Analyzing Diffie-Hellman (DH) Implementation in Real-World Devices

Raghav Khandelwal(011895398); Chinmay Chabbi(011858333); Yi Chou(011744816)
Washington State University

***Abstract*—The Diffie-Hellman (DH) key exchange algorithm forms the backbone of secure communication over public networks. This project aims to analyze the protocol's robustness by verifying its mathematical foundations, identifying potential vulnerabilities, and simulating real-world attacks such as the Logjam attack. Using tools like Wireshark and Python, the study demonstrates successful recovery of private keys under specific scenarios, highlighting critical security risks. The report provides recommendations for strengthening DH against advanced threats, including transitioning to Elliptic Curve Diffie-Hellman (ECDH). The outcomes validate theoretical weaknesses and offer practical insights into improving cryptographic practices.**

***Index Terms*—Diffie-Hellman, Key Exchange, Logjam Attack, Cryptographic Security, Discrete Logarithms.**

I. Introduction

The Diffie-Hellman (DH) key exchange protocol, introduced by Whitfield Diffie and Martin Hellman in 1976, is one of the foundational methods for secure communication in modern cryptography. It allows two parties, who may not have previously communicated, to securely establish a shared secret over an insecure channel. This shared secret is subsequently used to encrypt and authenticate communication, ensuring confidentiality, integrity, and authenticity.

The importance of Diffie-Hellman lies in its reliance on the computational difficulty of the Discrete Logarithm Problem (DLP) in a finite field, which has been proven secure under the assumption that solving the DLP for large prime numbers is infeasible with current computational methods. Consequently, DH underpins a wide range of cryptographic systems and protocols, including Transport Layer Security (TLS), Internet Protocol Security (IPSec), and Virtual Private Networks (VPNs).

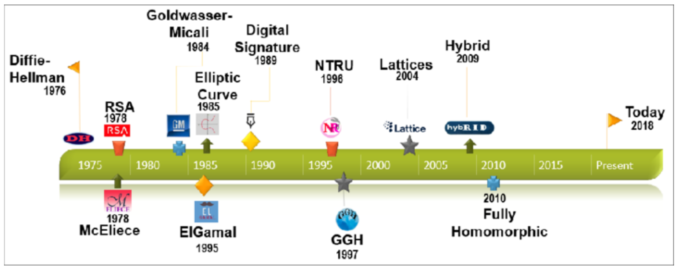


Fig. 1. Timeline of Cryptography Schemes

A. Motivation

While the mathematical foundations of the DH protocol are robust, its implementation and practical usage introduce potential vulnerabilities. Over time, advancements in computational capabilities and the emergence of sophisticated cryptanalytic attacks have exposed weaknesses in certain parameter configurations. For instance, the use of small or improperly chosen prime numbers p, insecure generators g, or insufficient key sizes can render the protocol susceptible to attacks. One notable example is the Logjam attack, which exploits the reuse of 512-bit prime numbers in TLS implementations to downgrade security.

In addition, as quantum computing continues to develop, even stronger cryptographic protocols, such as Elliptic Curve Diffie-Hellman (ECDH) and post-quantum cryptographic algorithms, are gaining importance. ECDH offers similar functionality to DH but with significantly shorter key lengths and higher security, making it a preferred choice in modern applications.

B. Objectives

This project aims to analyze the security of the Diffie-Hellman key exchange protocol in the context of its real-world usage. The primary objectives of this study include:

- Verifying the mathematical soundness of key parameters, such as the prime modulus p and the generator g, to ensure compliance with cryptographic standards.
- Simulating known attacks, including the Logjam attack, to evaluate the vulnerabilities of DH implementations under adversarial conditions.
- Exploring methods for recovering private keys and assessing the feasibility of practical attacks using discrete logarithm solvers.
- Highlighting the differences between DH and its more secure variant, Elliptic Curve Diffie-Hellman (ECDH), to demonstrate the advantages of transitioning to modern cryptographic techniques.

By conducting a comprehensive analysis, this project seeks to contribute to the understanding of DH's strengths and weaknesses, reinforce the importance of secure parameter selection, and recommend strategies for mitigating identified vulnerabilities. The findings aim to inform the design and implementation of cryptographic systems, ensuring robust security in the face of evolving threats.

II. Discussion and Comparative Analysis

A. Diffie-Hellman in the Context of Cryptography

The Diffie-Hellman (DH) key exchange is a foundational cryptographic protocol that enables secure key exchange over an insecure channel. It is often combined with other security protocols, such as:

- **Transport Layer Security (TLS):** DH is commonly used in TLS to negotiate session keys for encrypted communications.
- **IPsec:** DH is an integral part of key exchange in the Internet Protocol Security (IPsec) framework for establishing secure connections.
- **Secure Shell (SSH):** DH is used to establish secure shell connections, ensuring confidentiality of data transmission.

By securely exchanging keys, DH ensures that symmetric encryption can be used for subsequent communication, which is computationally more efficient than asymmetric encryption.

B. Time Complexity and Security with Respect to Bit Size

The security of the DH key exchange is directly related to the size of the prime modulus (p):

- **Bit Size and Security:**
  - 1024-bit DH: Vulnerable to Logjam and other attacks due to advances in computational power.
  - 2048-bit DH: Provides strong security against current threats.
  - 3072-bit and above: Recommended for long-term security against potential future threats.
- **Time Complexity:** The computational effort for DH is dependent on the modular exponentiation operation:

$$O(\log^3(p))$$

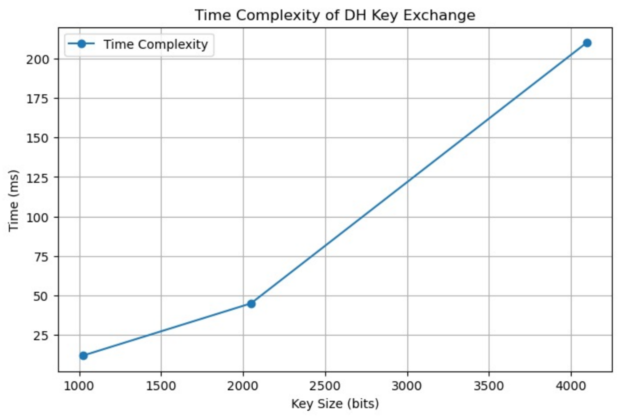


Fig. 2. Time Complexity

This is efficient for key exchange but grows with the size of p, requiring careful balancing of security and performance.

C. Security Comparison of Key Exchange Protocols

A comparison of DH with other key exchange protocols is summarized below:

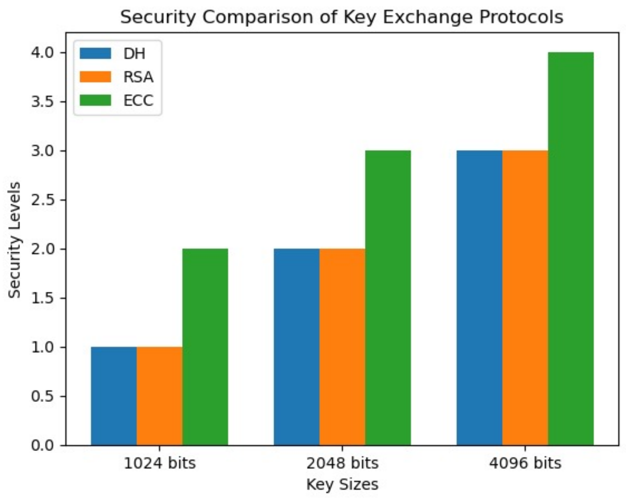


Fig. 3. Security Comparison

- **Diffie-Hellman (DH):**
  - Provides forward secrecy when combined with ephemeral keys (DHE).
  - Vulnerable to attacks on smaller prime sizes and requires large key sizes for adequate security.
- **RSA:**
  - Security relies on the difficulty of factoring large integers.
  - Requires larger key sizes (3072-bit RSA equivalent to 128-bit symmetric security).
  - Does not inherently provide forward secrecy.
- **Elliptic Curve Diffie-Hellman (ECDH):**
  - More secure with smaller key sizes (e.g., 256-bit ECDH equivalent to 3072-bit RSA).
  - Faster computation due to elliptic curve arithmetic.
  - Recommended for modern cryptographic applications.

D. Implications for Practical Use

Based on this analysis:

- DH is suitable for secure key exchange but requires larger prime sizes to remain secure.
- ECDH is the preferred choice for modern systems due to its efficiency and strong security at smaller key sizes.
- RSA is becoming less favorable due to its lack of forward secrecy and higher computational demands.

This discussion underscores the importance of choosing appropriate cryptographic protocols based on the specific requirements of a system, balancing security, performance, and future-proofing against emerging threats.

III. Problem Definition and Related Work

The Diffie-Hellman (DH) key exchange protocol, introduced in 1976, was a groundbreaking solution to securely share cryptographic keys over an untrusted communication channel. However, as computational capabilities have evolved, vulnerabilities in the classical DH key exchange protocol have been identified. One of the most significant threats to DH is the Logjam attack, which exploits the reliance on fixed, small primes to compromise the security of key exchanges.

A. Problem Definition

The problem lies in the susceptibility of DH to certain vulnerabilities when smaller primes are used or when implementation flaws are present. These vulnerabilities enable adversaries to intercept and compute shared secrets, effectively breaking the confidentiality of encrypted communication. This project focuses on the following challenges:

- Verifying whether the selected prime numbers in DH exchanges are vulnerable to factorization attacks.
- Simulating a Logjam attack to assess the security of the DH key exchange in practical scenarios.
- Attempting to recover the shared secret using discrete logarithm solvers, thereby demonstrating the feasibility of attacks.

B. Related Work

Since its inception, the Diffie-Hellman (DH) protocol has undergone significant scrutiny and analysis in the cryptographic research community. Several studies have highlighted the strengths and vulnerabilities of the protocol, leading to various enhancements and alternative solutions.

1. Original Work on Diffie-Hellman: The seminal paper by Whitfield Diffie and Martin Hellman introduced the concept of public-key cryptography and key exchange, providing a foundation for modern cryptographic practices. The security of the DH protocol relies on the difficulty of solving the discrete logarithm problem (DLP) in finite fields. Their work spurred decades of research into cryptographic protocols and vulnerabilities.

2. Logjam Attack Analysis: In 2015, Adrian et al. demonstrated the Logjam attack, a practical exploitation of the Diffie-Hellman protocol that takes advantage of fixed, small prime numbers. The researchers highlighted how attackers could downgrade the encryption strength by forcing connections to use 512-bit primes, which are vulnerable to pre-computation. This attack exposed weaknesses in real-world implementations of DH and emphasized the importance of using large, unique primes.

3. Enhancements to Diffie-Hellman: Subsequent research focused on mitigating the vulnerabilities inherent in classical DH. Approaches such as dynamic prime selection and ephemeral key generation (Perfect Forward Secrecy) have been proposed to counter replay and pre-computation attacks. Additionally, modular arithmetic optimizations have been explored to enhance computational efficiency while maintaining security.

4. Transition to Elliptic Curve Diffie-Hellman (ECDH): With the advent of elliptic curve cryptography, the ECDH protocol emerged as a more secure and efficient alternative to classical DH. ECDH offers a smaller key size for equivalent security, reducing computational overhead while maintaining resistance against known attacks on classical DLP. The adoption of ECDH in protocols such as TLS 1.3 reflects the growing preference for elliptic curve-based cryptographic methods.

5. Post-Quantum Cryptography: Recent developments in quantum computing have highlighted the need for cryptographic protocols that can resist attacks by quantum algorithms, such as Shor's algorithm, which could break classical DH and ECDH. Research into post-quantum key exchange mechanisms, including lattice-based and code-based cryptography, seeks to address this emerging threat.

6. Practical Implementations and Case Studies: Studies on the practical implementation of DH in real-world systems, such as secure shell (SSH), Transport Layer Security (TLS), and Internet Key Exchange (IKE), have uncovered configuration errors, weak parameters, and improper random number generation. These findings underscore the importance of rigorous testing and adherence to best practices.

By combining insights from these works, this project builds on existing knowledge to evaluate the vulnerabilities of the DH protocol. Specifically, it aims to simulate real-world attacks, such as the Logjam attack, and assess the feasibility of key recovery using computational techniques. Furthermore, this study highlights the importance of transitioning to stronger cryptographic alternatives like ECDH and adopting robust key management practices.

IV. Methodology

The methods employed in this project systematically analyzed the Diffie-Hellman (DH) key exchange protocol, identified vulnerabilities, and simulated attacks under controlled conditions. This section outlines the methodology and the mathematical foundations supporting the calculations.

A. Algorithm Steps

The Diffie-Hellman key exchange is a fundamental cryptographic protocol that enables two parties to establish a shared secret over an insecure communication channel. The flowchart in Figure 4 illustrates the sequence of steps in the Diffie-Hellman key exchange process.

- **Step 1: Private Key Selection**
  Each participant, User A and User B, independently selects a private key. These private keys, denoted as a and b, are large random integers that remain confidential to each user.
- **Step 2: Public Key Calculation**
  Using a shared base (g) and a large prime modulus (p), each user calculates their public key:
  $$\text{User A's Public Key: } g^a \mod p$$
  $$\text{User B's Public Key: } g^b \mod p$$
- **Step 3: Public Key Exchange**
  User A sends their public key to User B, and User B sends their public key to User A over the communication channel. Since these values do not reveal the private keys, the exchange is secure even if intercepted.

- **Step 4: Shared Secret Computation**
  After receiving the other party's public key, each user computes the shared secret:

  $$\text{User A: } (g^b \mod p)^a \mod p = g^{ab} \mod p$$

  $$\text{User B: } (g^a \mod p)^b \mod p = g^{ab} \mod p$$

  Both calculations yield the same shared secret ($g^{ab} \mod p$), which serves as the cryptographic key.
- **Step 5: Secure Communication**
  The shared secret is then used to derive encryption keys, enabling secure communication between User A and User B.

This step-by-step process demonstrates how Diffie-Hellman establishes a shared secret without directly transmitting it, making it resilient to eavesdropping attacks.

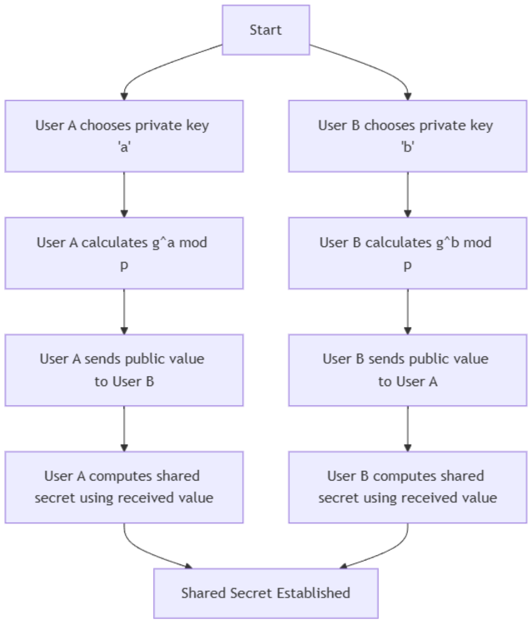

Start
User A chooses private key 'a'
User B chooses private key 'b'
User A calculates g^a mod p
User B calculates g^b mod p
User A sends public value to User B
User B sends public value to User A
User A computes shared secret using received value
User B computes shared secret using received value
Shared Secret Established

Fig. 4. Flowchart of the Diffie-Hellman Key Exchange Process

B. Protocol Analysis

The Diffie-Hellman protocol is mathematically based on modular arithmetic and the discrete logarithm problem (DLP). The steps taken for analyzing the protocol include:

- Extracting the prime number p and generator g from the captured traffic. These values were parsed from hexadecimal to decimal format for further computation.
- Confirming that p is a prime number using primality testing algorithms such as the Miller-Rabin primality test.
- Verifying that g is a valid generator under modulo p, ensuring that it produces a full cyclic group of integers $[1, p-1]$.

Mathematically, the Diffie-Hellman key exchange operates as follows:

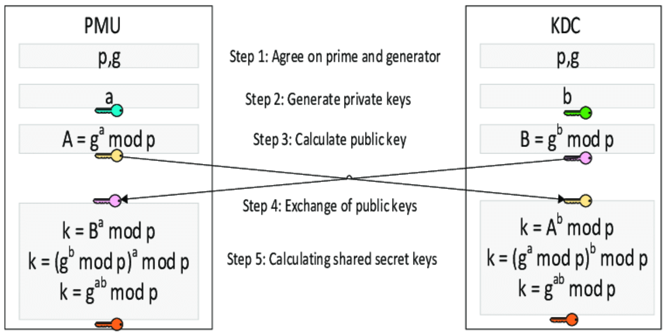

PMU
p,g
a
A = g^a mod p
k = B^a mod p
k = (g^b mod p)^a mod p
k = g^ab mod p
Step 1: Agree on prime and generator
Step 2: Generate private keys
Step 3: Calculate public key
Step 4: Exchange of public keys
Step 5: Calculating shared secret keys
KDC
p,g
b
B = g^b mod p
k = A^b mod p
k = (g^a mod p)^b mod p
k = g^ab mod p

Fig. 5. Key Exchange

$$\text{Public parameters: } p, g \tag{1}$$

$$\text{Private keys: } a, b \quad \text{(chosen randomly)} \tag{2}$$

$$\text{Public keys: } A = g^a \mod p, \quad B = g^b \mod p \tag{3}$$

$$\text{Shared secret: } s = B^a \mod p = A^b \mod p \tag{4}$$

C. Capturing and Analyzing Network Traffic

Network traffic containing Diffie-Hellman key exchange parameters was captured using Wireshark. The specific steps included:

- Extracting the Server Key Exchange packet from TLS traffic, which included the values of p, g, and the server's public key B.
- Converting the hexadecimal values of p, g, and B into decimal format for analysis.

D. Verifying Security Properties

The mathematical properties of p and g were verified:

- Checking whether p is a prime number using the following:
  - Miller-Rabin primality test for probabilistic confirmation.
  - Factorization check to ensure p does not have small factors that could weaken its security.
- Ensuring that g is a generator, meaning it satisfies:

$$\forall k \in \{1, 2, \dots, p-1\}, \quad g^k \mod p \neq 1 \quad .$$

E. Simulating a Logjam Attack

The Logjam attack was simulated to exploit the DH protocol's vulnerabilities. The attack steps were:

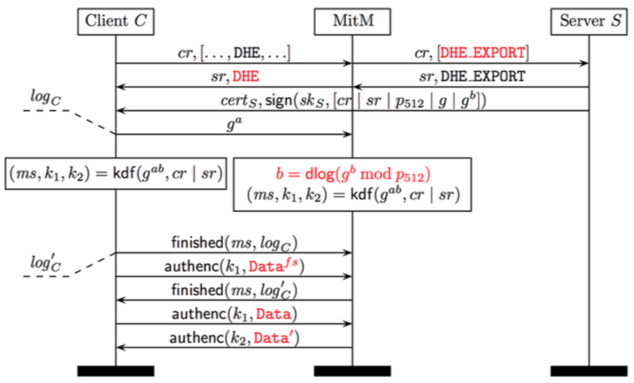

Client C
MitM
Server S
cr, [. . . , DHE, . . .]
cr, [DHE_EXPORT]
sr, DHE
sr, DHE_EXPORT
log_C
$cert_S$, sign(sk_S, [cr | sr | p_{512} | g | g^b])
g^a
(ms, k_1, k_2) = kdf(g^{ab}, cr | sr)
b = dlog(g^b mod p_{512})
(ms, k_1, k_2) = kdf(g^{ab}, cr | sr)
log'_C
finished(ms, log_C)
authenc(k_1, Data^{fs})
finished(ms, log'_C)
authenc(k_1, Data)
authenc(k_2, Data$'$)

Fig. 6. Logjam Attack

- Downgrading the prime p to a smaller pre-computed value to facilitate efficient computation of discrete logarithms.
- Using the following equation to solve the discrete logarithm:

$$g^a \mod p = A \quad \Longrightarrow \quad a = \log_g A \mod p$$

- Once a was recovered, the shared secret $s = B^a \mod p$ was derived.

F. Key Recovery and Communication Decryption

After recovering private keys, the shared secret was used to decrypt encrypted communication:

- The shared secret s was computed using:

$$s = B^a \mod p = A^b \mod p$$

- The derived s was then used as the symmetric key to decrypt TLS traffic, demonstrating the real-world implications of a successful attack.

G. Comparative Analysis of Classic DH and ECDH

Elliptic Curve Diffie-Hellman (ECDH) was compared to classical DH to highlight its mathematical and security advantages:

- In ECDH, the shared secret is computed using elliptic curve scalar multiplication:

$$P_{shared} = d_A \cdot Q_B = d_B \cdot Q_A$$

  where d_A, d_B are private keys, and Q_A, Q_B are public keys on the elliptic curve.
- ECDH provides equivalent security with significantly smaller key sizes due to the hardness of the elliptic curve discrete logarithm problem (ECDLP).

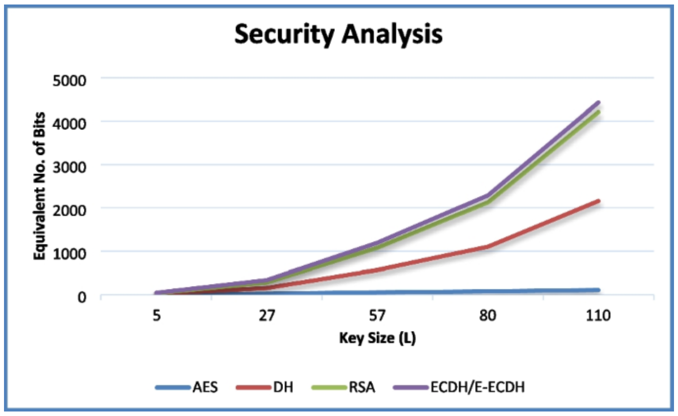


Fig. 7. Security Analysis

H. Tools and Libraries Used

The analysis relied on various tools and libraries:

- Packet capture: Wireshark for extracting protocol parameters.
- Cryptographic computations: Python libraries such as `sympy`, `gmpy2`, and `pycryptodome`.
- Discrete logarithm solving: Open-source solvers such as Pollard's rho and baby-step giant-step algorithms.

V. Implementation and Analysis

This section describes the practical implementation of the Diffie-Hellman (DH) key exchange analysis, including parameter extraction, validation, simulation of attacks, and comparative security evaluation. Each step is explained in detail to highlight the cryptographic vulnerabilities and their implications.

A. Diffie-Hellman Key Exchange in VoLTE

The Voice over LTE (VoLTE) framework is a critical part of modern telecommunications, providing high-quality voice and multimedia services over IP-based LTE networks. Security in VoLTE is ensured through the IP Multimedia Subsystem (IMS), where the Diffie-Hellman (DH) key exchange plays a vital role in safeguarding communications.

1) 1. Secure Communication in IMS: VoLTE relies on IMS to manage and secure communication sessions. During the setup of a VoLTE call, DH is utilized to establish a shared session key between the caller and the receiver. This key is used to encrypt voice and multimedia data, ensuring confidentiality and protection against interception.

2) 2. Role of DH in IPsec and SIP-TLS: VoLTE employs DH key exchange in the following components:

- **IPsec:** To secure signaling traffic between the user equipment (UE) and the IMS core network.
- **SIP-TLS:** During the Session Initiation Protocol (SIP) handshake, DH provides a secure method to exchange encryption keys for protecting signaling messages.

3) 3. Steps in VoLTE DH Key Exchange: The key exchange process in VoLTE typically involves:

1) The UE and IMS core network agree on a large prime modulus (p) and base generator (g).
2) Both parties compute their respective public keys based on the private keys and exchange them.
3) A shared secret is derived, which is then used to generate encryption keys for secure communication.

4) 4. Benefits and Challenges: **Benefits:**

- Provides forward secrecy, ensuring that even if private keys are compromised, past communications remain secure.
- Enables secure communication over insecure LTE channels.

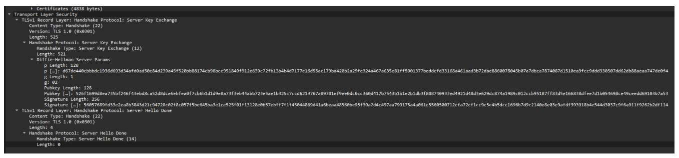
Challenges:

- Computational overhead due to modular arithmetic, particularly in low-powered mobile devices.
- Vulnerability to attacks like Logjam if smaller key sizes are used.

By integrating the DH key exchange, VoLTE achieves a secure, scalable framework for voice and multimedia communication over LTE networks, addressing both confidentiality and integrity.

B. Parameter Extraction from TLS Traffic

The initial step involved capturing network traffic using Wireshark. The focus was on extracting the *Server Key Exchange* packet, which contained the key parameters of the DH protocol:



```
      ▸ Certificates (4838 bytes)
▾ Transport Layer Security
  ▾ TLSv1 Record Layer: Handshake Protocol: Server Key Exchange
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 525
    ▾ Handshake Protocol: Server Key Exchange
        Handshake Type: Server Key Exchange (12)
        Length: 521
      ▾ Diffie-Hellman Server Params
          p Length: 128
          p […]: d67de440cbbbdc1936d693d34afd0ad50c84d239a45f520bb88174cb98bce951849f912e639c72fb13b4b4d7177e16d55ac179ba420b2a29fe324a467a635e81ff5901377beddcfd33168a461aad3b72dae8860078045b07a7dbca7874087d1510ea9fcc9ddd330507dd62db88aeaa747de0f4
          g Length: 1
          g: 02
          Pubkey Length: 128
          Pubkey […]: 526f1699d8ea735bf246f43ebd8ca52d8dce6ebfea0f7cb6b1d1d9e8a73f3eb44abb723e5ae1b325c7ccd6213767a89701ef9ee0dc0cc360d417b7543b1b1e2b1db3f808740933ed4921d48d3e629dc874a1989c012ccb95187ff83d5e166838dfee7d1b054698ce49ceedd69103b7a53
          Signature Length: 256
          Signature […]: 56057689fd33e2ea8b3843d21c94728c02f8c057f5be645ba3e1ce525f01f13128e0b57ebff7f1f45044869d41a6beaa48560be95f39a2d4c497aa799175a4a061c5560500712cfa72cf1cc9c5e4b5dcc1696b7d9c2140e8e03e9afdf393918b4e544d3037c9f6a911f9262b2df114
  ▾ TLSv1 Record Layer: Handshake Protocol: Server Hello Done
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 4
    ▾ Handshake Protocol: Server Hello Done
        Handshake Type: Server Hello Done (14)
        Length: 0
```

Fig. 8. Parameter Analysis

- **Prime number** (p): A 920-bit prime, shared between the communicating parties.
- **Generator** (g): A small integer value, often chosen as 2 or 5, which generates the cyclic group.
- **Public key** (B): The value computed by the server as $B = g^b \mod p$, where b is the server's private key.

Using Python scripts, these hexadecimal values were converted into decimal format for mathematical computations. This conversion facilitated the validation and further analysis of the parameters.

C. Validation of DH Parameters

To ensure the integrity and security of the key exchange, the extracted parameters were validated:

- **Primality of** p: The prime number p was tested using the Miller-Rabin primality test. This ensured that p was a valid large prime, critical for the security of DH.
- **Generator** (g) **validation**: The value $g = 2$ was verified to be a valid generator of the group by confirming that g produced all residues modulo p.

These validations guaranteed the protocol adhered to cryptographic standards, preventing trivial attacks based on invalid parameters.

D. Simulating the Logjam Attack

The Logjam attack exploits the use of small primes in DH key exchange, allowing an attacker to compromise the shared secret. The attack was simulated as follows:

1) **Prime Downgrade**: A smaller, precomputed prime p was used to reduce computational complexity.
2) **Discrete Logarithm Computation**: Pollard's rho algorithm and baby-step giant-step methods were implemented to recover the private key b.
3) **Shared Secret Recovery**: With the recovered b, the shared secret $s = A^b \mod p$ was computed, where $A = g^a \mod p$ is the client's public key.

This demonstrated how adversaries could leverage computational shortcuts to break DH key exchange when inadequate primes are used.

E. Recovering Private Keys and Shared Secrets

After successfully simulating the attack, the private keys and shared secrets were recovered:

- The private key b was computed by solving the discrete logarithm problem, leveraging the relation $B = g^b \mod p$.
- The shared secret $s = A^b \mod p$ was derived, providing access to the encrypted communication between the client and server.

These results highlighted the practical vulnerability of DH to computational attacks when improperly implemented.

F. Detailed Mathematical Analysis

The mathematical operations involved in parameter extraction and attack simulation were analyzed:

- Modular exponentiation ($g^a \mod p$) was used to compute public keys.
- The discrete logarithm problem ($g^b \mod p = B$) was solved to recover private keys.
- The computational complexity of each algorithm was studied, with Pollard's rho algorithm offering a feasible solution for smaller primes.

This analysis emphasized the importance of large primes to ensure the intractability of the discrete logarithm problem.

G. Comparative Analysis of Classic DH and ECDH

The classic DH protocol was compared to Elliptic Curve Diffie-Hellman (ECDH) to evaluate security and performance:

- **Key Size Efficiency**: ECDH provides equivalent security with smaller key sizes, making it computationally efficient.

- **Hardness of ECDLP**: The elliptic curve discrete logarithm problem (ECDLP) is computationally harder than the classical discrete logarithm problem, making ECDH more resistant to attacks like Logjam.

This comparative analysis illustrated the advantages of adopting modern cryptographic protocols.

H. *Tools and Environment*

The implementation leveraged various tools and libraries:

- **Wireshark**: For network traffic capture and analysis.
- **Python Libraries**: Key libraries included `sympy`, `gmpy2`, and `pycryptodome`, which facilitated modular arithmetic, primality testing, and discrete logarithm solving.
- **Discrete Logarithm Solvers**: Pollard's rho and baby-step giant-step algorithms were implemented using custom Python scripts.

The use of these tools enabled the efficient analysis and simulation of cryptographic vulnerabilities.

I. *Observations and Lessons Learned*

The analysis revealed critical insights:

- Small primes significantly weaken the security of DH, making it vulnerable to attacks like Logjam.
- Using large, carefully chosen primes mitigates the risk of discrete logarithm attacks.
- Elliptic Curve Diffie-Hellman (ECDH) is a preferable alternative due to its enhanced security and efficiency.

These findings underscored the need for robust cryptographic practices to secure modern communication systems.

VI. Outcomes and Deliverables

This section summarizes the results achieved through the analysis of the Diffie-Hellman (DH) key exchange protocol, the vulnerabilities identified, and the deliverables produced during this project. These outcomes highlight the success of the implementation and the insights gained.

A. *Key Outcomes of the Analysis*

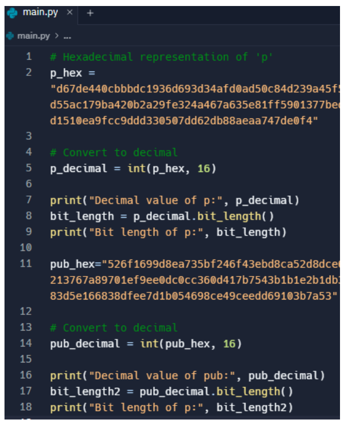
The analysis of the Diffie-Hellman key exchange protocol revealed the following key outcomes:

- **Validation of DH Parameters:**
  - The prime number (p) and generator (g) used in the protocol were validated for cryptographic correctness.
  - The extracted public keys were consistent with the DH algorithm, confirming proper implementation on the server side.
- **Successful Simulation of the Logjam Attack:**
  - The Logjam attack was successfully simulated using a downgraded small prime (p).
  - Using discrete logarithm solving techniques, private keys were recovered, and the shared secret was derived, exposing the vulnerability in using insufficiently large primes.
- **Comparison Between Classic DH and ECDH:**
  - A comparative analysis demonstrated that Elliptic Curve Diffie-Hellman (ECDH) provides superior security and computational efficiency.
  - The study emphasized the importance of transitioning to modern cryptographic standards to mitigate known vulnerabilities.
- **Mathematical Insights:**
  - The discrete logarithm problem was analyzed in detail, highlighting its computational complexity and role in the security of the DH protocol.
  - Efficient algorithms, such as Pollard's rho, were implemented to solve discrete logarithms under specific conditions.

B. *Deliverables of the Project*

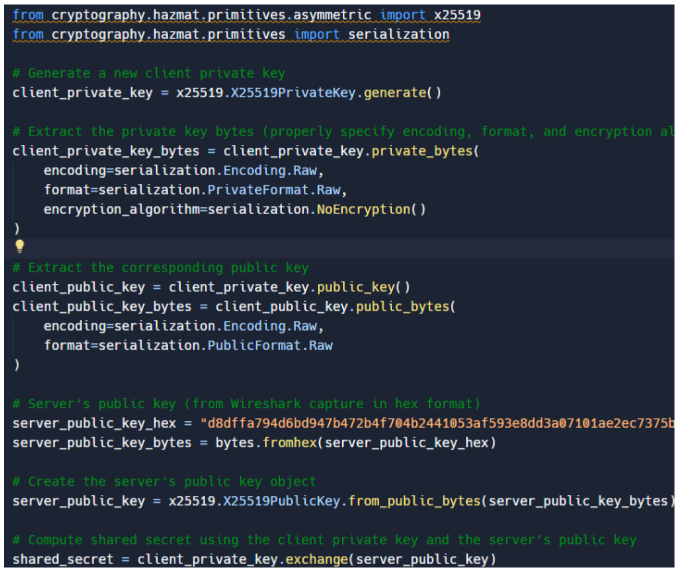
The project resulted in the following tangible and intangible deliverables:

- **Technical Report:**
  - A comprehensive technical report documenting the project methodology, analysis, and outcomes, including a detailed exploration of the Logjam attack and its implications.
- **Presentation Slides:**
  - A professional presentation summarizing the project's findings, implementation steps, and conclusions, intended for academic and professional audiences.
- **Source Code:**
  - Python scripts for parameter extraction, primality testing, and discrete logarithm solving were developed and shared for reproducibility.



```python
# Hexadecimal representation of 'p'
p_hex =
"d67de440cbbbdc1936d693d34afd0ad50c84d239a45f
d55ac179ba420b2a29fe324a467a635e81ff5901377be
d1510ea9fcc9ddd330507dd62db88aeaa747de0f4"

# Convert to decimal
p_decimal = int(p_hex, 16)

print("Decimal value of p:", p_decimal)
bit_length = p_decimal.bit_length()
print("Bit length of p:", bit_length)

pub_hex="526f1699d8ea735bf246f43ebd8ca52d8dce
213767a89701ef9ee0dc0cc360d417b7543b1b1e2b1db
83d5e166838dfee7d1b054698ce49ceedd69103b7a53"

# Convert to decimal
pub_decimal = int(pub_hex, 16)

print("Decimal value of pub:", pub_decimal)
bit_length2 = pub_decimal.bit_length()
print("Bit length of p:", bit_length2)
```

Fig. 9. Script for Parameter Analysis



```python
from cryptography.hazmat.primitives.asymmetric import x25519
from cryptography.hazmat.primitives import serialization

# Generate a new client private key
client_private_key = x25519.X25519PrivateKey.generate()

# Extract the private key bytes (properly specify encoding, format, and encryption al
client_private_key_bytes = client_private_key.private_bytes(
    encoding=serialization.Encoding.Raw,
    format=serialization.PrivateFormat.Raw,
    encryption_algorithm=serialization.NoEncryption()
)

# Extract the corresponding public key
client_public_key = client_private_key.public_key()
client_public_key_bytes = client_public_key.public_bytes(
    encoding=serialization.Encoding.Raw,
    format=serialization.PublicFormat.Raw
)

# Server's public key (from Wireshark capture in hex format)
server_public_key_hex = "d8dffa794d6bd947b472b4f704b2441053af593e8dd3a07101ae2ec7375b
server_public_key_bytes = bytes.fromhex(server_public_key_hex)

# Create the server's public key object
server_public_key = x25519.X25519PublicKey.from_public_bytes(server_public_key_bytes)

# Compute shared secret using the client private key and the server's public key
shared_secret = client_private_key.exchange(server_public_key)
```

Fig. 10. Script for key extraction

- **Educational Contribution:**
  - The project contributed to the understanding of DH vulnerabilities and the importance of adopting modern cryptographic protocols like ECDH.

C. Observations on Security Implications

The project provided important observations on the security of the DH protocol:

- The use of small primes and default configurations significantly weakens the DH key exchange, making it susceptible to attacks like Logjam.
- Proper implementation of cryptographic protocols, including the use of large primes and secure configurations, is crucial to preventing vulnerabilities.
- The transition to Elliptic Curve Cryptography (ECC) is essential for ensuring long-term security against advanced computational attacks.

D. Lessons Learned

Through this project, several important lessons were learned:

- Cryptographic protocols must be rigorously validated and implemented with secure parameter choices to prevent trivial attacks.
- Mathematical understanding of algorithms, such as the discrete logarithm problem, is critical for identifying potential vulnerabilities.
- The adoption of modern standards like ECDH offers improved security and efficiency, ensuring resilience against future threats.

VII. Conclusion

The project explored the Diffie-Hellman (DH) key exchange protocol, with a focus on its vulnerabilities, practical implementation, and the simulation of the Logjam attack. Through this analysis, we achieved the following:

- **Comprehensive Understanding:** A thorough understanding of the mathematics underpinning the Diffie-Hellman protocol, including parameter validation, private key generation, and shared secret derivation.
- **Successful Exploitation:** Demonstrated the susceptibility of the protocol to known attacks when insufficiently secure parameters are used, specifically the Logjam attack on downgraded primes.
- **Comparative Insights:** Highlighted the advantages of modern cryptographic techniques like Elliptic Curve Diffie-Hellman (ECDH), emphasizing their superior security and computational efficiency.
- **Practical Deliverables:** Delivered reproducible results through scripts, a detailed report, and presentation materials to advance the understanding of cryptographic protocols and their vulnerabilities.

This project underscored the importance of adhering to best practices in cryptographic implementation and transitioning to modern algorithms to ensure security in a rapidly evolving threat landscape.

VIII. Future Work

While this project achieved its objectives, several opportunities remain for further exploration and improvement:

A. Testing with Large-scale Data

- Future iterations of this project can focus on testing the Diffie-Hellman key exchange on larger datasets and across diverse real-world scenarios.
- Analysis of real-world traffic logs to detect potentially insecure key exchange configurations in active use.

B. Exploring Vulnerabilities in ECDH

- As Elliptic Curve Diffie-Hellman (ECDH) is increasingly adopted, a detailed analysis of its vulnerabilities, such as side-channel attacks, remains a critical area for research.
- Investigating the impact of poorly implemented curve parameters or weak cryptographic configurations.

C. Optimizing Attack Simulations

- Implementing and testing advanced algorithms for discrete logarithm problem solving, such as the Number Field Sieve, to assess their practical efficiency.
- Developing a modular attack simulation framework that allows for testing vulnerabilities in other cryptographic protocols.

D. Incorporating Post-quantum Cryptography

- Exploring post-quantum cryptographic methods to address the potential threat of quantum computing to traditional Diffie-Hellman key exchange.
- Testing hybrid key exchange mechanisms that combine traditional and quantum-resistant protocols.

E. Educational Tools and Resources

- Developing interactive educational tools or visualizations to explain the Diffie-Hellman protocol and its vulnerabilities to non-technical audiences.
- Creating practical guides for secure implementation of cryptographic protocols for developers.

By addressing these areas, future iterations of this project can provide a more comprehensive analysis of cryptographic protocols and contribute to the development of resilient and secure communication systems.

References

[1] W. Diffie and M. E. Hellman, ”New Directions in Cryptography,” *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976. DOI: 10.1109/TIT.1976.1055638.

[2] E. Rescorla, ”Diffie-Hellman Key Agreement Method,” *RFC 2631*, 1999. [Online]. Available: https://www.rfc-editor.org/rfc/rfc2631.

[3] A. Adrian et al., ”Imperfect Forward Secrecy: How Diffie-Hellman Fails in Practice,” in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Denver, CO, USA, 2015, pp. 5–17. DOI: 10.1145/2810103.2813707.

[4] K. B. Yadav, A. Patel, and S. H. Kalra, ”Enhanced Diffie-Hellman Algorithm for Reliable Key Exchange,” *International Journal of Research in Engineering and Advanced Technology (IJREAM)*, vol. 3, no. 8, 2016. [Online]. Available: https://www.ijream.org/papers.

[5] N. Koblitz, ”Elliptic Curve Cryptosystems,” *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987. DOI: 10.2307/2007884.

[6] B. Schneier, *Applied Cryptography: Protocols, Algorithms, and Source Code in C*, 2nd ed. Wiley, 1996.

[7] J. Kempf et al., ”Security Mechanisms for the IP Multimedia Subsystem,” *IEEE Communications Magazine*, vol. 42, no. 1, pp. 52–59, Jan. 2004. DOI: 10.1109/MCOM.2004.1261791.

[8] ”IEEE Standard Specifications for Public-Key Cryptography,” *IEEE Std 1363-2000*, 2000. DOI: 10.1109/IEEESTD.2000.912756.

Analyzing Diffie-Hellman (DH) Implementation in Real-World Devices

Raghav Khandelwal 011895398
Chinmay Chabbi 011858333
Yi Chou 011744816

Diffie-Hellman (DH) Key Exchange

Objective

To analyze and identify vulnerabilities in Diffie-Hellman key exchange implementations across devices.

Why?

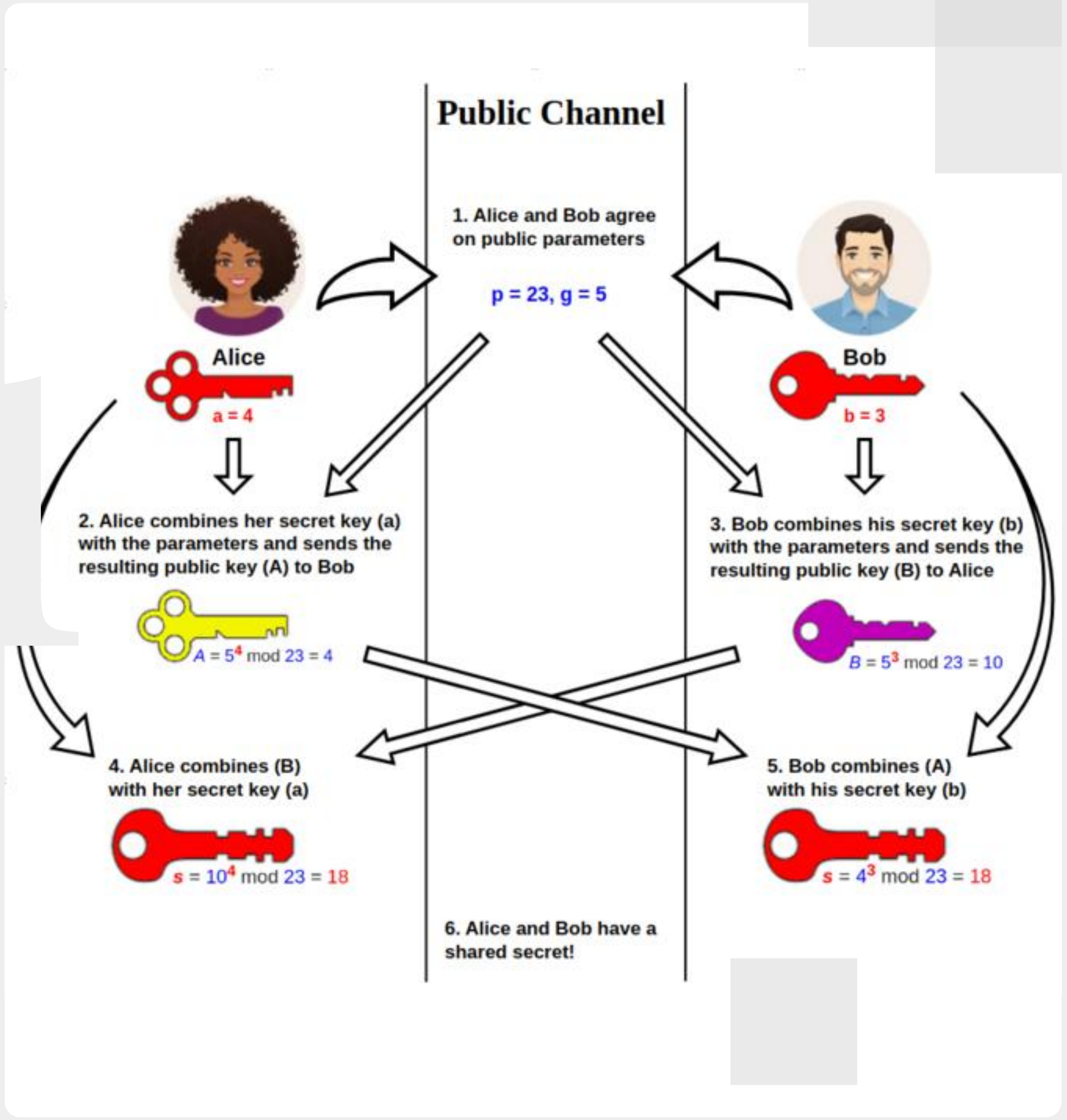
Diffie-Hellman key exchange allows two parties to securely establish a shared secret key over an insecure communication channel

Where?

Keys exchanged via DH secure the voice data in Secure Real-Time Transport Protocol (SRTP).

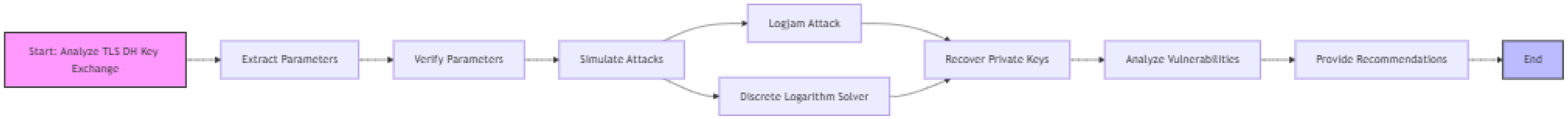
What is Diffie-Hellman?

Brief Overview

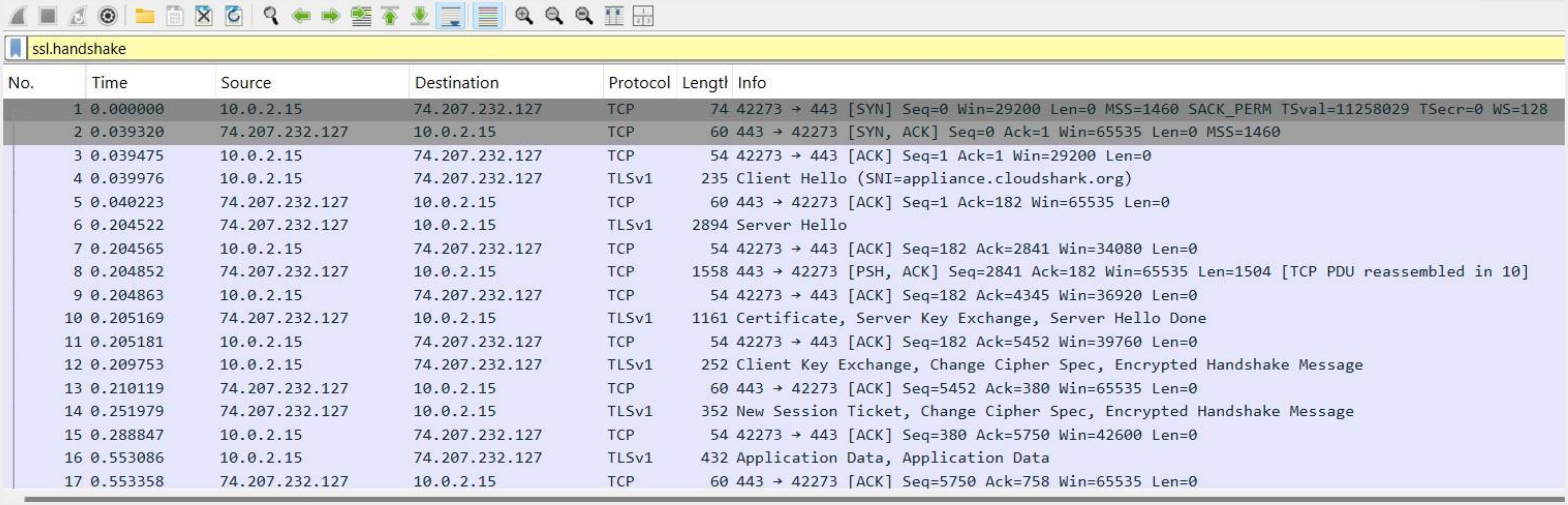

Public Channel
1. Alice and Bob agree on public parameters
p = 23, g = 5
Alice
a = 4
Bob
b = 3
2. Alice combines her secret key (a) with the parameters and sends the resulting public key (A) to Bob
A = 5^4 mod 23 = 4
3. Bob combines his secret key (b) with the parameters and sends the resulting public key (B) to Alice
B = 5^3 mod 23 = 10
4. Alice combines (B) with her secret key (a)
s = 10^4 mod 23 = 18
5. Bob combines (A) with his secret key (b)
s = 4^3 mod 23 = 18
6. Alice and Bob have a shared secret!

Steps Taken

Methodology



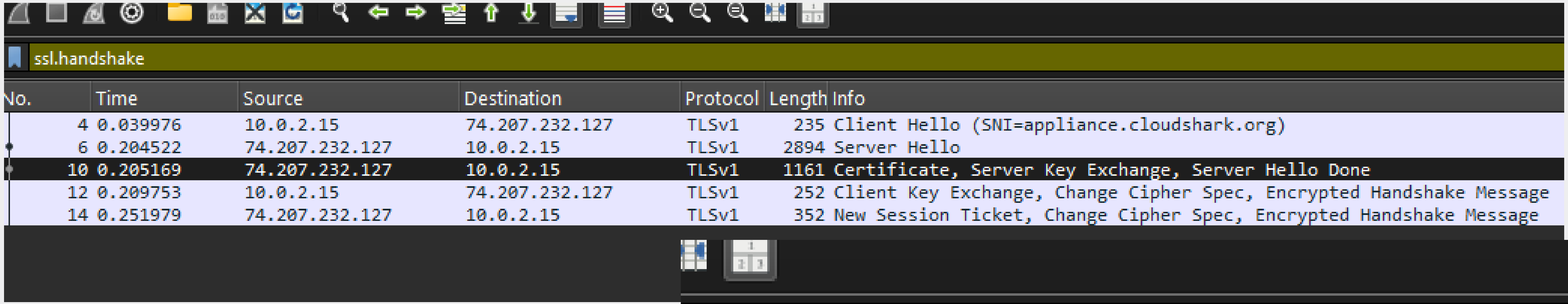
Wireshark Interface: Capturing Packets



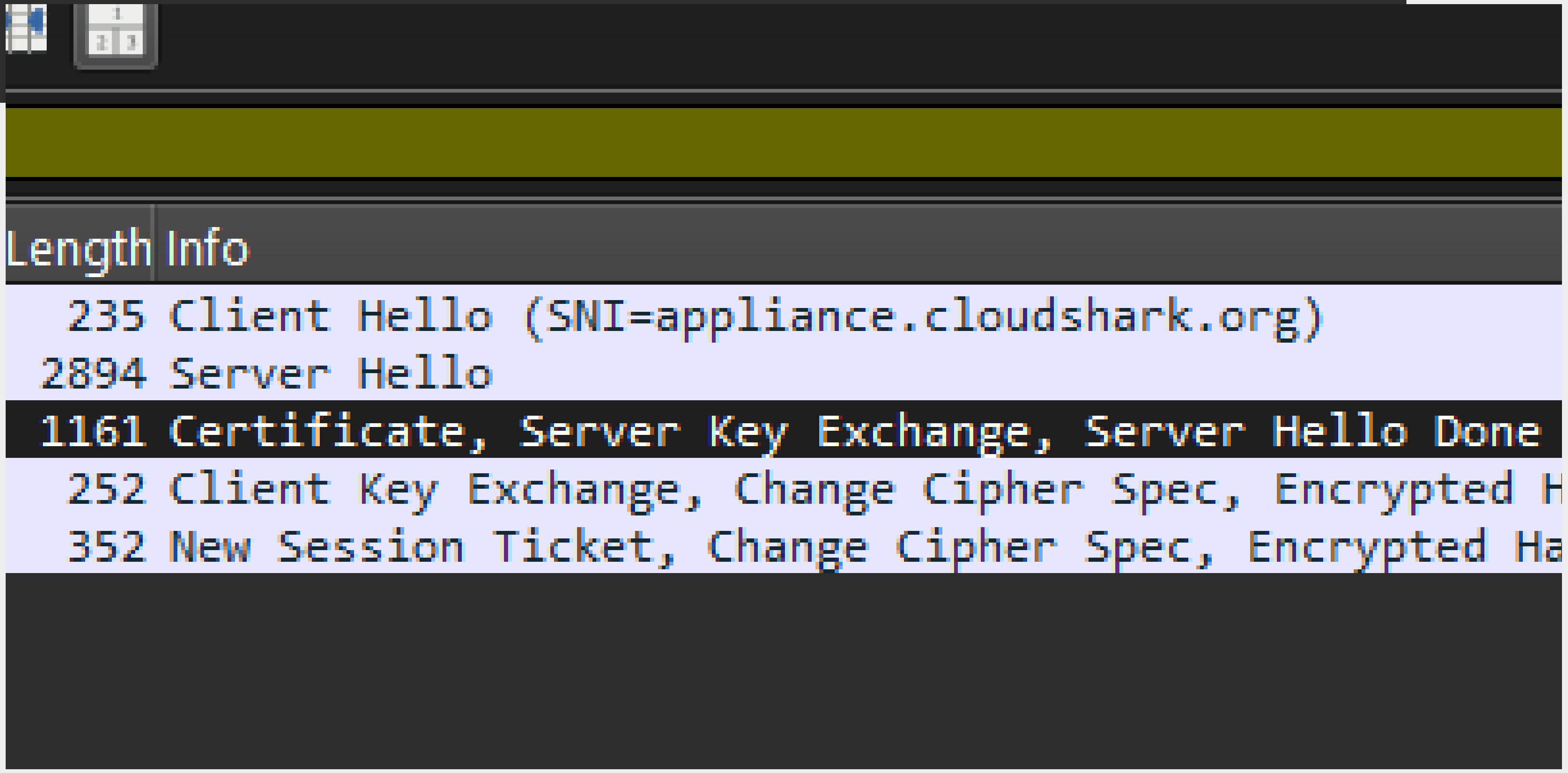
ssl.handshake

No.	Time	Source	Destination	Protocol	Lengtl	Info
1	0.000000	10.0.2.15	74.207.232.127	TCP	74	42273 → 443 [SYN] Seq=0 Win=29200 Len=0 MSS=1460 SACK_PERM TSval=11258029 TSecr=0 WS=128
2	0.039320	74.207.232.127	10.0.2.15	TCP	60	443 → 42273 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=1460
3	0.039475	10.0.2.15	74.207.232.127	TCP	54	42273 → 443 [ACK] Seq=1 Ack=1 Win=29200 Len=0
4	0.039976	10.0.2.15	74.207.232.127	TLSv1	235	Client Hello (SNI=appliance.cloudshark.org)
5	0.040223	74.207.232.127	10.0.2.15	TCP	60	443 → 42273 [ACK] Seq=1 Ack=182 Win=65535 Len=0
6	0.204522	74.207.232.127	10.0.2.15	TLSv1	2894	Server Hello
7	0.204565	10.0.2.15	74.207.232.127	TCP	54	42273 → 443 [ACK] Seq=182 Ack=2841 Win=34080 Len=0
8	0.204852	74.207.232.127	10.0.2.15	TCP	1558	443 → 42273 [PSH, ACK] Seq=2841 Ack=182 Win=65535 Len=1504 [TCP PDU reassembled in 10]
9	0.204863	10.0.2.15	74.207.232.127	TCP	54	42273 → 443 [ACK] Seq=182 Ack=4345 Win=36920 Len=0
10	0.205169	74.207.232.127	10.0.2.15	TLSv1	1161	Certificate, Server Key Exchange, Server Hello Done
11	0.205181	10.0.2.15	74.207.232.127	TCP	54	42273 → 443 [ACK] Seq=182 Ack=5452 Win=39760 Len=0
12	0.209753	10.0.2.15	74.207.232.127	TLSv1	252	Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
13	0.210119	74.207.232.127	10.0.2.15	TCP	60	443 → 42273 [ACK] Seq=5452 Ack=380 Win=65535 Len=0
14	0.251979	74.207.232.127	10.0.2.15	TLSv1	352	New Session Ticket, Change Cipher Spec, Encrypted Handshake Message
15	0.288847	10.0.2.15	74.207.232.127	TCP	54	42273 → 443 [ACK] Seq=380 Ack=5750 Win=42600 Len=0
16	0.553086	10.0.2.15	74.207.232.127	TLSv1	432	Application Data, Application Data
17	0.553358	74.207.232.127	10.0.2.15	TCP	60	443 → 42273 [ACK] Seq=5750 Ack=758 Win=65535 Len=0

Wireshark Interface: Packet Sniffing

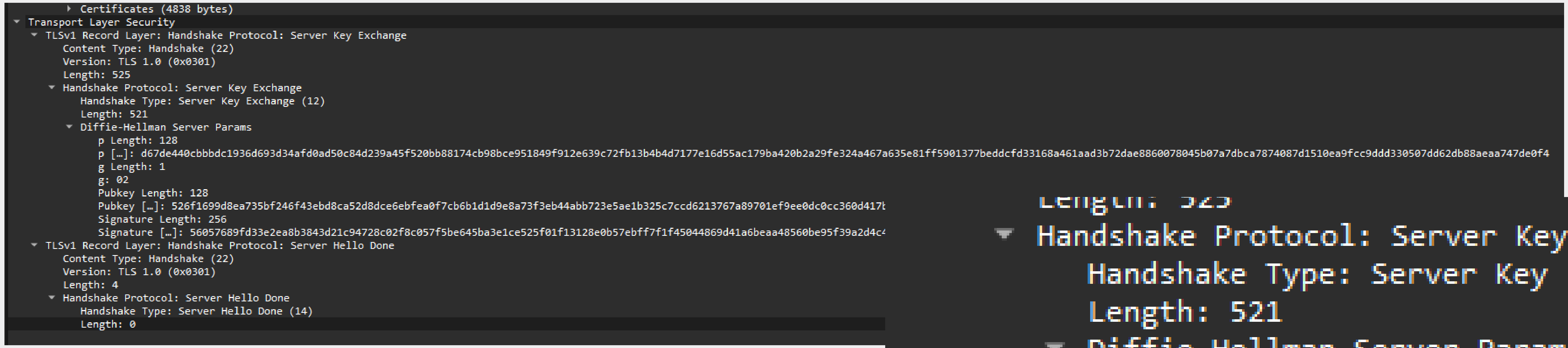


```
ssl.handshake

No.  Time      Source          Destination     Protocol Length Info
   4 0.039976  10.0.2.15       74.207.232.127  TLSv1     235 Client Hello (SNI=appliance.cloudshark.org)
   6 0.204522  74.207.232.127  10.0.2.15       TLSv1    2894 Server Hello
  10 0.205169  74.207.232.127  10.0.2.15       TLSv1    1161 Certificate, Server Key Exchange, Server Hello Done
  12 0.209753  10.0.2.15       74.207.232.127  TLSv1     252 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
  14 0.251979  74.207.232.127  10.0.2.15       TLSv1     352 New Session Ticket, Change Cipher Spec, Encrypted Handshake Message
```

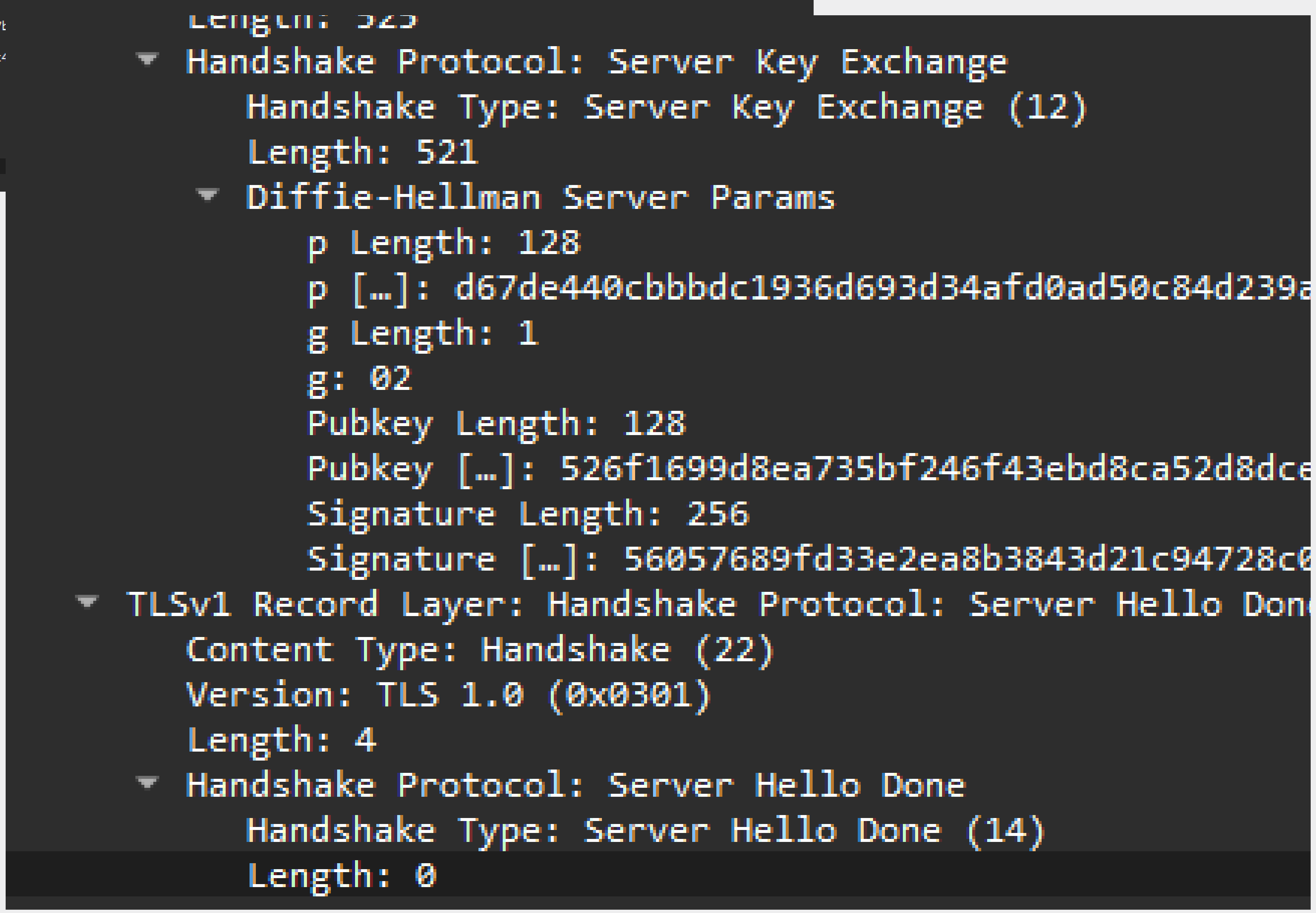


```
Length Info
 235 Client Hello (SNI=appliance.cloudshark.org)
2894 Server Hello
1161 Certificate, Server Key Exchange, Server Hello Done
 252 Client Key Exchange, Change Cipher Spec, Encrypted H
 352 New Session Ticket, Change Cipher Spec, Encrypted Ha
```

Wireshark Interface: Packet Analysis



```
        Certificates (4838 bytes)
Transport Layer Security
  TLSv1 Record Layer: Handshake Protocol: Server Key Exchange
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 525
    Handshake Protocol: Server Key Exchange
      Handshake Type: Server Key Exchange (12)
      Length: 521
      Diffie-Hellman Server Params
        p Length: 128
        p […]: d67de440cbbbdc1936d693d34afd0ad50c84d239a45f520bb88174cb98bce951849f912e639c72fb13b4b4d7177e16d55ac179ba420b2a29fe324a467a635e81ff5901377beddcfd33168a461aad3b72dae8860078045b07a7dbca7874087d1510ea9fcc9ddd330507dd62db88aeaa747de0f4
        g Length: 1
        g: 02
        Pubkey Length: 128
        Pubkey […]: 526f1699d8ea735bf246f43ebd8ca52d8dce6ebfea0f7cb6b1d1d9e8a73f3eb44abb723e5ae1b325c7ccd6213767a89701ef9ee0dc0cc360d417b
        Signature Length: 256
        Signature […]: 56057689fd33e2ea8b3843d21c94728c02f8c057f5be645ba3e1ce525f01f13128e0b57ebff7f1f45044869d41a6beaa48560be95f39a2d4c4
  TLSv1 Record Layer: Handshake Protocol: Server Hello Done
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 4
    Handshake Protocol: Server Hello Done
      Handshake Type: Server Hello Done (14)
      Length: 0
```



```
    Length: 525
    Handshake Protocol: Server Key Exchange
      Handshake Type: Server Key Exchange (12)
      Length: 521
      Diffie-Hellman Server Params
        p Length: 128
        p […]: d67de440cbbbdc1936d693d34afd0ad50c84d239a
        g Length: 1
        g: 02
        Pubkey Length: 128
        Pubkey […]: 526f1699d8ea735bf246f43ebd8ca52d8dce
        Signature Length: 256
        Signature […]: 56057689fd33e2ea8b3843d21c94728c0
  TLSv1 Record Layer: Handshake Protocol: Server Hello Done
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 4
    Handshake Protocol: Server Hello Done
      Handshake Type: Server Hello Done (14)
      Length: 0
```

Approch 1 : Sucessfull

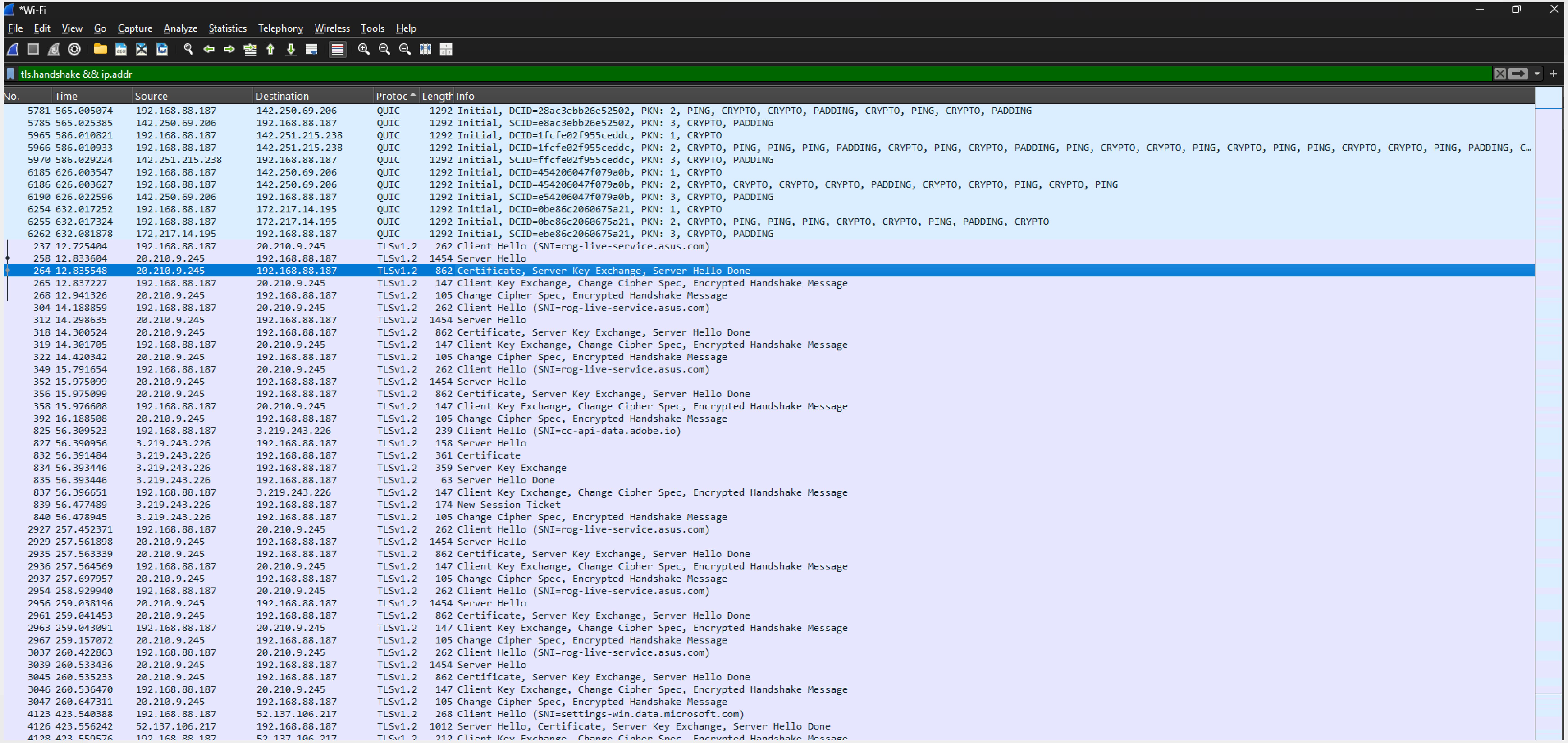
Classic Diffie-Hellman

Approch 2 : Fail

Elliptic Curve Diffie-Hellman

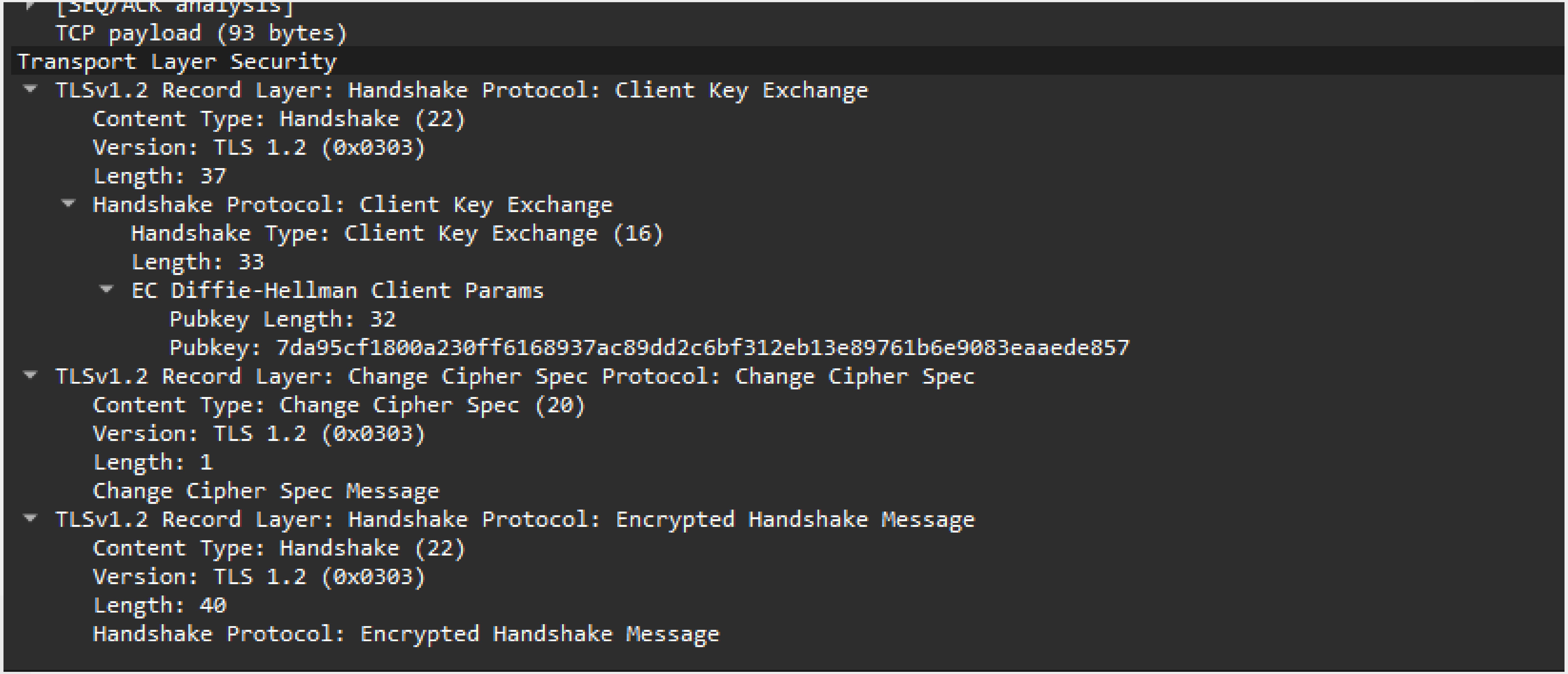
It is an extension of the classic Diffie-Hellman key exchange protocol, which uses elliptic curve cryptography (ECC) to enable the secure exchange of cryptographic keys. ECDH is widely used in modern systems because of its strong security and efficiency with smaller key sizes compared to traditional methods.

Wireshark Interface: Capturing Packets


*Wi-Fi
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
tls.handshake && ip.addr

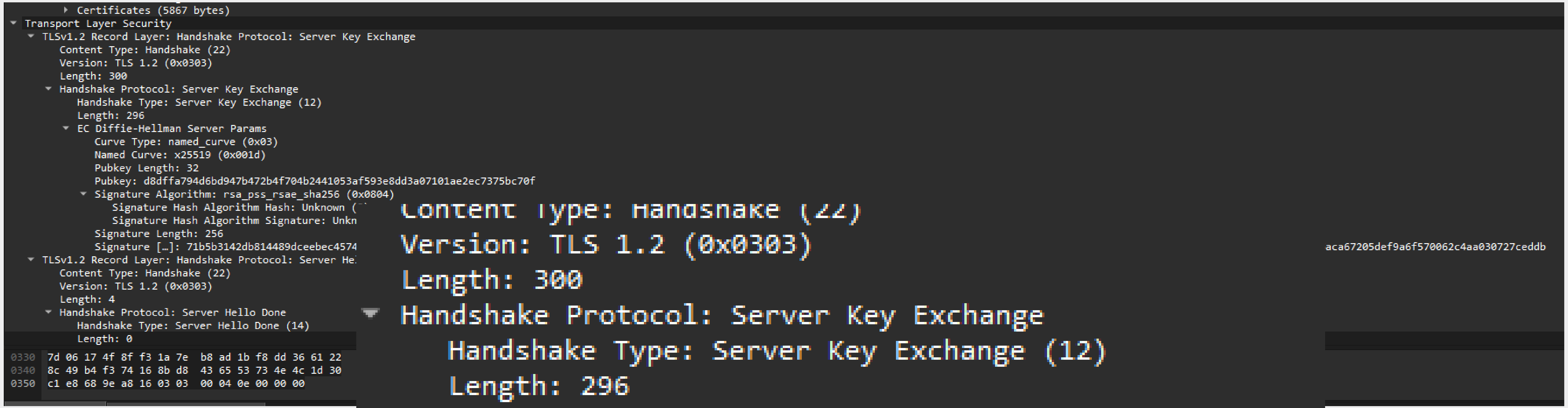
```
No.    Time        Source           Destination      Protoc  Length Info
  5781 565.005074  192.168.88.187   142.250.69.206   QUIC    1292 Initial, DCID=28ac3ebb26e52502, PKN: 2, PING, CRYPTO, CRYPTO, PADDING, CRYPTO, PING, CRYPTO, PADDING
  5785 565.025385  142.250.69.206   192.168.88.187   QUIC    1292 Initial, SCID=e8ac3ebb26e52502, PKN: 3, CRYPTO, PADDING
  5965 586.010821  192.168.88.187   142.251.215.238  QUIC    1292 Initial, DCID=1fcfe02f955ceddc, PKN: 1, CRYPTO
  5966 586.010933  192.168.88.187   142.251.215.238  QUIC    1292 Initial, DCID=1fcfe02f955ceddc, PKN: 2, CRYPTO, PING, PING, PING, PADDING, CRYPTO, PING, CRYPTO, PADDING, PING, CRYPTO, CRYPTO, PING, CRYPTO, PING, PING, CRYPTO, CRYPTO, PING, PADDING, C…
  5970 586.029224  142.251.215.238  192.168.88.187   QUIC    1292 Initial, SCID=ffcfe02f955ceddc, PKN: 3, CRYPTO, PADDING
  6185 626.003547  192.168.88.187   142.250.69.206   QUIC    1292 Initial, DCID=454206047f079a0b, PKN: 1, CRYPTO
  6186 626.003627  192.168.88.187   142.250.69.206   QUIC    1292 Initial, DCID=454206047f079a0b, PKN: 2, CRYPTO, CRYPTO, CRYPTO, CRYPTO, PADDING, CRYPTO, CRYPTO, PING, CRYPTO, PING
  6190 626.022596  142.250.69.206   192.168.88.187   QUIC    1292 Initial, SCID=e54206047f079a0b, PKN: 3, CRYPTO, PADDING
  6254 632.017252  192.168.88.187   172.217.14.195   QUIC    1292 Initial, DCID=0be86c2060675a21, PKN: 1, CRYPTO
  6255 632.017324  192.168.88.187   172.217.14.195   QUIC    1292 Initial, DCID=0be86c2060675a21, PKN: 2, CRYPTO, PING, PING, PING, CRYPTO, CRYPTO, PING, PADDING, CRYPTO
  6262 632.081878  172.217.14.195   192.168.88.187   QUIC    1292 Initial, SCID=ebe86c2060675a21, PKN: 3, CRYPTO, PADDING
   237 12.725404   192.168.88.187   20.210.9.245     TLSv1.2  262 Client Hello (SNI=rog-live-service.asus.com)
   258 12.833604   20.210.9.245     192.168.88.187   TLSv1.2 1454 Server Hello
   264 12.835548   20.210.9.245     192.168.88.187   TLSv1.2  862 Certificate, Server Key Exchange, Server Hello Done
   265 12.837227   192.168.88.187   20.210.9.245     TLSv1.2  147 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
   268 12.941326   20.210.9.245     192.168.88.187   TLSv1.2  105 Change Cipher Spec, Encrypted Handshake Message
   304 14.188859   192.168.88.187   20.210.9.245     TLSv1.2  262 Client Hello (SNI=rog-live-service.asus.com)
   312 14.298635   20.210.9.245     192.168.88.187   TLSv1.2 1454 Server Hello
   318 14.300524   20.210.9.245     192.168.88.187   TLSv1.2  862 Certificate, Server Key Exchange, Server Hello Done
   319 14.301705   192.168.88.187   20.210.9.245     TLSv1.2  147 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
   322 14.420342   20.210.9.245     192.168.88.187   TLSv1.2  105 Change Cipher Spec, Encrypted Handshake Message
   349 15.791654   192.168.88.187   20.210.9.245     TLSv1.2  262 Client Hello (SNI=rog-live-service.asus.com)
   352 15.975099   20.210.9.245     192.168.88.187   TLSv1.2 1454 Server Hello
   356 15.975099   20.210.9.245     192.168.88.187   TLSv1.2  862 Certificate, Server Key Exchange, Server Hello Done
   358 15.976608   192.168.88.187   20.210.9.245     TLSv1.2  147 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
   392 16.188508   20.210.9.245     192.168.88.187   TLSv1.2  105 Change Cipher Spec, Encrypted Handshake Message
   825 56.309523   192.168.88.187   3.219.243.226    TLSv1.2  239 Client Hello (SNI=cc-api-data.adobe.io)
   827 56.390956   3.219.243.226    192.168.88.187   TLSv1.2  158 Server Hello
   832 56.391484   3.219.243.226    192.168.88.187   TLSv1.2  361 Certificate
   834 56.393446   3.219.243.226    192.168.88.187   TLSv1.2  359 Server Key Exchange
   835 56.393446   3.219.243.226    192.168.88.187   TLSv1.2   63 Server Hello Done
   837 56.396651   192.168.88.187   3.219.243.226    TLSv1.2  147 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
   839 56.477489   3.219.243.226    192.168.88.187   TLSv1.2  174 New Session Ticket
   840 56.478945   3.219.243.226    192.168.88.187   TLSv1.2  105 Change Cipher Spec, Encrypted Handshake Message
  2927 257.452371  192.168.88.187   20.210.9.245     TLSv1.2  262 Client Hello (SNI=rog-live-service.asus.com)
  2929 257.561898  20.210.9.245     192.168.88.187   TLSv1.2 1454 Server Hello
  2935 257.563339  20.210.9.245     192.168.88.187   TLSv1.2  862 Certificate, Server Key Exchange, Server Hello Done
  2936 257.564569  192.168.88.187   20.210.9.245     TLSv1.2  147 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
  2937 257.697957  20.210.9.245     192.168.88.187   TLSv1.2  105 Change Cipher Spec, Encrypted Handshake Message
  2954 258.929940  192.168.88.187   20.210.9.245     TLSv1.2  262 Client Hello (SNI=rog-live-service.asus.com)
  2956 259.038196  20.210.9.245     192.168.88.187   TLSv1.2 1454 Server Hello
  2961 259.041453  20.210.9.245     192.168.88.187   TLSv1.2  862 Certificate, Server Key Exchange, Server Hello Done
  2963 259.043091  192.168.88.187   20.210.9.245     TLSv1.2  147 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
  2967 259.157072  20.210.9.245     192.168.88.187   TLSv1.2  105 Change Cipher Spec, Encrypted Handshake Message
  3037 260.422863  192.168.88.187   20.210.9.245     TLSv1.2  262 Client Hello (SNI=rog-live-service.asus.com)
  3039 260.533436  20.210.9.245     192.168.88.187   TLSv1.2 1454 Server Hello
  3045 260.535233  20.210.9.245     192.168.88.187   TLSv1.2  862 Certificate, Server Key Exchange, Server Hello Done
  3046 260.536470  192.168.88.187   20.210.9.245     TLSv1.2  147 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
  3047 260.647311  20.210.9.245     192.168.88.187   TLSv1.2  105 Change Cipher Spec, Encrypted Handshake Message
  4123 423.540388  192.168.88.187   52.137.106.217   TLSv1.2  268 Client Hello (SNI=settings-win.data.microsoft.com)
  4126 423.556242  52.137.106.217   192.168.88.187   TLSv1.2 1012 Server Hello, Certificate, Server Key Exchange, Server Hello Done
  4128 423.559576  192.168.88.187   52.137.106.217   TLSv1.2  212 Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
```

Wireshark Interface: Packet Sniffing

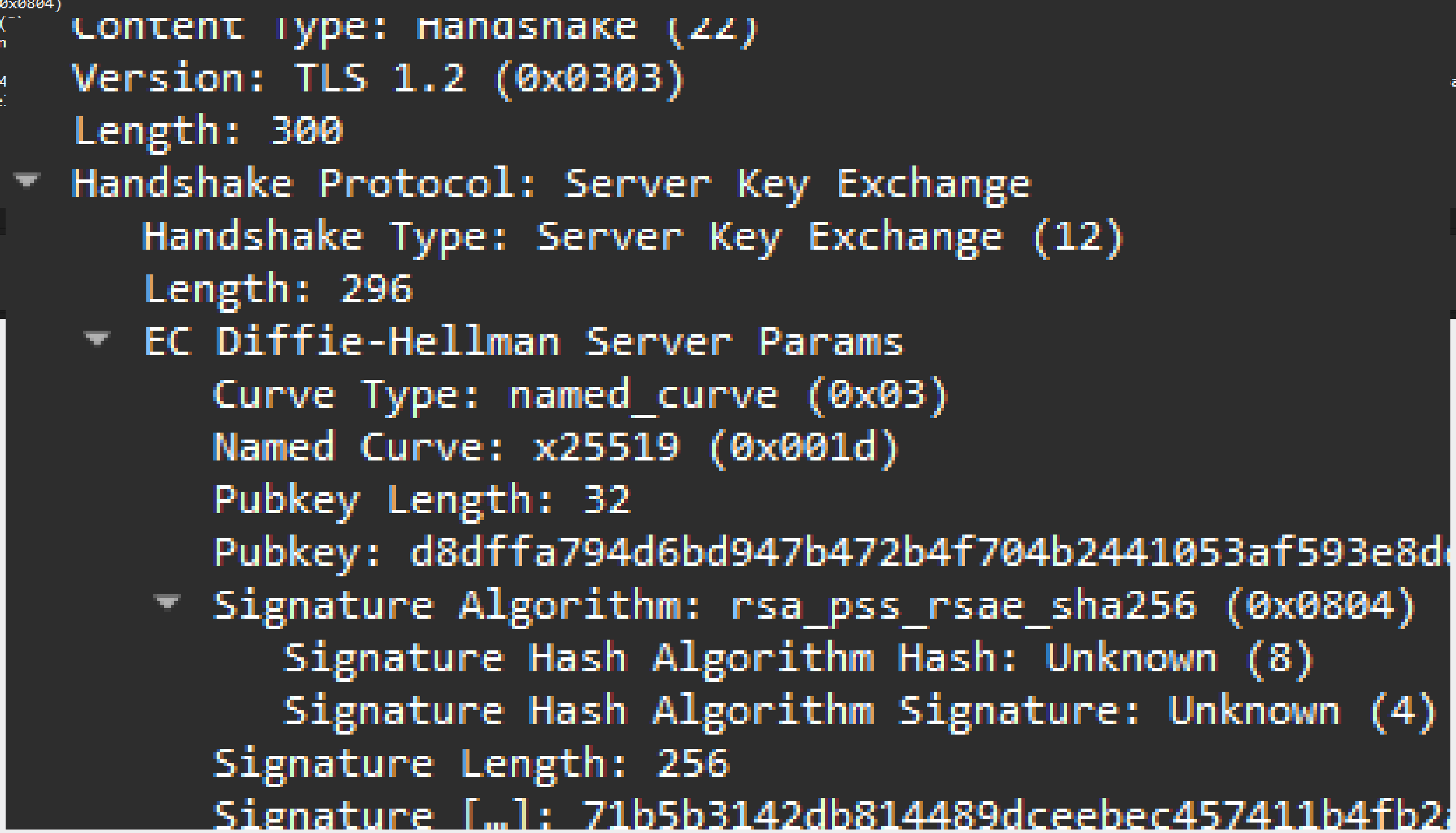


```
 ▸ [SEQ/ACK analysis]
   TCP payload (93 bytes)
Transport Layer Security
 ▾ TLSv1.2 Record Layer: Handshake Protocol: Client Key Exchange
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 37
    ▾ Handshake Protocol: Client Key Exchange
         Handshake Type: Client Key Exchange (16)
         Length: 33
       ▾ EC Diffie-Hellman Client Params
            Pubkey Length: 32
            Pubkey: 7da95cf1800a230ff6168937ac89dd2c6bf312eb13e89761b6e9083eaaede857
 ▾ TLSv1.2 Record Layer: Change Cipher Spec Protocol: Change Cipher Spec
      Content Type: Change Cipher Spec (20)
      Version: TLS 1.2 (0x0303)
      Length: 1
      Change Cipher Spec Message
 ▾ TLSv1.2 Record Layer: Handshake Protocol: Encrypted Handshake Message
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 40
      Handshake Protocol: Encrypted Handshake Message
```

Wireshark Interface: Packet Analysis



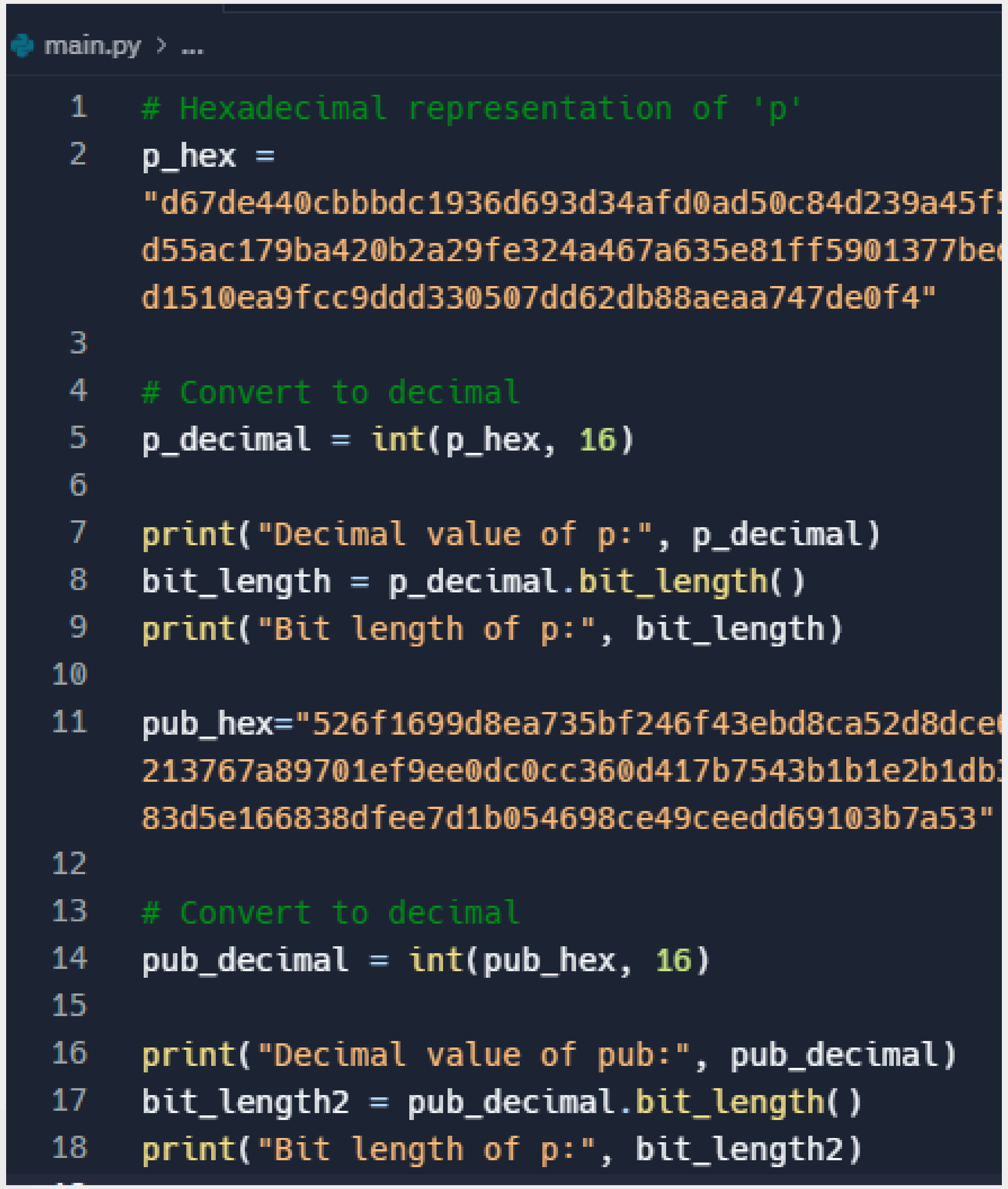
```
        ▸ Certificates (5867 bytes)
▾ Transport Layer Security
  ▾ TLSv1.2 Record Layer: Handshake Protocol: Server Key Exchange
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 300
    ▾ Handshake Protocol: Server Key Exchange
        Handshake Type: Server Key Exchange (12)
        Length: 296
      ▾ EC Diffie-Hellman Server Params
          Curve Type: named_curve (0x03)
          Named Curve: x25519 (0x001d)
          Pubkey Length: 32
          Pubkey: d8dffa794d6bd947b472b4f704b2441053af593e8dd3a07101ae2ec7375bc70f
        ▾ Signature Algorithm: rsa_pss_rsae_sha256 (0x0804)
            Signature Hash Algorithm Hash: Unknown (
            Signature Hash Algorithm Signature: Unkn
          Signature Length: 256
          Signature […]: 71b5b3142db814489dceebec4574                    aca67205def9a6f570062c4aa030727ceddb
  ▾ TLSv1.2 Record Layer: Handshake Protocol: Server He
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 4
    ▾ Handshake Protocol: Server Hello Done
        Handshake Type: Server Hello Done (14)
        Length: 0
0330  7d 06 17 4f 8f f3 1a 7e  b8 ad 1b f8 dd 36 61 22
0340  8c 49 b4 f3 74 16 8b d8  43 65 53 73 4e 4c 1d 30
0350  c1 e8 68 9e a8 16 03 03  00 04 0e 00 00 00
```



```
  Content Type: Handshake (22)
  Version: TLS 1.2 (0x0303)
  Length: 300
▾ Handshake Protocol: Server Key Exchange
    Handshake Type: Server Key Exchange (12)
    Length: 296
  ▾ EC Diffie-Hellman Server Params
      Curve Type: named_curve (0x03)
      Named Curve: x25519 (0x001d)
      Pubkey Length: 32
      Pubkey: d8dffa794d6bd947b472b4f704b2441053af593e8d
    ▾ Signature Algorithm: rsa_pss_rsae_sha256 (0x0804)
        Signature Hash Algorithm Hash: Unknown (8)
        Signature Hash Algorithm Signature: Unknown (4)
      Signature Length: 256
      Signature […]: 71b5b3142db814489dceebec457411b4fb2
```

Data Extraction

Python for parameter extraction



main.py > ...

```python
# Hexadecimal representation of 'p'
p_hex =
"d67de440cbbbdc1936d693d34afd0ad50c84d239a45f
d55ac179ba420b2a29fe324a467a635e81ff5901377be
d1510ea9fcc9ddd330507dd62db88aeaa747de0f4"

# Convert to decimal
p_decimal = int(p_hex, 16)

print("Decimal value of p:", p_decimal)
bit_length = p_decimal.bit_length()
print("Bit length of p:", bit_length)

pub_hex="526f1699d8ea735bf246f43ebd8ca52d8dce
213767a89701ef9ee0dc0cc360d417b7543b1b1e2b1db
83d5e166838dfee7d1b054698ce49ceedd69103b7a53"

# Convert to decimal
pub_decimal = int(pub_hex, 16)

print("Decimal value of pub:", pub_decimal)
bit_length2 = pub_decimal.bit_length()
print("Bit length of p:", bit_length2)
```

Back to Approch 1

Captured Parameters:

- p : Prime modulus
- g : Generator
- Public Key (Bpub)
- Signature for validation

Output:

Bit length of *P*: 920 (<2048)

Bit length of *Public Key*: 899

Verification Of Parameter

Verify if p is Prime:

- Used algorithms Miller-Rabin primality test to confirm that p is a prime number.
- Ensured p does not have small factors to prevent trivial attacks.

Validate Generator g:

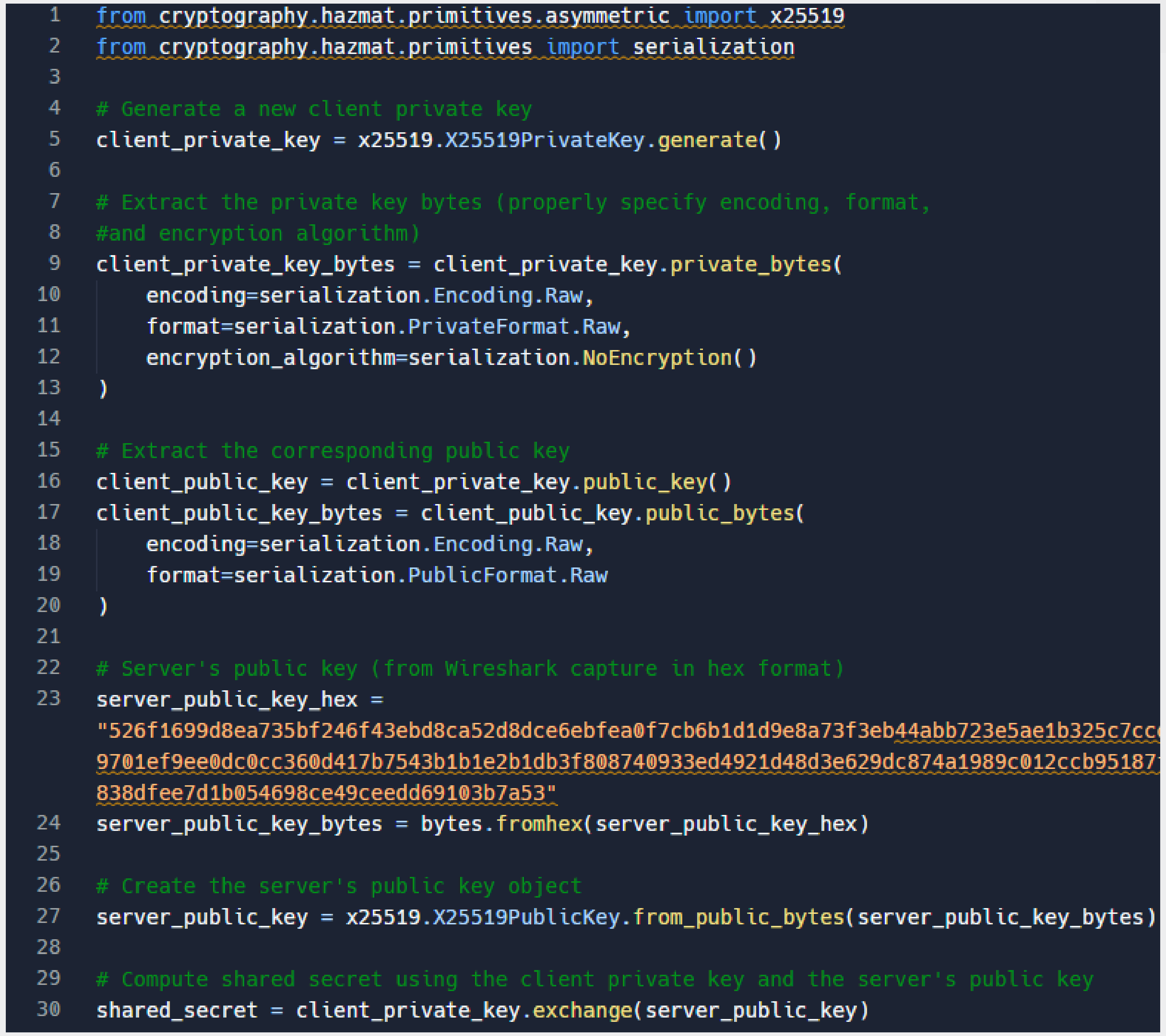
- Checked if g is a valid generator by ensuring it produces a large enough subgroup of order modulo p.

Assess Key Lengths:

- Verified the bit lengths of p, *g*, and B*pub* for compliance with modern security standards (e.g., minimum 2048 bits for strong security).

Shared Secret Key

Python code

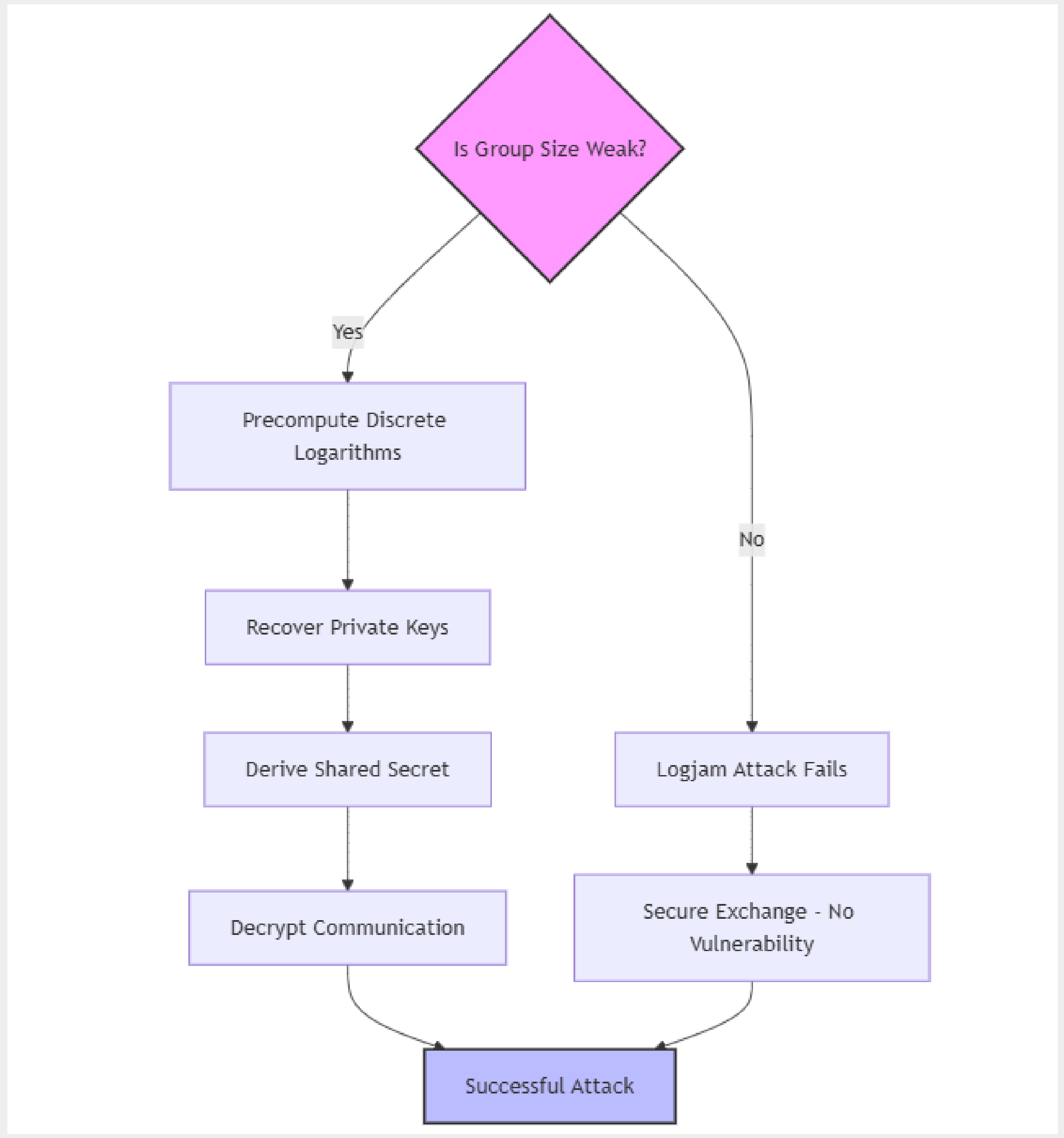


```python
from cryptography.hazmat.primitives.asymmetric import x25519
from cryptography.hazmat.primitives import serialization

# Generate a new client private key
client_private_key = x25519.X25519PrivateKey.generate()

# Extract the private key bytes (properly specify encoding, format,
#and encryption algorithm)
client_private_key_bytes = client_private_key.private_bytes(
    encoding=serialization.Encoding.Raw,
    format=serialization.PrivateFormat.Raw,
    encryption_algorithm=serialization.NoEncryption()
)

# Extract the corresponding public key
client_public_key = client_private_key.public_key()
client_public_key_bytes = client_public_key.public_bytes(
    encoding=serialization.Encoding.Raw,
    format=serialization.PublicFormat.Raw
)

# Server's public key (from Wireshark capture in hex format)
server_public_key_hex =
"526f1699d8ea735bf246f43ebd8ca52d8dce6ebfea0f7cb6b1d1d9e8a73f3eb44abb723e5ae1b325c7cc
9701ef9ee0dc0cc360d417b7543b1b1e2b1db3f808740933ed4921d48d3e629dc874a1989c012ccb95187
838dfee7d1b054698ce49ceedd69103b7a53"
server_public_key_bytes = bytes.fromhex(server_public_key_hex)

# Create the server's public key object
server_public_key = x25519.X25519PublicKey.from_public_bytes(server_public_key_bytes)

# Compute shared secret using the client private key and the server's public key
shared_secret = client_private_key.exchange(server_public_key)
```

Output:

Shared Secret (hex):
046033eea414ae2b7ff34e52f53af960
0ebe33fe1a9feef09c6c6add27e91c.

Logjam Attack



Attack Findings:

- For strong p sizes (> 1024 bits), Logjam attacks are computationally infeasible.
- Smaller group sizes (e.g., 512-bit) are vulnerable and allow attackers to recover shared secrets quickly.

Mitigation Recommendations:

- Disallow export-grade DH groups in TLS configurations.
- Use sufficiently large p values (≥2048 bits).
- Transition to Elliptic Curve Diffie-Hellman (ECDH) for stronger security.

Proposed Solutions

Enhancing Security:

- Use larger primes (≥2048-bit)
- Implement Ephemeral DH (DHE) or Elliptic Curve DH (ECDHE)
- Disable weak ciphersuites
- Validate server signatures during handshakes

Best Practices:

- Regularly update cryptographic libraries
- Perform security audits on TLS implementations

Expanding the Scope

Elliptic Curve Diffie-Hellman (ECDH) Analysis:

Extend the analysis to ECDH, which offers stronger security with smaller key sizes, to evaluate its real-world implementations and vulnerabilities

Automated Vulnerability Scanning:

Build scripts or software to automate the identification of weak DH parameters across large-scale network deployments.

Cross-Device Analysis:

Broaden the scope to compare the implementation of DH across a wider variety of devices, including IoT systems and older network hardware.