message:", decrypted\_message)

Output :-



**Step 1: Input Two Prime Numbers (p and q)**

* You input two prime numbers (p and q), which are used to generate the RSA keys.

**Step 2: Compute n and phi\_n**

* **n = p \* q**: n is the product of the two primes and is used in both the public and private keys.
* **phi\_n = (p - 1) \* (q - 1)**: This is Euler’s Totient function, used to find e (public key exponent) and d (private key exponent).

**Step 3: Choose the Public Key Exponent e**

* The program randomly chooses e such that 1 < e < phi\_n and gcd(e, phi\_n) = 1 (i.e., e and phi\_n are relatively prime).

**Step 4: Compute the Private Key Exponent d**

* **d = modInverse(e, phi\_n)**: d is the modular inverse of e mod phi\_n, and it’s part of the private key.

**Step 5: Output Public and Private Keys**

* The public key is **(e, n)**, and the private key is **(d, n)**.

**Step 6: Input an Integer Message (M)**

* You input an integer M (the plaintext message you want to encrypt).

**Step 7: Encrypt the Message**

* The encryption formula is **C = M^e mod n**, where C is the ciphertext (the encrypted message).

**Step 8: Decrypt the Ciphertext**

* The decryption formula is **M = C^d mod n**, which retrieves the original integer message.

**Final Output**

* The program shows both the encrypted message and the decrypted message. You can see that after decryption, the original message M is recovered.

**Summary of Key Formulas:**

* **Encrypt**: C = M^e mod n (Ciphertext)
* **Decrypt**: M = C^d mod n (Plaintext)

This process ensures secure encryption and decryption using the RSA algorithm!

**8.** **Implement the Diffie-Hellman Key Exchange algorithm for a given problem.**

The **Diffie-Hellman key exchange** is a cryptographic method used to securely share a secret key between two parties (e.g., Alice and Bob) over an insecure channel. This secret key can be used for encrypting and decrypting messages in future communication.

**How Diffie-Hellman Works :**

1. **Public Values**:
   * Two publicly known numbers are agreed upon:
     + p: A large prime number (modulus).
     + g: A primitive root modulo p (base).
2. **Private Keys**:
   * Both parties (Alice and Bob) select their **private keys** (random numbers) that are kept secret:
     + Alice's private key = a.
     + Bob's private key = b.
3. **Public Keys**:
   * Each party calculates and exchanges their **public keys** using the formula:
     + Alice computes her public key as A = g^a mod p.
     + Bob computes his public key as B = g^b mod p.
4. **Shared Secret Calculation**:
   * After exchanging public keys, both parties compute a **secret key** using each other’s public key and their own private key:
     + Alice calculates: secret\_key\_A = B^a mod p.
     + Bob calculates: secret\_key\_B = A^b mod p.
5. **Result**:
   * Both Alice and Bob now have the same secret key (secret\_key\_A = secret\_key\_B), which they can use for secure communication.

This secret key is never transmitted directly; it's computed independently by each party, ensuring secure key exchange over an insecure channel.

#include <iostream>

#include <cmath>

#include <cstdlib>

using namespace std;

// Function to compute (a^b) % p

long long power(long long a, long long b, long long p) {

    long long result = 1;

    a = a % p;  // Ensure 'a' is in the range [0, p-1]

    while (b > 0) {

        // If b is odd, multiply 'a' with the result

        if (b % 2 == 1)

            result = (result \* a) % p;

        // Divide b by 2

        b = b / 2;

        // Square 'a'

        a = (a \* a) % p;

    }

    return result;

}

// Diffie-Hellman key exchange

void diffie\_hellman(long long q, long long alpha) {

    // Both parties choose their private keys (a and b)

    long long a = rand() % (q-2) + 1;  // Alice's private key

    long long b = rand() % (q-2) + 1;  // Bob's private key

    // Calculate public keys (A = alpha^a % q and B = alpha^b % q)

    long long A = power(alpha, a, q);  // Alice's public key

    long long B = power(alpha, b, q);  // Bob's public key

    // Each party computes the shared secret key

    long long shared\_secret\_alice = power(B, a, q);  // Alice computes the shared secret

    long long shared\_secret\_bob = power(A, b, q);    // Bob computes the shared secret

    cout << "Alice's Public Key (A): " << A << endl;

    cout << "Bob's Public Key (B): " << B << endl;

    cout << "Alice's Shared Secret: " << shared\_secret\_alice << endl;

    cout << "Bob's Shared Secret: " << shared\_secret\_bob << endl;

    // Both shared secrets should match

    if (shared\_secret\_alice == shared\_secret\_bob) {

        cout << "Shared secret successfully established!" << endl;

    } else {

        cout << "Error: Shared secrets do not match!" << endl;

    }

}

int main() {

    // Prime number p and generator g

    long long q = 23;  // Example prime number

    long long alpha = 5;   // Example primitive root

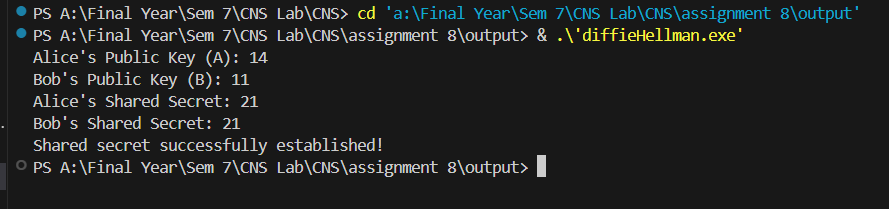
    // Perform Diffie-Hellman key exchange

    diffie\_hellman(q, alpha);

    return 0;

}

Output :-



**9. Calculate the message digest of a text using the SHA-1 algorithm.**

The SHA (Secure Hash Algorithm) is a family of cryptographic hash functions that are designed to take an input (or message) and return a fixed-size string of bytes. This output, commonly referred to as a digest, is unique for every unique input. SHA is widely used for data integrity verification, digital signatures, and password hashing.

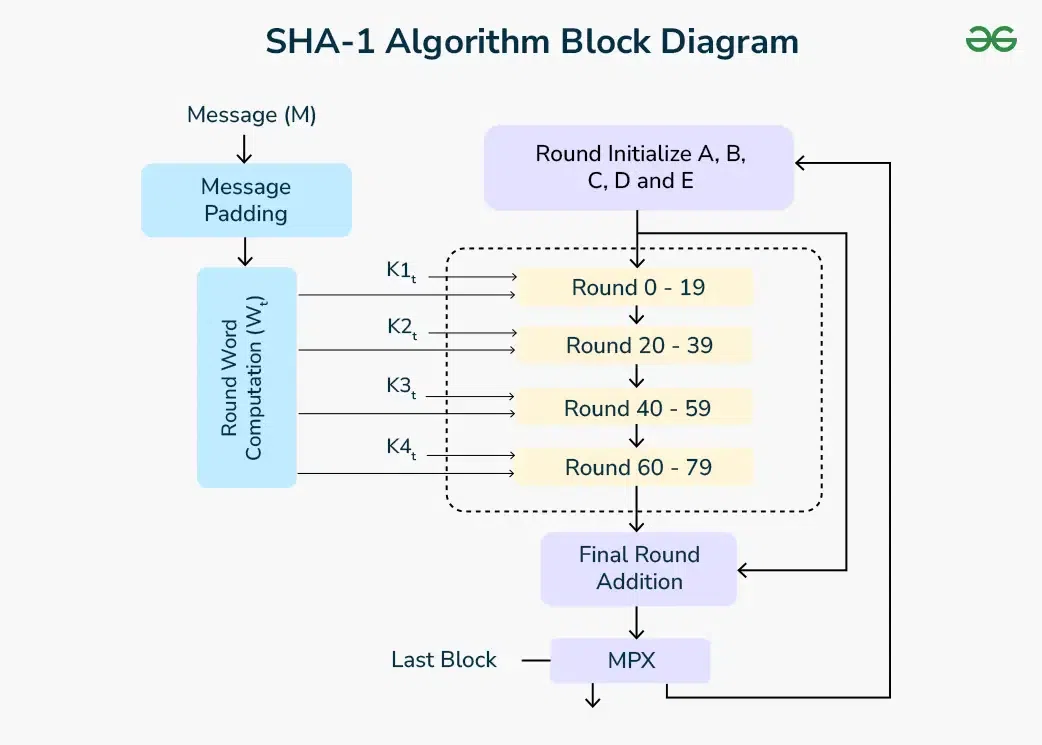
Key Characteristics of SHA:

1. Fixed Output Size: Regardless of the input size, SHA functions produce a fixed-length hash (digest). For example, SHA-256 produces a 256-bit (32-byte) hash.
2. Pre-image Resistance: Given a hash value, it is computationally infeasible to find an input that hashes to that value.
3. Small Change, Big Impact: A small change in the input should drastically change the hash output (this is known as the avalanche effect).
4. Collision Resistance: It should be extremely difficult to find two different inputs that produce the same hash output.
5. One-Way Function: It is computationally hard to reverse the hash and obtain the original input.

How SHA Works:

The SHA algorithms are designed to take an arbitrary-length input and process it through a series of mathematical transformations to produce a fixed-size hash. The process generally involves:

1. Padding the Input: The input is padded to ensure its length is compatible with the block size of the algorithm.
2. Breaking the Input into Blocks: The input is divided into fixed-size blocks.
3. Iterative Processing: Each block of the input is processed iteratively through the SHA compression function, which involves bitwise operations, modular arithmetic, and mixing functions.
4. Final Output: After all blocks are processed, the final output is the hash (or digest).



import java.math.BigInteger;

import java.security.MessageDigest;

import java.security.NoSuchAlgorithmException;

public class SHA1 {

    public static String encryptThisString(String input) {

        try {

            MessageDigest md = MessageDigest.getInstance("SHA-1");

            byte[] messageDigest = md.digest(input.getBytes());

            BigInteger no = new BigInteger(1, messageDigest);

            String hashtext = no.toString(16);

            while (hashtext.length() < 40) {

                hashtext = "0" + hashtext;

            }

            return hashtext;

        }

        catch (NoSuchAlgorithmException e) {

            throw new RuntimeException(e);

        }

    }

    public static void main(String args[]) throws NoSuchAlgorithmException {

        System.out.println("HashCode Generated by SHA-1 for:");

        String input1 = "Shreya";

        System.out.println(input1 + " : " + encryptThisString(input1));

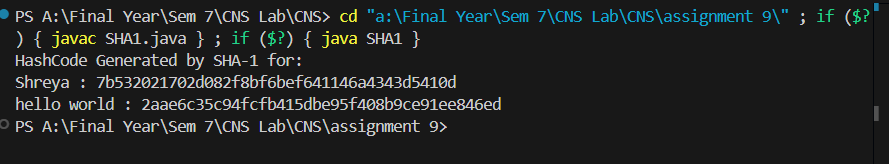
        String input2 = "hello world";

        System.out.println(input2 + " : " + encryptThisString(input2));

    }

}

**Output :-**

****

**Import Statements**:

* BigInteger: Used to convert the hashed byte array into a readable format (hexadecimal).
* MessageDigest: Used to perform the SHA-1 hashing.
* NoSuchAlgorithmException: Handles cases where the requested hashing algorithm (SHA-1) is unavailable.

**encryptThisString method**:

* Takes a string as input and converts it into a SHA-1 hash.
* **MessageDigest** instance for "SHA-1" is created.
* The input string is converted to a byte array and hashed.
* The hashed byte array is converted into a positive BigInteger.
* The BigInteger is converted to a hexadecimal string.
* Leading zeros are added until the hash is 40 characters long.
* The final hash string is returned.

**main method**:

* Takes two strings, "Shreya" and "hello world", hashes them using the encryptThisString method, and prints the hashes.

**10. Implement the SIGNATURE SCHEME – Digital Signature Standard**

The Digital Signature Standard (DSS) defines algorithms for generating and verifying digital signatures, specifically using the Digital Signature Algorithm (DSA). DSA is a public key algorithm that allows a user to generate a digital signature that can be verified by others to confirm the authenticity and integrity of a message.

The process of digital signing consists of:

1. Key Generation: Generate a pair of public and private keys.
2. Signature Generation: The message is hashed, and the signature is generated using the private key.
3. Signature Verification: The signature is verified using the public key.

A diagram of a communication system

Description automatically generated

import java.security.KeyPair;

import java.security.KeyPairGenerator;

import java.security.PrivateKey;

import java.security.PublicKey;

import java.security.SecureRandom;

import java.security.Signature;

import java.util.Scanner;

public class DSS {

    private static final String SIGNING\_ALGORITHM = "SHA256withRSA";

    private static final String RSA = "RSA";

    private static Scanner sc;

    public static byte[] Create\_Digital\_Signature(byte[] input, PrivateKey Key) throws Exception {

        Signature signature = Signature.getInstance(SIGNING\_ALGORITHM);

        signature.initSign(Key);

        signature.update(input);

        return signature.sign();

    }

    public static KeyPair Generate\_RSA\_KeyPair() throws Exception {

        SecureRandom secureRandom = new SecureRandom();

        KeyPairGenerator keyPairGenerator = KeyPairGenerator.getInstance(RSA);

        keyPairGenerator.initialize(2048, secureRandom);

        return keyPairGenerator.generateKeyPair();

    }

    public static boolean Verify\_Digital\_Signature(byte[] input, byte[] signatureToVerify, PublicKey key) throws Exception {

        Signature signature = Signature.getInstance(SIGNING\_ALGORITHM);

        signature.initVerify(key);

        signature.update(input);

        return signature.verify(signatureToVerify);

    }

    public static String bytesToHex(byte[] bytes) {

        StringBuilder hexString = new StringBuilder(2 \* bytes.length);

        for (int i = 0; i < bytes.length; i++) {

            String hex = Integer.toHexString(0xff & bytes[i]);

            if (hex.length() == 1) {

                hexString.append('0');

            }

            hexString.append(hex);

        }

        return hexString.toString().toUpperCase();

    }

    public static void main(String args[]) throws Exception {

        String input = "WCE Sangli";

        KeyPair keyPair = Generate\_RSA\_KeyPair();

        byte[] signature = Create\_Digital\_Signature(input.getBytes(), keyPair.getPrivate());

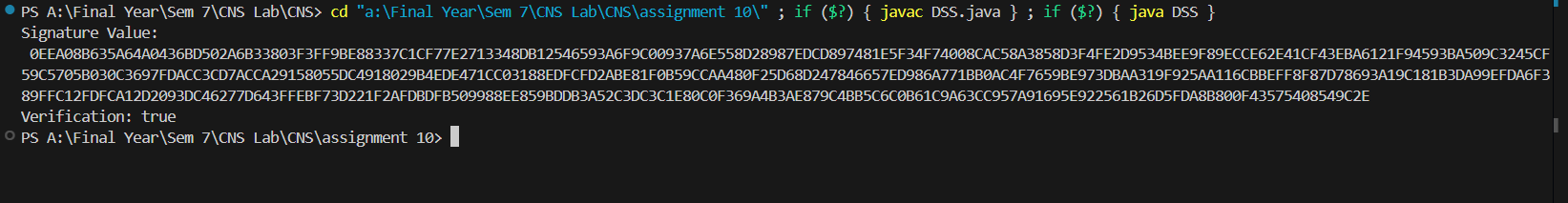
        System.out.println("Signature Value:\n " + bytesToHex(signature));

        System.out.println("Verification: " + Verify\_Digital\_Signature(input.getBytes(), signature, keyPair.getPublic()));

    }

}

Output :-



**1. Key Pair Generation**

* **Generate RSA Key Pair**:
  + The program uses the KeyPairGenerator class to create a pair of RSA keys (private and public).
  + The initialize method sets the key size to 2048 bits, ensuring strong security. A SecureRandom instance is used to add randomness to the key generation process.

**2. Digital Signature Creation**

* **Create Digital Signature**:
  + The Create\_Digital\_Signature method takes an input string (in byte form) and the private key.
  + A Signature object is initialized with the signing algorithm (SHA-256 with RSA).
  + The input data is fed into the signature object, and the sign method generates a unique digital signature based on the input and the private key.
  + The resulting signature is a byte array.

**3. Signature Representation**

* **Convert Signature to Hexadecimal**:
  + The program includes a helper method, bytesToHex, which converts the byte array of the signature into a readable hexadecimal string format. This makes it easier to display the signature.

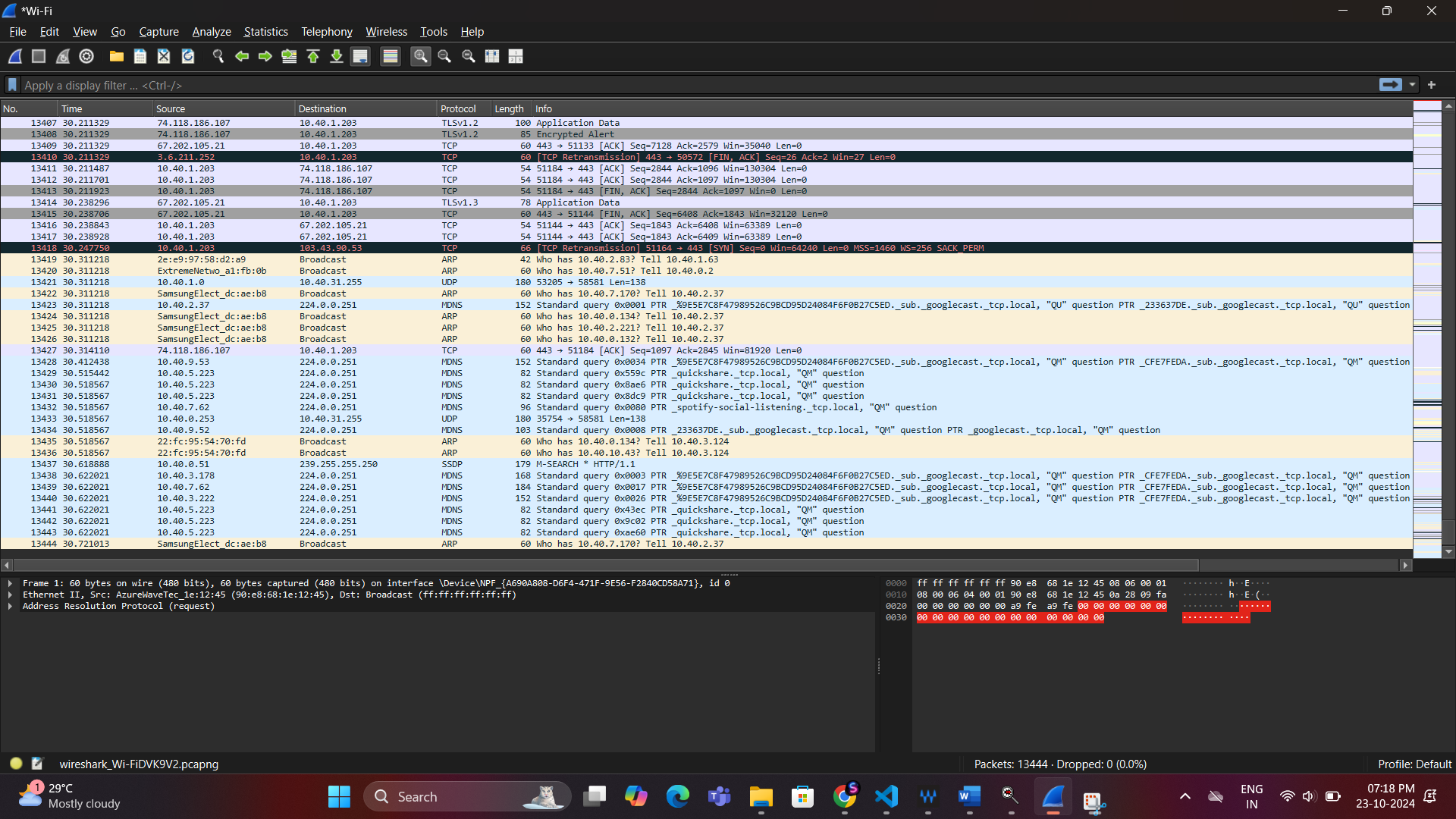
**4. Signature Verification**

* **Verify Digital Signature**:
  + The Verify\_Digital\_Signature method takes the original input, the generated signature, and the public key.
  + It initializes another Signature object for verification.
  + The method checks if the signature matches the input data using the public key.
  + It returns true if the signature is valid (indicating the data has not been tampered with) and false if it is not.

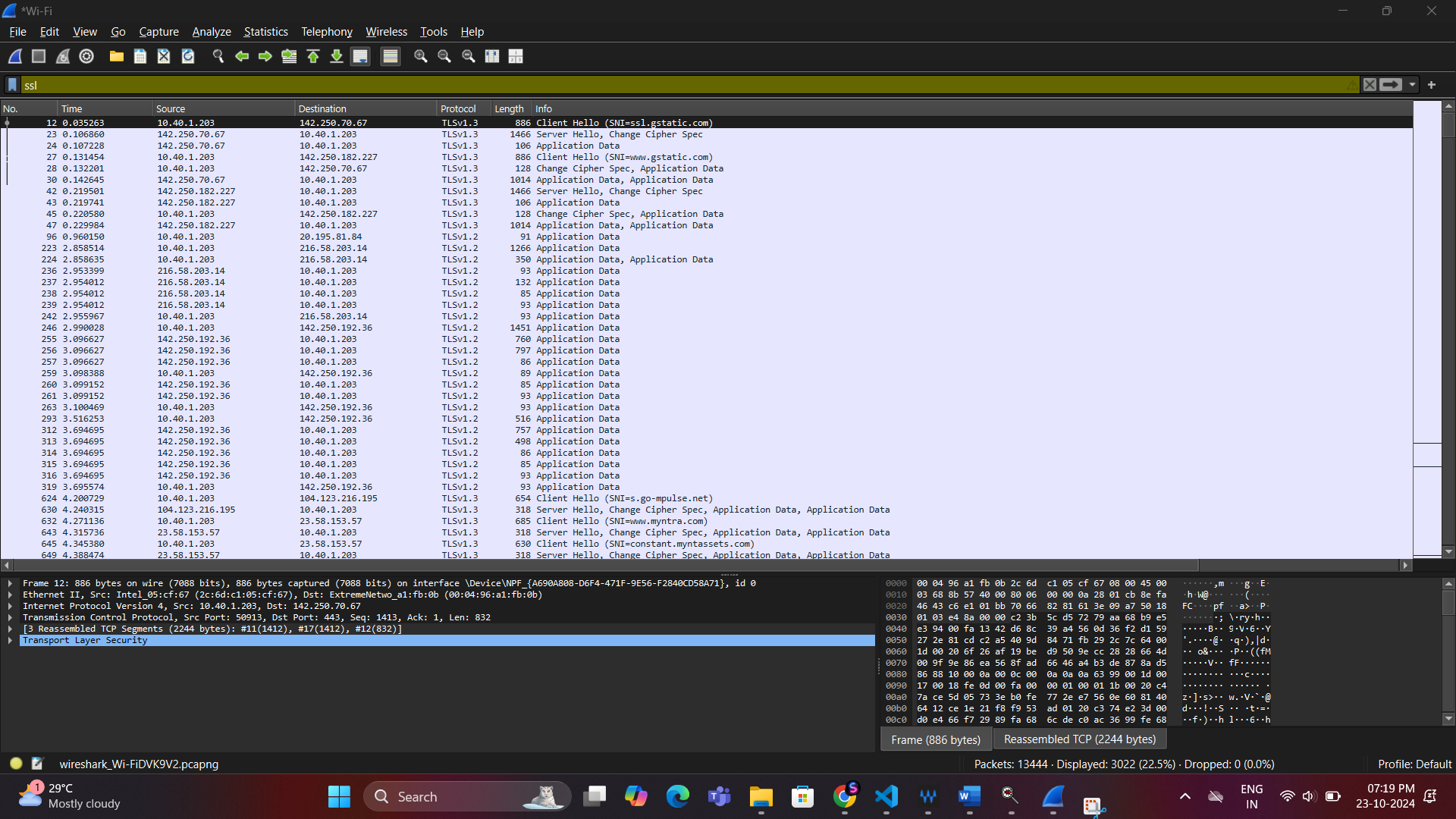
**11. Demonstration of SSL using Wireshark**

**Website :-** [www.myntra.com](http://www.myntra.com)

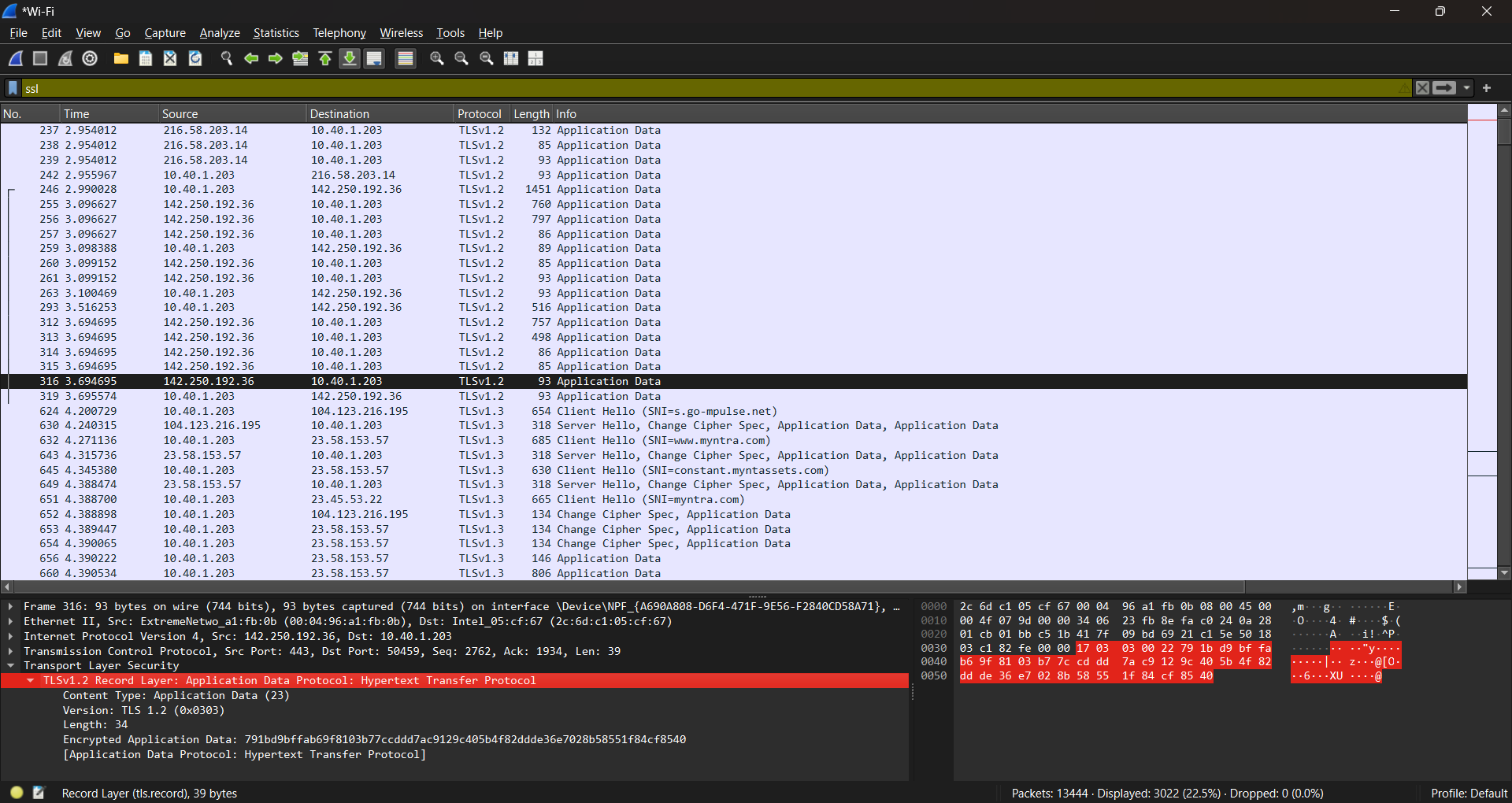
1)Open then trace



2) Inspect the Trace

****

**Q1.What is the Content-Type for a record containing “Application Data”?**

****

•Content-Type/Opaque Type: This field in the TLS record layer indicates the type of data contained in the record. It's used to distinguish between different phases of the SSL/TLS protocol.

•Value 23 (hexadecimal 0x17): Represents Application Data, which is encrypted data exchanged after the SSL/TLS handshake is complete. At this stage, the communication between the client and server is secured, and this data typically contains actual application-level content like HTTP requests or responses (for HTTPS).

Common Content-Type Values:

•20 (0x14): Change Cipher Spec

•21 (0x15): Alert

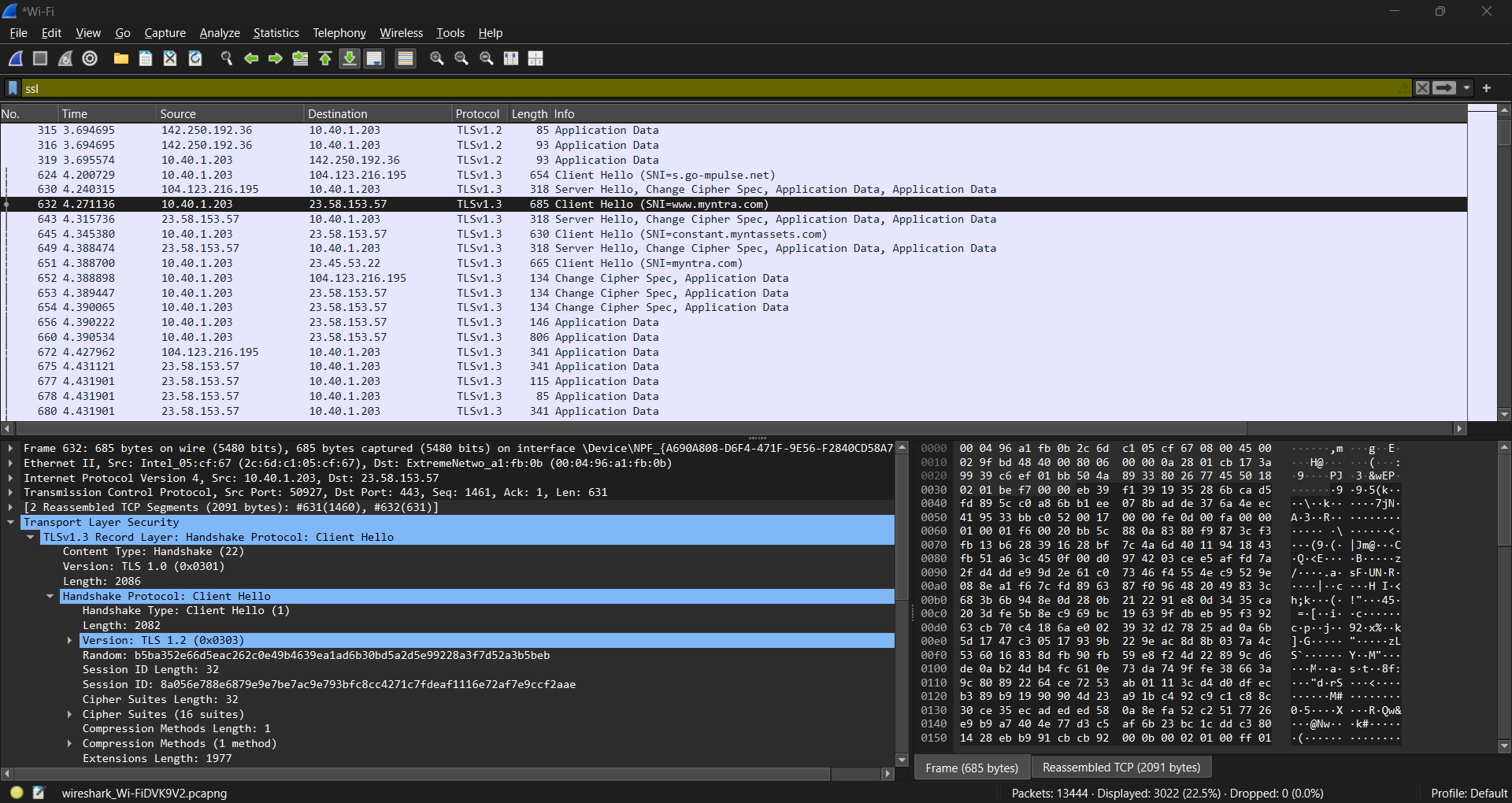
•22 (0x16): Handshake

•23 (0x17): Application Data

In our case, since the value is 23, it means the TLS record is carrying Application Data, which is the encrypted content sent after the secure session has been established. This is typically where user data, such as web page content in HTTPS, is transmitted securely.

3) The SSL Handshake

Client Hello :-



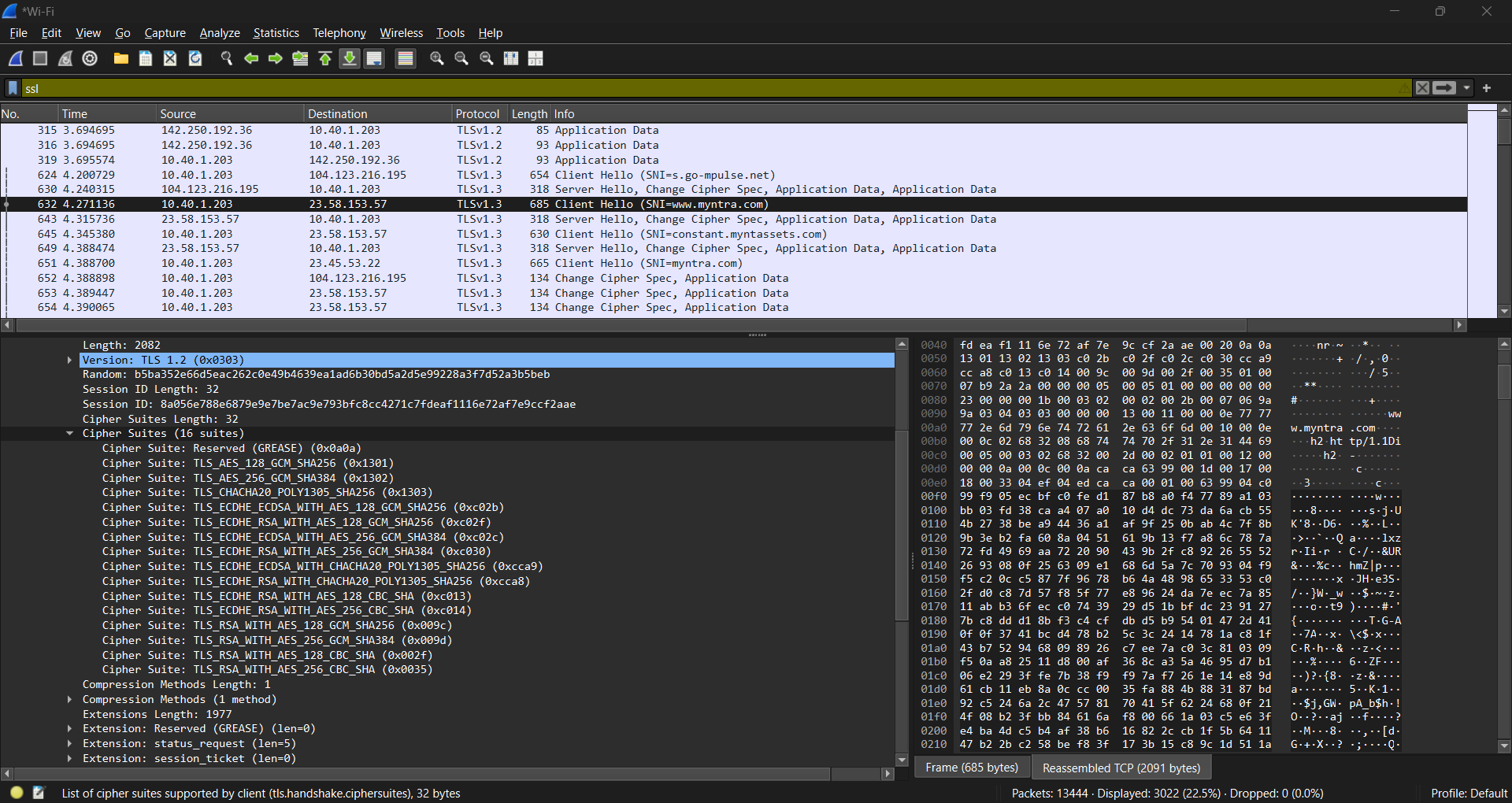
Q2.What version constant is used in the trace, and which version of TLS does it represent?

=>

Version constant: (0x0301)

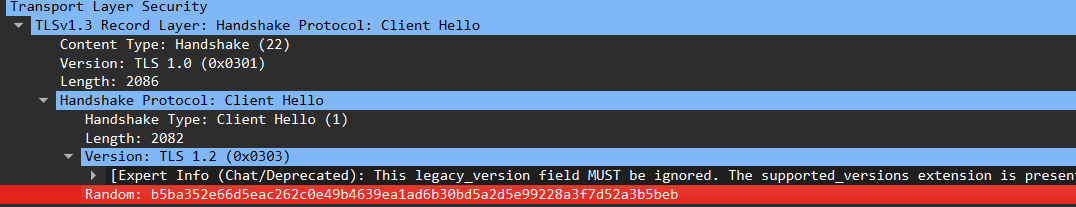
Version used: TLS 1.0

Q3.Which encryption algorithms are supported by client?



Q4. How long in bytes is the random data in the Hellos? Both the Client and Server include this random data(a nonce) to allow the establishment of session keys.

=>



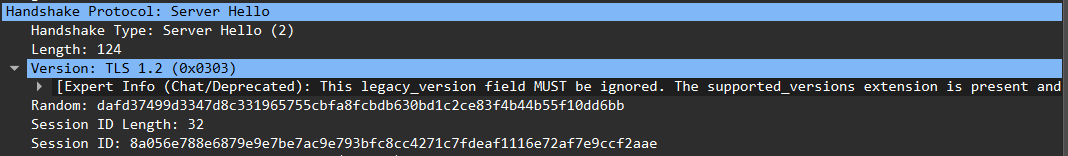
The Client Hello and Server Hello messages of the TLS handshake, a random nonce is used to help establish the session keys. This random data is always 32 bytes long.

Q5. How long in bytes is the session identifier sent by the server? This identifier allows later

resumption of the session with an abbreviated handshake when both the client and server

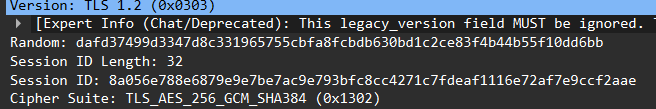
indicate the same value.

=>

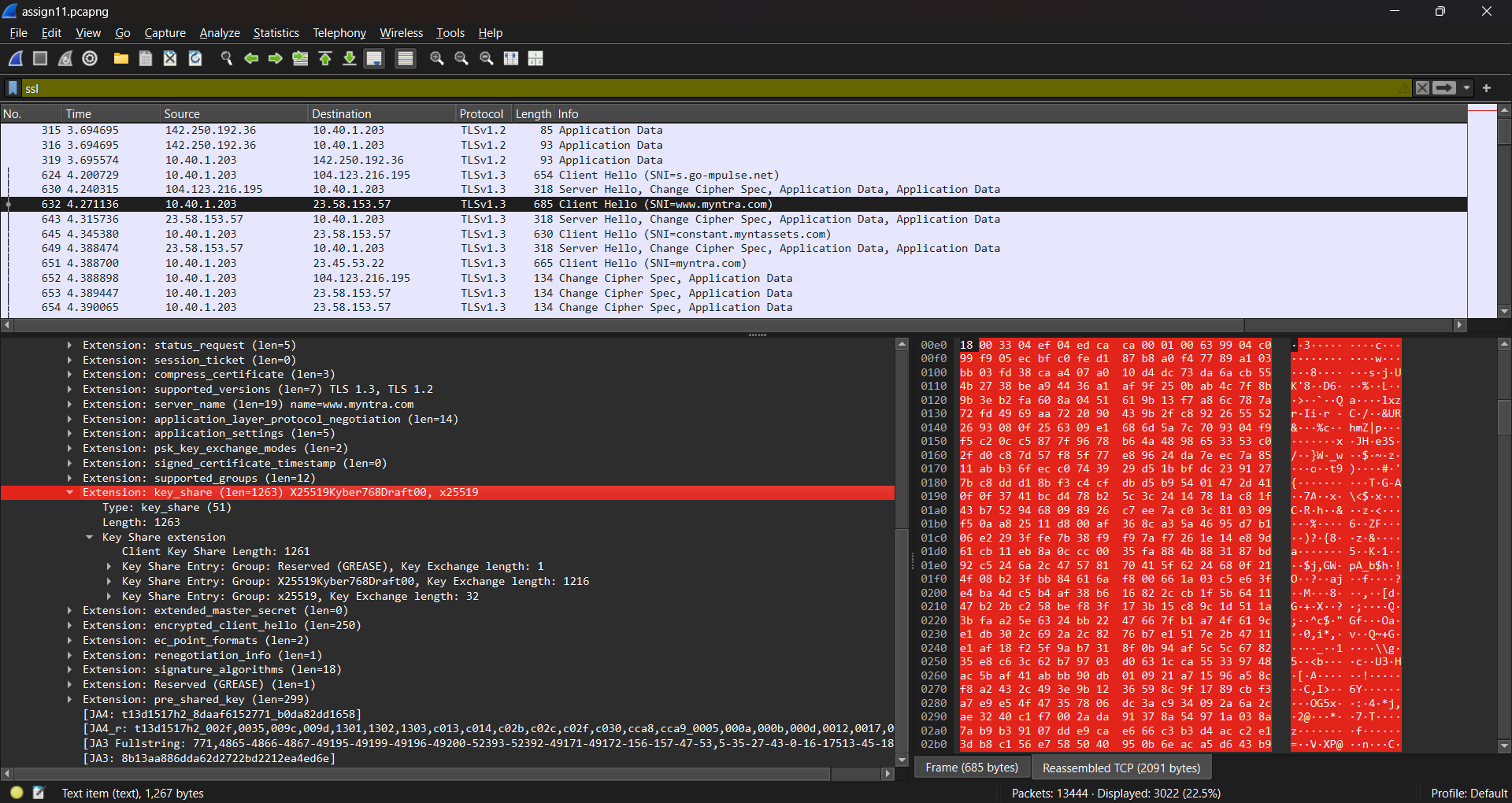


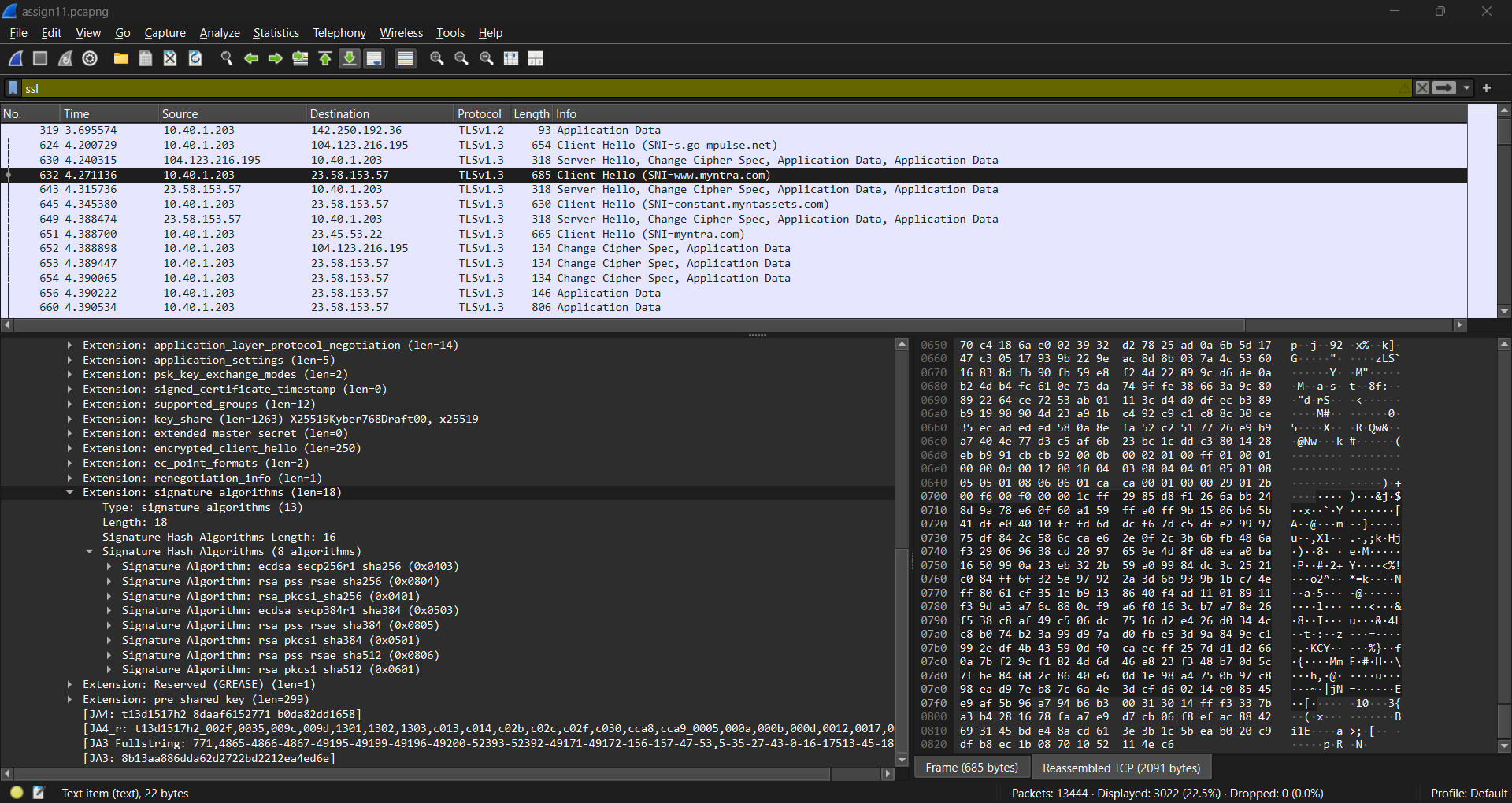
* Session ID Length: The Session ID Length is 32 bytes.
* Session ID:The Session ID is displayed in hexadecimal format. This is the actual identifier used for session resumption.

Q6. What Cipher method is chosen by the Server? The Client will list the different cipher methods it supports, and the Server will pick one of these methods to use.

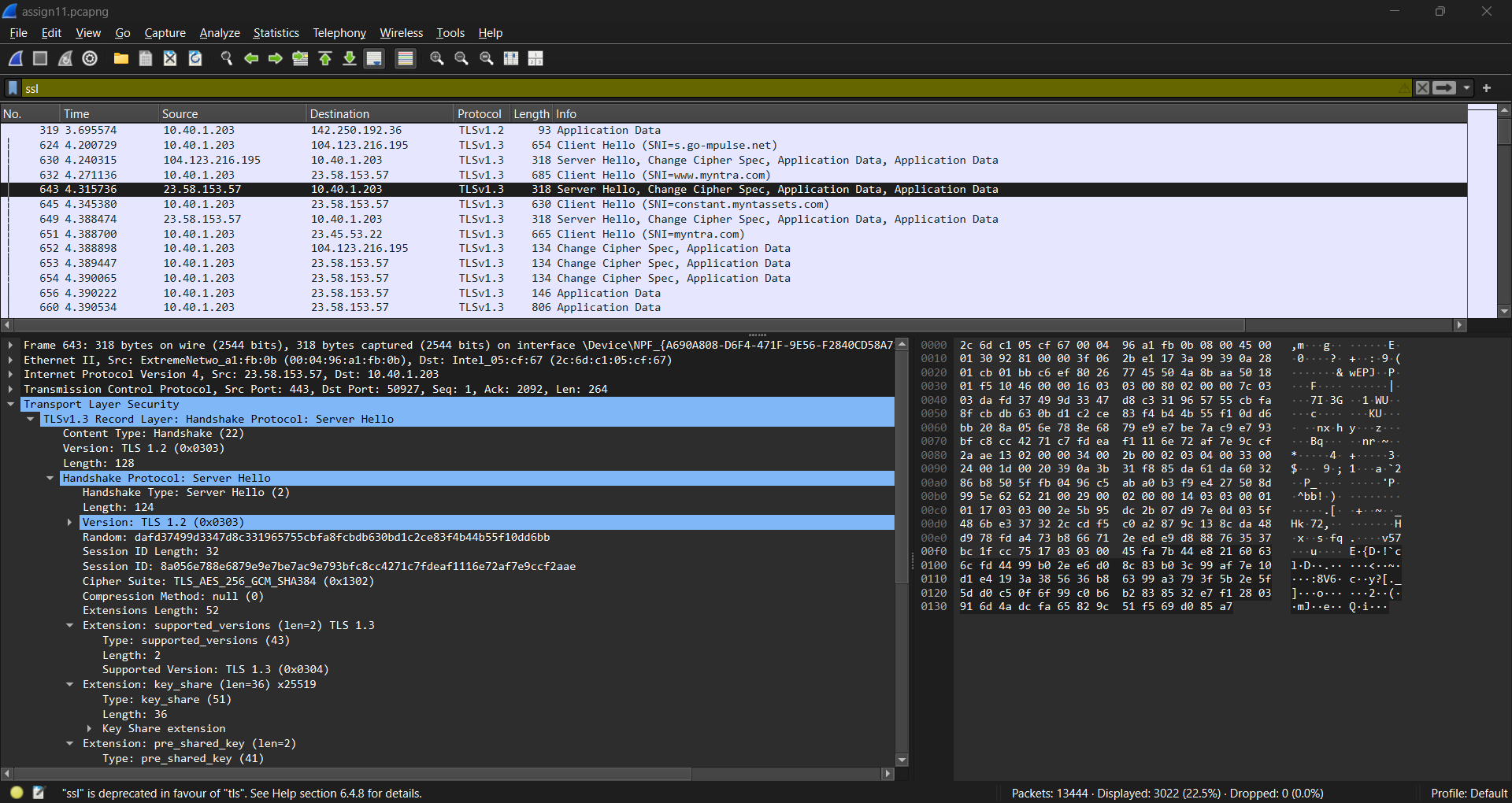
=> 

**Key exchange :-**



****

**Server hello:-**

****

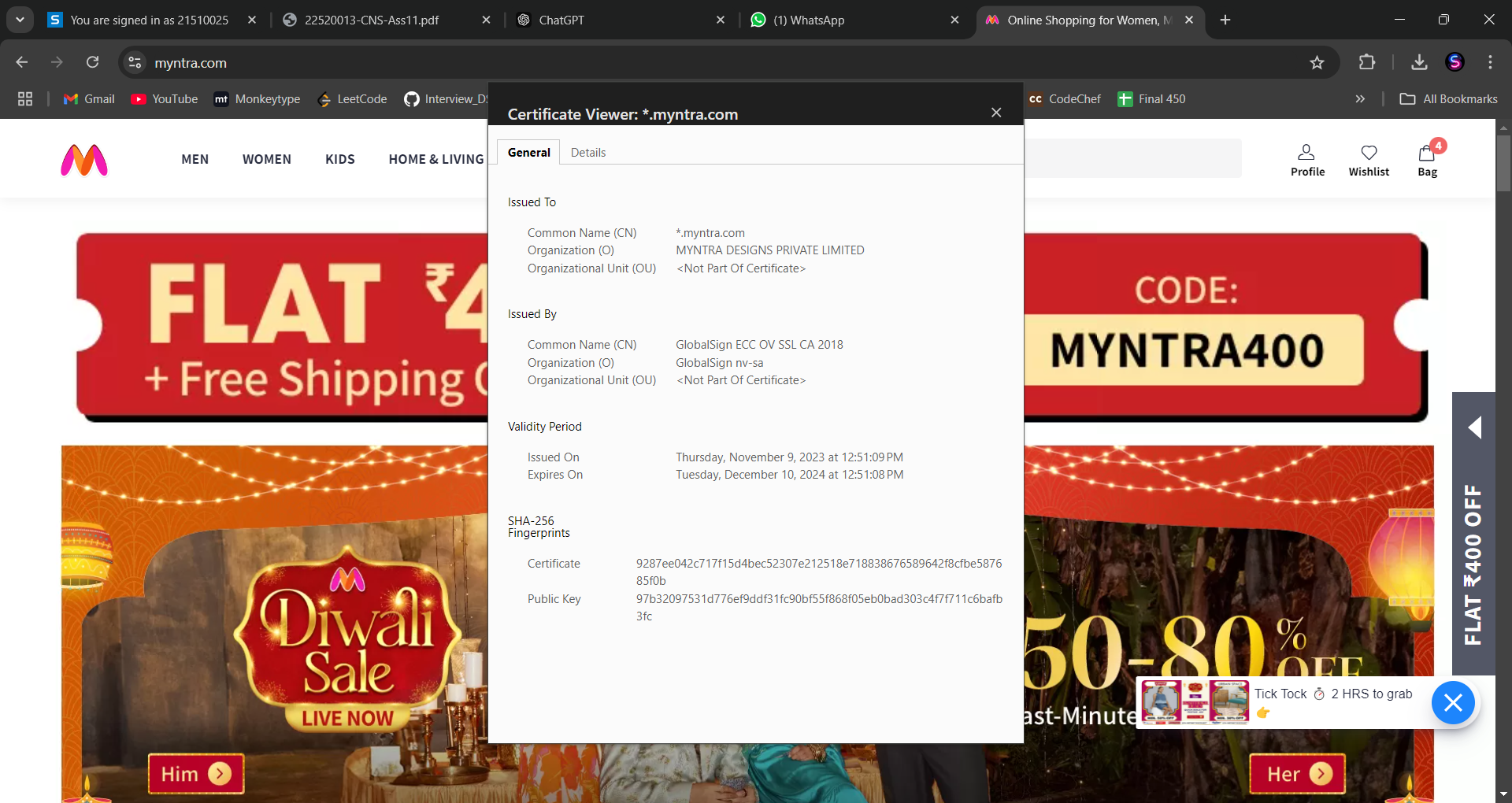
Q7. 7. Who sends the Certificate, the client, the server, or both? A certificate is sent by one party to let the other party authenticate that it is who it claims to be.

Client → Server: Client Hello

Server → Client: Server Hello, Server Certificate (and potentially a Server Key Exchange message)

Client → Server: (optional) Client Certificate (if mutual authentication is required)

Primarily, the server sends the certificate to the client.



Who sends the Certificate, the client, the server, or both? A certificate is sent by one party to let the other party authenticate that it is who it claims to be.

Client → Server: Client Hello

Server → Client: Server Hello, Server Certificate (and potentially a Server Key Exchange message)

Client → Server: (optional) Client Certificate (if mutual authentication is required)

Primarily, the **server** sends the certificate to the **client**.