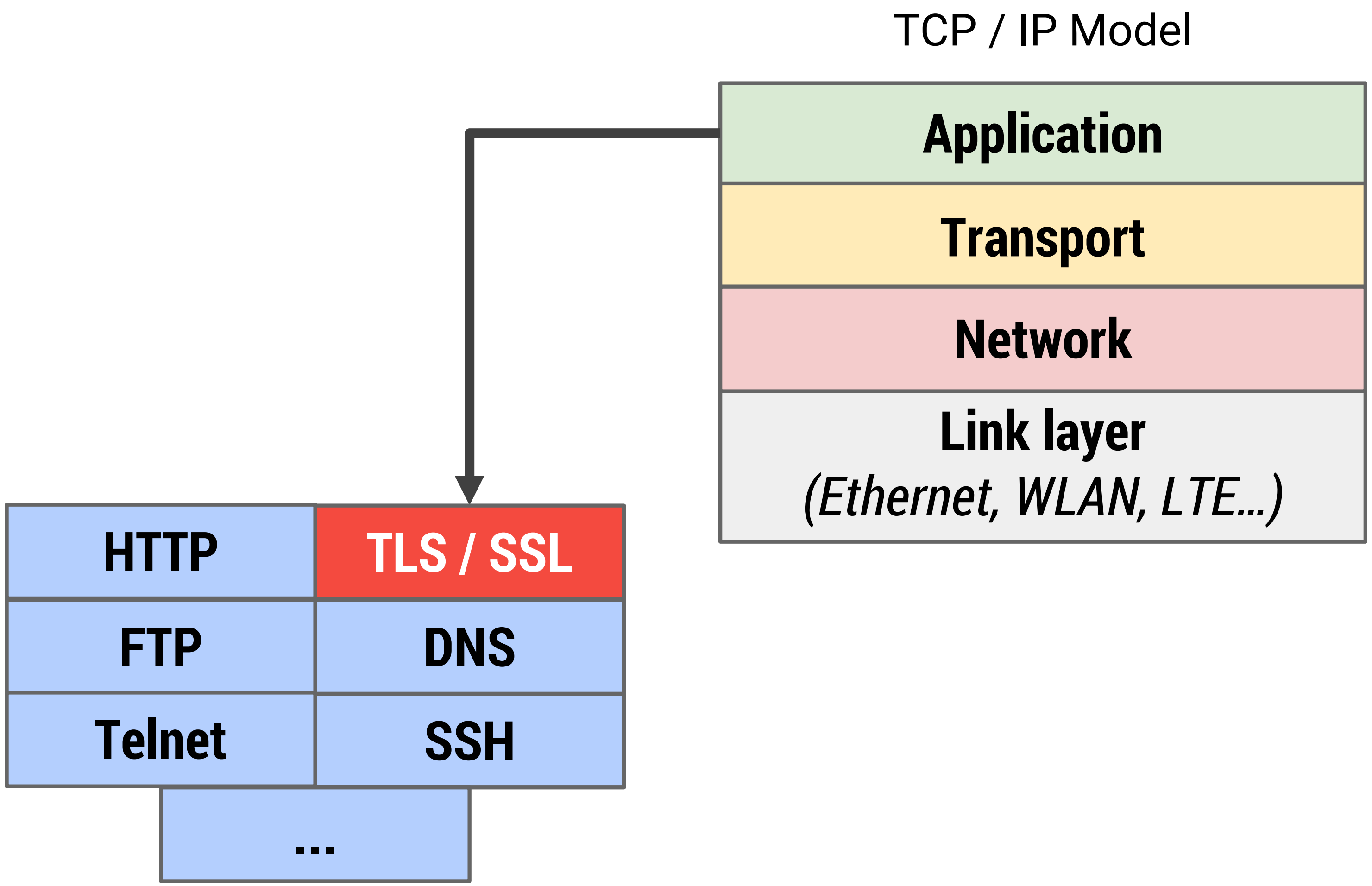
Application Layer – TLS / SSL

Information Security 2019

Johannes Feichtner
johannes.feichtner@iaik.tugraz.at

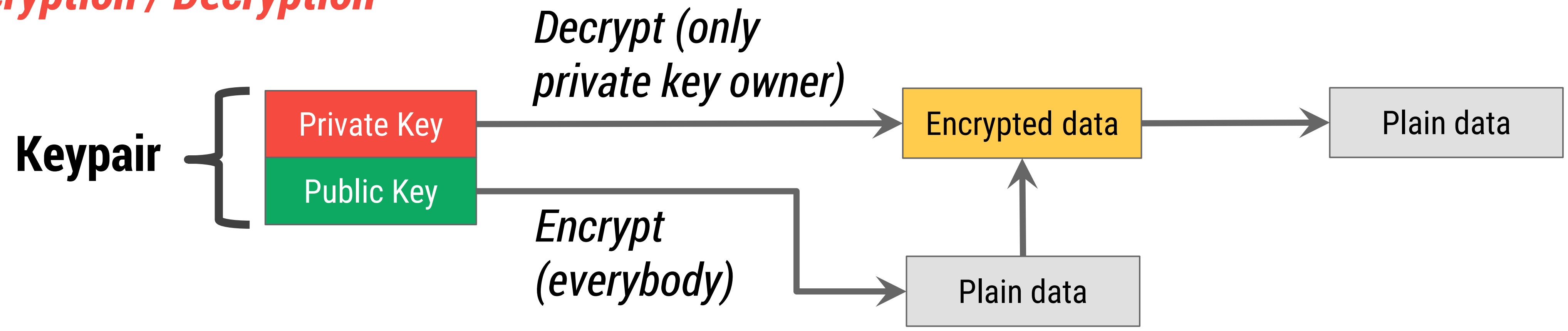
Outline

- Crypto Crash Course
- TLS Handshake
- Properties
  - Cipher Suites
  - Perfect Forward Secrecy
- Security
  - HSTS
  - Certificate Pinning (HPKP)



Crypto Crash Course

Asymmetric Cryptography: Encryption / Decryption



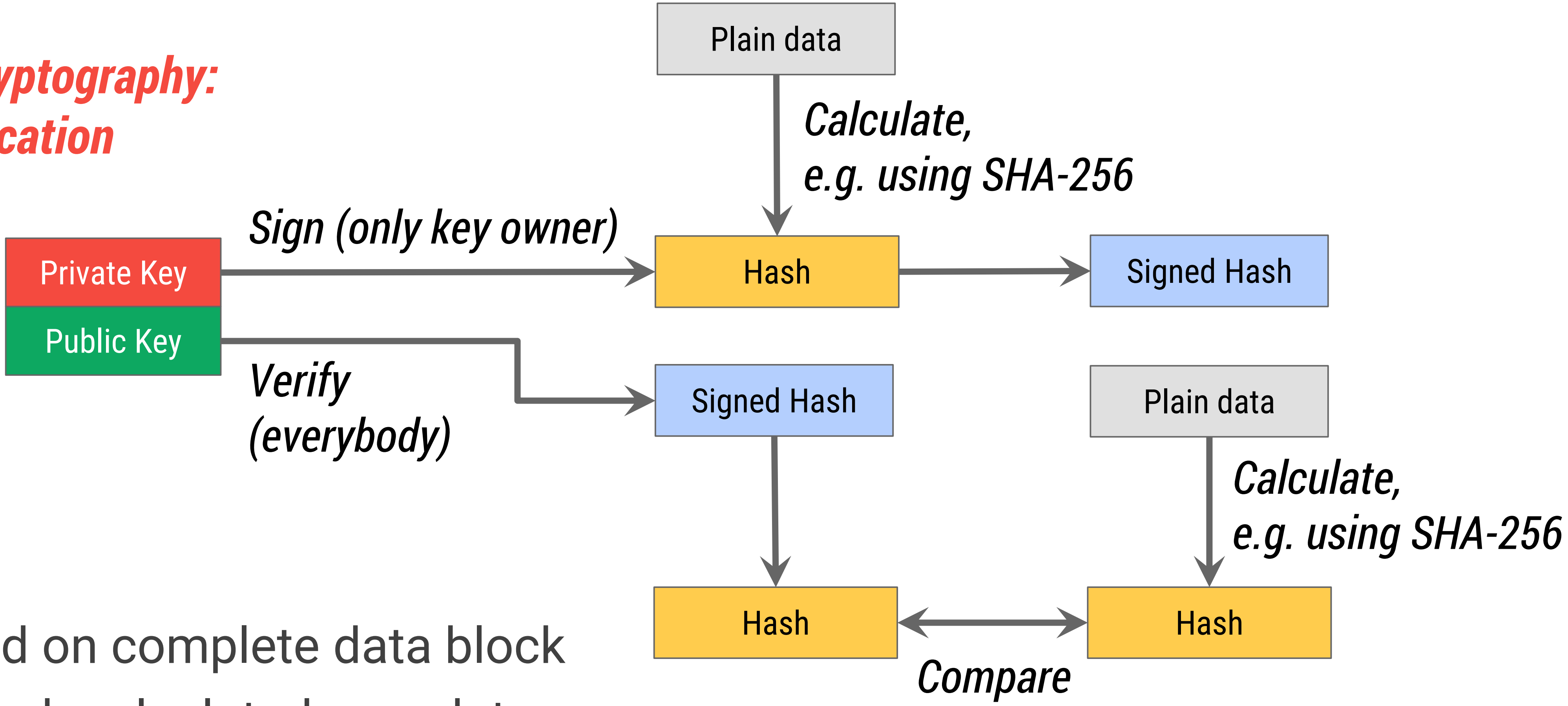
Private Key = Really private, only the owner should have it

Public Key = Everyone can have it

- Typically only small data is encrypted with asymmetric keys (performance!)
- Asymmetric schemes often encrypt („wrap“) symmetric keys

Crypto Crash Course

Asymmetric Cryptography: Signing / Verification



Signature

- Not applied on complete data block
- Instead, hash calculated over data
  → signed / verified

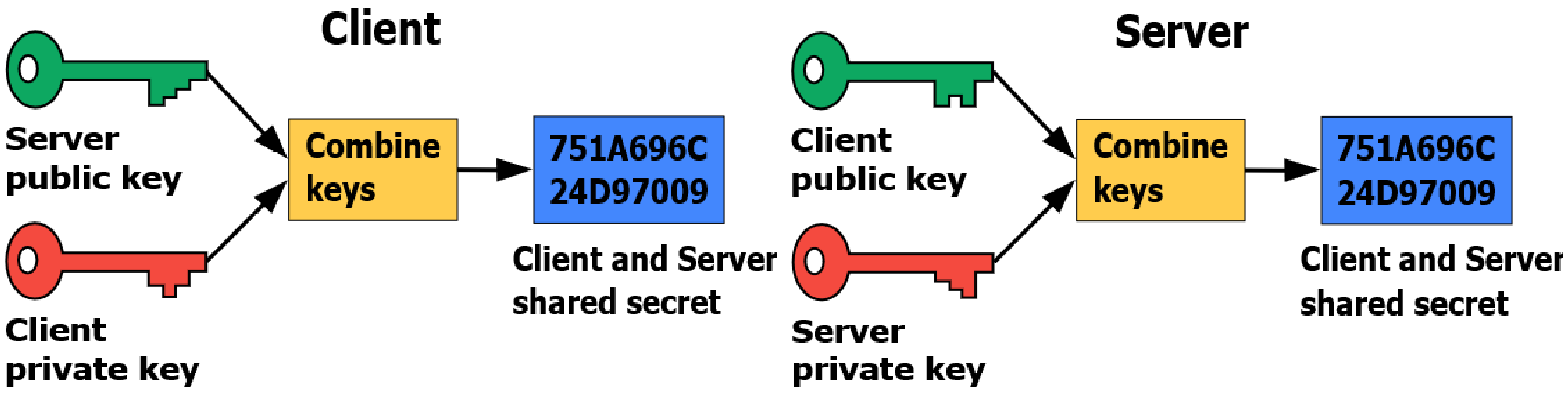
Verification: Comparison if hashes match

Diffie-Hellman (DH)

Basic idea

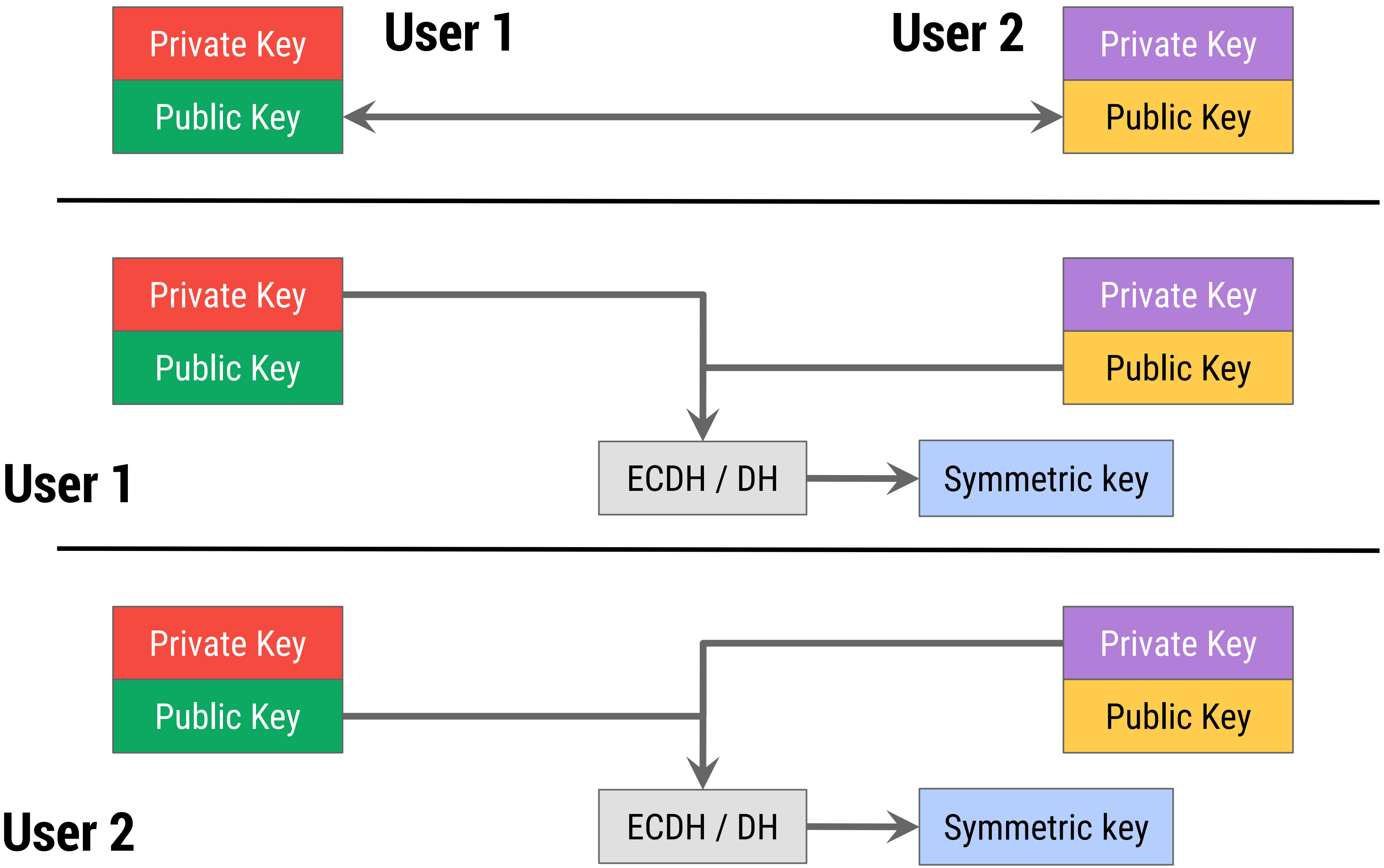
- Server determines DH parameters + generates key pair
- Sends parameters + public key to client

- Client uses DH parameters (of server) + generates key pair
- Client sends public key to server
- Both calculate same secret



Crypto Crash Course

Asymmetric Cryptography: Key Agreement



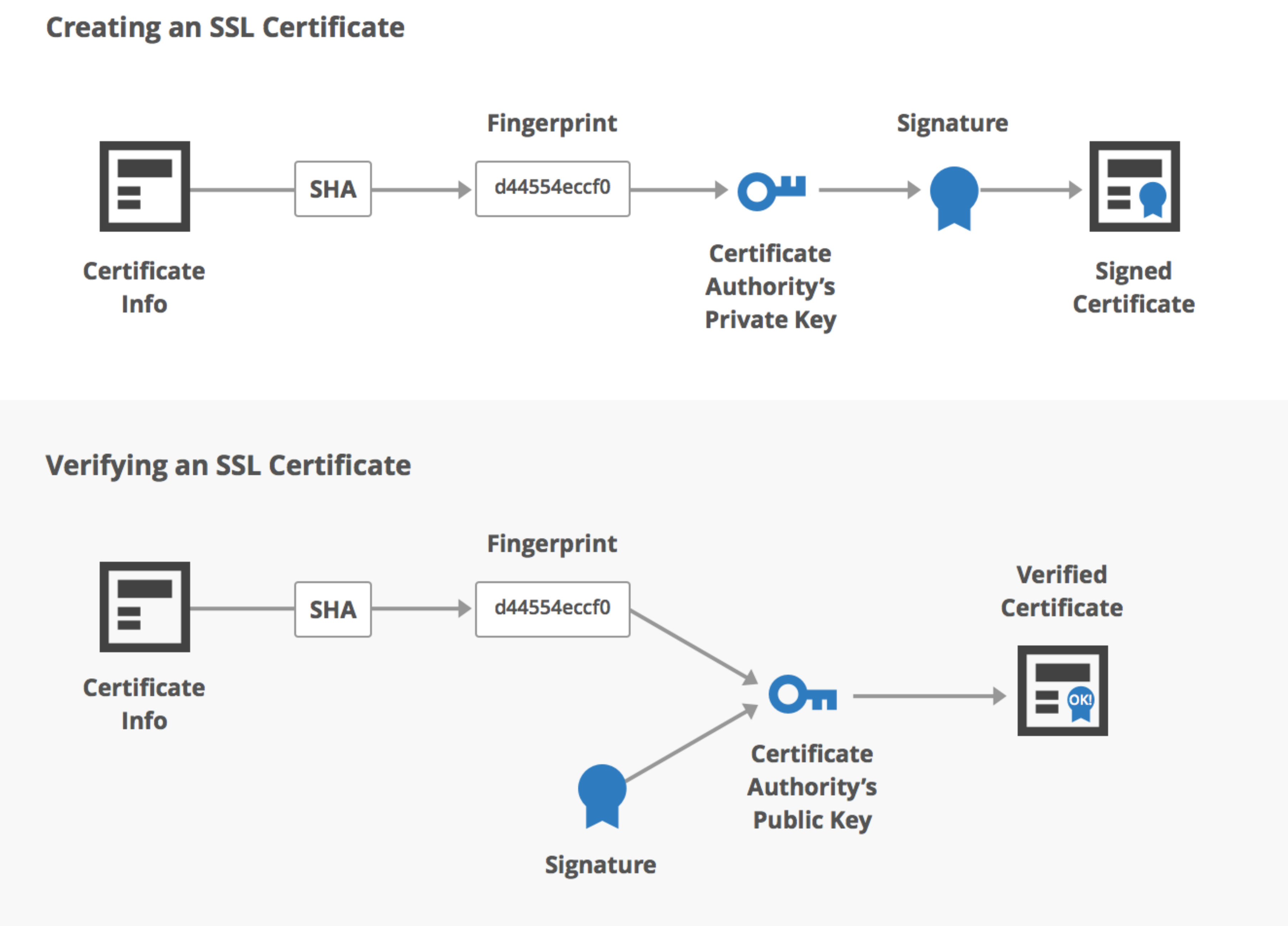
Exchange public keys

Agree on symmetric key with user 1 private key and user 2 public key

Agree on symmetric key with user 2 private key and user 1 public key

IAIK TU Graz

X.509 Certificates



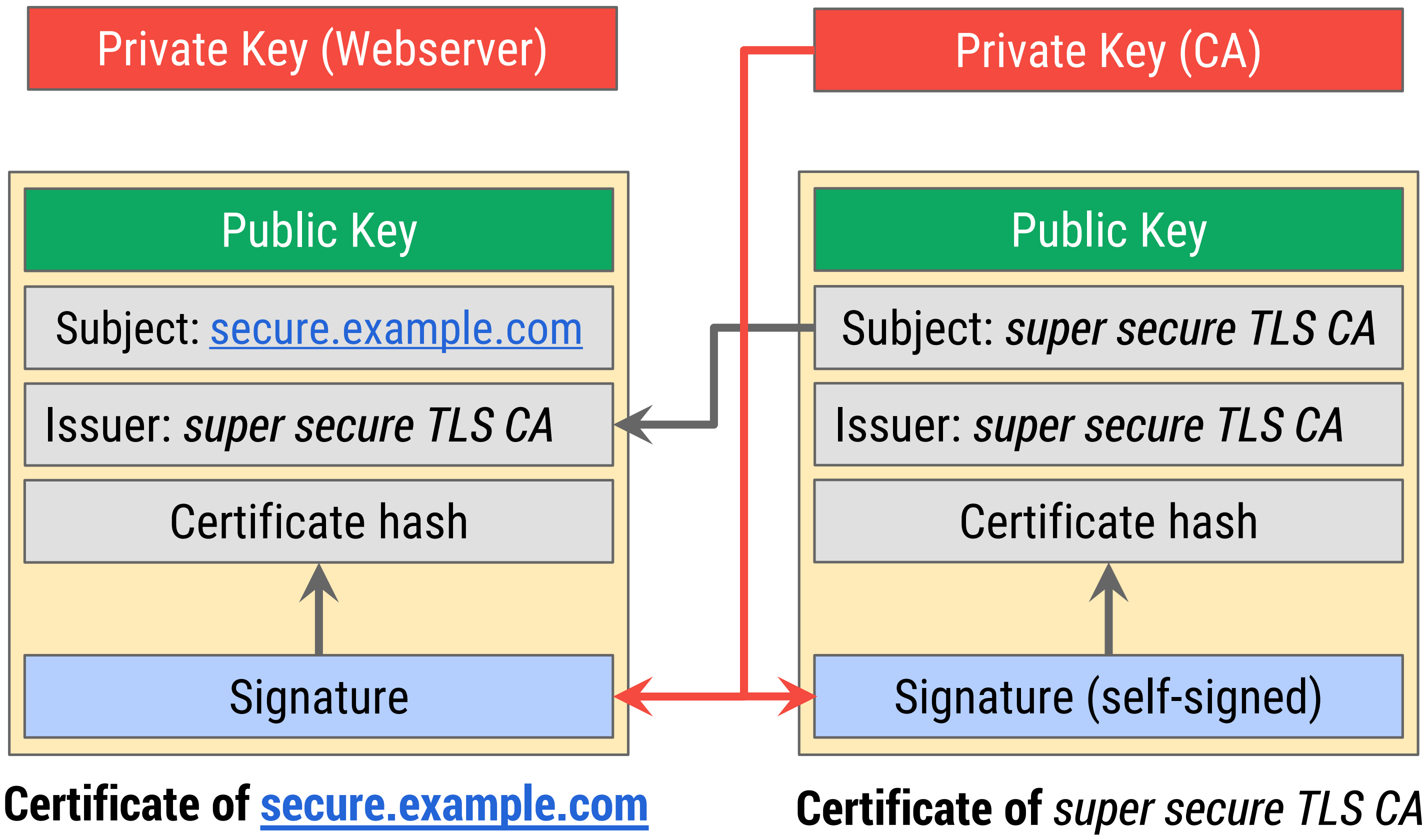
Source: https://goo.gl/egFCjg

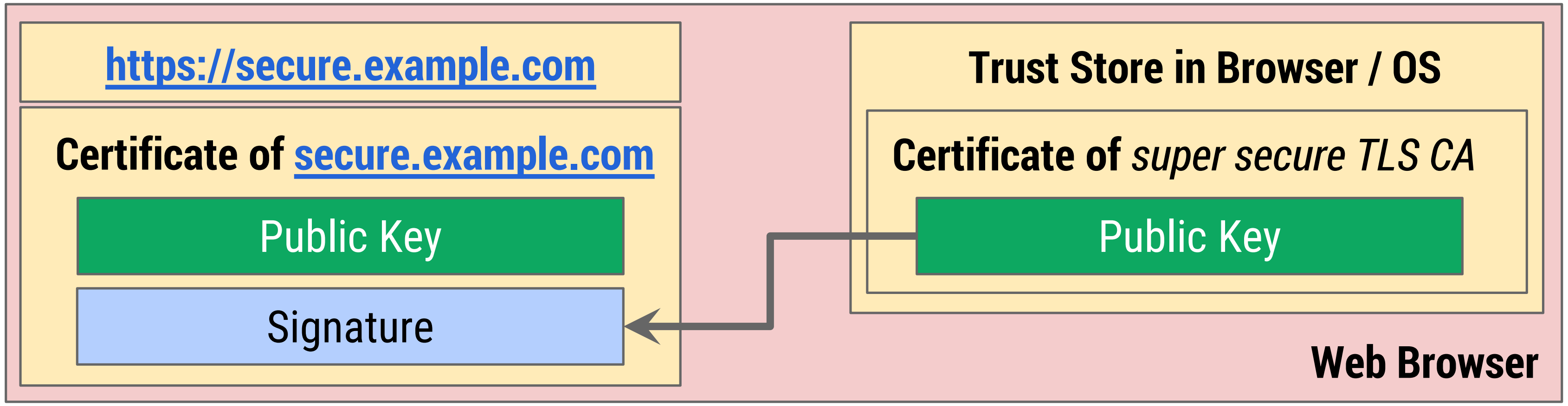
X.509 Certificates

Validation

Web Browser gets host cert during TLS handshake

1. Verify hostname matches certificate subject
2. Verify signature





IAIK TU Graz

Transport Layer Security

TLS Introduction

Basics

- Key protocol for secure communication
  - HTTPS, VPNs, for any secure communication based on certificates

- Designed to operate on TCP (for reliability reasons)
  - Later adapted to support datagram protocols also, e.g. UDP
    → Datagram Transport Layer Security (DTLS), RFC 6347

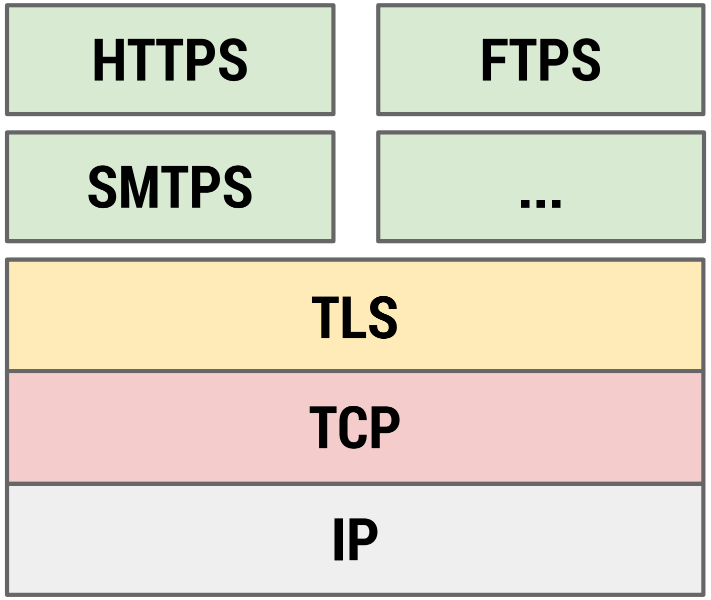
- Initial development by Netscape in the 90s
  - Named „Secure Sockets Layer“ (SSL)
  - Later standardized by IETF → renamed to TLS

TLS Versions

- 1995: First public release of proprietary SSL 2.0
  - Critical security flaws briefly afterwards
  - Usage prohibited in 2011 (RFC 6176)
- 1996: SSL 3.0, RFC 6101, deprecated in June 2015 (RFC 7568)
- 1999: TLS 1.0, RFC 2246
  - No „dramatic changes“ but no more interoperability between SSL 3.0 & TLS 1.0
  - Includes downgrade option to SSL 3.0 → weakens security!
- 2006: TLS 1.1, RFC 4346
- 2008: TLS 1.2, RFC 5246: Removed old ciphers, bugfixes
- 2018: TLS 1.3, RFC 8446 (Proposed Standard): Drop weak ciphers

TLS Services

All applications running TLS are provided with three essential services



Authentication
Verify identity of client and server

Data Integrity
Detect message tampering and forgery, e.g. malicious Man-in-the-middle

Encryption
Ensure privacy of exchanged communication

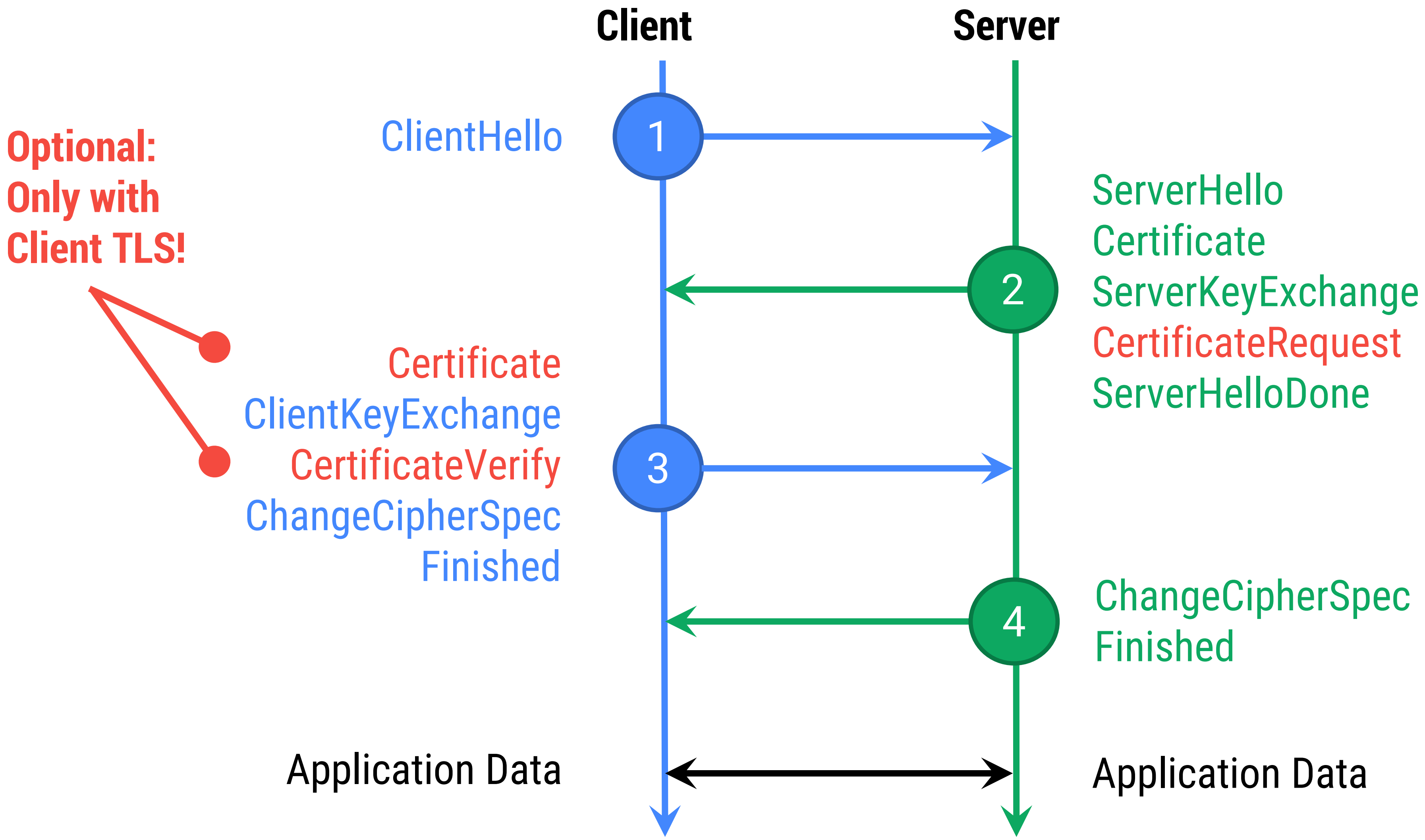
Note: Technically, not all services are required to be used
→ Can raise risk for security issues!

TLS 1.2 Handshake

RFC 5246

= Establish parameters for cryptographically secure data channel

Full handshake scenario!



Client: ClientHello

With TCP connection setup on port 443, clients initiate the TLS negotiation

Message contains

- Highest supported TLS version
- Random number (for key exchange)
- Session ID
  - If existing session should be resumed
  - Kind of „keep-alive“ across requests
- Suggested cipher suites
- Supported compression methods
- Extensions

```
◢ Secure Sockets Layer
  ◢ TLSv1.2 Record Layer: Handshake Protocol: Client Hello
      Content Type: Handshake (22)
      Version: TLS 1.0 (0x0301)
      Length: 189
    ◢ Handshake Protocol: Client Hello
        Handshake Type: Client Hello (1)
        Length: 185
        Version: TLS 1.2 (0x0303)
      ◢ Random
          GMT Unix Time: Jul 26, 1992 07:13:56.000000000 Mitteleurop�ische
          Random Bytes: 6f2575d1f037b52c7651ee3b59cf418baf22c251f88b18bb...
        Session ID Length: 0
        Cipher Suites Length: 22
      ◢ Cipher Suites (11 suites)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_GCM_SHA256 (0xc02b)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA (0xc00a)
          Cipher Suite: TLS_ECDHE_ECDSA_WITH_AES_128_CBC_SHA (0xc009)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_CBC_SHA (0xc013)
          Cipher Suite: TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA (0xc014)
          Cipher Suite: TLS_DHE_RSA_WITH_AES_128_CBC_SHA (0x0033)
          Cipher Suite: TLS_DHE_RSA_WITH_AES_256_CBC_SHA (0x0039)
          Cipher Suite: TLS_RSA_WITH_AES_128_CBC_SHA (0x002f)
          Cipher Suite: TLS_RSA_WITH_AES_256_CBC_SHA (0x0035)
          Cipher Suite: TLS_RSA_WITH_3DES_EDE_CBC_SHA (0x000a)
        Compression Methods Length: 1
      ▷ Compression Methods (1 method)
        Extensions Length: 122
      ▷ Extension: server_name
      ▷ Extension: Extended Master Secret
      ▷ Extension: renegotiation_info
      ▷ Extension: elliptic_curves
      ▷ Extension: ec_point_formats
      ▷ Extension: SessionTicket TLS
      ▷ Extension: next_protocol_negotiation
      ▷ Extension: Application Layer Protocol Negotiation
      ▷ Extension: status_request
      ▷ Extension: signature_algorithms
```

Server: ServerHello

Response to ClientHello if server finds common set of algorithms

Message contains

- Chosen TLS version
- Random number (for key exchange)
- Session ID
  - If supported / enabled by server
- Chosen cipher suite
  - No list, only the selected one
- Chosen compression method
- Common extensions

```
◢ Secure Sockets Layer
  ◢ TLSv1.2 Record Layer: Handshake Protocol: Server Hello
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 80
    ◢ Handshake Protocol: Server Hello
        Handshake Type: Server Hello (2)
        Length: 76
        Version: TLS 1.2 (0x0303)
      ◢ Random
          GMT Unix Time: Aug  9, 1975 01:08:47.000000000 Mitteleurop�ische
          Random Bytes: 42daa757f0afd1e705b3582f064c771b86257810a8018290...
        Session ID Length: 0
        Cipher Suite: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256 (0xc02f)
        Compression Method: null (0)
        Extensions Length: 36
      ▷ Extension: server_name
      ▷ Extension: renegotiation_info
      ▷ Extension: ec_point_formats
      ▷ Extension: SessionTicket TLS
      ▷ Extension: Application Layer Protocol Negotiation
```

If no match on TLS version and cipher suite
→ Handshake abort with error, e.g.
Firefox: „SSL_ERROR_NO_CYPHER_OVERLAP "
Chrome: „ERR_SSL_VERSION_OR_CIPHER_MISMATCH"

Server: Certificate

Server sends X.509v3 certificate chain

- Server‘s certificate has to be the first certificate
- Each following (intermediate) certificate must certify the preceding one
- Root certificates can be excluded
  - Browsers need to know them anyway

```
◢ Secure Sockets Layer
  ◢ TLSv1.2 Record Layer: Handshake Protocol: Certificate
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 3203
    ◢ Handshake Protocol: Certificate
        Handshake Type: Certificate (11)
        Length: 3199
        Certificates Length: 3196
      ◢ Certificates (3196 bytes)
          Certificate Length: 1938
        ◢ Certificate: 3082078e30820676a00302010202100b335018920af117cd... (id-at-commonName=online.tugraz.at,id-at-organizationalUnitName=Zentraler Informatikdienst
          ▷ signedCertificate
          ▷ algorithmIdentifier (sha256WithRSAEncryption)
            Padding: 0
            encrypted: 8d119078e946ad1308f06c1ddf898f64d54b9b2836487af7...
          Certificate Length: 1252
        ▷ Certificate: 308204e0308203c8a00302010202100b5c3435675b2467c0... (id-at-commonName=TERENA SSL High Assurance CA 3,id-at-organizationName=TERENA,id-at-local
```

Server: ServerKeyExchange

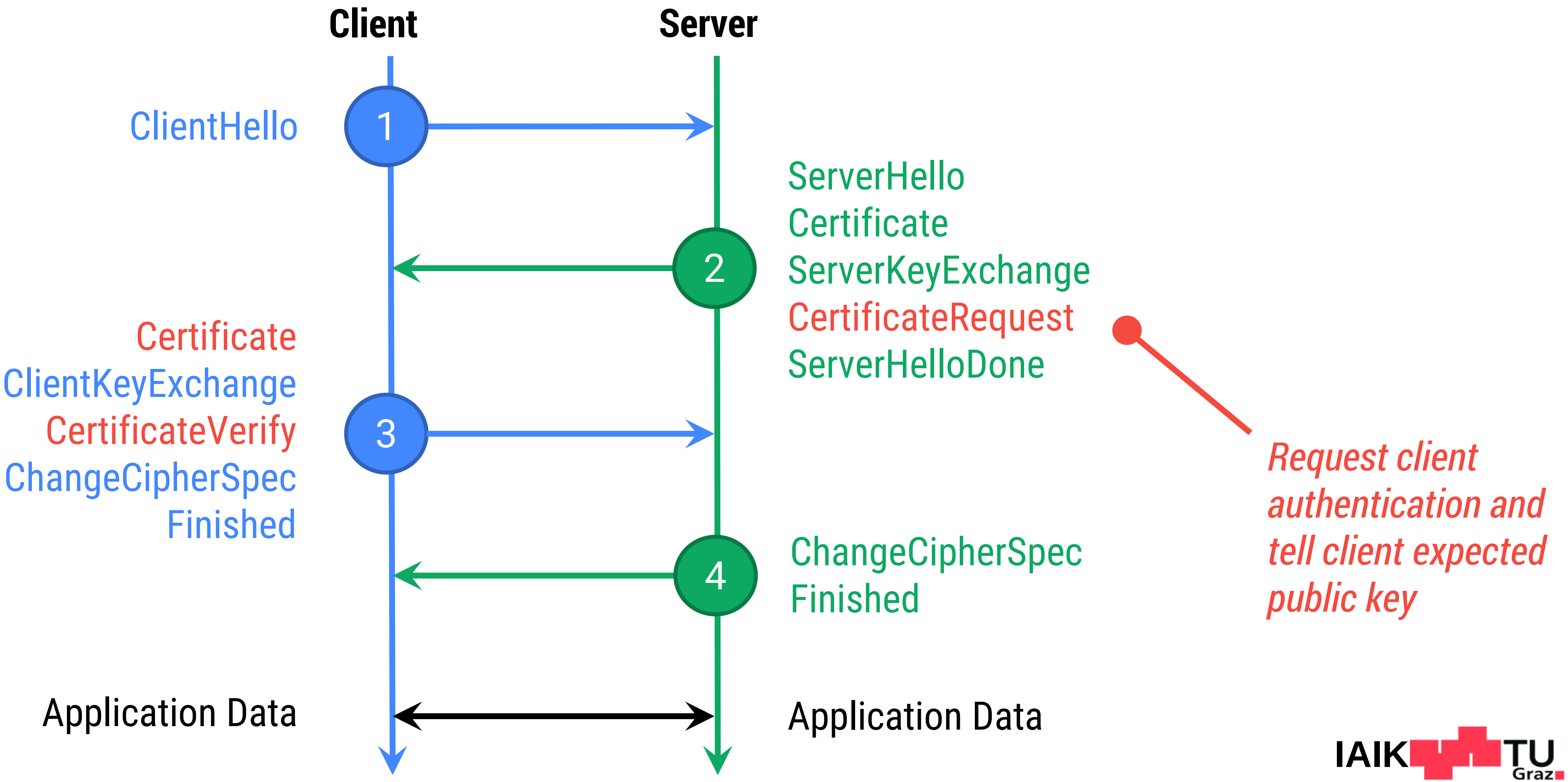
- Carry additional data needed for key exchange
  - Only sent when required for specified protocol
  - Our example: Parameters for ECDH

- Often this information is already within the certificate, e.g. if key exchange is RSA

```
◢ Secure Sockets Layer
  ◢ TLSv1.2 Record Layer: Handshake Protocol: Server Key Exchange
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 333
    ◢ Handshake Protocol: Server Key Exchange
        Handshake Type: Server Key Exchange (12)
        Length: 329
      ◢ EC Diffie-Hellman Server Params
          Curve Type: named_curve (0x03)
          Named Curve: secp256r1 (0x0017)
          Pubkey Length: 65
          Pubkey: 0465cdb560ea3a18bf633275625192a87cf2962309f144c2...
        ◢ Signature Hash Algorithm: 0x0601
            Signature Hash Algorithm Hash: SHA512 (6)
            Signature Hash Algorithm Signature: RSA (1)
          Signature Length: 256
          Signature: 70b0f4efbf6357fe43c0e1943051ae775c338ed374fa926e...
```

Server: CertificateRequest

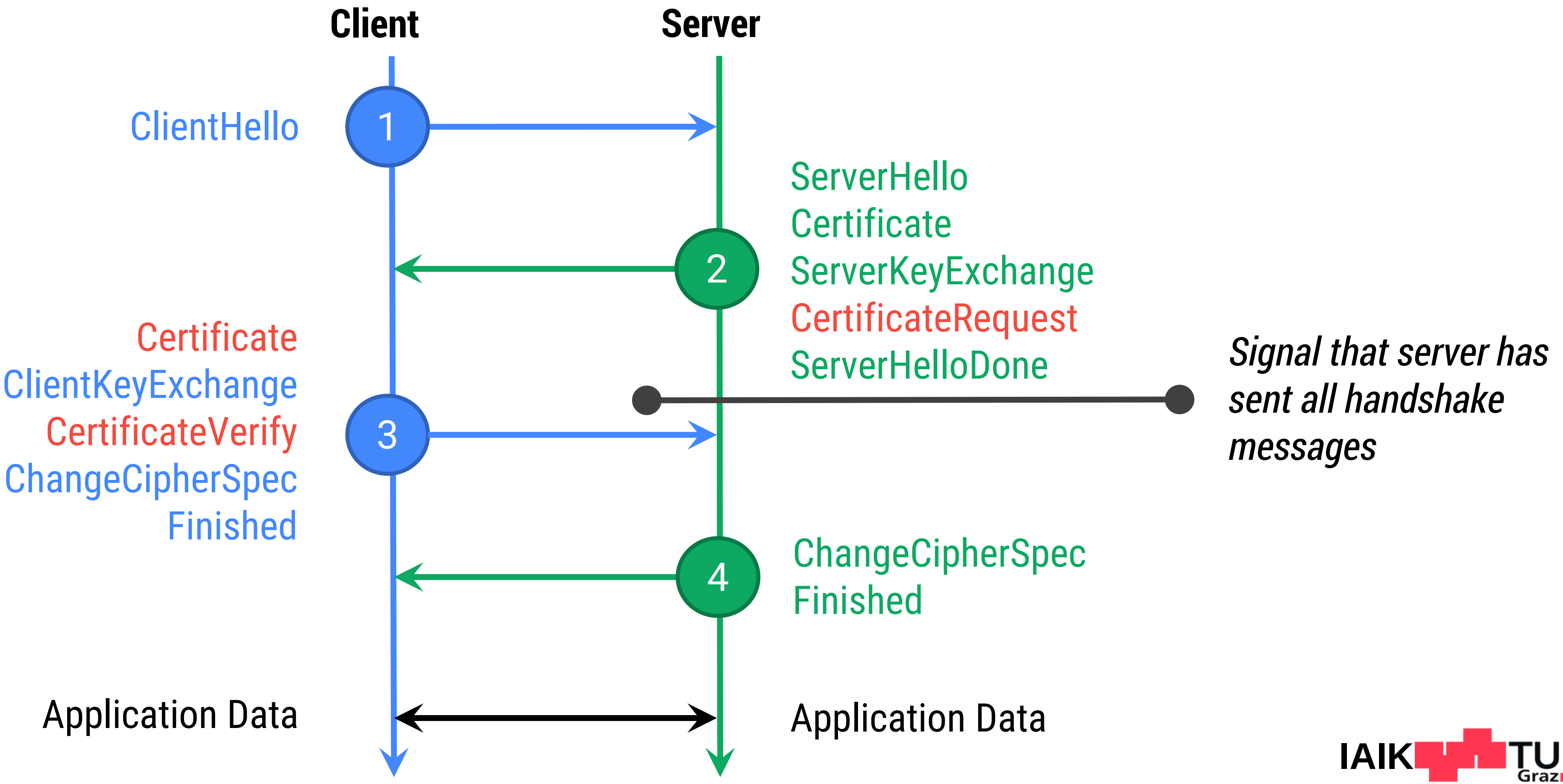
Only with Client TLS!



IAIK TU Graz

Server: ServerHelloDone

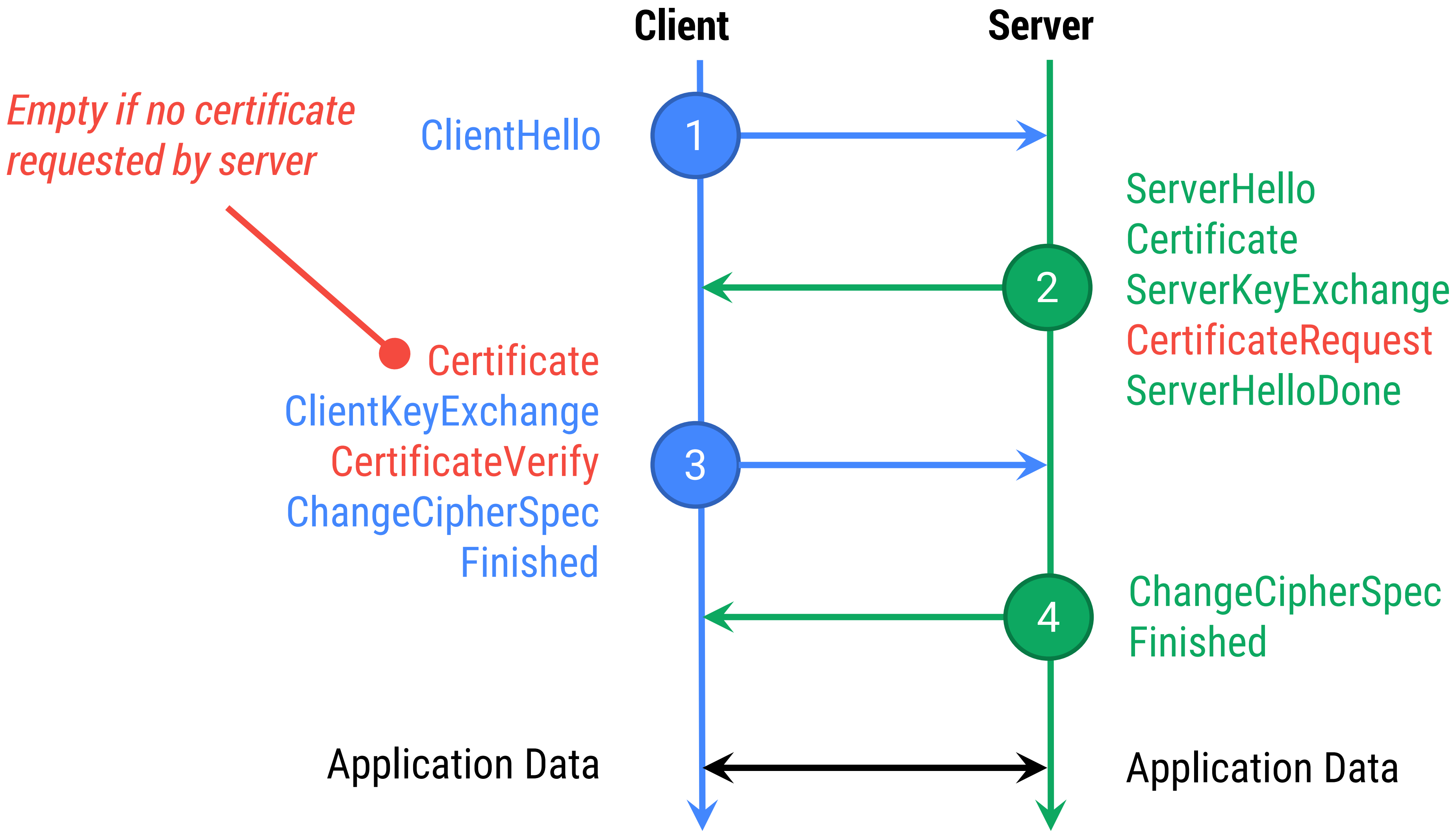
```
◢ TLSv1.2 Record Layer: Handshake Protocol: Server Hello Done
    Content Type: Handshake (22)
    Version: TLS 1.2 (0x0303)
    Length: 4
  ◢ Handshake Protocol: Server Hello Done
      Handshake Type: Server Hello Done (14)
      Length: 0
```



IAIK TU Graz

Client: Certificate

Only with Client TLS!



IAIK TU Graz

Client: ClientKeyExchange

Carries client‘s contribution (= preMaster secret) to key exchange

- Content depends on used cipher
  - If RSA is used, an RSA-encrypted secret is transfered
  - If Diffie Hellman (DH) is used, only the parameters are sent
    → enables both parties to agree on same preMaster secret
  - If *ephemeral* Diffie Hellman (DHE) is used, message contains client‘s DH public key

```
◢ Secure Sockets Layer
  ◢ TLSv1.2 Record Layer: Handshake Protocol: Client Key Exchange
      Content Type: Handshake (22)
      Version: TLS 1.2 (0x0303)
      Length: 70
    ◢ Handshake Protocol: Client Key Exchange
        Handshake Type: Client Key Exchange (16)
        Length: 66
      ◢ EC Diffie-Hellman Client Params
          Pubkey Length: 65
          Pubkey: 0440a27d25db5e4e3cc49a61356feeef85f9d825fdd04254...
```

Client: ClientKeyExchange

Example: RSA is used for key exchange

Step 1

- Client generates „PreMaster secret“ (48 random bytes)
- PreMaster secret encrypted with public key of server certificate
- Server decrypts PreMaster secret with private RSA key

Step 2

- Master secret (= session key) is derived by server and client

PRF = Pseudo-Random Function

```
masterSecret = PRF(preMasterSecret, „master secret“,
  ClientHello.random + ServerHello.random)[0..47]
```

Client: ClientKeyExchange – Security

RSA

- Simpler than others but with a fundamental weakness
  - PreMaster secret encrypted with server‘s public key
  - Anyone with access to private key can recover preMaster secret
  - Using preMaster secret → master secret recomputable

Diffie Hellman

- Security depends on quality of chosen parameters
  - If server sends weak or insecure parameters → compromise security of session

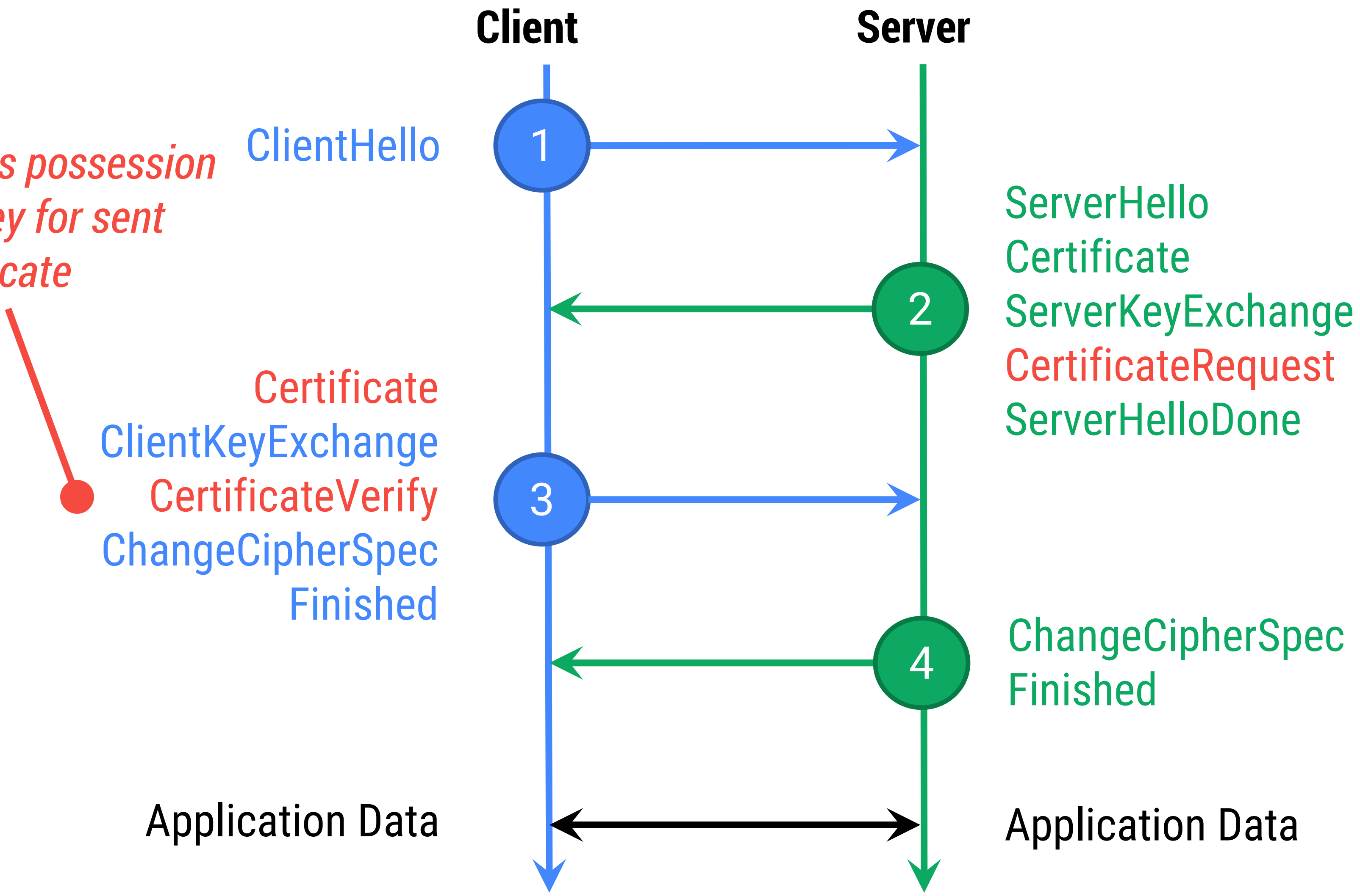
- Solution is to use standardized domain parameters of varying strength

IAIK TU Graz

Client: CertificateVerify

Only with Client TLS!

Client proves possession of private key for sent client certificate



IAIK TU Graz

Client & Server: ChangeCipherSpec

Signal that one party has all needed parameters,
has generated encryption keys and is switching to encryption

```
◢ 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
```

Sent by client and server as soon as they are ready…

IAIK TU Graz

Client & Server: Finished

Signal that handshake is complete

- Purpose is to verify integrity of entire handshake
  - Content is already encrypted

- Message contains hash of all handshake messages

```
verify_data = PRF(masterSecret, finishedLabel, hash(handshakeMessages))
```

  - Integrity of Finished message itself is guaranteed by negotiated MAC algorithm
  - Both parties decrypt message → check hash values

```
▽ TLSv1 Record Layer: Handshake Protocol: Encrypted Handshake Message
    Content Type: Handshake (22)
    Version: TLS 1.0 (0x0301)
    Length: 36
    Handshake Protocol: Encrypted Handshake Message
```

TLS Handshake Summary

1. Client starts handshake, sends parameters to Server
2. Server chooses common connection parameters
3. Server sends his certificate chain
4. If needed for key exchange → Server sends needed parameters to client
5. Server informs client that everything is done

6. Client sends parameters for key exchange to Server
7. Client switches to encrypted communication and informs Server about this
8. Client sends checksum (MAC) of all sent and received handshake messages to Server

9. Server switches to encrypted communication and informs client about this
10. Server also sends MAC of handshake messages

IAIK TU Graz

TLS Record

Byte	+0	+1	+2	+3
0	Content type			
1..4	Version		Length	
5..n	*Payload*			
n..m	MAC			
m..p	Padding (block ciphers only)			

Source: http://goo.gl/7zig7b

Typical workflow

- Record protocol receives application data
- Received data is divided into blocks (max. 16 KB per record)
- Add message authentication code (MAC)
- Data is encrypted using negotiated masterSecret

TLS Properties

Overview

Cryptographic aspects of TLS are fully configurable by cipher suites.
→ Define exactly how security will be implemented

Defines the following attributes

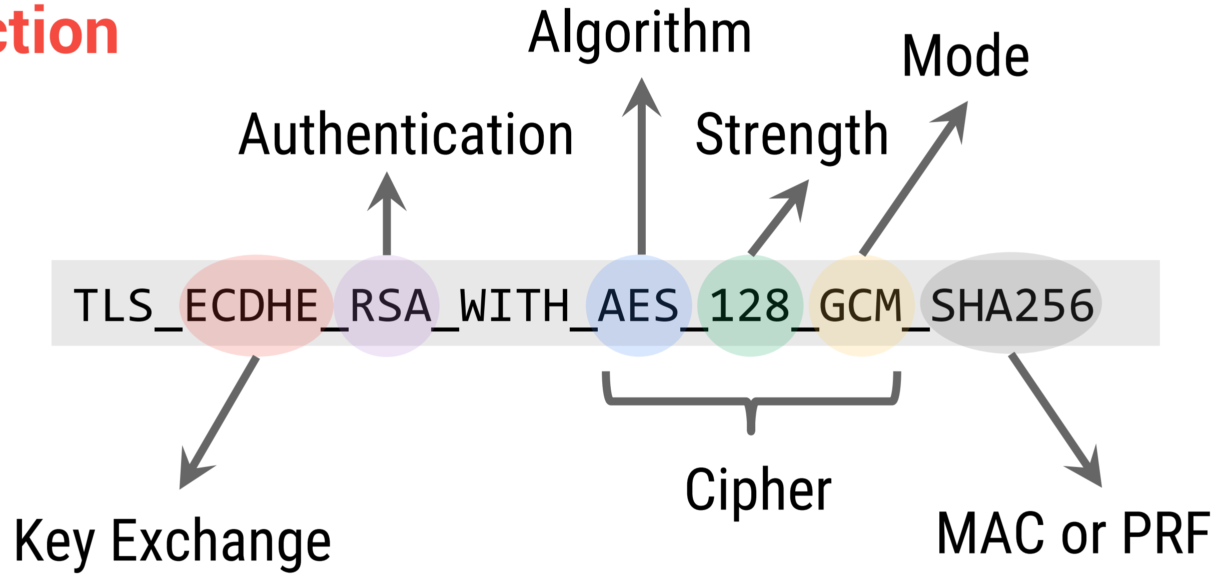
● Key exchange	RSA, DH, DHE, ECDH, ECDHE
● Authentication	RSA, DSA, DSS, ECDSA
● Hash function for MAC	MD5, SHA-1, SHA-256, SHA-512
● Encryption algorithm & key size	*none*, RC4, (3)DES, AES, …

→ Ensure TLS principles: Authenticity, Integrity, Confidentiality
Key exchange is a requirement for integrity and confidentiality

Note: RSA can be used for key exchange and authentication!

Cipher Suites

Name construction



Different notations

- IANA: TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256
- OpenSSL: ECDHE-RSA-AES128-GCM-SHA256
- PolarSSL: TLS-ECDHE-RSA-WITH-AES-128-GCM-SHA256

→ *[SSL|TLS], [Key Exchange], [Authentication], [Bulk cipher], [MAC]*

IAIK TU Graz

Cipher Suites

Key Exchange

`TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256`

DH = Diffie Hellman
DHE = Diffie Hellman Ephemeral
ECDH = Elliptic Curve Diffie Hellman
ECDHE = Ellliptic Curve Diffie Hellman Ephemeral

Note

- ECDH/ECDHE is similar to DH/DHE but **faster**!

→ ECDH keys with elliptic curves instead of DH parameters

→ Table of equivalent key lengths:

Symmetrisch	RSA / DH	ECDH
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	512

ECRYPT2 Yearly Report on Algorithms and Keysizes (2012)

Cipher Suites

```
openssl ciphers -v

TLS_AES_256_GCM_SHA384        TLSv1.3 Kx=any      Au=any   Enc=AESGCM(256) Mac=AEAD
TLS_CHACHA20_POLY1305_SHA256  TLSv1.3 Kx=any      Au=any   Enc=CHACHA20/POLY1305(256)…
TLS_AES_128_GCM_SHA256        TLSv1.3 Kx=any      Au=any   Enc=AESGCM(128) Mac=AEAD
ECDHE-RSA-AES256-GCM-SHA384   TLSv1.2 Kx=ECDH     Au=RSA   Enc=AESGCM(256) Mac=AEAD
ECDHE-ECDSA-AES256-GCM-SHA384 TLSv1.2 Kx=ECDH     Au=ECDSA Enc=AESGCM(256) Mac=AEAD
ECDHE-RSA-AES256-SHA384       TLSv1.2 Kx=ECDH     Au=RSA   Enc=AES(256)    Mac=SHA384
ECDHE-ECDSA-AES256-SHA384     TLSv1.2 Kx=ECDH     Au=ECDSA Enc=AES(256)    Mac=SHA384
ECDHE-RSA-AES256-SHA          SSLv3 Kx=ECDH       Au=RSA   Enc=AES(256)    Mac=SHA1
ECDHE-ECDSA-AES256-SHA        SSLv3 Kx=ECDH       Au=ECDSA Enc=AES(256)    Mac=SHA1
SRP-DSS-AES-256-CBC-SHA       SSLv3 Kx=SRP        Au=DSS   Enc=AES(256)    Mac=SHA1
SRP-RSA-AES-256-CBC-SHA       SSLv3 Kx=SRP        Au=RSA   Enc=AES(256)    Mac=SHA1
SRP-AES-256-CBC-SHA           SSLv3 Kx=SRP        Au=SRP   Enc=AES(256)    Mac=SHA1
DHE-DSS-AES256-GCM-SHA384     TLSv1.2 Kx=DH       Au=DSS   Enc=AESGCM(256) Mac=AEAD
(...)
```

For complete list, see http://goo.gl/Jg5wUp

Cipher Suites in the Browser

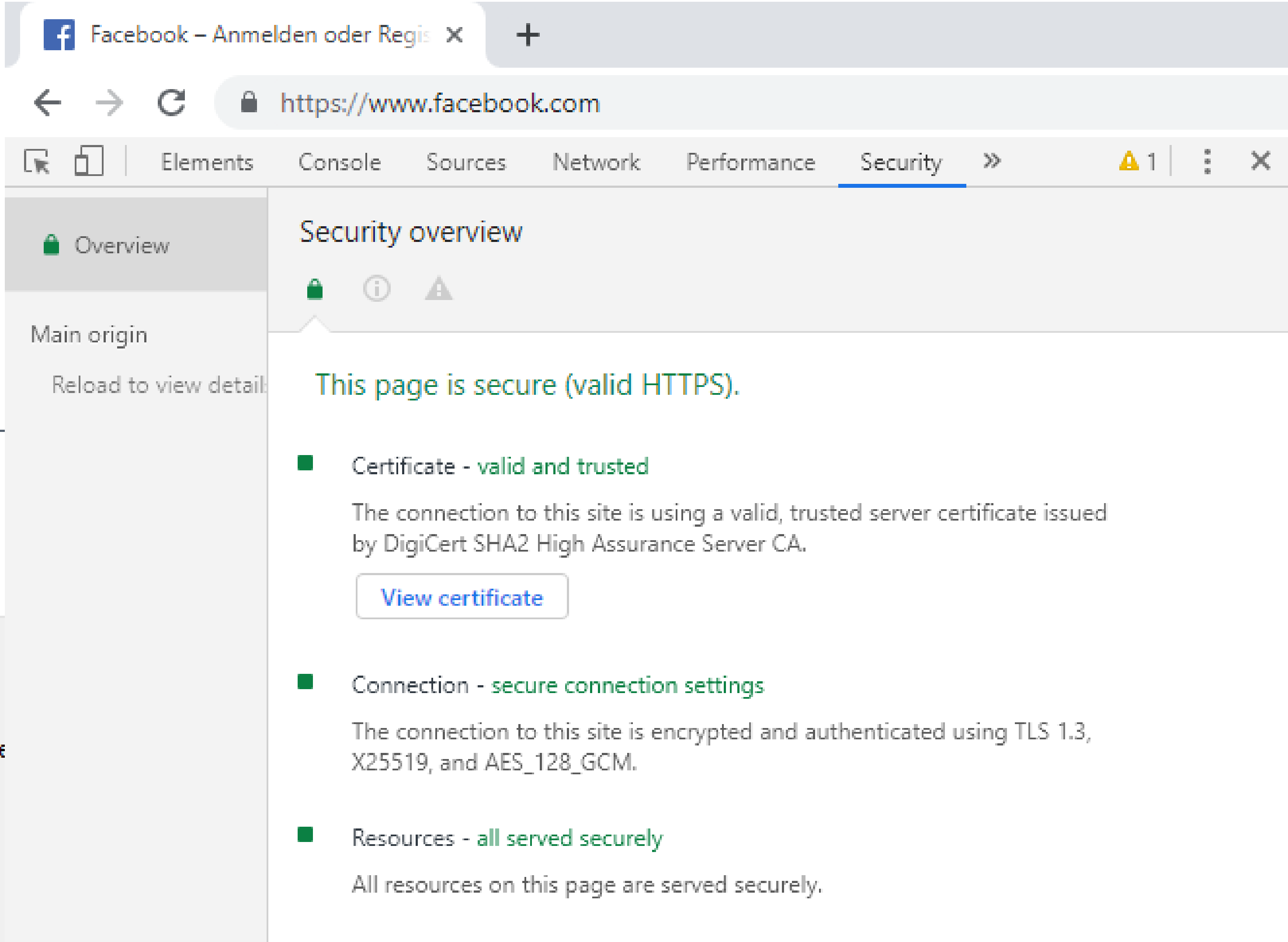
Which are offered by your client?

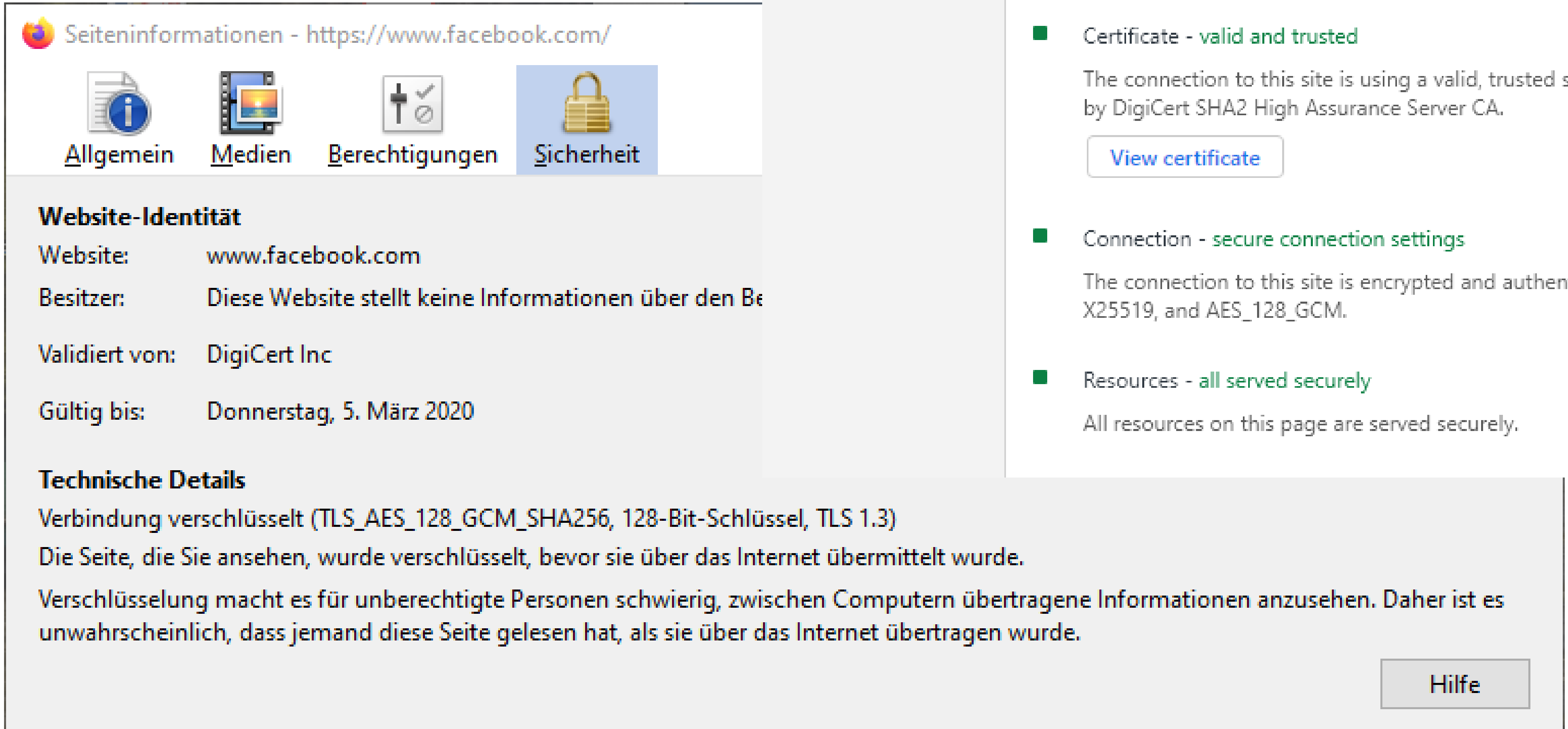
- Depends on used library
  - Internet Explorer (Edge): Cryptography Service Provider (CSP)
  - Mozilla Firefox: Network Security Services (NSS)
  - Google Chrome: NSS with own adaptions
  - Apple Safari: SecureTransport
  - Android: AndroidOpenSSL and BouncyCastle (modified)

→ Modern browsers prefer AES-GCM and AES-CBC

Find out your preferences at https://www.howsmyssl.com

Cipher Suites in the Browser





(Perfect) Forward Secrecy

Compromise of long-term keys should not compromise past session keys

Without Forward Secrecy

- Security of all connections depend on server‘s private key
- If broken or stolen → previous communication can be decrypted

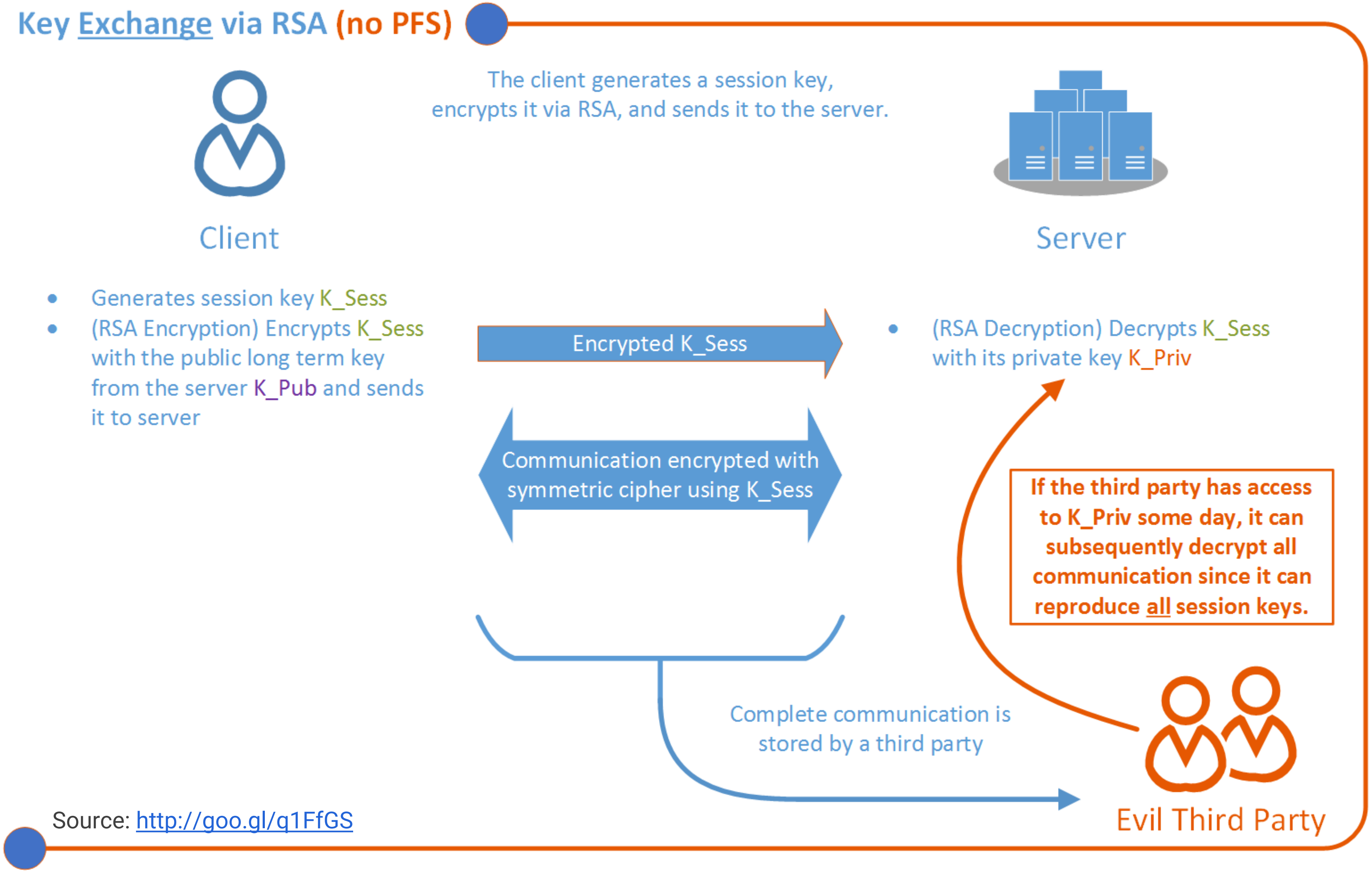
Why is this possible?

- During the handshake, the client creates a preMaster secret
- Encrypted using the server‘s public (RSA) key it is sent to the server
  - Server uses his private key to decrypt it → calculate common masterSecret

→ If you have the private key, you can decrypt past and future data!!

IAIK TU Graz

Without PFS



Source: http://goo.gl/q1FfGS

(Perfect) Forward Secrecy

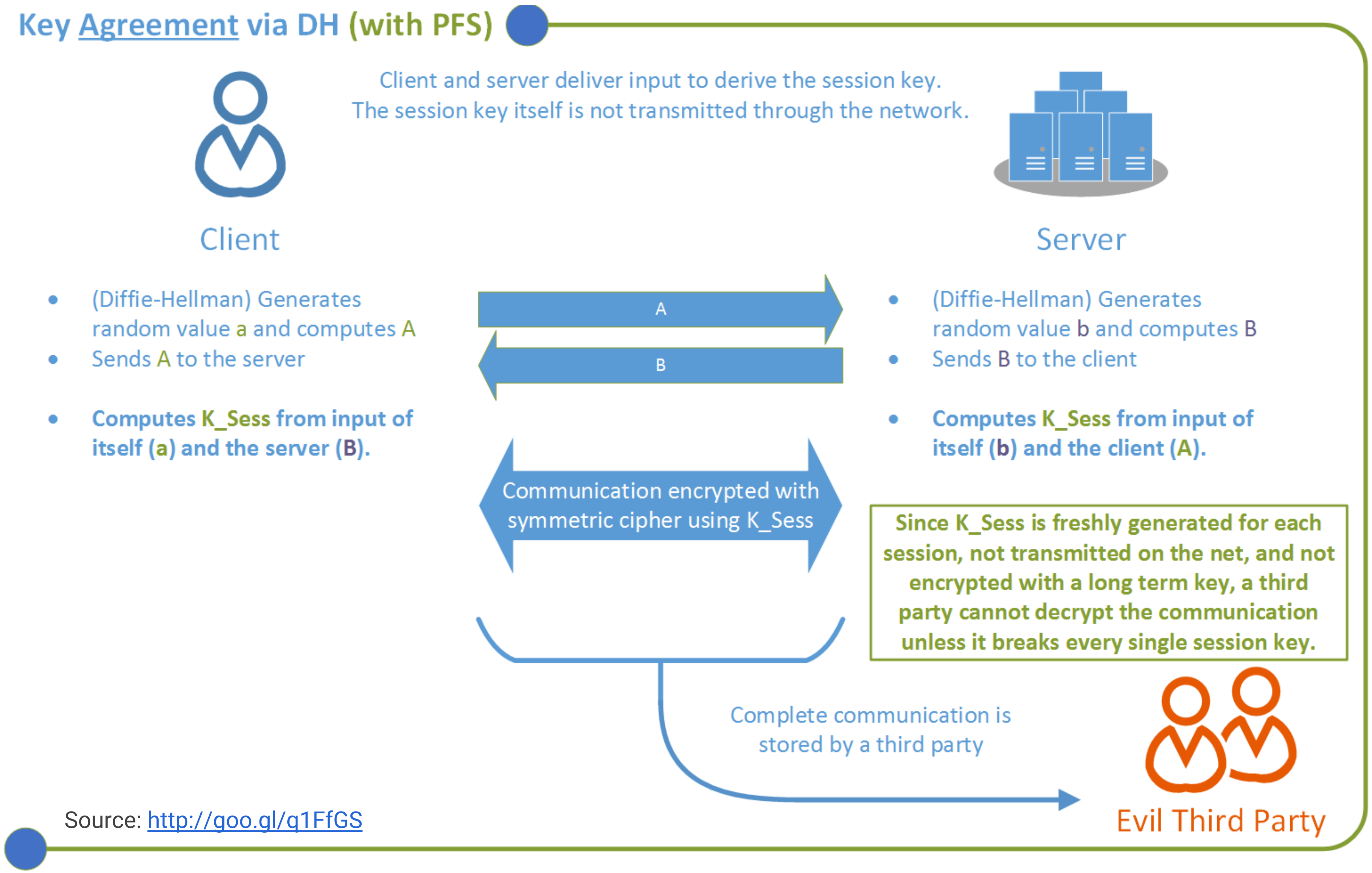
With Forward Secrecy

- Server generates a *short-living („ephemeral“)* Diffie-Hellman keypair
  - DHE = Diffie-Hellman *Ephemeral*
  - ECDHE = Elliptic Curve Diffie-Hellman *Ephemeral*
- Server signs the public key of this DH pair with the private key of the server‘s certificate
  - Can be RSA or ECDSA depending on the certificate
- Client receives the signed public DH key, checks if signature is verifiable using public key of the previously received server‘s certificate

→ Instead of „Key transport“ (RSA), forward secrecy works with „Key agreement“!

With PFS

Note: This graphic misses the key signing part!



Source: http://goo.gl/q1FfGS

IAIK TU Graz

(Perfect) Forward Secrecy

Security

- For every new session, client & server generate new Diffie-Hellman parameters
  - If compromised somehow → attacker could only read this particular session

- Attacking the session key
  - If parameters are securely chosen, brute-force should not be possible
    - E.g. use 2048-bit or stronger Diffie-Hellman groups with „safe“ primes

- Attacking the server‘s private key
  - With PFS, only used to sign ephemeral public DH keys sent to the client
  - If broken or leaked → would not compromise past sessions

- Hacking the server: Attacker only gets current session keys & key for signatures

(Perfect) Forward Secrecy

How to get Forward Secrecy?

- Server needs at least TLS 1.2 + offer PFS supporting cipher suite
- Important: Only key exchange with DHE or ECDHE offers forward secrecy!
  - Cipher suite, e.g DHE-RSA-AES128-SHA or ECDHE-ECDSA-AES128-SHA

Test servers

- https://www.ssllabs.com/ssltest/
- http://demoapps.a-sit.at/ssl-tool/
- https://testssl.sh
- https://github.com/nabla-c0d3/sslyze

- Examples on how not to configure servers: https://badssl.com
  - Small DH groups, weak ciphers, etc.

(Perfect) Forward Secrecy

SSL/TLS Server Test for online.tugraz.at

Server information

Hostname: online.tugraz.at
IP address: 129.27.2.210
Port: 443
Server type: Apache

Forward Secrecy

Weak

Insecure / Vulnerable

Cipher suite	Key exchange	MAC	Cipher	Key length
TLS_DHE_RSA_WITH_AES_256_GCM_SHA384	DHE RSA 2048 bits	SHA384	AES/GCM/NoPadding	256 bits
TLS_DHE_RSA_WITH_AES_256_CBC_SHA256	DHE RSA 2048 bits	SHA256	AES/CBC/NoPadding	256 bits
TLS_ECDHE_RSA_WITH_AES_256_GCM_SHA384	ECDHE RSA 256 bits	SHA384	AES/GCM/NoPadding	256 bits
TLS_ECDHE_RSA_WITH_AES_256_CBC_SHA384	ECDHE RSA 256 bits	SHA384	AES/CBC/NoPadding	256 bits
TLS_DHE_RSA_WITH_AES_128_GCM_SHA256	DHE RSA 2048 bits	SHA256	AES/GCM/NoPadding	128 bits
TLS_DHE_RSA_WITH_AES_128_CBC_SHA256	DHE RSA 2048 bits	SHA256	AES/CBC/NoPadding	128 bits
TLS_ECDHE_RSA_WITH_AES_128_GCM_SHA256	ECDHE RSA 256 bits	SHA256	AES/GCM/NoPadding	128 bits

TLS Security

Overview

Problem

Attacks often based on downgrades HTTPS → HTTP („SSLStrip“)

- Variant A
  - Web page offers HTTP and HTTPS version
  - Attacker injects HTTP links to force user to use weak HTTP communication
- Variant B
  - Web page offers HTTPS only
  - Attacker uses proxy server (Man-in-the-middle) and translates to HTTP communication

Solution? HTTP Strict Transport Security (HSTS)

HSTS

RFC 6797

= Tell browser that all connections to a domain are HTTPS only

→ Specified via HTTP header that can only be sent during valid HTTPS request

```
Strict-Transport-Security: max-age=10886400; includeSubDomains
```

in seconds

Optional (recommended)

Browser remembers (for specified max-age period) that it should only request HTTPS resources for this site (and optionally subdomains)

→ Effectively prevents „SSL Stripping“ attacks!

HTSS

But: What if an attacker has control over the initial HTTPS requests?

Scenario

- Attacker would strip HSTS headers
- Browsers would not know HSTS should be active

Solution

- Browsers ship with „preloaded“ HSTS lists → Sites that *always* require HTTPS
- Add „preload“ header and add domain here: https://hstspreload.appspot.com

```
Strict-Transport-Security: max-age=10886400; includeSubDomains; preload
```

Man-in-the-Middle

Problem

You are not presented the „correct“ certificate for a domain

- Variant A
  - Attacker malevolently exchanges certificate with self-generated one
  - Client connects and attacker redirects data transfer
- Variant B
  - Certificate Authority (CA) is compromised
  - Attacker generates trusted certificate and exchanges it

Solution? HTTP Public Key Pinning (HPKP)

Certificate Pinning (HPKP)

RFC 7469

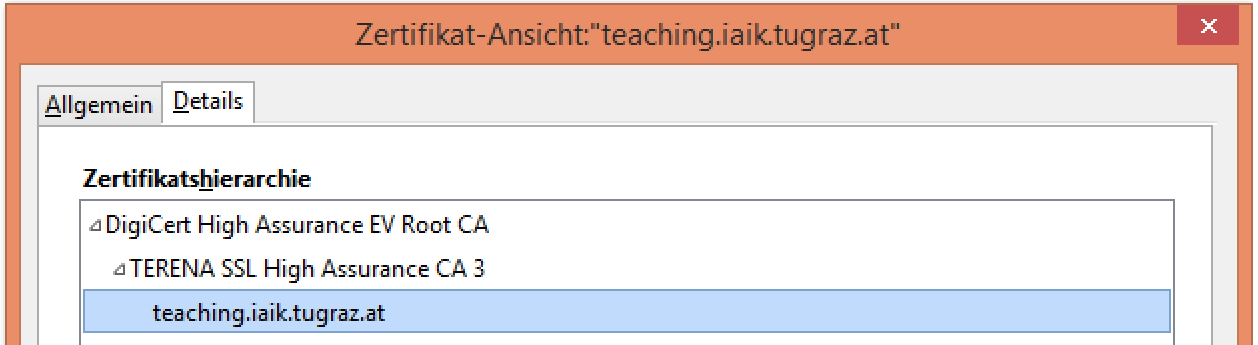
Problem

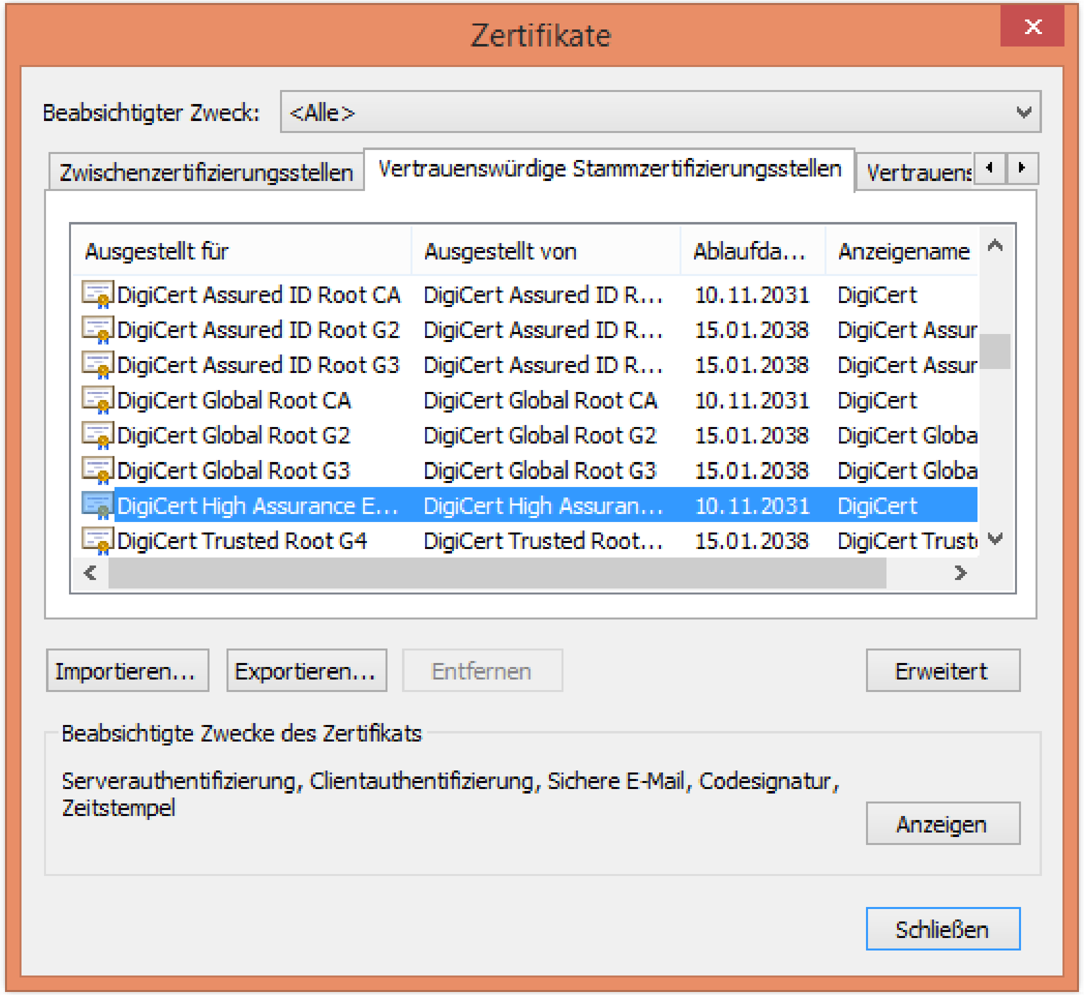
Our browsers trust ~130 CAs („Trust Store“)

How is trust established?

1. Browser compares DNS hostname with subject name in certificate

2. Upon match, check if certificate issued by trusted CA





Certificate Pinning

RFC 7469

Scenario

Usually the certificate chain for google.com looks as follows:

```
GlobalSign Root CA – R2
    GTS CA 1O1
        google.com
```

Now:

- Assume „TÜRKTRUST Elektronik Sunucu Sertifikası Hizmetleri“ issues a certificate for google.com
- A webserver for google.com is setup, DNS entries are rewritten to point at that server and the user is forwarded there → would he notice?

```
TÜRKTRUST Elektronik Sunucu Sertifikası Hizmetleri
    e-islem.kktcmerkezbankasi.org
        google.com
```

No, he would not!
https://goo.gl/QVfHYV

IAIK TU Graz

Certificate Pinning

Another scenario

1. Attacker has access to trusted CA, issues certificates for arbitrary hostnames
2. Attacker performs MITM attack using previously generated certificate

→ Attacker could replace any TLS certificate, browser would still trust it

Remedy?

- Remember hash values („pins“) of public keys associated with certificates
- If PIN changes (= certificate changes), drop connection even if certificate would be trustworthy and DNS name matches with cert‘s subject name
- PINs either stored in browser (or mobile app) or sent via HTTP header

Certificate Pinning

```
Public-Key-Pins:
pin-sha256="GRAH5Ex+kB4cCQi5gMU82urf+6kEgbVtzfCSkw55AGk=";
pin-sha256="lERGk61FITjzyKHcJ89xpc6aDwtRkOPAU0jdnUqzW2s=";
max-age=15768000; includeSubDomains
```

How to generate PINs?

- Get SHA-256 hash value of public key of server certificate
- Base-64 encoding of hash and inserting into header

Advantages

- Defeats MITM attacks
- PIN can also be stored in browser

Disadvantages

- “Trust-on-first-use“ mechanism (like HSTS)
- Many things can go wrong while setup
- You **must** have >= 2 PINs

Outlook

- 24.01.2020
  - TLS Vulnerabilities & Attacks
  - DNS Security
- 31.01.2020
  - Lecture Exam

IAIK TU Graz