

# TLS 핸드셰이크 상세 분석

<https://youtu.be/ESAAStI4sGQ>



- **Transport Layer Security**

- 컴퓨터 네트워크 상에서 안전한 통신을 보장하기 위한 암호화 프로토콜
- 일반적으로 TCP(Transmission Control Protocol, 전송 제어 프로토콜) 위에서 동작
- TLS 1.3은 2018년에 표준화된 최신 버전
  - 기존 버전보다 더 강화된 보안성, 지연 시간 감소, 단순화된 설정 제공
  - Client-Server 사이의 인증, 데이터 암호화, 무결성 제공

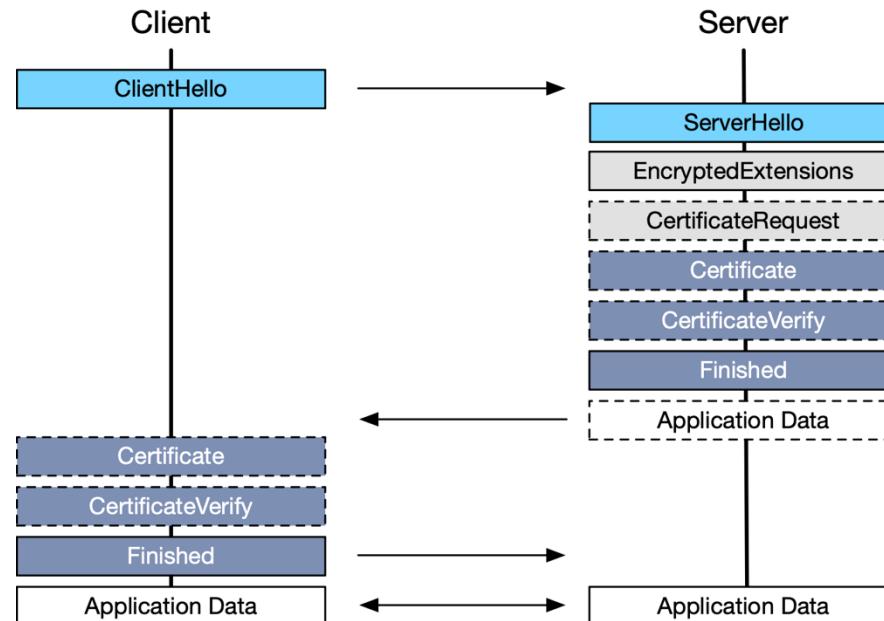
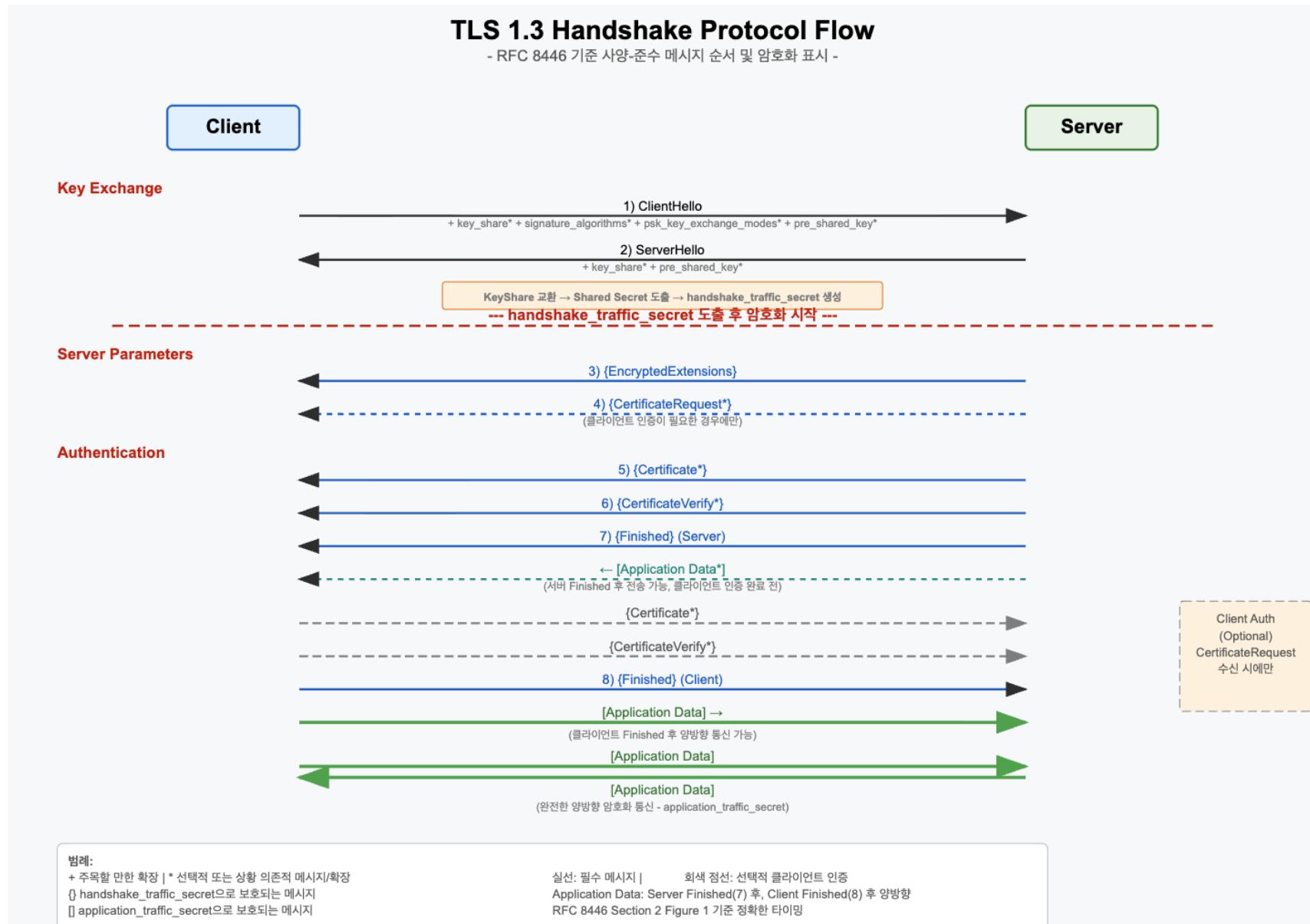


Fig. 1: TLS 1.3 Message Flow (based on RFC 8446)

# TLS Handshake

- 클라이언트와 서버가 암호화된 세션을 수립하기 위한 초기 과정
  - 암호 스펙 협상, 키 교환, 인증, 무결성 검증을 수행
  - TLS 1.3은 1 RTT(Round Trip Time)로 축소되어 지연시간을 크게 감소시킴
- 
- 주요 단계:
    - 1) ClientHello / ServerHello → 키 교환 및 알고리즘 협상
    - 2) Shared Secret 생성 → handshake\_secret, master\_secret 파생
    - 3) 인증 및 Finished 메시지 교환
    - 4) Application Data 단계 전환

# TLS 1.3 Handshake 프로토콜 Flow



# ClientHello

- 클라이언트가 세션 시작 시 지원 가능한 암호 스위트, 키교환 방식, 서명 알고리즘 등을 서버에게 알림
- 핵심 필드
  - Legacy\_version = 0x0303(호환 표기)
  - supported\_verisons = 0x0304(확장필드에 적혀있음)
  - cipher\_suites : TLS\_AES\_128\_GCM\_SHA256, TLS\_AES\_256\_GCM\_SHA384 등
  - key\_share(확장) : 클라이언트가 생성한 ECDHE 키 쌍의 공개키
    - FFDHE :  $X = g^x \text{ mod } p$
    - ECDHE(주로 사용) :  $X = x \cdot G$ (타원곡선 점) ( $x \leftarrow \text{Random}()$ )
  - signature\_algorithms : RSA-PSS, ECDSA, EdDSA(서명 용도)
  - (선택적)supported\_groups, pre\_shared\_key, alpn 등
- 32-byte client\_random 생성 : 세션 식별 및 다운그레이드 탐지 용도
- Client 전송(평문)

구분	필드명	설명		bytes
Header	legacy_version	항상 0x0303 (TLS 1.2 호환용)	평문	2
Header	random	32 바이트 난수 (다운그레이드 탐지 가능)	평문	32
Header	legacy_session_id	옛 세션 ID (비워둘 수 없음, 0~32 바이트)	평문	1 (길이) + 0~32
Header	cipher_suites	지원 암호 스위트 목록 (각 스위트 2 바이트)	평문	2 (목록 길이) + 2×n
Header	legacy_compression_methods	항상 1 개의 값(0x00)만 포함	평문	1 (길이) + 1 = 2
Extensions	extensions	실제 핵심 TLS 1.3 협상 정보 포함	평문	2 (전체 길이) + N

# DHE(Ephemeral Diffie-Hellman)

- 매 연결마다 새로 뽑는(=Ephemeral) DH 개인키로 키 합의 하는 방식
- 장점
  - 전방 안전성(Forward Secrecy) 확보
  - 세션이 끝난 뒤 서버의 장기 비밀키가 유출되어도 과거 트래픽 복호화 불가
- 구현 형태
  - FFDHE : 유한체(정수 모듈러  $p$ ) 위의 DH
  - ECDHE : 타원곡선 위의 DH
- TLS 1.3 메시지에서
  - ClientHello.key\_share에 클라이언트가 지원 그룹들의 KeyShare(공개값)들을 넣어 전송하고, ServerHello.key\_share로 서버가 하나 골라 자기 공개값을 회신  
→ 양쪽이 공유 비밀키 생성

# ServerHello

- 채택한 암호 스위트/키교환 그룹을 통지하고 서버의 ECDHE 공개키 제공
- 핵심 필드
  - Legacy\_version = 0x0303(호환 표기)
  - supported\_verisons = 0x0304(확장필드에 적혀있음)
  - Cipher\_suite : 채택된 1개
  - Key\_share : 서버 공개키  $Y$ 
    - FFDHE :  $Y = g^y \text{ mod } p$
    - ECDHE(주로 사용) :  $Y = y \cdot G(\text{타원곡선 점}) \quad (y \leftarrow \text{Random}())$
- 공유 비밀(shared\_secret) :  
FFDHE :  $\text{shared} = g^{xy}$    **ECDHE : shared =  $(xy) \cdot G$**
- 키 스케줄 시작 : **early\_secret → handshake\_secret → client/ server\_handshake\_traffic\_secret**

구분	필드명	설명	암호화 여부	길이 (bytes)
Header	legacy_version	항상 0x0303 (TLS 1.2 고정, 호환성용)	평문	2 bytes
Header	random	32바이트 난수 (마지막 8바이트는 다운그레이드 Sentinel 가능)	평문	32 bytes
Header	legacy_session_id_echo	클라이언트의 legacy_session_id 그대로 반환 (호환성 필드)	평문	1 byte 길이 + 0~32 bytes 데이터
Header	cipher_suite	서버가 선택한 단일 암호 스위트 (ex: 0x1301)	평문	2 bytes
Header	legacy_compression_method	항상 0x00 (압축 미사용)	평문	1 byte
Extensions	extensions 블록	아래 확장 필드 목록으로 구성됨 (각각 길이 앞에 2바이트 length prefix 포함)	평문	2 bytes (전체 길이 필드) + N bytes 데이터

# ServerHello

```
struct {
    ProtocolVersion legacy_version = 0x0303;    // 항상 TLS 1.2
    Random random;
    opaque legacy_session_id<0..32>;
    CipherSuite cipher_suites<2..2^16-2>;
    opaque legacy_compression_methods<1..2^8-1>;
    Extension extensions<8..2^16-1>;           // 여기 안에 supported_versions 포함
} ClientHello;
```

Extension: supported\_versions

Type: supported\_versions (43)  
Length: 5  
Supported Versions length: 4  
**Supported Version: TLS 1.3 (0x0304)**  
**Supported Version: TLS 1.2 (0x0303)**

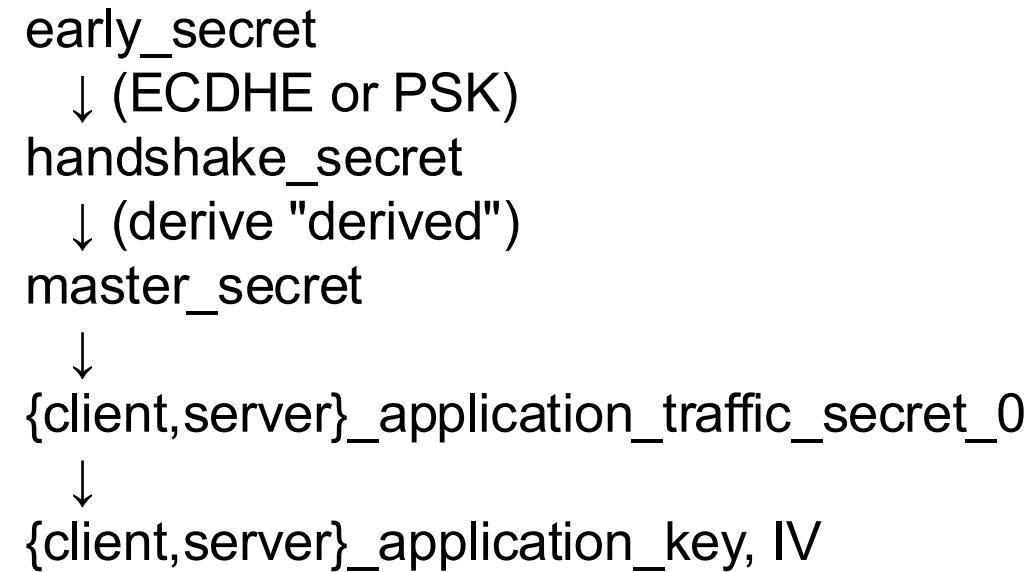
**ServerHello:**

legacy\_version = 0x0303  
...  
**Extension: supported\_versions**  
**Supported Version: TLS 1.3 (0x0304)**

# ServerHello 이후 HKDF 기반 키 스케줄링 과정

[키 스케줄링 흐름도 요약]

HKDF : HMAC을 기반으로 하는 키 유도함수



# ServerHello 이후 HKDF 기반 키 스케줄링 과정

- PSK(Pre-Share Key)
  - 미리 공유된 키(외부로 사전 배포된 키, 이전 세션에서 발급된 세션 티켓 등)로 새 연결을 빠르게 수립
  - TLS 1.3에서 사용위치
    - 세션 재개(resumption) : 서버가 이전에 연결해준 클라이언트에게 준 resumption ticket(=PSK)로 다음 연결을 1-RTT로 빠르게 재 수립
    - 0-RTT 데이터(optional) : PSK로 Early Data를 보낼 수 있음(재전송/리플레이 위험, 서버 정책으로 제한적)
  - TLS 1.3에서 사용되는 모드
    - psk\_ke : PSK만으로 키 파생  
→ 가장 빠르지만, 새로운 전방 안전성 없음 – PSK가 유출되면 해당 세션은 위험
    - psk\_dhe\_ke : PSK+(EC)DHE 를 함께 사용하여 전방 안전성 확보
  - TLS 1.3 메시지에서
    - ClientHello.pre\_shared\_key(PSK ID 목록, 바인더 포함, psk\_key\_exchange\_modes(psk\_ke/psk\_dhe\_ke 선호)로 협상

# ServerHello 이후 HKDF 기반 키 스케줄링 과정

- 1단계 : ServerHello 도착 시점

클라이언트가 ServerHello를 수신하면, 서버의 공개키  $Y$  를 획득함

이 시점에서 ECDHE 공유키 (shared\_secret) 계산 가능함

$$\text{Client: } \text{shared\_secret} = x \cdot Y$$

$$\text{Server: } \text{shared\_secret} = y \cdot X$$

두 값은 동일하며, 바로 다음 단계의 HKDF 입력값으로 사용됨

즉, ServerHello 수신 후, 클라이언트와 서버가 동일한 shared\_secret 생성함

# ServerHello 이후 HKDF 기반 키 스케줄링 과정

- 2단계 : HKDF 기반 키 스케줄 시작

세션에서 사용할 모든 키의 **seed** 생성 과정 시작됨

early\_secret → handshake\_secret → master\_secret

## step 1) early\_secret

아무 입력이 없는 초기 상태에서 생성됨 (=가장 처음 "빈 상태(0)"에서 시작)

`early_secret = HKDF-Extract(0, 0)`

PSK 재개 세션에서는 여기에 PSK 값이 여기에서 사용됨

# ServerHello 이후 HKDF 기반 키 스케줄링 과정

- 2단계 : HKDF 기반 키 스케줄 시작

## step 2) handshake\_secret

실제 ECDHE 공유키를 결합하여 시드 생성

```
handshake_secret =  
    HKDF-Extract(  
        Derive-Secret(early_secret, "derived", ""),  
        shared_secret  
    )
```

즉, handshake\_secret = early\_secret + ECDHE 결과를 섞은 값임

# ServerHello 이후 HKDF 기반 키 스케줄링 과정

(Handshake 단계 하위) **client/server handshake traffic secret** 생성(핸드쉐이크 트래픽용 secret 파생)  
handshake\_secret으로부터 서버와 클라이언트 각각이 쓸 암호화 키의 근본값(secret)을 뽑습니다.  
(=클라이언트/서버 방향별 트래픽 secret 파생)

```
client_handshake_traffic_secret =  
    Derive-Secret(handshake_secret,  
                  "c hs traffic",  
                  transcript_hash(ClientHello...ServerHello))
```

```
server_handshake_traffic_secret =  
    Derive-Secret(handshake_secret,  
                  "s hs traffic",  
                  transcript_hash(ClientHello...ServerHello))
```

**transcript\_hash**는 지금까지 교환된 핸드쉐이크 메시지를 의미 -> 메시지 무결성이 반영된 키 파생 구조임

즉, 서버와 클라이언트가 각자 사용할 **traffic\_secret**을 생성하는 단계

# ServerHello 이후 HKDF 기반 키 스케줄링 과정

## step 3) Master Secret 도출

```
master_secret =  
    HKDF-Extract(  
        Derive-Secret(handshake_secret, "derived", ""),  
        0  
    )
```

- Handshake\_secret 단계에서 마지막으로 파생됨
- 0을 입력값으로 사용하여 새로운 상위 시드 생성
- 이 master\_secret이 이후 모든 애플리케이션의 키의 뿌리(root)가 됨

이 master\_secret은 이후 Finished 메시지 직후

{client, server} application\_traffic\_secret\_0으로 전환됨.

즉, Finished 이후부터는 handshake 단계의 키 대신 application 단계 키로 세션이 보호됨

# ServerHello 이후 HKDF 기반 키 스케줄링 과정

- 3단계 : 실제 트래픽 키로 확장

이 secret들로부터 실제로 암호화를 수행할 대칭키와 IV 가 생성됨

```
client_handshake_key, client_handshake_iv =  
    HKDF-Expand(client_handshake_traffic_secret)
```

```
server_handshake_key, server_handshake_iv =  
    HKDF-Expand(server_handshake_traffic_secret)
```

이 시점부터 서버의 다음 메시지들 (EncryptedExtensions, Certificate, Finished 등)은  
server\_handshake\_key로 암호화되어 전송 됨.

# Application Traffic Secret 파생 및 의미

- Application Traffic Secret 파생

```
client_application_traffic_secret_0 =
    Derive-Secret(master_secret, "c ap traffic",
                  transcript_hash(ClientHello...Finished_S))

server_application_traffic_secret_0 =
    Derive-Secret(master_secret, "s ap traffic",
                  transcript_hash(ClientHello...Finished_S))
```

- Finished(Server) 메시지까지의 transcript\_hash를 입력으로 사용
- 클라이언트용, 서버용 각각 독립적인 secret이 생성됨
- 여기서 파생된 secret으로 실제 Application Data 암호화 키와 IV 생성됨

# ServerHello 이후 HKDF 기반 키 스케줄링 과정

- Client\_application\_traffic\_secret\_0 ?
  - 클라이언트 →서버 방향의 Application Data 암호화 키를 생성하기 위한 seed
  - \_0은 초기 세션 상태(첫 번째 트래픽 시크릿)을 의미  
(이후 KeyUpdate 메시지 발생 시 \_1, \_2 , ..로 갱신됨)

```
client_application_key, client_application_iv =  
    HKDF-Expand(client_application_traffic_secret_0)
```

서버는 server\_application\_traffic\_secret\_0을 동일한 방식으로 사용

# TLS 1.3 Handshake

## 3) EncryptedExtensions

- 암호화되어야 할 서버 측 확장들(extensions) 전달
- 예시 : server\_name(SNI), max\_fragment\_length, alph, record\_size\_limit 등
- 보호키 : server\_handshake\_traffic\_secret 로 AEAD보호

## 4) CertificateRequest(선택)

- 클라이언트 인증이 필요할 때만 전송  
어떤 신뢰 상위(허용 CA), 서명 알고리즘, 인증서 요구 조건 등을 지정
- 보호 키 : server\_handshake\_traffic\_secret
- 클라이언트 인증이 불필요하다면 이 단계는 스kip되고, 이후 클라이언트 인증 관련 메시지들도 생략됨

# TLS 1.3 Handshake

## 5) Certificate(서버 인증서)

- 서버 신원 증명 자료(X.509 chain) 전달
- 서버 EE 인증서(+중간 CA)가 들어있음  
공개키 유형은 RSA-PSS 또는 ECDSA/EdDSA
- 클라이언트 검사 : 체인/유효기간/정책/이름 일치(SAN)/ 해지 등
- 보호 키 : server\_handshake\_traffic\_secret



서버의 인증서 하나만으로는 신뢰할 수 없음.  
따라서 상위 인증서(Intermediate CA) 와 최상위 인증서(Root CA) 하여  
“이 인증서는 신뢰할 만한 기관이 발급한 것”임을 보장합니다.  
이를 인증서 체인(Certificate Chain) 또는 X.509 chain이라고 부름

## 6) Certificate Verify

- 서버가 인증서의 개인키를 실제로 보유함을 증명
- 서명 입력 : 그 시점까지의 handshake transcript hash를 자신의 개인키로 서명
  - **handshake transcript hash** : 지금까지 주고받은 모든 핸드쉐이크 메시지의 요약본(hash)  
$$\text{transcript\_hash} = \text{Hash}(\text{ClientHello} \parallel \text{ServerHello} \parallel \text{EncryptedExtensions} \parallel \text{Certificate})$$
- 알고리즘 : RSA-PSS, ECDSA, EdDSA
- 보호 키 : server\_handshake\_traffic\_secret

# TLS 1.3 Handshake

## 7) Finished(서버)

- 서버 핸드쉐이크 종료 + transcript(지금까지 수행한 핸드쉐이크 요약본) 무결성 증명
- finished\_key = HKDF-Expand-Label(server\_handshake\_traffic\_secret, "finished", "")  
verify\_data = HMAC(finished\_key, transcript\_hash(ClientHello..CertificateVerify\_S))
- 보호 키 : server\_handshake\_traffic\_secret
- 이 메시지 직후 서버는 **server\_application\_traffic\_secret\_0**로 전환해 Application Data를 곧바로 보낼 수 있음

클라이언트 인증을 사용하는 경우

7과 8 사이에 클라이언트 측 Certificate / CertificateVerify가 추가됨

## 8) Finished(클라이언트)

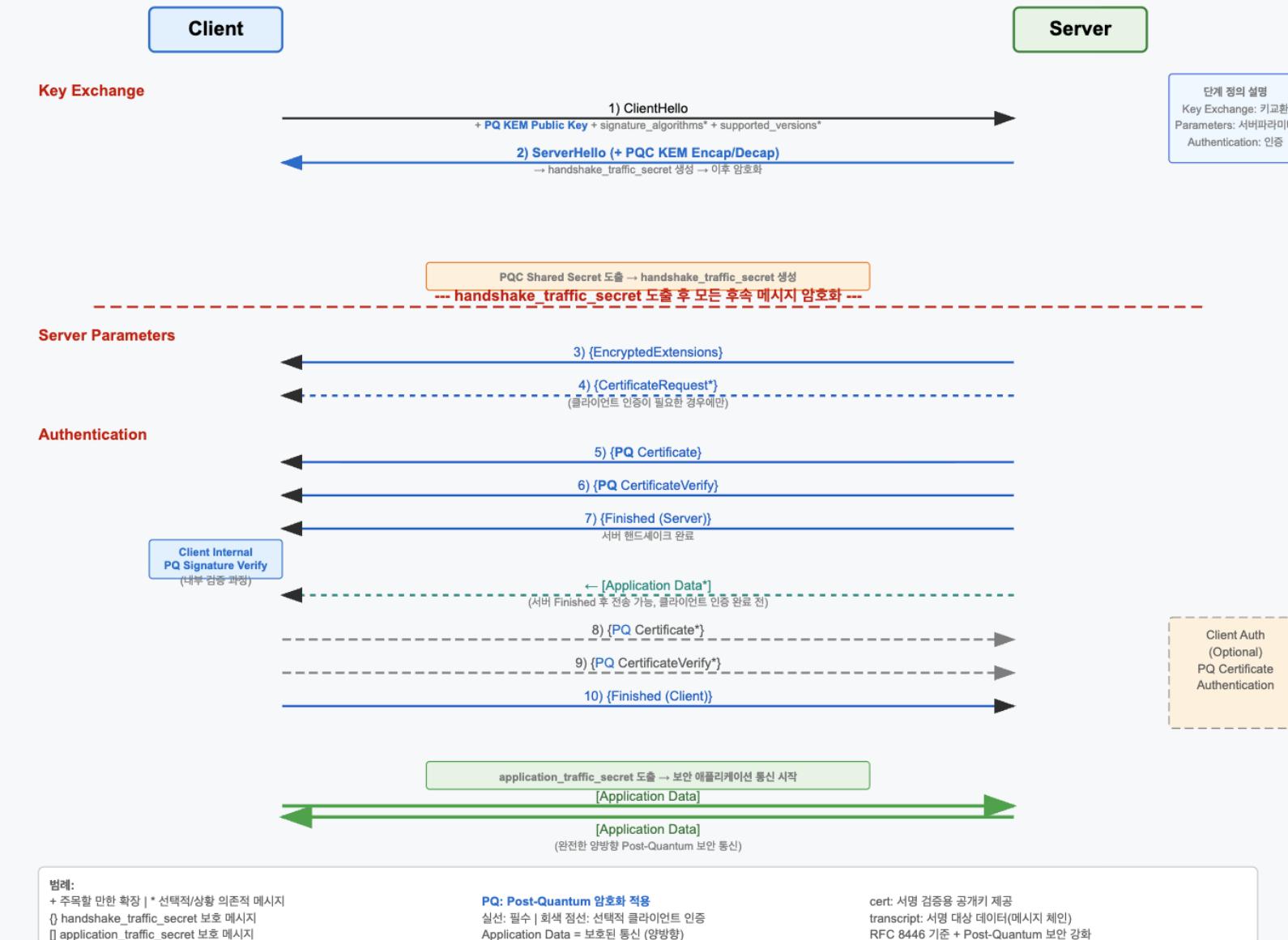
- 클라이언트도 핸드쉐이크 종료 + transcript 무결성 증명
- 보호 키 : client\_handshake\_traffic\_secret
- 이후 상태 : {client, server}\_**application\_traffic\_secret\_0**로 전환 완료  
즉, 양방향 Application Data 가능

# TLS 1.3 Handshake

단계	방향	메시지	암호화 여부	사용 키(레코드 보호 키)	비고
1	C → S	<b>ClientHello</b>	평문	—	key_share 포함
2	S → C	<b>ServerHello</b>	평문	—	여기까지 평문
3	S → C	<b>EncryptedExtensions</b>	암호화	<b>server_handshake_traffic_key</b>	ServerHello 이후부터 핸드셰이크 키 사용 시작
4	S → C	<b>CertificateRequest</b> (선택)	암호화	<b>server_handshake_traffic_key</b>	클라이언트 인증 쓸 때만
5	S → C	<b>Certificate</b> (서버)	암호화	<b>server_handshake_traffic_key</b>	서버 인증서 체인
6	S → C	<b>CertificateVerify</b> (서버)	암호화	<b>server_handshake_traffic_key</b>	transcript 서명 포함
7	S → C	<b>Finished</b> (서버)	암호화	<b>server_handshake_traffic_key</b>	서버 측 핸드셰이크 종료 신호
8	C → S	<b>Finished</b> (클라이언트)	암호화	<b>client_handshake_traffic_key</b>	이 직후 앱키로 전환 가능
9	양방향	<b>Application Data</b>	암호화	<b>{client,server}_application_traffic_key</b>	이후 전부 앱키로 보호

## TLS 1.3 Post-Quantum Handshake Protocol Flow

- RFC 8446 기준 Post-Quantum 암호화 적용 버전 -



Q & A