Authenticated Key Exchange (in TLS)

Kenny Paterson

Information Security Group

@kennyog ; www.isg.rhul.ac.uk/~kp

ROYAL HOLLOWAY UNIVERSITY OF LONDON

Overview

- Overview of TLS Key Exchange
- Attacks on TLS Handshake Protocol
  - RSA encryption in TLS
  - TLS Renegotiation and Triple Handshake Attack
  - Cross-protocol attacks including FREAK and LOGJAM
- Security proofs for TLS
- The future of key exchange in TLS

TLS Overview

SSL = Secure Sockets Layer.

- Developed by Netscape in mid 1990s.
- SSLv1 broken at birth.
- SSLv2 flawed in several ways, now IETF-deprecated (RFC 6176).
- SSLv3 now considered broken (POODLE + RC4 attacks), but still widely supported.

TLS = Transport Layer Security.

- IETF-standardised version of SSL.
- TLS 1.0 in RFC 2246 (1999).
- TLS 1.1 in RFC 4346 (2006).
- TLS 1.2 in RFC 5246 (2008).
- TLS 1.3 currently in development in IETF.

Importance of TLS

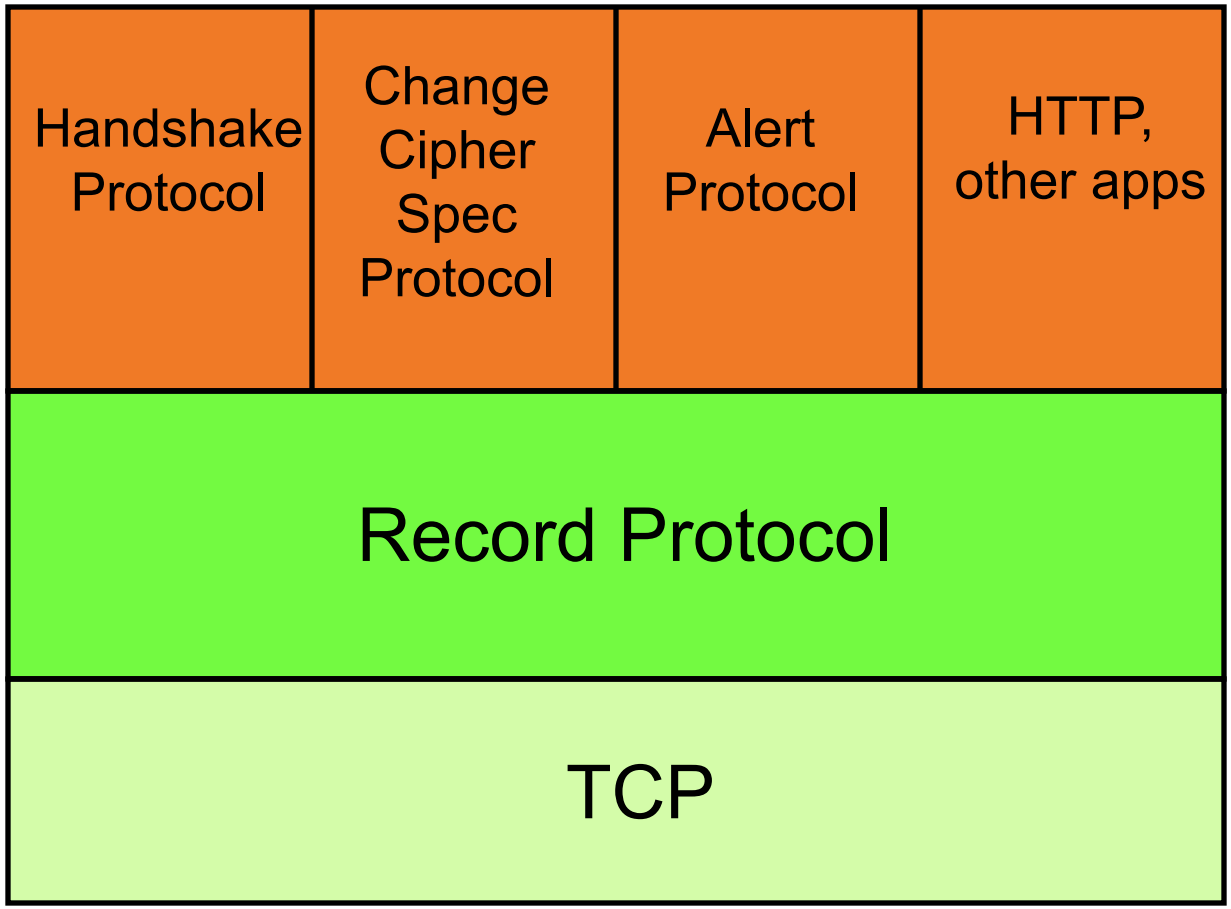
Originally designed for secure e-commerce, now used much more widely.

- Retail customer access to online banking facilities.
- Access to gmail, facebook, Yahoo, etc.
- Mobile applications, including banking apps.
- Payment infrastructures.

TLS has become the *de facto* secure protocol of choice.

- Used by hundreds of millions of people and devices every day.
- So we need to analyse it, in order to find and remove flaws.

TLS Protocol Architecture



TLS Record Protocol

TLS Record Protocol provides:

- Data origin authentication, integrity.
- Confidentiality.
- Anti-replay using sequence numbers.
- Optional compression.
- For **streams** of application layer messages.

Achieved via stateful Authenticated Encryption

- MAC-Encode-Encrypt using (CBC-mode or RC4) + HMAC up to TLS 1.1.
- Option for "proper" AE in TLS 1.2, typically AES-GCM, also now ChaCha20+ Poly1305

TLS Record Protocol subject to many attacks and analyses

- Padding oracles, BEAST, CRIME, Lucky 13, RC4, short MAC,…

TLS Handshake Protocol – Goals

Establishes keys (and IVs) needed by the Record Protocol.

- Via establishment of the TLS `master_secret` and subsequent key derivation.

Provides authentication of server (usually) and client (rarely)

- Using public key cryptography supported by digital certificates.
- Or pre-shared key (less commonly).

Protects negotiation of all cryptographic parameters.

- SSL/TLS version number.
- Encryption and hash algorithms.
- Authentication and key establishment methods.
- To prevent version rollback and ciphersuite downgrade attacks.

TLS Handshake Protocol – Key Establishment

TLS supports several key establishment mechanisms.

Method used is negotiated during the Handshake Protocol itself.

Client sends list of *ciphersuites* it supports in `ClientHello`; server selects one and tells client in `ServerHello`.

e.g. `TLS_RSA_WITH_AES_256_CBC_SHA256`

e.g. `TLS_ECDHE_RSA_WITH_RC4_128_SHA`

https://www.thesprawl.org/research/tls-and-ssl-cipher-suites/

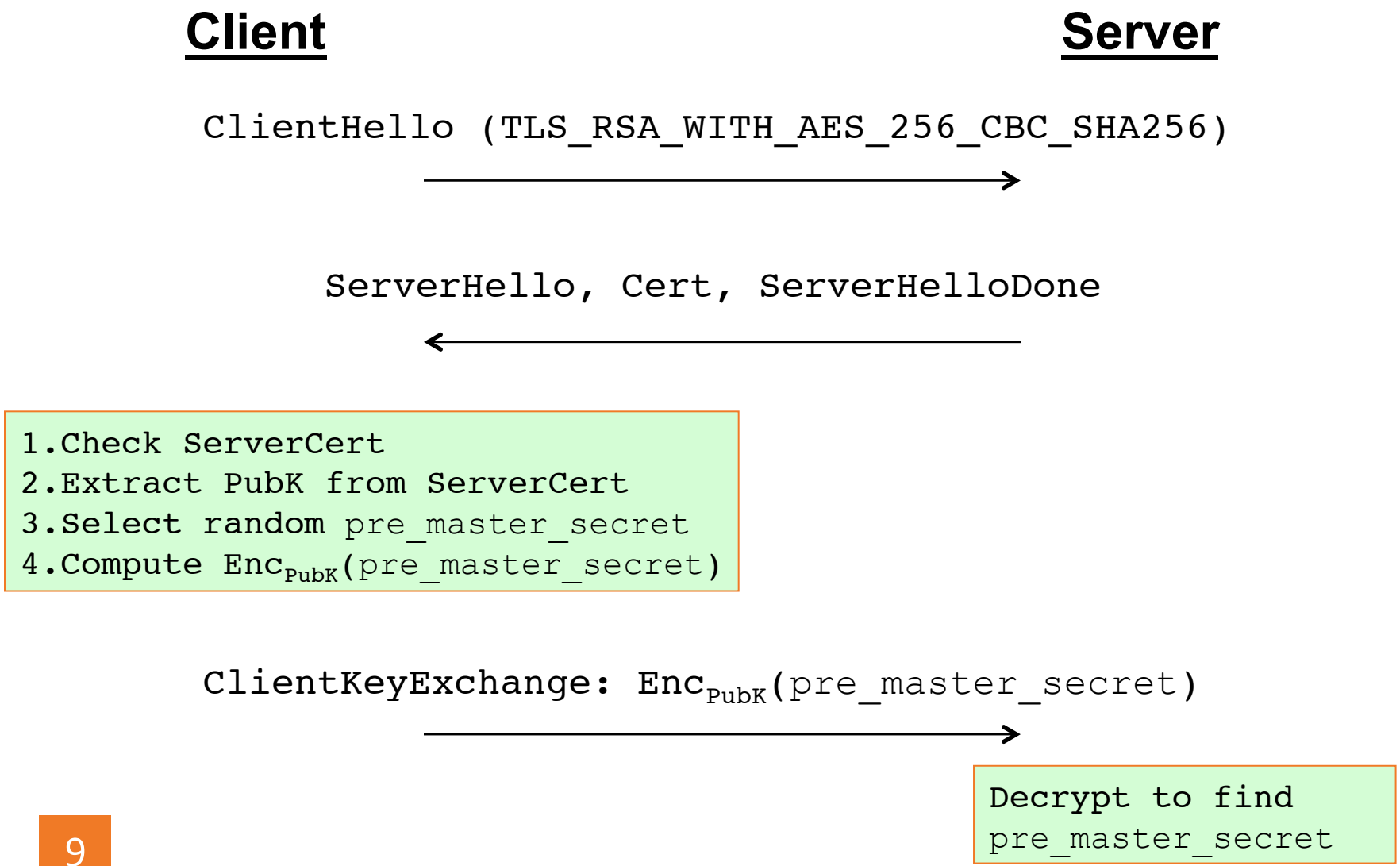
Common choice is RSA encryption.

Server sends RSA public key and certificate chain after `ServerHello`.

Client chooses `pre_master_secret`, encrypts using public RSA key of server, sends to server.

RSA encryption is based on PKCS#1 v1.5 padding method.

TLS Handshake Protocol – RSA-based Key Establishment (Simplified)



TLS Handshake Protocol – Key Establishment

Static Diffie-Hellman

Server certificate contains DH parameters (group, generator g) and static DH value g^x.

Client chooses y, computes g^y and sends to server.

$$\texttt{pre\_master\_secret} = g^{xy}.$$

Ephemeral Diffie-Hellman

Server and Client exchange fresh Diffie-Hellman components g^x, g^y in a group chosen by server.

Server values (and nonces) signed to provide server authentication.

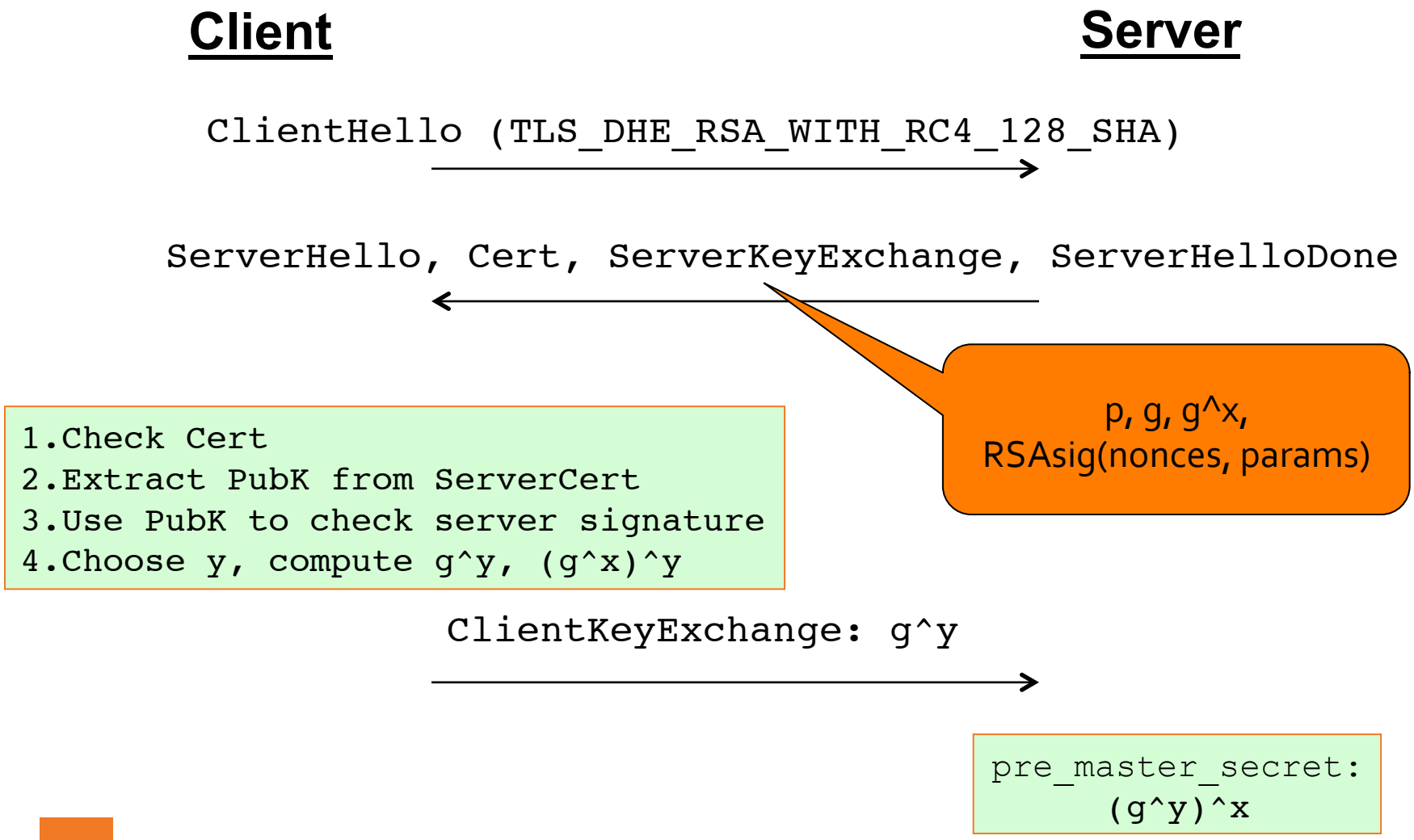
Typically mod p group or elliptic curve group.

Anonymous Diffie-Hellman

Each side sends Diffie-Hellman values in group chosen by server, but no authentication of these values.

Vulnerable to man-in-middle attacks.

TLS Handshake Protocol – Ephemeral DH-based Key Establishment (Simplified)



TLS Handshake Protocol – Key Establishment Notes

Typical `ClientHello` offers many different ciphersuites, choice of which to use is made by server.

`ClientHello` also offers SSL/TLS version number; server replies with its choice.

- Semantics: client: I support up to version x; server: I will use version y ≤ x.
- Legacy servers do not implement this correctly, and fail if they don't support version x.
- Typical client behaviour: try again with lower version in a fresh handshake, with no memory of previous offer(s) carried over.
- Security consequence: an active MITM can force client/server to roll back to **lowest** SSL/TLS version they are both willing to use!
- POODLE attack exploits this to roll back to SSL_3 and perform Moeller attack on $SSLv_3$ padding.

TLS Handshake Protocol – Key Establishment Notes

`ClientHello` and `ServerHello` contain 32-byte nonces.

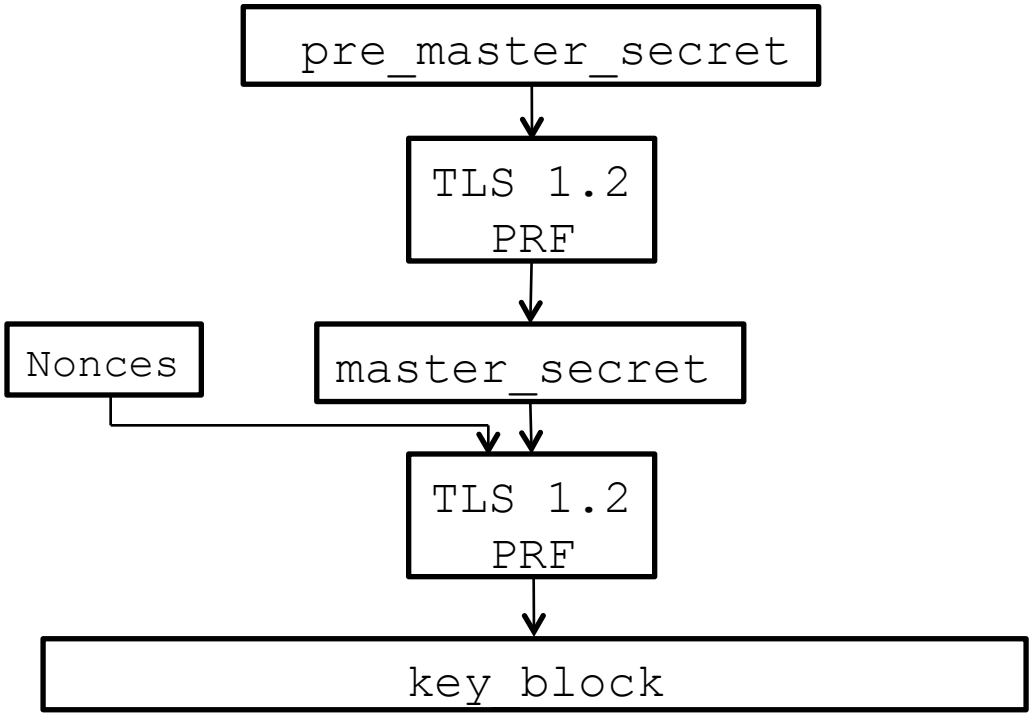
- Signed by server in DH-based ciphersuites, involved in key derivation (next slide).

An attacker who knows the RSA private key can passively eavesdrop on all RSA-based sessions!

An attacker who can predict client's choice of `pms` or DH private value can passively eavesdrop on all sessions!

- And nonces may already leak information about state of client or server PRNG.
- See Checkoway et al. (USENIX Security 2014) for more.

TLS Key Derivation

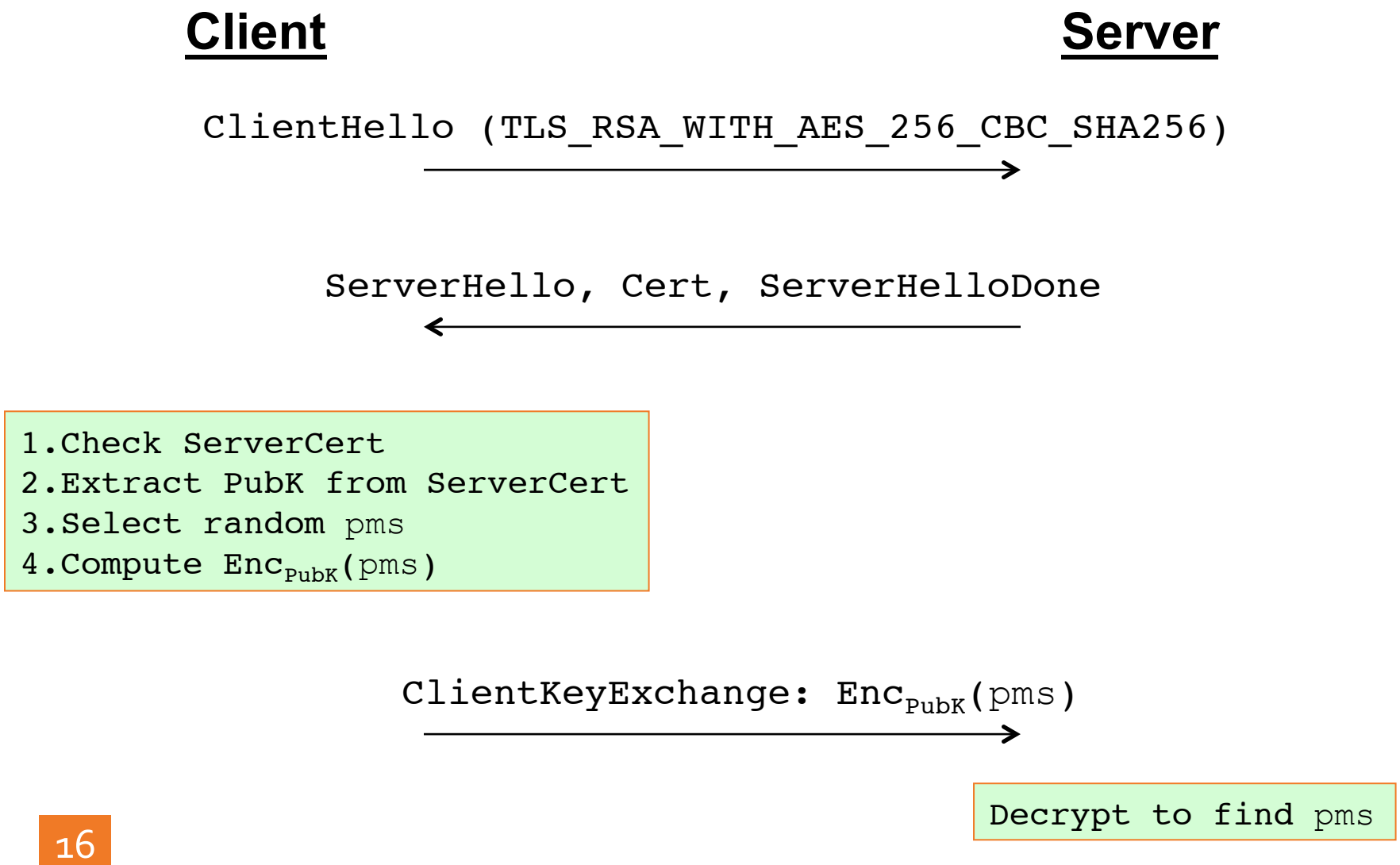


TLS Key Derivation

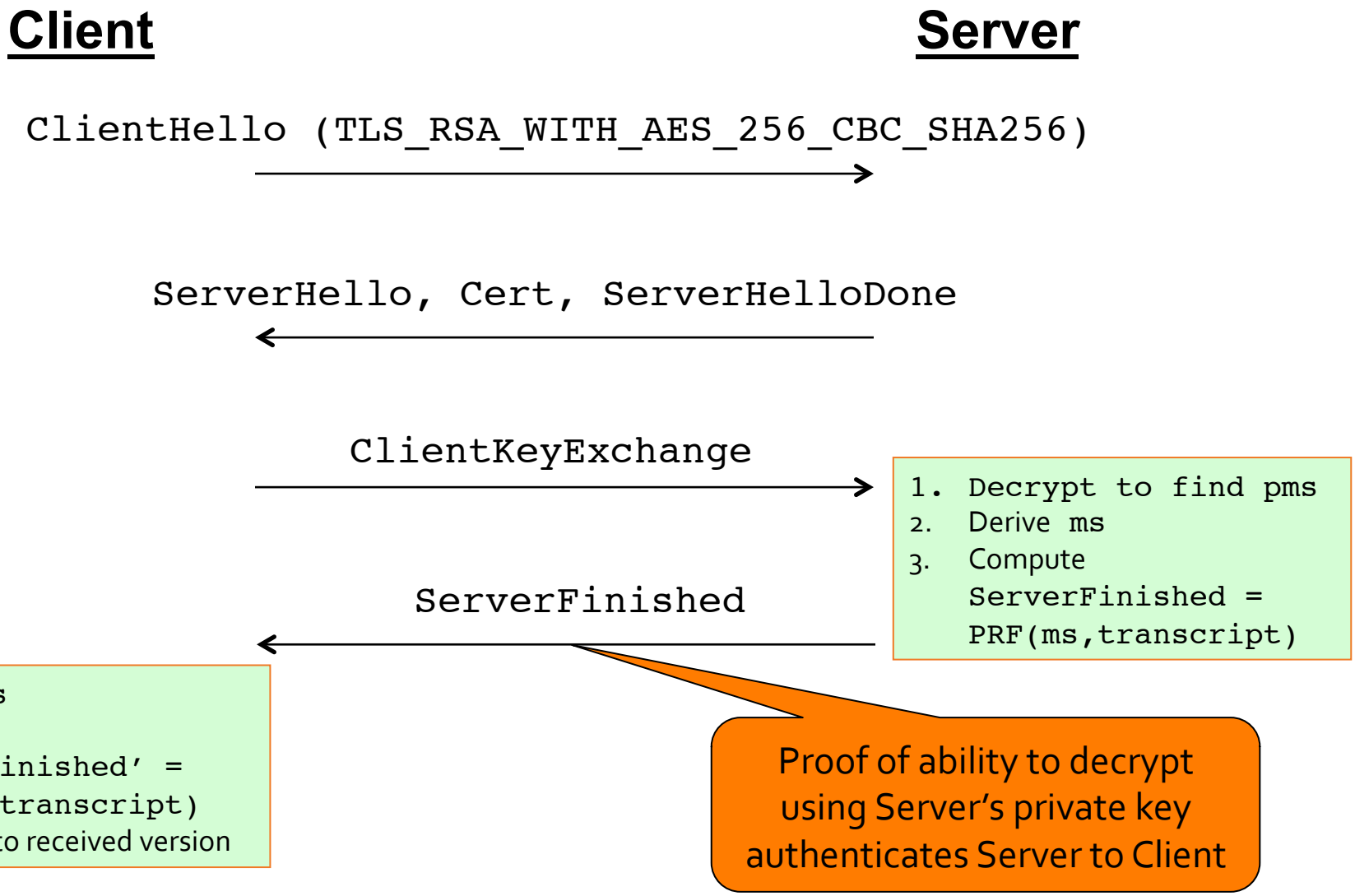
Keys used by MAC and encryption algorithms in the Record Protocol are derived from `pms`:

- Derive `ms` from `pms` using TLS Pseudo-Random Function (PRF).
- Default PRF for TLS1.2 is built by iterating HMAC-SHA256 in a specified way
- Derive key_block from `ms` and client/server nonces exchanged during Handshake Protocol.
- Again using the TLS PRF in TLS1.2.
- Split up key_block into MAC keys, encryption keys and IVs for use in Record Protocol as needed.
- NB: neither client nor server identity is not involved in key derivation.

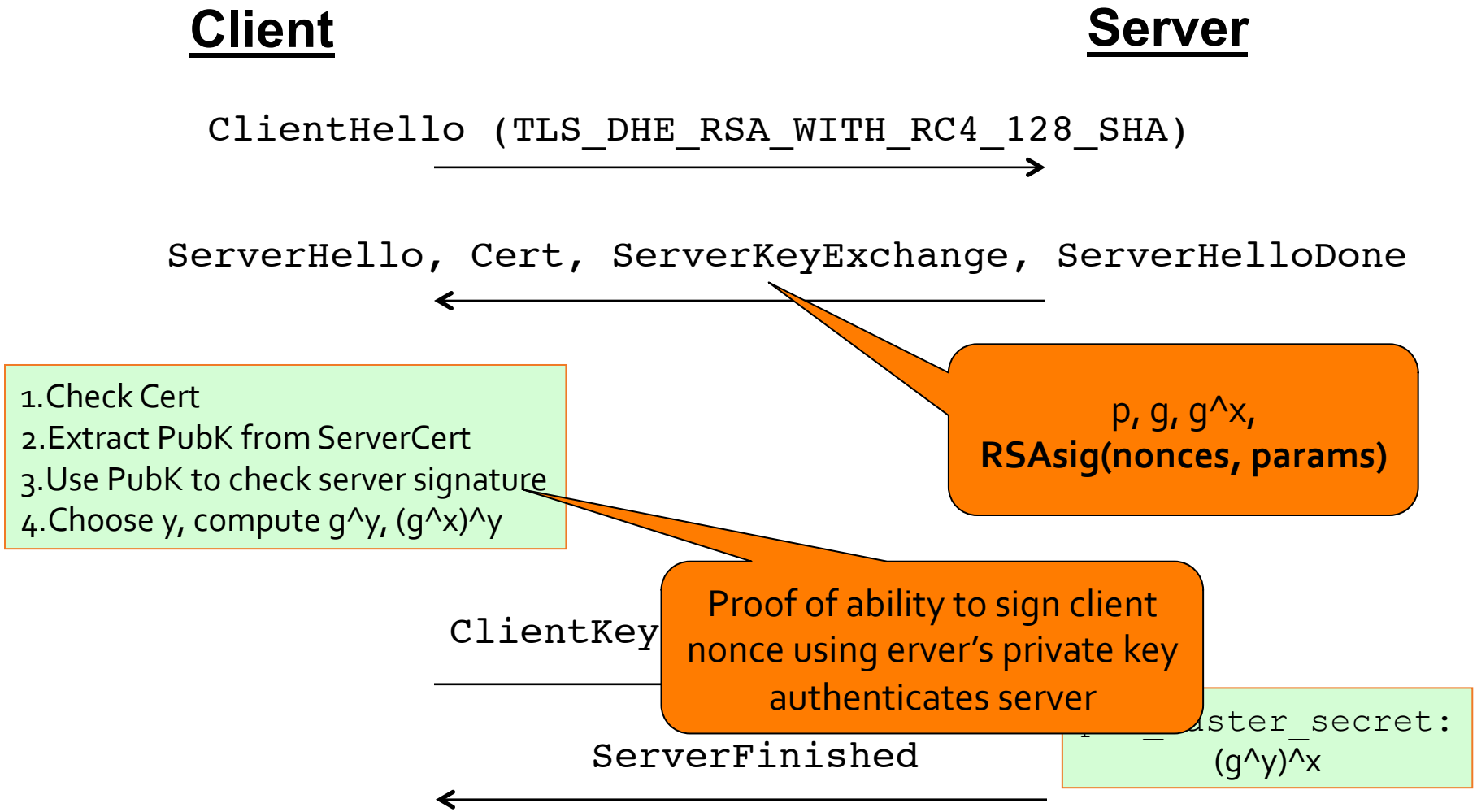
TLS Handshake Protocol – RSA-based Authentication?



TLS Handshake Protocol – RSA-based Authentication



TLS Handshake Protocol – Authentication for Ephemeral DH-based Key Establishment



TLS Handshake Protocol – Authentication

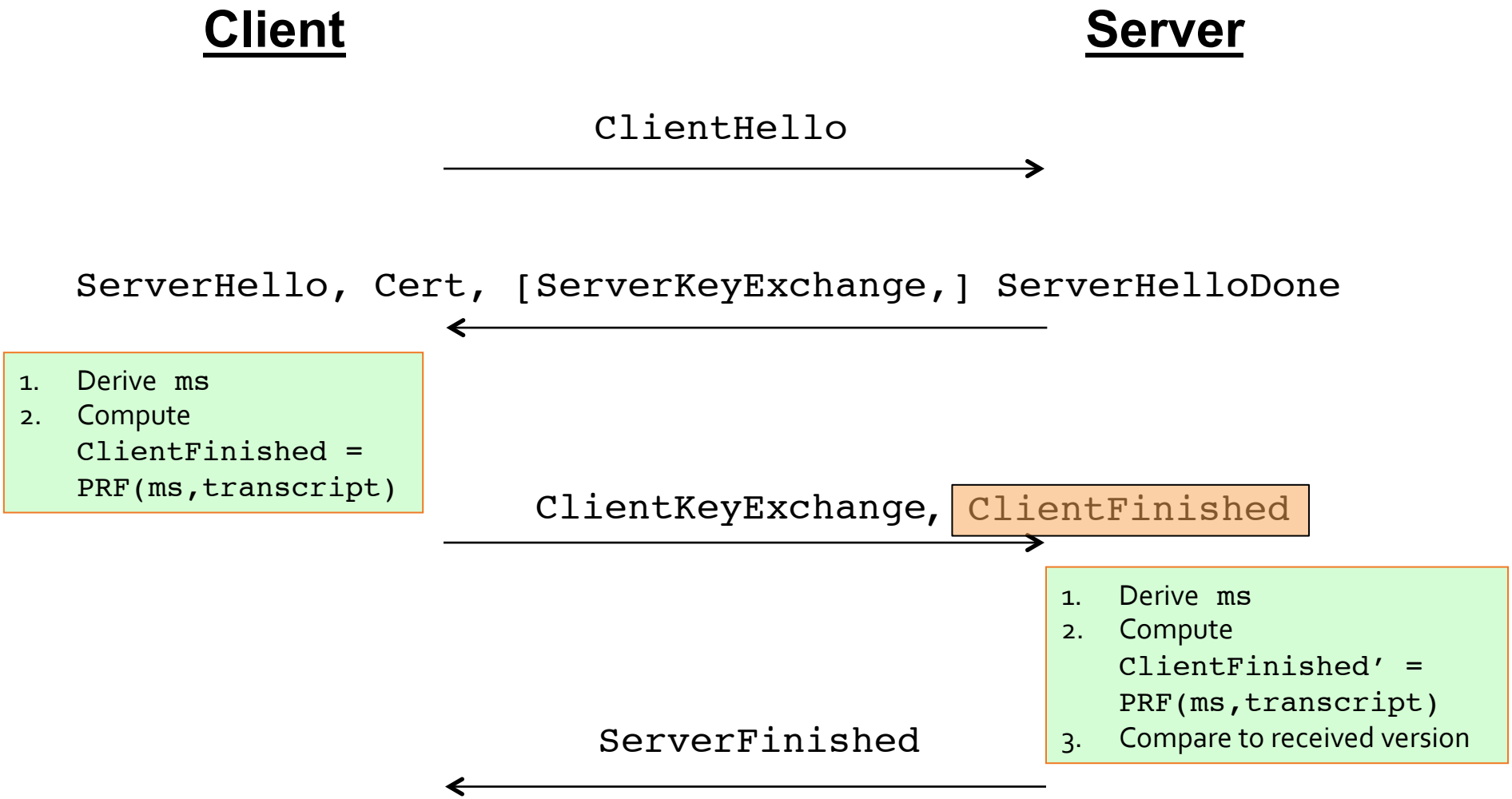
TLS supports several different entity authentication mechanisms for clients and servers.

Method used is negotiated along with key exchange method during the Handshake Protocol itself.

RSA: Ability of server to decrypt `pms` using its private key, derive `ms` from `pms` and then generate correct PRF value in ServerFinished message.

DHE/ECDHE: Ability of server to sign `ClientNonce` using its private key.

TLS Handshake Protocol – `ClientFinished`



TLS Handshake Protocol – `Finished` Messages

TLS `Finished` messages enable each side to check that both views of the Handshake Protocol are the same.

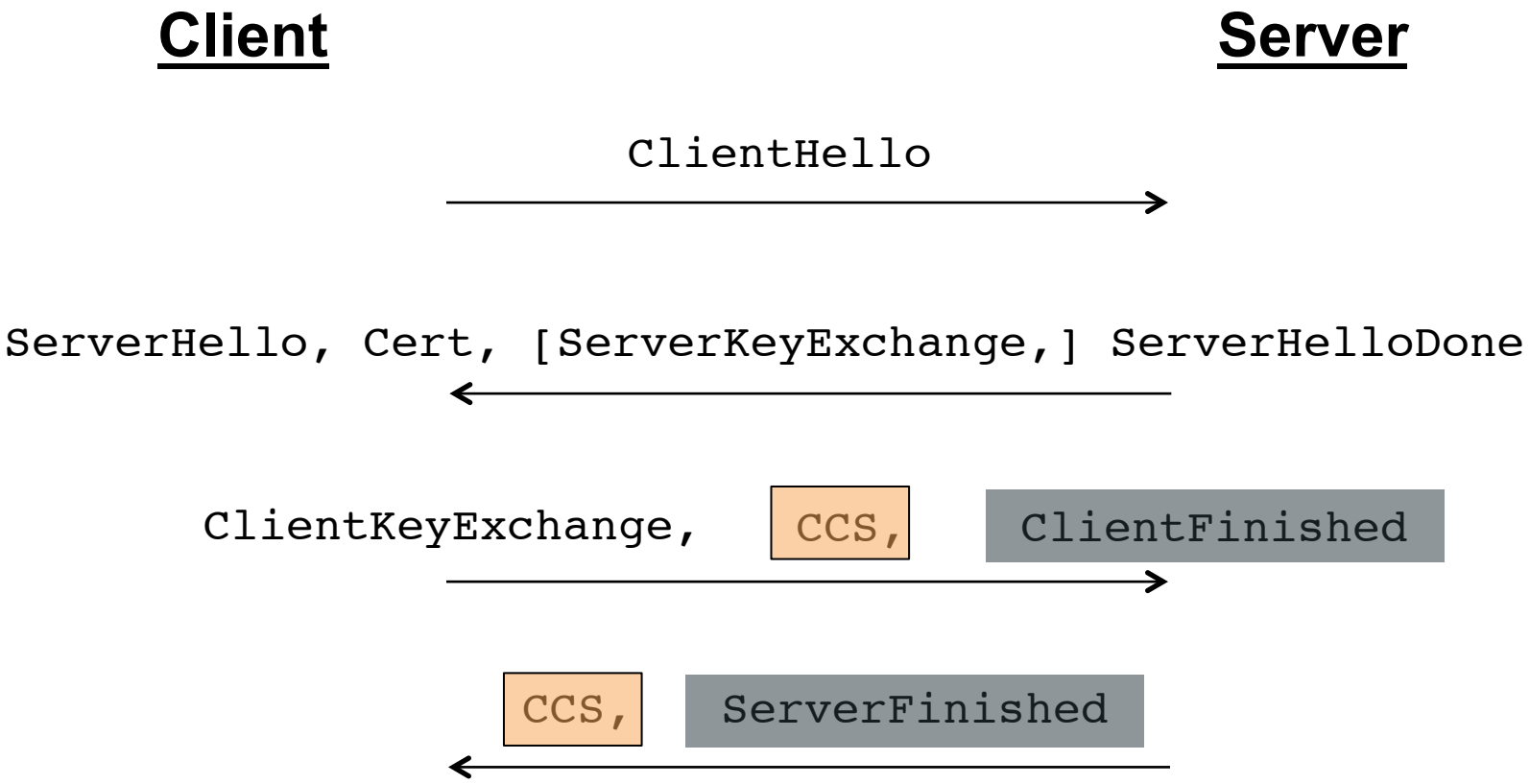
Computed as `PRF(ms,transcript)` where `transcript` = sender's view of all protocol messages sent and received up to *this* point.

Compared by recipient to expected value; protocol aborts if mismatch is observed.

Designed to prevent version rollback and ciphersuite downgrade attacks.

Ineffective if attacker can compute `ms` during protocol run.

TLS Handshake Protocol – ChangeCipherSpec



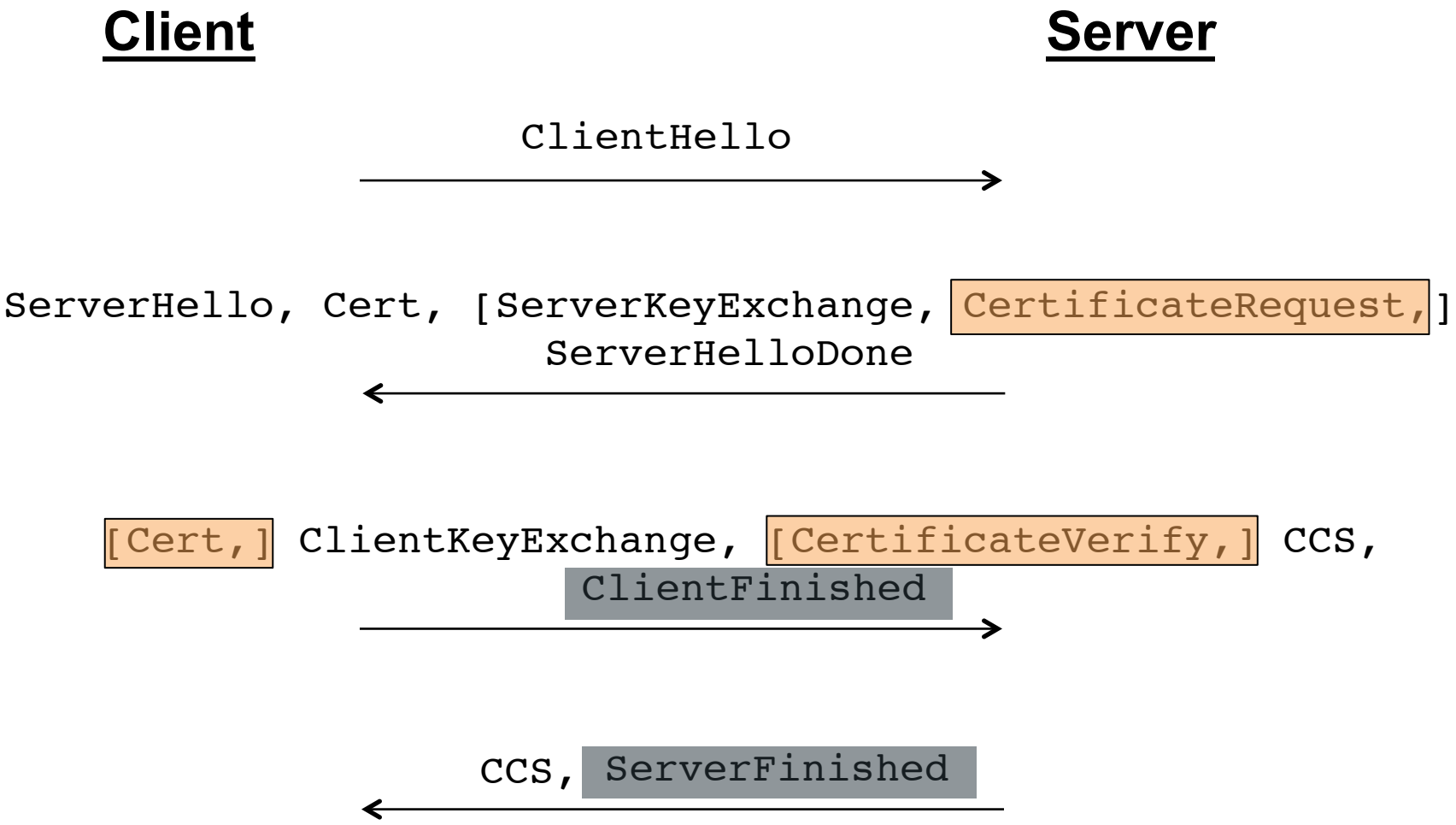
TLS Handshake Protocol – CCS Messages

`ChangeCipherSpec` messages enable parties to inform each other that they are switching to the recently agreed keys in the Record Protocol.

Here, this means that all subsequent messages are protected using the agreed ciphersuite (e.g. `AES_256_CBC_SHA256`).

Not part of the Handshake Protocol, so not included in transcripts.

TLS Handshake Protocol – Client Authentication



TLS Handshake Protocol – Client Authentication

Client authentication is optional and rarely used in the web setting.

Server requests client's certificate in its `Hello` message.

Client responds with:

- `Cert`: client's certificate (chain).
- `CertificateVerify`: signature on protocol transcript to this point.

TLS Handshake Protocol – Additional Features

The TLS Handshake Protocol also supports *renegotiation* and *session resumption*.

Renegotiation allows re-keying and change of ciphersuite during a session.

> For example, to force strong client-side authentication before access to a particular resource on the server is allowed.

Initiated by client sending `ClientHello` or server sending `ServerHelloRequest`.

> Followed by full run of Handshake Protocol.
>
> Over existing Record Protocol.

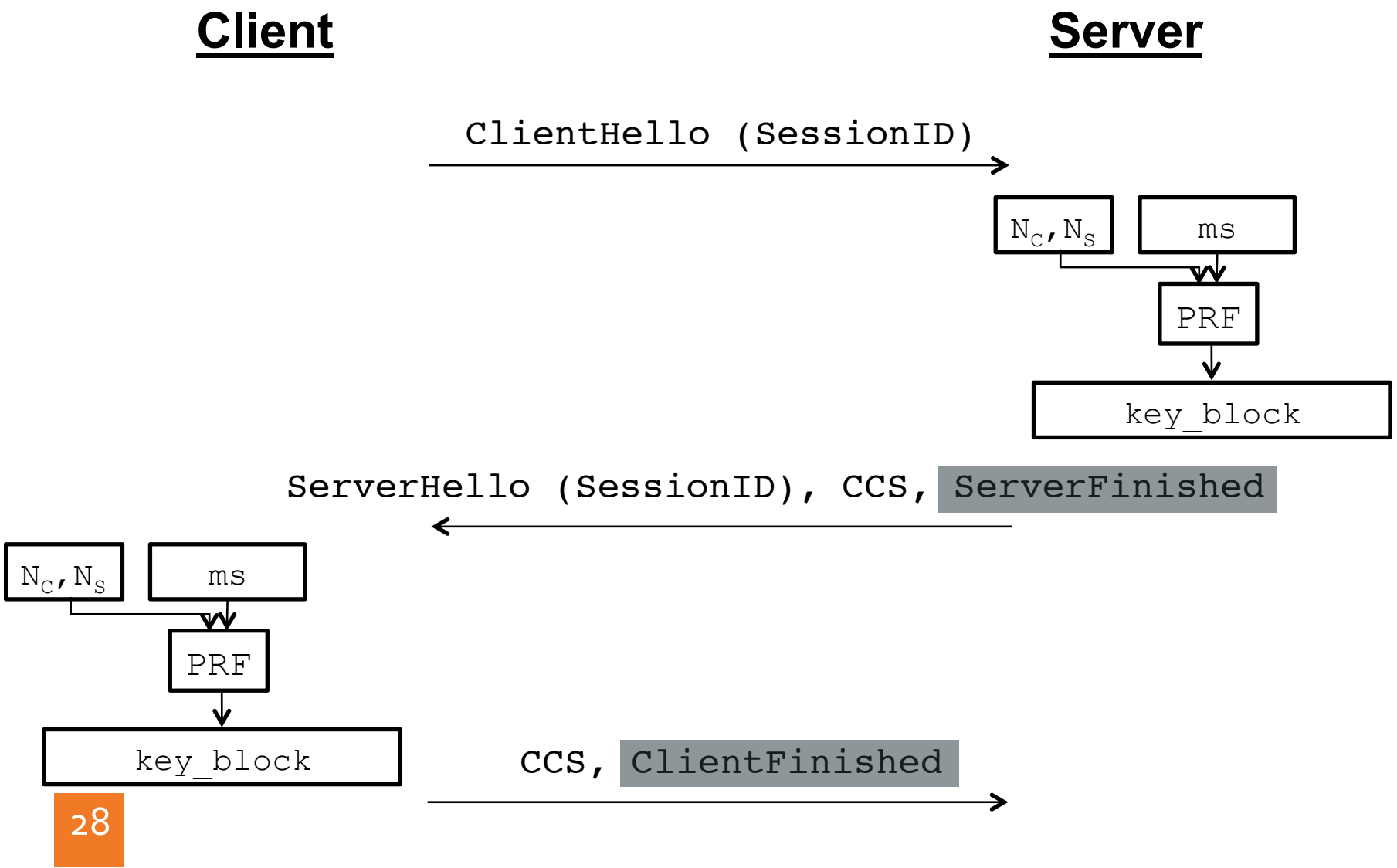
TLS Handshake Protocol – Session Resumption

Session resumption allows authentication and shared secrets to be reused across multiple, parallel *connections* in a single session.

E.g., allows fetching multiple resources from same website without re-doing full, expensive Handshake Protocol.

Client and Server quote existing `SessionID` and exchange fresh nonces.

TLS Handshake Protocol – Session Resumption



TLS Sessions and Connections

Session concept:

Sessions are created by the Handshake Protocol.

Session state defined by session ID and set of cryptographic parameters (encryption and hash algorithm, master secret, certificates) negotiated in Handshake Protocol.

Each session can carry multiple **parallel** *connections*.

Connection concept:

Keys for multiple connections are derived from a single `ms` created during one run of the full Handshake Protocol.

Session resumption Handshake Protocol runs exchange new nonces.

These nonces are combined with existing `ms` to derive keys for each new connection.

Avoids repeated use of expensive Handshake Protocol.

TLS Extensions

Many *extensions* to TLS exist.

Allows extended capabilities and security features.

Examples:

- Renegotiation Indicator Extension (RIE), RFC 5746.
- Application layer protocol negotiation (ALPN), RFC 7301.
- Authorization Extension, RFC 5878.
- Server Name Indication, Maximum Fragment Length Negotiation, Truncated HMAC, etc, RFC 6066.

TLS Handshake Protocol Complexity

The TLS Handshake Protocol is very complex:

- The protocol is "self-negotiating".
- There are many options and extensions.
- There are interactions with other protocols (CCS, Record Protocol, Alert Protocol).
- There is no clear state machine or API in the specification.

What could possibly go wrong?

A Selection of Attacks on the TLS Handshake Protocol

TLS Handshake Protocol Attacks

Up until 2009, the TLS Handshake Protocol survived relatively unscathed.

Notable exception: Bleichenbacher's attack on RSA encryption used in TLS (Crypto 1998).

- Exploits fact that RSA encryption scheme used (PKCS#1 v1.5) is not CCA secure.
- Recovers master secret for a target session using roughly 2^{20} interactions with server.
- Attack was addressed in TLS 1.0 via a specific countermeasure rather than change of scheme.

RSA Encryption in TLS

PKCS documents are a series of "standards" published by RSA.

PKCS#1 defines how to use RSA for encryption.

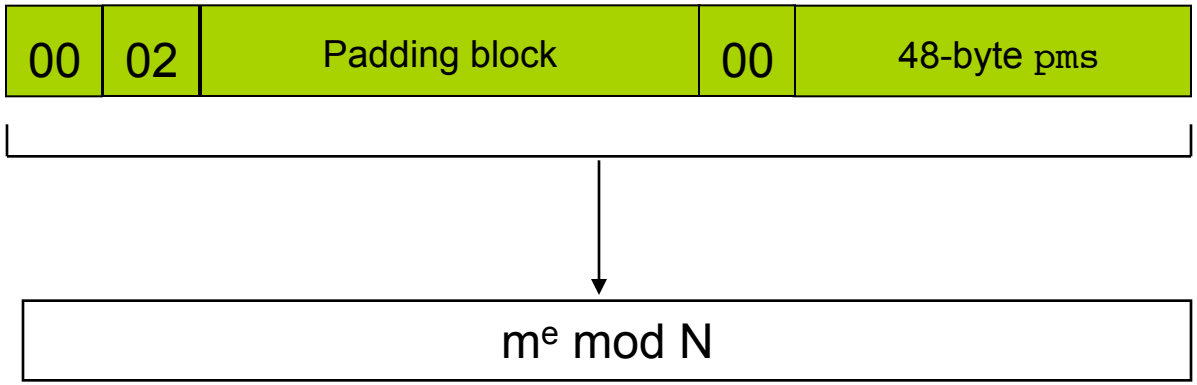
Version 1.5: November 1993, also RFC 2313.

Version 2.0: October 1998, also RFC 2427.

Version 2.1: February 2003, also RFC 3447.

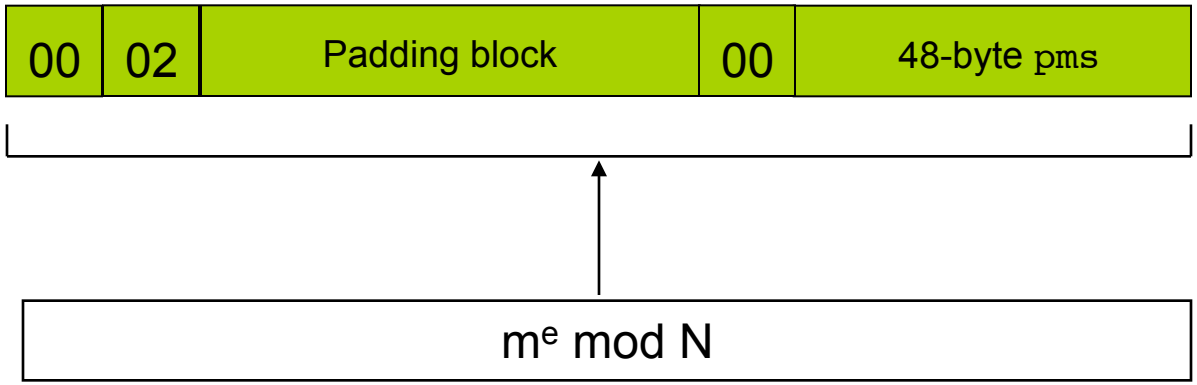
PKCS#1 version 1.5 is used in the TLS Handshake Protocol.

PKCS#1 v1.5, block type 2



- Plaintext must begin with "00 02" bytes.
- Padding block consists of *at least* 8 non-zero bytes.
- Should be terminated by "00" byte.
- Last 48 bytes are used as `pms`.
  - Additional complication: most significant two bytes are set to client TLS version.

PKCS#1 v1.5, block type 2



Think about sanity checking after decryption:

- Check for "00 02"?
- Check for at least 8 non-zero padding bytes or just *some* non-zero bytes?
- Check for a 00-byte? Or just extract last 48 bytes?
- Demand 00-byte to be in exactly the right position?
- Check for TLS version number?

Bleichenbacher's Attack

- Exact decryption processing is not specified.
- Different implementations exhibit different behaviours.

- Suppose that we have an oracle that on input *c* outputs whether $x := c^d \bmod N$ begins with byte pattern "00 02".

- If oracle output is "yes", then we have an inequality:

$$2B \leq x \bmod N < 3B$$

where $B = 2^{8(k-2)}$ and k is the number of bytes in modulus N.

Bleichenbacher's Attack

- Suppose attacker records c*, the RSA ciphertext encrypting the unknown `pms` for a target session.
- Attacker calls the "00 02" oracle on many, carefully selected inputs of the form $s^e c^* \bmod N$.
- Each "yes" output gives an inequality of the form:

$$2B \leq s \times \texttt{pms} \bmod N < 3B$$

  where s is known.

- By analysing responses from the oracle, the attacker can eventually reconstruct `pms`.
- Roughly 2^{20} oracle queries are needed.

Bleichenbacher's Attack

In the TLS context:

The required oracle is realised using error messages arising from server processing of attacker-generated `ClientKeyExchange` messages.

Countermeasures?

- Switch to using CCA-secure variant of RSA encryption, e.g. RSA-OAEP (cannot create "related" ciphertexts that are valid).
- Add protocol-specific countermeasures.

Bleichenbacher and TLS1.0 (1999)

TLS 1.0 was published in RFC 2246, Jan 1999, shortly after adoption of RSA-OAEP into PKCS#1v2.0.

TLS 1.0 still uses PKCS#1v1.5, despite Bleichenbacher's attack:

> *The best way to avoid vulnerability to this attack is to treat incorrectly formatted messages in a manner indistinguishable from correctly formatted RSA blocks. Thus, when it receives an incorrectly formatted RSA block, a server should generate a random 48-byte value and proceed using it as the premaster secret. Thus, the server will act identically whether the received RSA block is correctly encoded or not.*

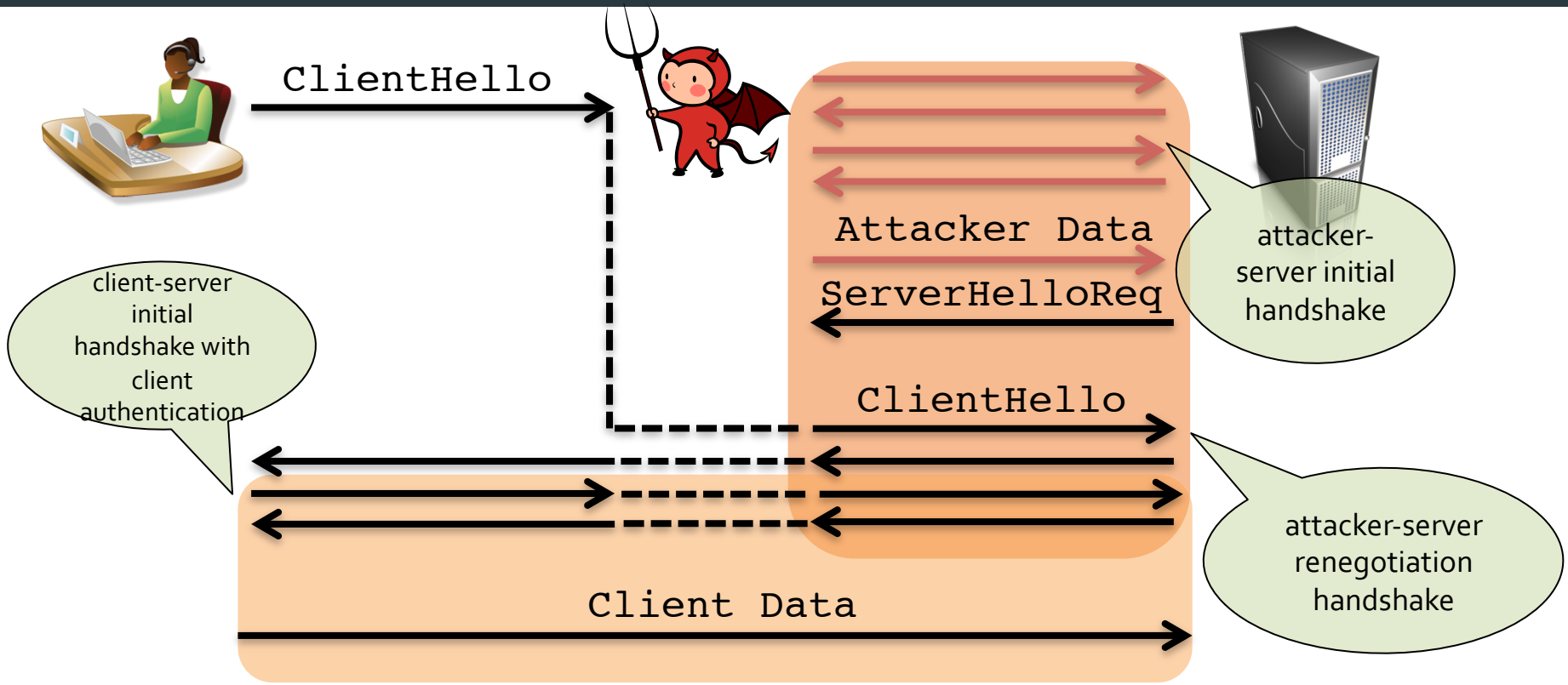
TLS 1.1, RFC 4346 (2006)

[PKCS1B] defines a newer version of PKCS#1 encoding that is more secure against the Bleichenbacher attack. However, for maximal compatibility with TLS 1.0, TLS 1.1 retains the original encoding. No variants of the Bleichenbacher attack are known to exist provided that the above recommendations are followed.

Over-optimistic: several implementations still get it wrong, and there's now a long literature of Bleichenbacher-style attacks against RSA implementations (not just in TLS).

Bardou et al. (Crypto 2012), Jager et al. (Esorics 2012),…

Renegotiation Attack (Ray and Dispensa, Rex, 2009)



Client view: single handshake, sends `ClientData`.

Server view: two handshakes, receives `AttackerData` | | `ClientData` from authenticated client.

Overall effect: attacker injects `AttackerData` as if from trusted source.

Renegotiation

- Renegotiation attack due to Ray and Dispensa, also Rex (2009).
- Server treats data as coming from either side of client authentication as being a single unit from an authenticated source.
- TLS specification does not really say how to handle this situation.
    - Flush buffer of received fragments upon renegotiation?
    - Signal to application that authentication status has changed?
    - Highlights lack of API specification for TLS.
- Attack addressed via Renegotiation Indication Extension (RIE), RFC 5746.
    - Include and verify information about previous handshakes in any renegotiation.
    - Could also disable renegotiation on server.

Triple Handshake Attack (Bhargavan et al, IEEE S&P 2014)

- Triple Handshake attack: renegotiation attack rebooted.

- Complex attack leveraging lack of identities in key derivation + resumption + renegotiation.
  - Even first step in the attack (UKS attack) breaks certain authentication protocols relying on TLS.

- Attack highlights that RIE fix for renegotiation attack is not robust in the context of the full TLS Handshake Protocol.
  - Renegotiation status gets lost across resumptions.

Certificate Processing Bugs

Many problems have been discovered in code for certificate processing.

- Fahl et al. (CCS 2012)
- Georgiev et al. (CCS 2012)
- GnuTLS bug (CVE-2014-0092)
- Apple goto fail (CVE-2014-1266)
  - Affecting Apple iOS 6.x before 6.1.6 and 7.x before 7.0.6, Apple TV 6.x before 6.0.2, and Apple OS X 10.9.x before 10.9.2.

Apple `goto fail`

```
SSLVerifySignedServerKeyExchange(SSLContext *ctx, bool isRsa, SSLBuffer signedParams,
                 uint8_t *signature, UInt16 signatureLen)
{
        OSStatus      err;
        ...

        if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
                goto fail;
        if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) !=
                goto fail;
                goto fail;
        if ((err = SSLHashSHA1.final(&hash                    HashOut)) != 0)
                goto fail;
        ...

fail:
        SSLFreeBuffer(&signedHashes);
        SSLFreeBuffer(&hashCtx);
        return err;
}
```

Causes all server signature processing on client to be bypassed!

Meaning that MITM attacker can trivially spoof ***any*** TLS server!

CCS Mishandling Bug (CVE 2014-0224)

OpenSSL implementation of TLS will accept `ChangeCipherSpec` message at any point in the TLS Handshake.

So MITM attacker can inject it at point of his choosing.

Result is that TLS key derivation is carried out with a zero-length master secret.

Leading to predictable session keys.

Cross-ciphersuite Attacks

Recall server signature format in `ServerKeyExchange`:

sig(nonces, params)

Format of params depends on type of key exchange: mod *p* DH parameters or ECDH parameters.

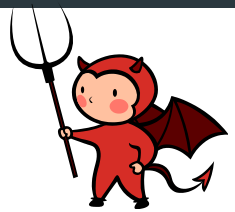
But *type* of parameters is not itself signed.

Instead, it's inferred by client from the ciphersuite, for which agreement is only verified via `Finished` messages.

Cross-ciphersuite Attack (Mavrogiannopoulos et al, CCS 2012)

ClientHello

TLS_DHE_RSA…



ClientHello'

TLS_ECDHE_RSA…

Server accepts TLS_ECDHE_RSA…

ServerHello,
Cert,
ServerKeyExchange

Changed by MITM back to TLS_DHE_RSA…

ServerHello',
Cert,
ServerKeyExchange

Contains server's ECDHE parameters and RSA signature on nonces and parameters*

ECDHE params are interpreted as being DHE params; RSA signature is valid!

With probability 2^{-40}, DHE params are parsed as having g = 0, 1 or -1!

*server needs to support "explicit prime curve" option, RFC 4492.

Cross-ciphersuite Attack (Mavrogiannopoulos et al, CCS 2012)

- Attack requires server to support "explicit prime curve" option (RFC 4492).
- Attack requires client to accept weak DH parameters (g = 0, 1 or -1).
  - Enabling MITM to compute `pms` and correct `ServerFinished` message to complete the handshake.
- Success rate can be boosted by repeatedly sending `ClientHello'` message within TLS timeout on client (tens of seconds).
- Attack possible because server signature does not cover type of ciphersuite, nor TLS extensions specifying use of ECC.

FREAK and LOGJAM Attacks

`EXPORT` ciphersuites:

```
0x000003    TLS_RSA_EXPORT_WITH_RC4_40_MD5
0x000006    TLS_RSA_EXPORT_WITH_RC2_CBC_40_MD5
0x000008    TLS_RSA_EXPORT_WITH_DES40_CBC_SHA
0x00000B    TLS_DH_DSS_EXPORT_WITH_DES40_CBC_SHA
0x00000E    TLS_DH_RSA_EXPORT_WITH_DES40_CBC_SHA
0x000011    TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA
0x000014    TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA
```

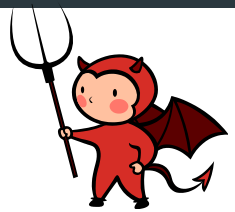
(and more)

- Introduced in the 90s in the era of export control.
- Maximum 512-bit RSA keys and 512-bit primes for DH/DHE.
- Repurpose `ServerKeyExchange` message to transport "ephemeral" RSA/DH/DHE keys.
- Until recently, still supported by around 25% of servers…

FREAK Attack (Beurdouche et al, IEEE S&P 2015)

ClientHello

TLS_RSA…



ClientHello'

TLS_RSA_EXPORT…

Server accepts TLS_RSA_EXPORT…

ServerHello,
Cert,
ServerKeyExchange

Changed by MITM back to TLS_RSA…

ServerHello',
Cert,
ServerKeyExchange

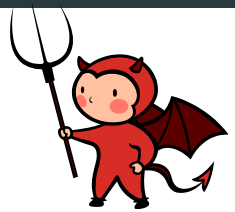
Buggy client processes this and accepts 512-bit RSA key for pms transport

Contains server's 512-bit RSA public key and RSA signature on nonces and parameters

FREAK Attack (Beurdouche et al, IEEE S&P 2015)

ClientHello

TLS_RSA…



ClientHello'

TLS_RSA_EXPORT…

ServerHello,
Cert,
ServerKeyExchange

ServerHello',
Cert,
ServerKeyExchange

ClientKeyExchange,
CCS,
ClientFinished

Attacker pre-factors 512-bit RSA key, and can now decrypt to get pms.

CCS,
ServerFinished

Attacker succeeds in impersonating server.

FREAK Attack (Beurdouche et al, IEEE S&P 2015)

- Attack relies on buggy clients accepting ServerKeyExchange containing 512-bit RSA key when no such message was expected.
  - Many clients were vulnerable (https://www.smacktls.com/).
- Export RSA keys are meant to be ephemeral, but hard to generate RSA moduli in practice, so made persistent.
- Cost of factoring 512-bit modulus: approximately $100 on Amazon EC2.
- Attack arises because of common code paths in implementations, coupled with state machine failures.
  - Explored in-depth in Berdouche et al paper.

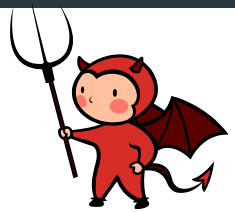
LOGJAM Attack (Adrian et al, 2015)

- LOGJAM = Cross-ciphersuite + FREAK.
  - Active attacker changes `TLS_DHE_RSA...` to `TLS_DHE_RSA_EXPORT...`
  - Server responds with weak DH parameters signed under its RSA key.
  - Client accepts these (signature does not include ciphersuite details).
  - Attacker solves 512-bit DLP before client times out.
  - Attacker can then create correct `ServerFinished` message to impersonate server.
- Difficult to perform in practice, but not impossible for three-letter agency.
  - Servers use small numer of common primes *p.*
  - Precomputation allows each 512-bit DLP to be solved in around 90s.

LOGJAM Attack (Adrian et al, 2015)

ClientHello

TLS_DHE_RSA…



ClientHello'

TLS_DHE_RSA_EXPORT…

ServerHello,
Cert,
ServerKeyExchange

Contains server's 512-bit **DHE parameters** and RSA signature on nonces and parameters

ServerHello',
Cert,
ServerKeyExchange

Attacker uses x and g^y to compute pms

ClientKeyExchange
(g^y), CCS,
ClientFinished

Attacker solves DLP for g, g^x to compute server's private value x .

CCS,
ServerFinished

Attacker succeeds in impersonating server.

Security Proofs for TLS Handshake Protocol

- Models for analysis of key exchange protocols are fairly mature.
  - Beginning with [BR93].

- But they are also quite complex.
  - Multiple, interacting parties.
  - Multiple sessions at each party.
  - Adversary given complete control of the network ("adversary is the network").
  - Adversarial access to various session keys, long-term secrets, ephemeral values, randomness.
  - Forward security?
  - Corruption of parties?
  - Dishonest long-term key pair generation?
  - Modeling of CA/PKI?
  - Authentication and key security properties

Security Proofs for TLS Handshake Protocol

Additional barriers to analysis for TLS:

- Protocol not designed with provable security in mind.
- Protocol version and ciphersuite are negotiated during the protocol itself and verified later via Finished messages.
- Different methods for authentication and key exchange in one protocol.
- Unilateral and mutual authentication modes
- Renegotiation and session resumption features.
- The session key is used in the Handshake Protocol itself (so can't prove usual "indistinguishability of session keys" property).
- RSA encryption used in TLS is not IND-CCA secure.
- Cross-ciphersuite attacks.
- Static DH harder to analyse then usual ephemeral DH because server acts as "fixed base DH oracle" by calculating and using g^{xy} for fixed g, g^x.
- What to leave out and what to include when modelling the protocol?

Security Proofs for TLS Handshake Protocol

[MSW08]:

- Analysis of model of RSA-based key exchange in TLS
- Assumed RSA encryption to be CCA-secure.

[JKSS12]:

- ACCE security notion, combining security of Handshake and Record Protocols.
- Overcomes "indistinguishability barrier" from use of session key in Handshake Protocol.
- Analysis of cryptographic core of single *mutually authenticated ephemeral DH key exchange* ciphersuites.
- Using a fairly faithful model of the core TLS Protocol.

Security Proofs for TLS Handshake Protocol

[KPW13]:

- Adapt ACCE to unilateral case.
- Analysis of cryptographic core of individual (*unilateral and mutually authenticated) RSA, static DH and ephemeral DH key exchange* ciphersuites.
- Modular approach:
  - Extract Key Encapsulation Mechanism (KEM) from Handshake Protocol.
  - (S)ACCE security of TLS follows from constrained CCA security of this KEM.
  - Analyse the KEM for various ciphersuites.

[GKS13]:

- Formal security treatment of renegotiation and Ray-Dispensa-Rex attack.

Security Proofs for TLS Handshake Protocol

[BFKPS13, BFKPSZ14]:

- Develop reference implementation of TLS in F#/F7: MITLS.
- Complete implementation of basic Handshake Protocol, plus renegotiation, and resumption.
- Implementation of Record Protocol and Alert Protocol.
- Encoding of security properties via typing, enabling formal verification of security down to computational assumptions, using a type checker.
- Some code modules idealised.
- Interoperable and quite fast.
- Used several elements to discover Triple Handshake, FREAK and LOGJAM attacks.

TLS 1.3

Main objectives:

- Reduce latency (0-RTT key establishment).
- Protocol simplification (reducing options and removing broken ciphersuites).

Two main Handshake Protocol options under consideration:

- Signed Diffie-Hellman.
- OPT-TLS (implicit DH).

Concluding Remarks

- The TLS Handshake Protocol is extraordinarily complex.
  - Much more so than typical key exchange protocols appearing in the literature.
  - Some of that complexity is necessary.
  - Some of it stems from a desire for flexibility.
  - All of it creates a challenge for analysis (and opportunities for finding attacks).
  - The gap between theoretical analyses and real-world security is closing.
- Some **design** errors were made, but not many.
- Desire to support **legacy** features has been the source of problems.
- Lack of formal state-machine description, lack of API specification, and sheer complexity of specifications have led to many serious **implementation** errors.