TLS 1.3

CHITRANG SRIVASTAVA

Agenda

- Recap of RSA, DH & Elliptic Curve
- Recap of TLS 1.2
- Introduction to TLS 1.3
- Authenticated Encryption & Additional Data (AEAD)

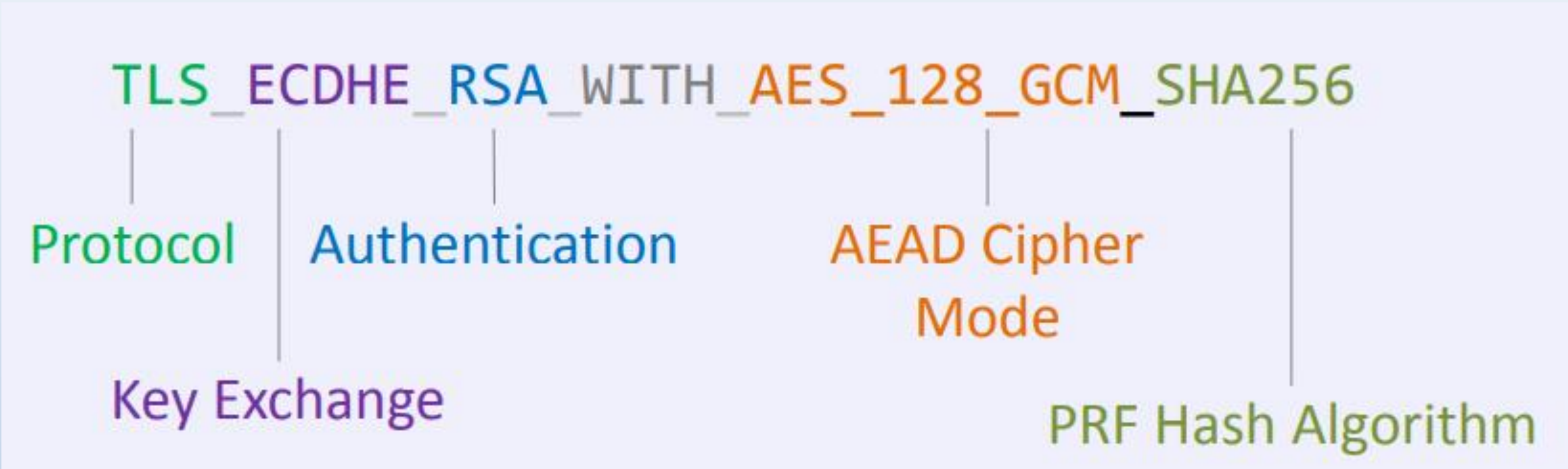
TLS/SSL Brief History

Developed originally by Netscape

Protocol	Published	Status
SSL 1.0	Unpublished	Unpublished
SSL 2.0	1995	Deprecated in 2011 (RFC 6176)
SSL 3.0	1996	Deprecated in 2015 (RFC 7568)
TLS 1.0	1999	Deprecation planned in 2020[11]
TLS 1.1	2006	Deprecation planned in 2020[11]
TLS 1.2	2008 (RFC 5246)	
TLS 1.3	2018 (RFC 8446)	

Source: Wikipedia

CipherSuite

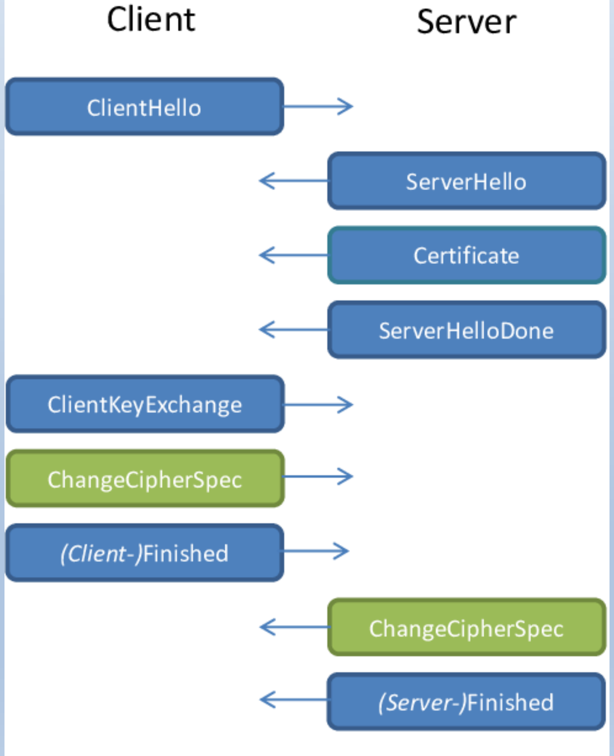


TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA256

When GCM is used, there is no per-record HMAC; integrity is **obtained from the GCM mode itself. So the hash function specified in the cipher suite is used only for the PRF and othe**r handshake-related usages. Source: StackExchange

TLS 1.2 Handshake with RSA

- Pre_Master_Secret = Random 48 bytes generated and encrypted using Server's public key.
- Master Secret = PRF(PMS, Client.Random, Server.Random, "master secret")
- Many keys derived like IV, Read-Write session key, read-write MAC keys, PRF(MS, "key expansion".
- ChangeCipherSec
- Client Finished, hash of all handshake message.
- 2 Round Trips.



PRF

- PRF(“secret” + “label” + “non-secret”)
- `master_secret = PRF(pre_master_secret, "master secret", ClientHello.random + ServerHello.random)`

- `PRF(SecurityParameters.master_secret,"key expansion", SecurityParameters.server_random +SecurityParameters.client_random);`
- `PRF(master_secret, “client finished”, Hash(handshake_messages))`

```
P_hash(secret, seed) = HMAC_hash(secret, A(1) + seed) +
                             HMAC_hash(secret, A(2) + seed) +
                             HMAC_hash(secret, A(3) + seed) + ...

   where + indicates concatenation.

   A() is defined as:

      A(0) = seed
      A(i) = HMAC_hash(secret, A(i-1))

   P_hash can be iterated as many times as necessary to produce the
   required quantity of data.  Reference
```

SHA (Secure Hash Algorithm)

- **Pre-image resistance.**

Given a hash value h it should be difficult to find any message m such that h = hash(m).

- **Second pre-image resistance.**

Given an input m1, it should be difficult to find a different input m2 such that hash(m1) = hash(m2).

- **Collision resistance.**
- It should be difficult to find two different messages m1 and m2 such that hash(m1) = hash(m2).
- SHA1(160 bits), SHA2(224/256 bits), SHA384, SHA-3
- https://www.youtube.com/watch?v=E4FL9Tv-X-k
- https://8gwifi.org/MessageDigest.jsp

Message Authentication Code (MAC)

- MAC is a tag of data computed with a key
- HMAC is one such MAC algorithm which is a recipe for turning hash functions into MAC i.e. HMAC-SHA256

This definition is taken from RFC 2104:

$$\mathrm{HMAC}(K, m) = \mathrm{H}\Big((K' \oplus opad) \parallel \mathrm{H}\big((K' \oplus ipad) \parallel m \big) \Big)$$

$$K' = \begin{cases} \mathrm{H}(K) & K \text{ is larger than block size} \\ K & \text{otherwise} \end{cases}$$

where

H is a cryptographic hash function

m is the message to be authenticated

K is the secret key

K' is a block-sized key derived from the secret key, *K*; either by padding to the right with 0s up to the block size, or by hashing down to less than the block size first and then padding to the right with zeros

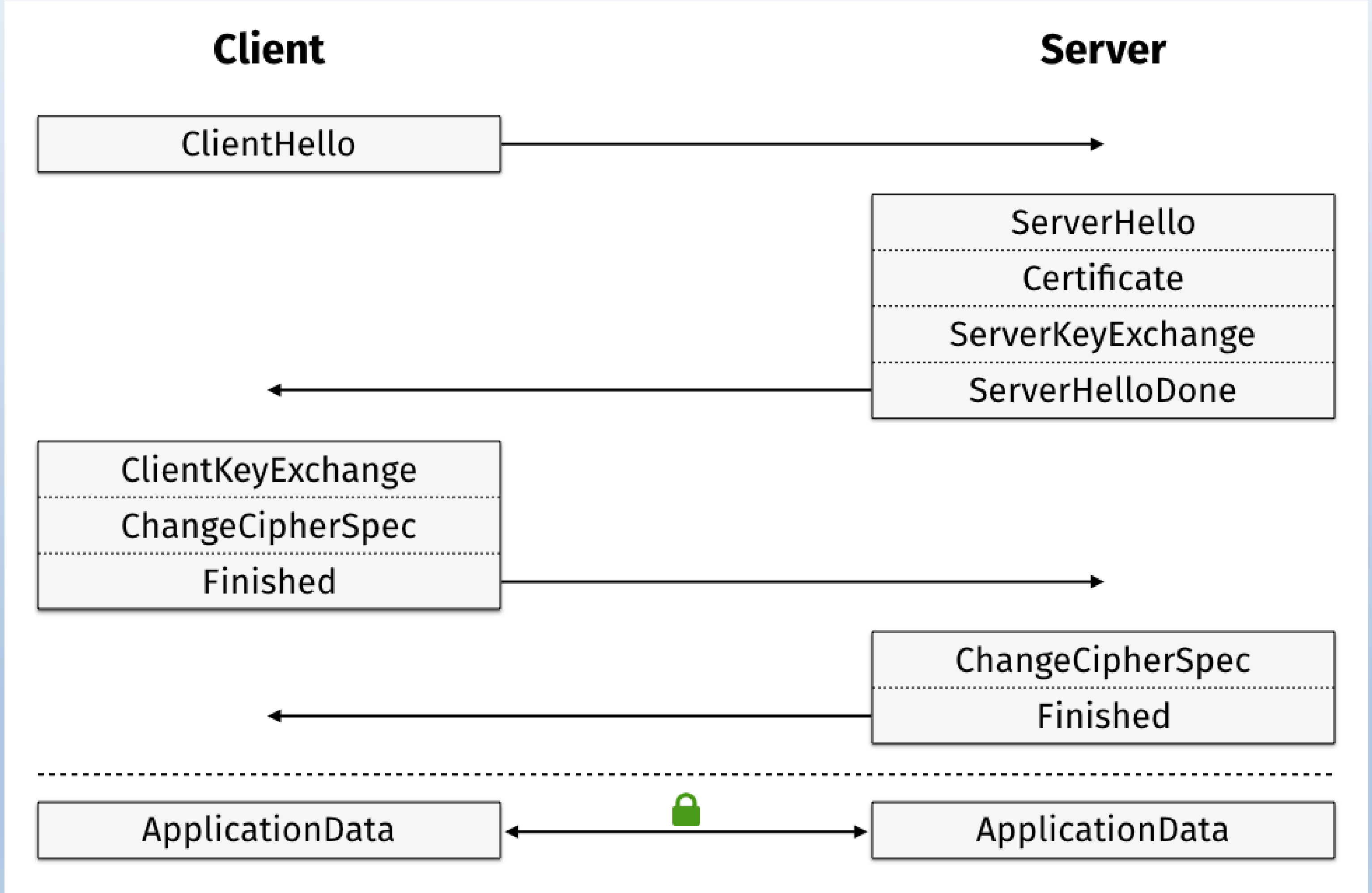
‖ denotes concatenation

⊕ denotes bitwise exclusive or (XOR)

opad is the block-sized outer padding, consisting of repeated bytes valued 0x5c

ipad is the block-sized inner padding, consisting of repeated bytes valued 0x36

TLS 1.2 Handshake with DH



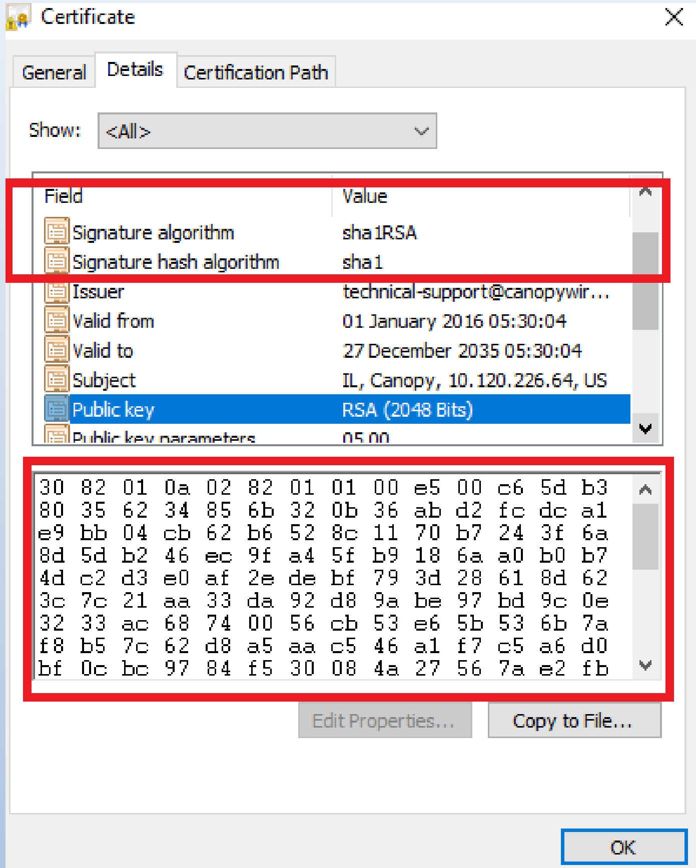
Generating Keys

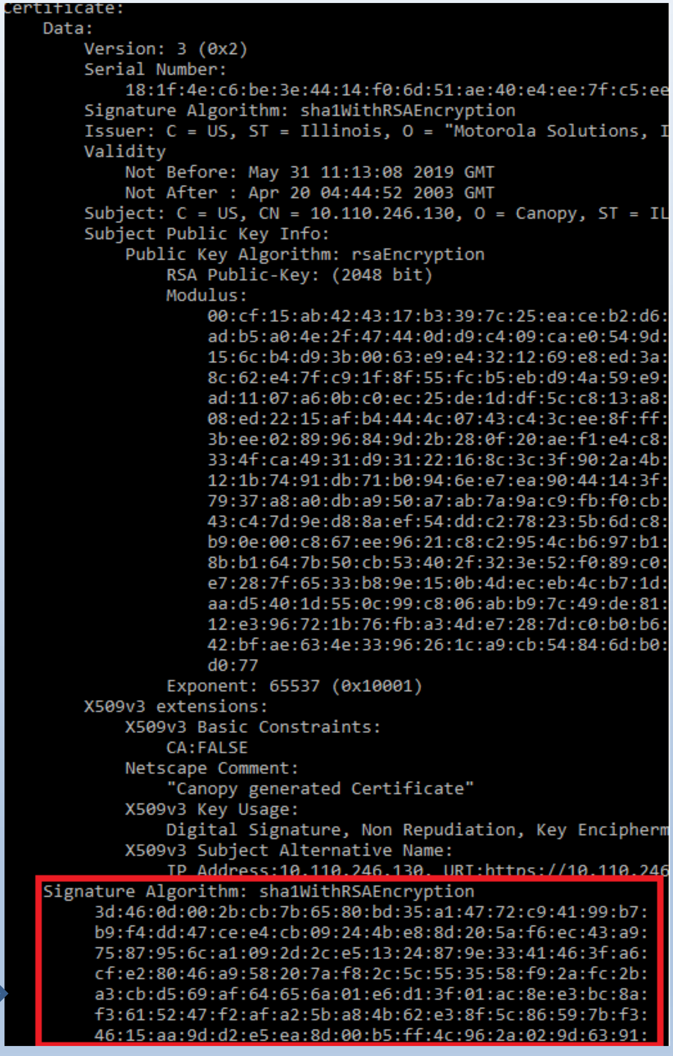
- Master_Secret = PRF(Client.Random+ Server.Random + Pre-Master-Secret + “master secret”)
- Key_Expansion = PRF(Client.Random+ Server.Random + Master-Secret + “key expansion”)

```
client_write_MAC_secret[] server_write_MAC_secret[]
client_write_key[] server_write_key[]
client_write_IV[] server_write_IV[]
```

Messages from **client to server** are **encrypted** with the **client write key**, and the **server** uses the **client write key** to **decrypt** them. Messages from server to client are encrypted with the server write key, and the client uses the server write key to decrypt them.

Certificate





Signed by **CA private key** after taking **SHA1** of Server certificate contents.

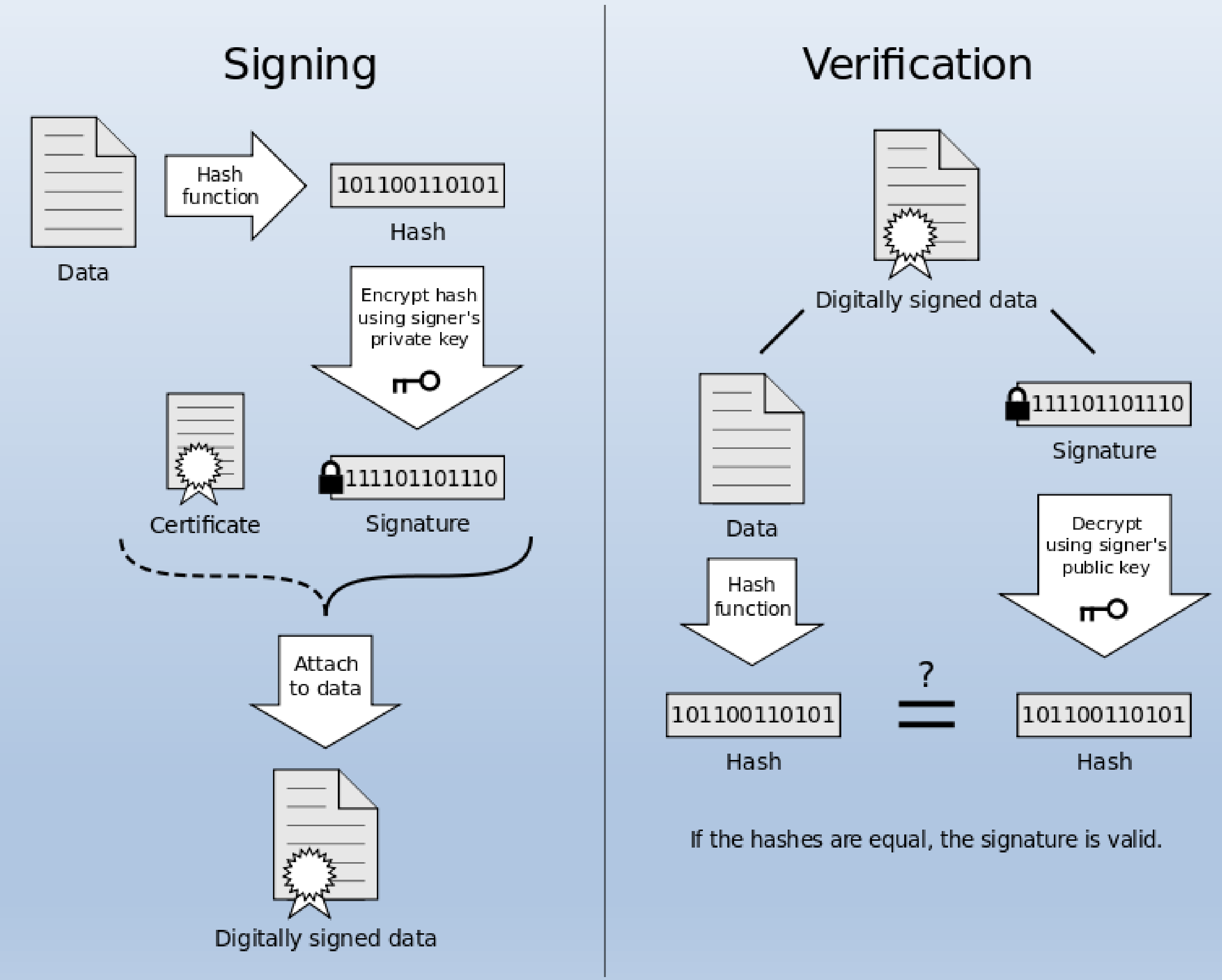
https://shattered.it/

Certificate Signing Request(CSR)

- Generate a key_pair(private, public)
- Fill up details like CN/C/ST/L/O/OU etc.
- Take a Hash of above data , Sign everything with your private key and append this data as “Signature Algorithm”. Signing the CSR proves ownership of the private key corresponding to the public key in the CSR. 1

```
csr100@IN01-760Y0G2:/mnt/c/Users/csr100/Downloads/Canopy_certs/2$openssl req -text -noout -verify -in myECC.csr
verify OK
Certificate Request:
    Data:
        Version: 1 (0x0)
        Subject: C = IN, ST = KA, L = BLR, O = Cambium Networks Inc, OU = Cambium BLR, CN = Chitrang Srivastava ECC secp384r1
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (384 bit)
                pub:
                    04:2c:a5:10:8f:b7:75:75:88:d4:fc:22:2b:41:ca:
                    7c:31:1c:82:39:7e:ef:e7:54:0b:75:0b:c1:7b:c7:
                    2a:5a:c9:d9:3e:ed:23:46:28:64:87:5d:69:0b:2b:
                    a4:1a:75:18:ac:d6:8d:43:44:b0:f8:31:3c:59:ac:
                    eb:4b:1c:23:e5:4f:be:bd:56:e3:a9:7a:05:a2:e3:
                    b3:66:a7:24:8b:3d:5d:b2:c3:40:01:9c:f7:54:00:
                    3d:96:1b:cd:9e:6b:fe
                ASN1 OID: secp384r1
                NIST CURVE: P-384
        Attributes:
            a0:00
    Signature Algorithm: ecdsa-with-SHA256
         30:65:02:31:00:db:80:3c:93:ec:56:d3:21:82:ed:4a:fd:f0:
         8d:41:78:eb:08:eb:22:c1:6a:e3:d8:f6:5a:e5:43:a4:b8:f6:
         6a:03:00:03:0c:ba:7d:bf:2d:44:58:03:9b:ce:70:c2:20:02:
         30:0f:9e:57:b2:db:fa:1c:aa:f2:a5:b2:fb:ac:58:7c:74:16:
         b8:99:45:49:81:3f:9b:81:f2:15:41:0f:b5:b9:69:5a:80:cf:
         a2:4c:48:06:34:6d:f6:c3:57:23:82:f6:df
```

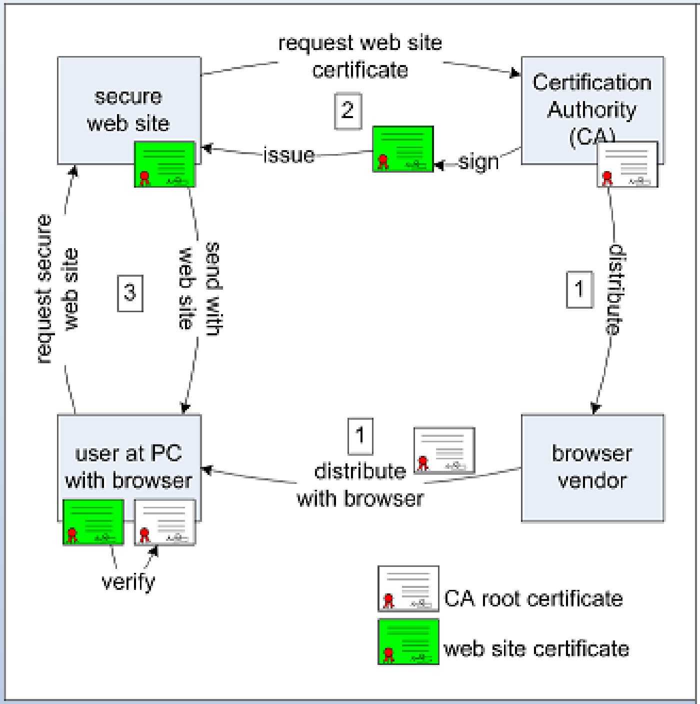
Signing

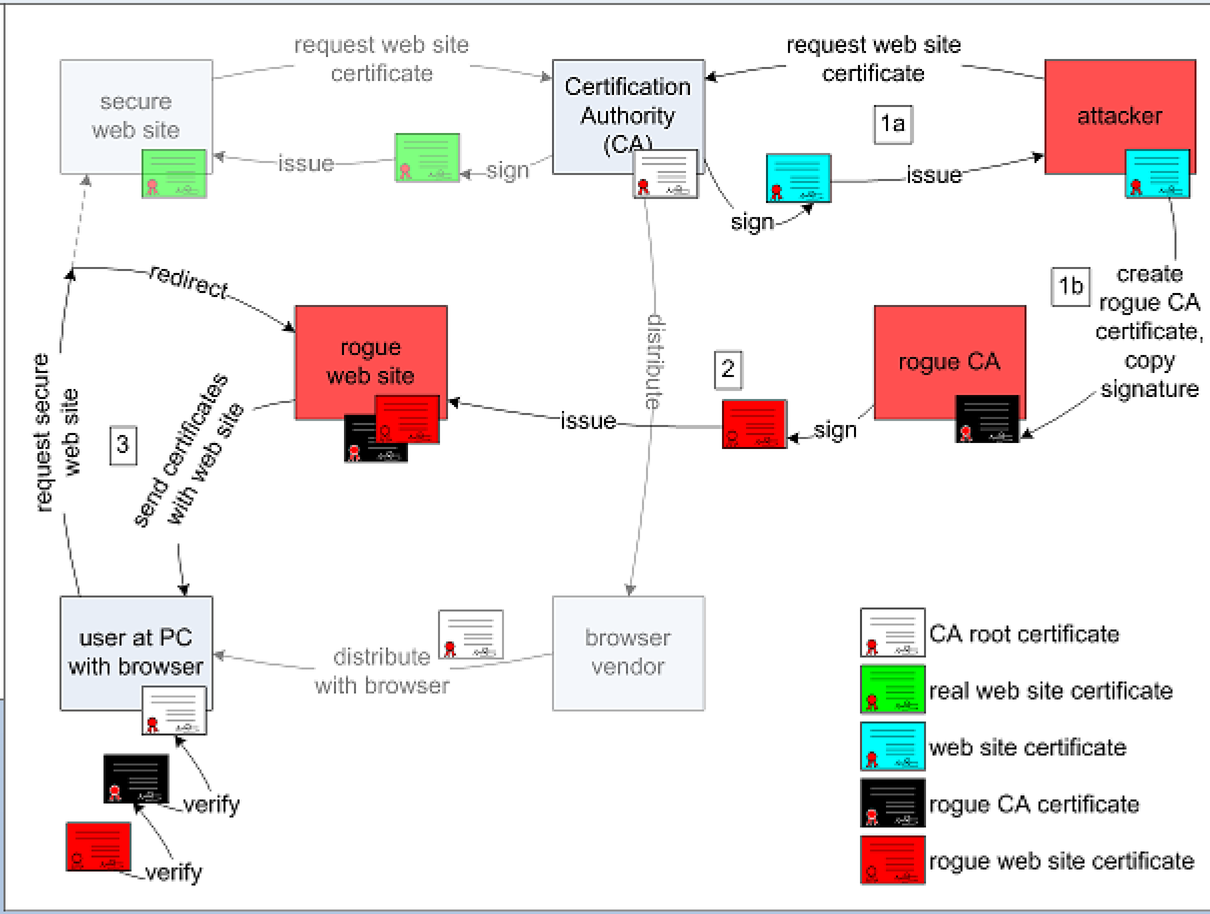

Signing
Data
Hash function
101100110101
Hash
Encrypt hash using signer's private key
Certificate
111101101110
Signature
Attach to data
Digitally signed data
Verification
Digitally signed data
Data
111101101110
Signature
Hash function
Decrypt using signer's public key
101100110101
?
=
101100110101
Hash
Hash
If the hashes are equal, the signature is valid.

```
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number: 1234605616436508555 (0x112233445566778b)
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = IN, ST = KA, O = Cambium Networks Inc, OU = Cambium BLR, CN = PMP 450 BLR
        Validity
            Not Before: Sep  3 10:32:43 2019 GMT
            Not After : Aug 31 10:32:43 2029 GMT
        Subject: C = IN, ST = KA, O = Cambium Networks Inc, OU = Cambium BLR, CN = Chitrang Srivastava ECC secp384r1
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (384 bit)
                pub:
                    04:2c:a5:10:8f:b7:75:75:88:d4:fc:22:2b:41:ca:
                    7c:31:1c:82:39:7e:ef:e7:54:0b:75:0b:c1:7b:c7:
                    2a:5a:c9:d9:3e:ed:23:46:28:64:87:5d:69:0b:2b:
                    a4:1a:75:18:ac:d6:8d:43:44:b0:f8:31:3c:59:ac:
                    eb:4b:1c:23:e5:4f:be:bd:56:e3:a9:7a:05:a2:e3:
                    b3:66:a7:24:8b:3d:5d:b2:c3:40:01:9c:f7:54:00:
                    3d:96:1b:cd:9e:6b:fe
                ASN1 OID: secp384r1
                NIST CURVE: P-384
        X509v3 extensions:
            X509v3 Basic Constraints:
                CA:FALSE
            Netscape Comment:
                OpenSSL Generated Certificate
            X509v3 Subject Key Identifier:
                A4:96:0A:E1:4E:30:AC:1D:37:14:EB:5D:C0:A8:44:6E:C3:63:5A:C5
            X509v3 Authority Key Identifier:
                keyid:EB:54:31:9C:00:9A:E3:48:BB:42:51:BC:14:38:3F:75:47:54:03:87

    Signature Algorithm: sha256WithRSAEncryption
         72:23:67:81:9b:96:35:12:97:f3:30:af:73:7e:99:7d:d4:ec:
         ad:fb:41:d7:60:68:9d:06:2b:8e:b5:c3:c4:d5:74:40:cb:f7:
         b8:78:0a:3c:cc:0f:ea:8c:54:2b:22:0c:36:72:a2:a5:16:25:
         f7:dc:d0:74:28:b9:05:50:57:70:3c:9a:80:30:be:32:79:2b:
         58:13:cc:f3:52:ed:d2:2a:be:3c:84:27:21:cf:5b:90:1e:c6:
         33:a1:54:11:3a:87:49:6e:94:b9:da:18:69:12:30:c9:df:bc:
         8a:1b:de:22:6d:72:08:9e:6d:39:9a:09:2c:27:35:1f:eb:c7:
         ee:f1:87:7b:ec:d4:59:3e:11:6f:04:1b:1f:e5:41:16:6a:cc:
         79:7a:bf:2a:6e:82:53:41:f6:72:ec:1e:c7:ac:08:ce:14:0b:
         21:c4:17:0a:00:89:cb:df:7d:44:42:aa:bf:d7:9d:e3:3d:a3:
         87:3e:78:2c:e6:7a:f5:f3:b2:f4:fd:2c:a3:d5:39:83:5a:50:
```

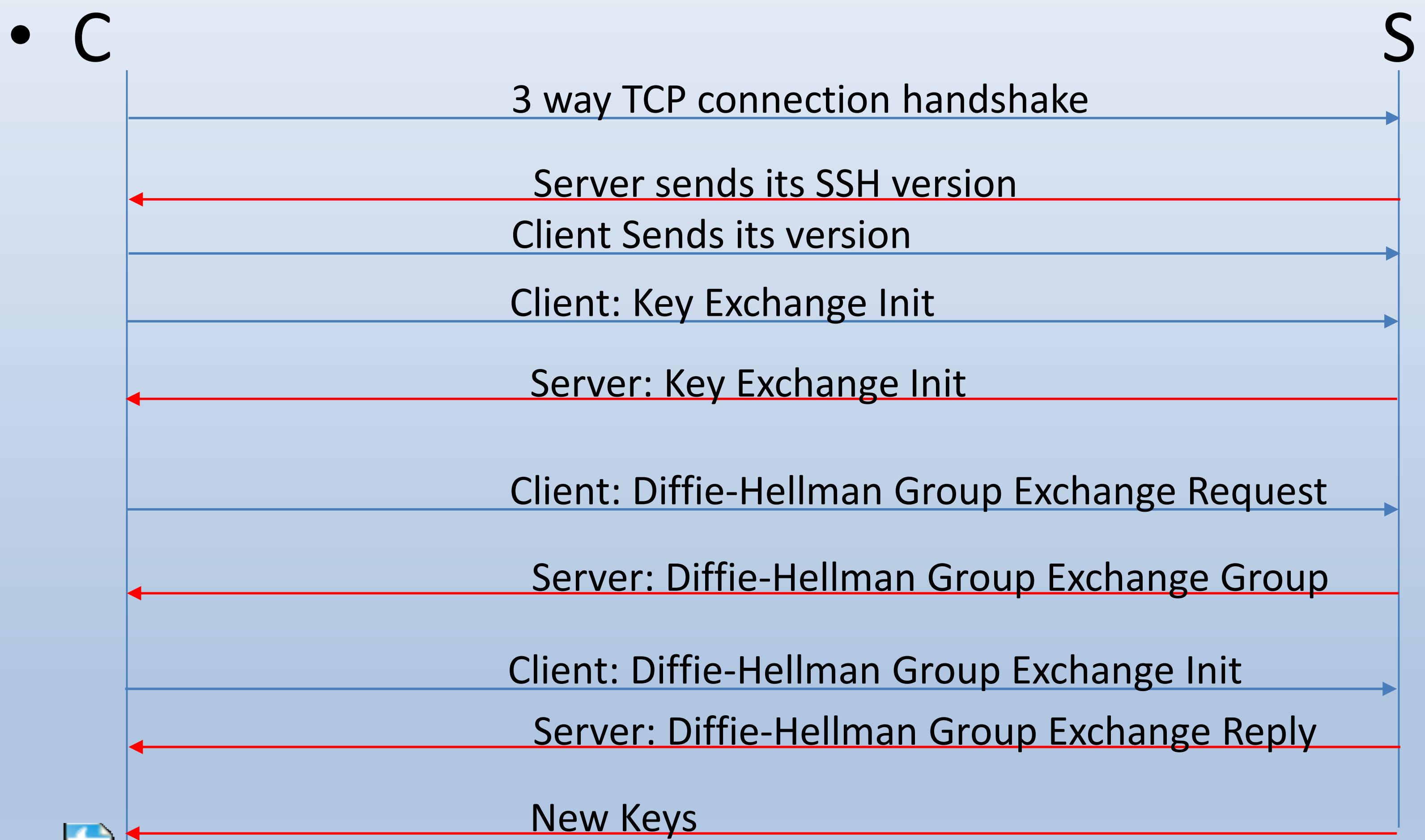
SHA-1 Collision Attack





https://www.win.tue.nl/hashclash/rogue-ca/

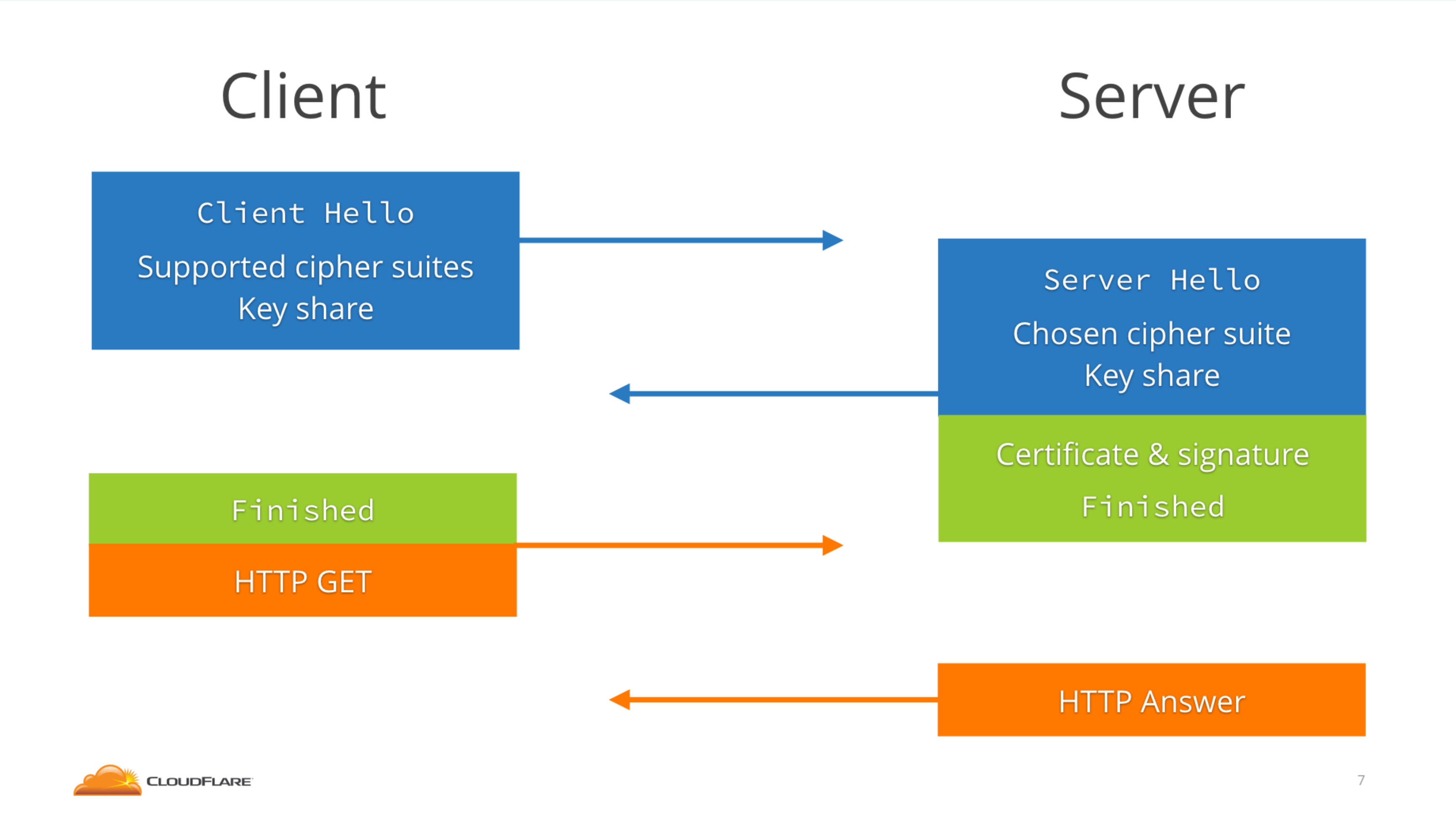
How SSH works



ssh-server-client.pcapng

[RFC 4253]

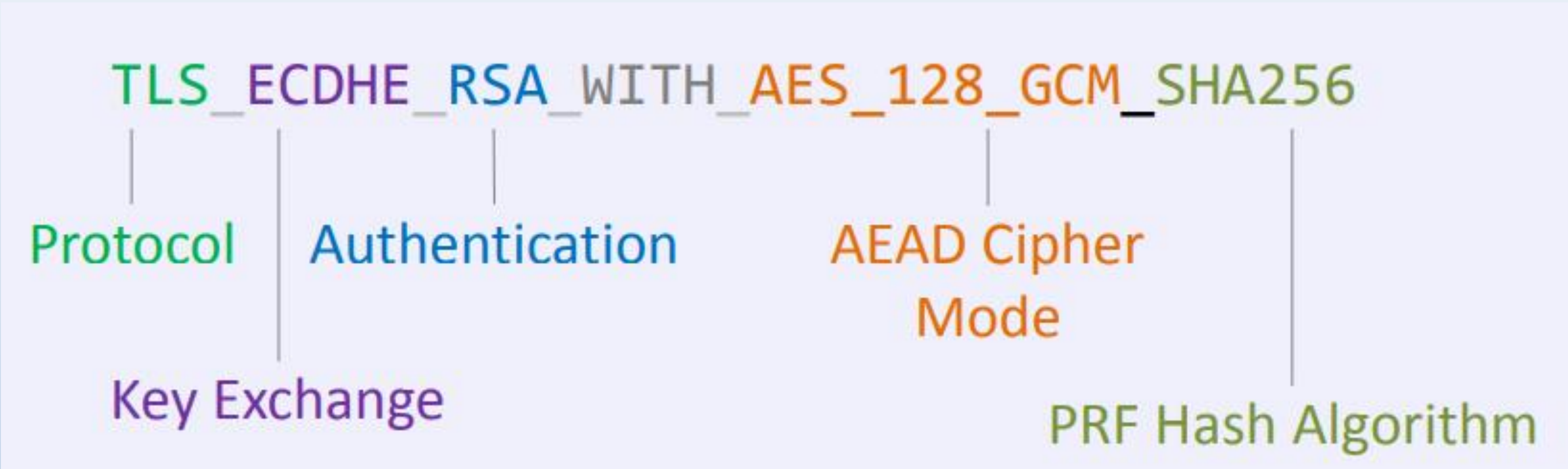
TLS 1.3 Handshake



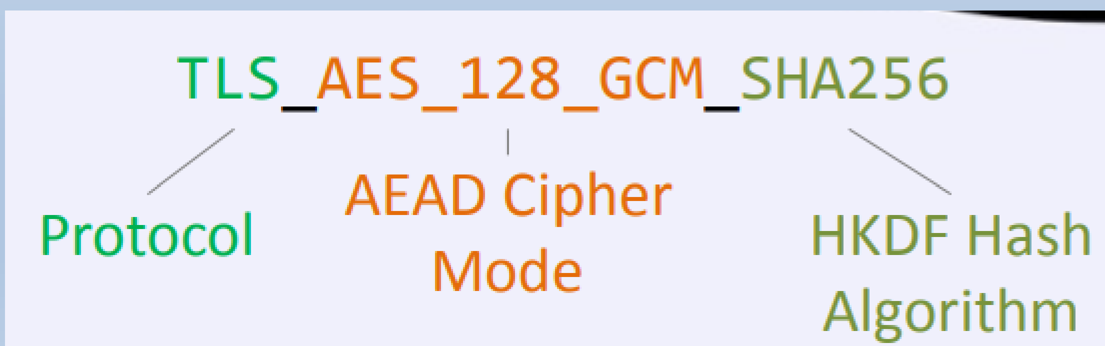
Major Difference between TLS 1.2 & TLS 1.3

• Simple cipher suites (just 5 of them) specifying encryption algorithm & HKDF.
• Perfect Forward Secrecy is mandatory.
• Certificate is sent encrypted in **ServerHello** as opposed to plain text in TLS 1.2
• 1-RTT, 0-RTT as opposed to 2-RTT or more in TLS 1.2
• PRF is replaced by HKDF.
• Only support AEAD cipher suites. AES in GCM mode and Chach20-Poly1305. Encrypt-then-MAC, Mac-then-Encrypt is all phased out.
• Compression, Renegotiation is removed

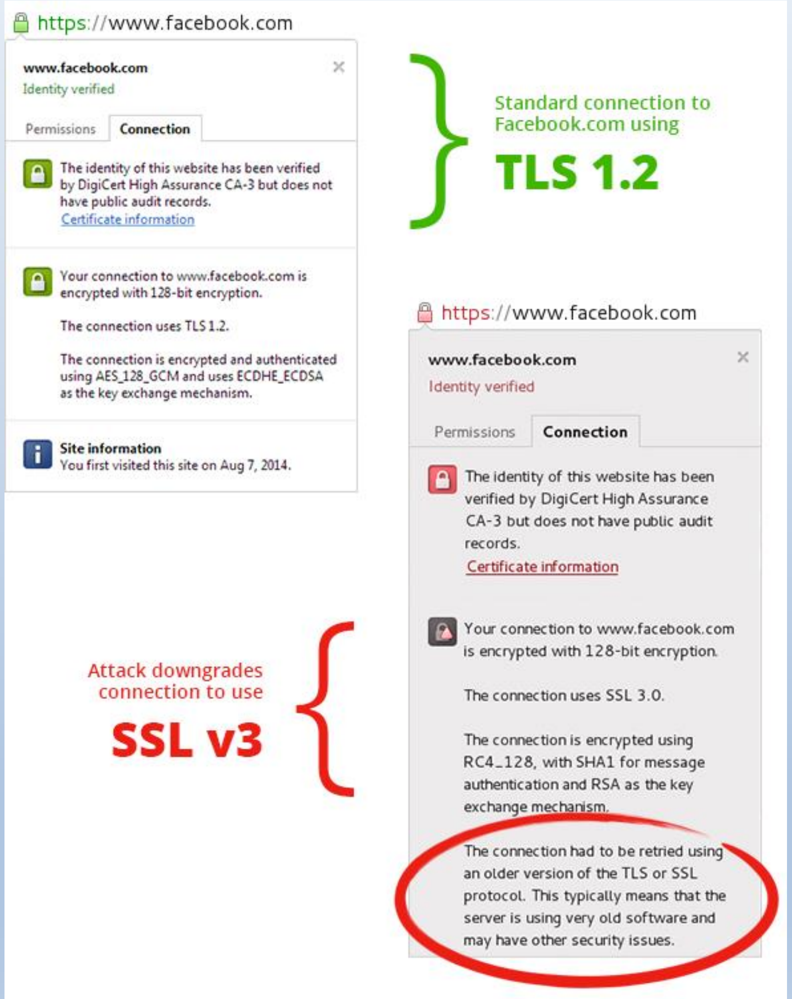
CipherSuites Difference



TLS 1.3



Downgrade Attack



Downgrade Attack

- ClientHello(CH) is dropped and then client send CH with lower SSL version of Cipher Suites.
- Sever Replies TLS 1.3 ServerRandom last 8 bytes has DOWNGRD01
- ServerHello is compromised ;SCSV(Signalling Cipher Supported Version)

its presence in the Client Hello message serves as

a backwards-compatible signal from the client to the server.

For backard compatibility, ClienHello version remain 1.2 instead a new “supported_version” extension is added which list 1.3 and hence TLS 1.3 server knows that client wants 1.3, TLS 1.2 will simply ignore it and do TLS 1.2 More use cases and here

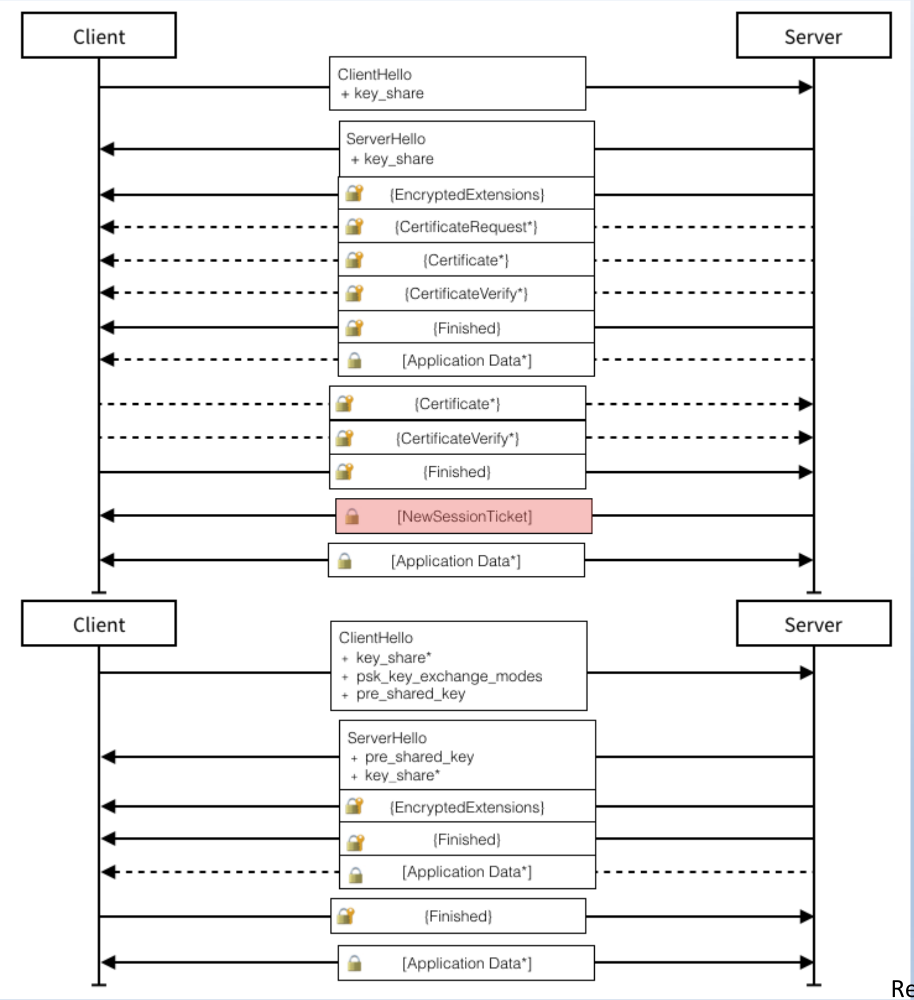
HKDF

- Client_public_key + server_private_key +SHA(ClientHello+ServerHello) is fed to HKDF
- Extract & Expand

0-RTT

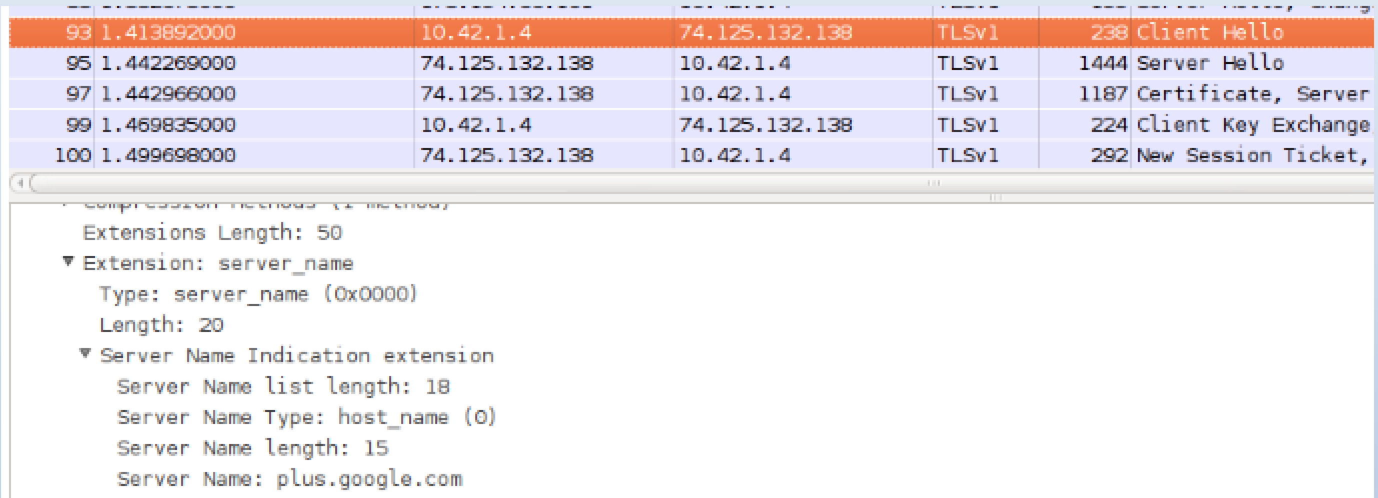
- Opens up risk fore replay attack.
- During 1st session establishment, server give client 'Session Ticket' which client uses in subsequent connection.
- The client also sends a key share, so that client and server can switch to a new fresh key for the actual HTTP response and the rest of the connection.

TLS 1.3 Session Resumption



Refrence:
https://www.davidwong.fr/tls13/

TLS Proxy

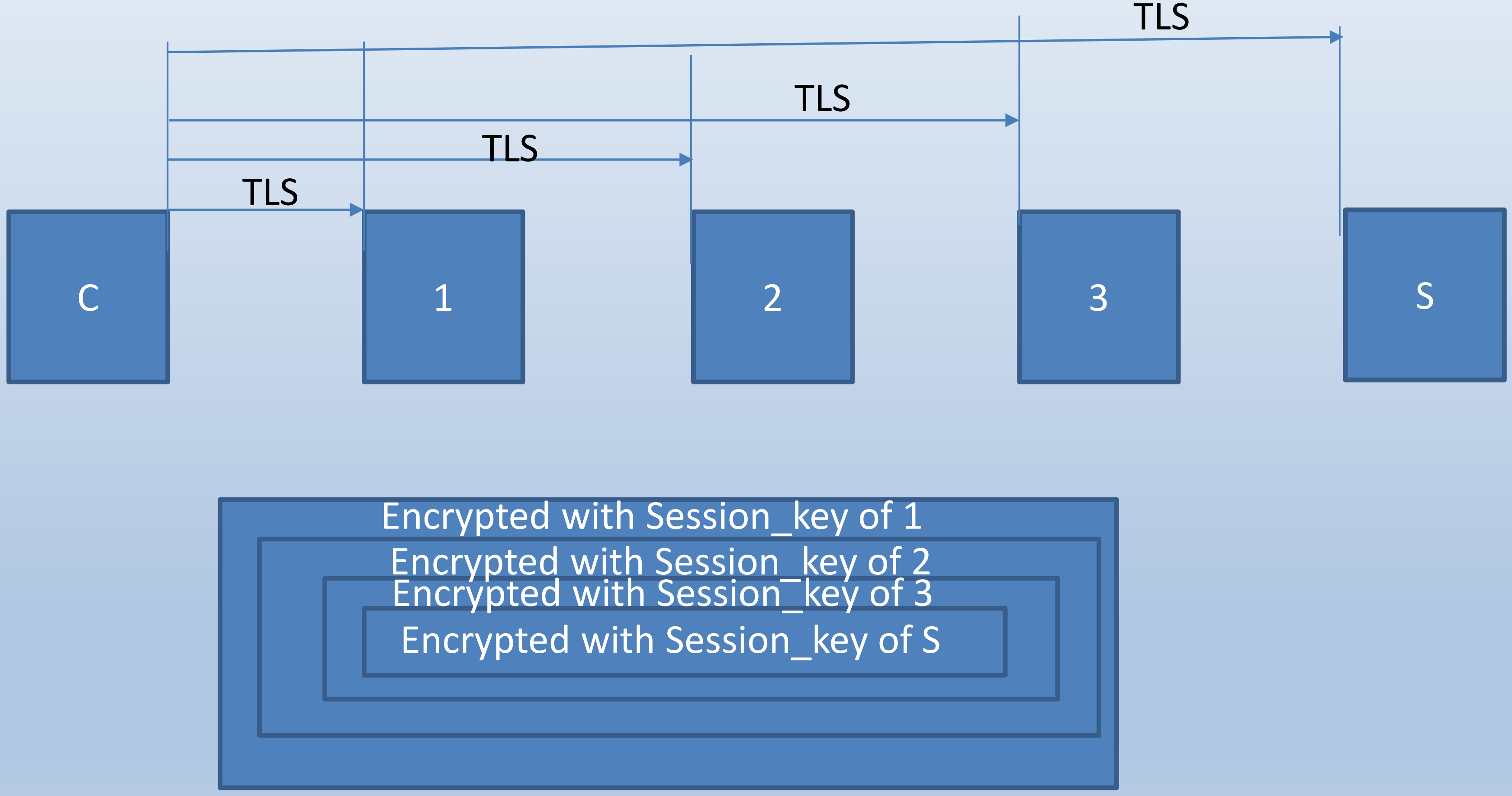


- https://docs.mitmproxy.org/stable/concepts-howmitmproxyworks/
- https://arxiv.org/pdf/1407.7146.pdf

TLS Proxy

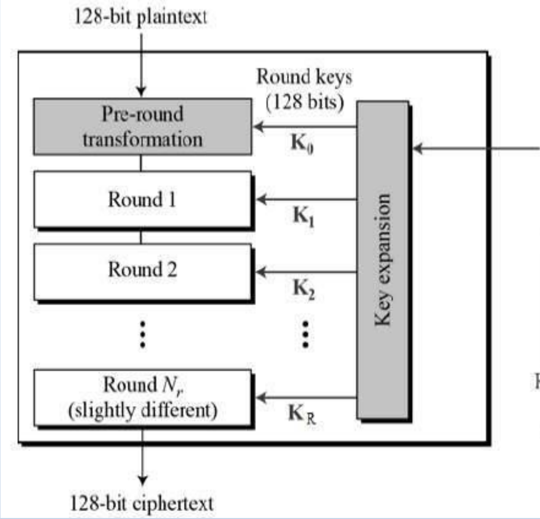
- Each & every connection have to be monitored while in TLS 1.2 certificate message is in plaintext and monitoring can be selective.
- Static DH /RSA is not allowed.

TOR



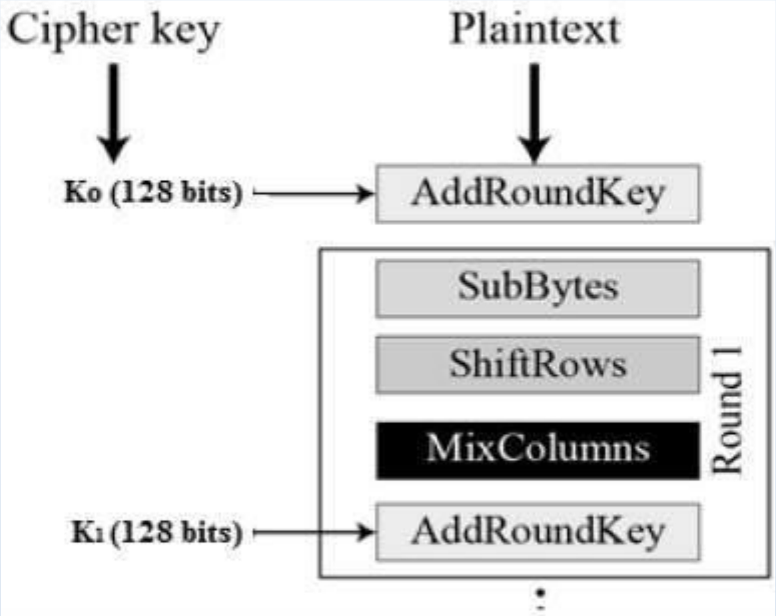
TLS
TLS
TLS
TLS
C
1
2
3
S
Encrypted with Session_key of 1
Encrypted with Session_key of 2
Encrypted with Session_key of 3
Encrypted with Session_key of S

AES(Advance Encryption Standard)

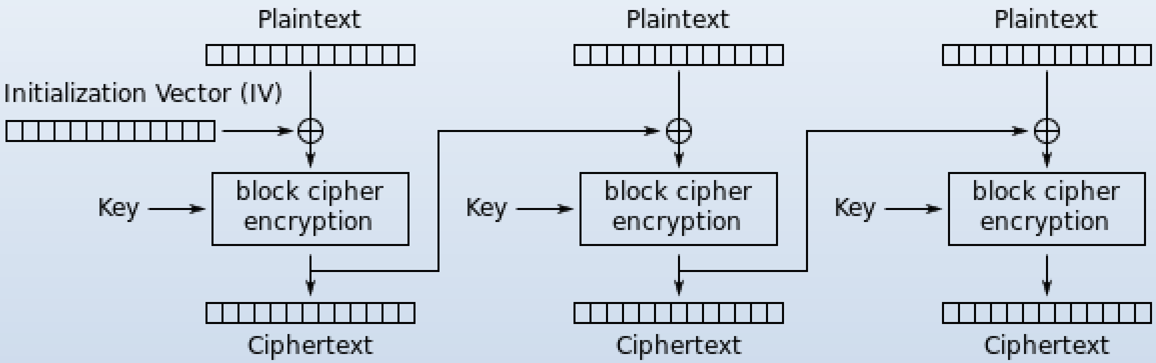


R	Key size
10	128
12	192
14	256

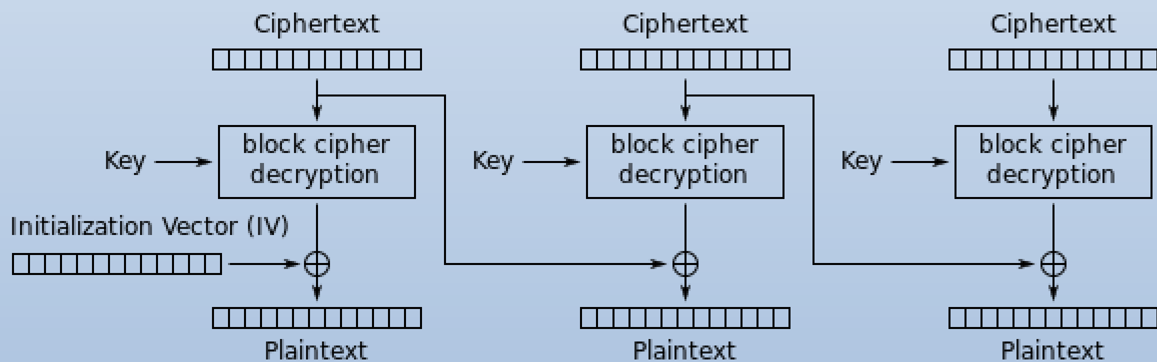
Relationship between number of rounds(R) and cipher key size



AES(Advance Encryption Standard)



Cipher Block Chaining (CBC) mode encryption



Cipher Block Chaining (CBC) mode decryption

AEAD(Authenticated Encryption Additional Data)