6 Transport Layer Security Protocol

6.1 Internet Security Protocols

Authenticated key exchange protocols are at the core of Internet security protocols: they authenticate one or more of the parties communicating, and provide the establishment of a session key that is then used to encrypt application data. There are several protocols in widespread use to secure various applications. The most prominent are the following:

Transport Layer Security (TLS). Formerly known as the *Secure Sockets Layer* (SSL) protocol. The TLS standards are developed and maintained by the Internet Engineering Task Force (IETF) TLS working group. TLS operates over the TCP/IP protocol stack and is used to protect web traffic (using HTTPS), file transfers, email transport, and many other applications. To date there have been two versions of SSL (SSL v2 and SSL v3) and three versions of TLS (TLS 1.0, TLS 1.1, and TLS 1.2); the next version, TLS 1.3, was submitted for standardisation in March 2018. A variant called *Datagram TLS* (DTLS) is used for protection of datagrams transmitted over UDP. See Table 6.1 for a list of versions of TLS and related protocols.

Secure Shell (SSH). SSH operates over the TCP/IP protocol stack and is primarily for used for securing remote command-line logins, replacing the insecure Telnet protocol. SSH can also be used for file transfer (secure copy (scp)), as well as a rudimentary virtual private network. The current version, SSH v2, is incompatible with the prior SSH v1.

Internet Protocol Security (IPsec). The IPsec protocol suite operates at the IP layer of the IETF protocol stack, and is automatically applied to all application data above it. It can be used in two modes: *transport mode* authenticates and encrypts the contents of an IP packet, but leaves the headers unchanged; and *tunnel mode* authenticates and encrypts an entire IP packet, including the headers, and then encapsulates that as the payload of a new IP packet. IPsec is typically run in one of three architectures: host-to-host (directly securing a connection between two computers), host-to-gateway (for example, a remote user

C. Boyd et al., *Protocols for Authentication and Key Establishment*, Information Security and Cryptography, https://doi.org/10.1007/978-3-662-58146-9_6

Table 6.1: Versions of SSL/TLS and Datagram TLS (DTLS)

SSL v2	1995	[355]
SSL v3	1996	[283]
TLS 1.0	1999	[249]
TLS 1.1	2006	[250]
TLS 1.2	2008	[251]
DTLS 1.0	2006	[627]
DTLS 1.2	2012	[628]

connecting to a corporate network via a virtual private network), and gateway-to-gateway (for example, a gateway connecting all computers in a branch office to the head office).

There are several other special-purpose Internet protocols that make use of authenticated key exchange, including the Tor anonymity network and secure instant messaging protocols such as Off-the-Record (OTR) messaging, and the Axolotl ratchet/Signal protocol. Appendix A summarizes many of these.

In this chapter, we examine the Transport Layer Security protocol in detail. TLS is interesting owing to its widespread use, the complexities in its design compared with academic key exchange protocols, and the many weaknesses and flaws found in the protocol and its implementations. TLS also differs from academic key exchange protocols in the sense that it directly combines key exchange and authenticated encryption to establish a secure channel.

6.2 Background on TLS

The Secure Sockets Layer protocol was developed by Netscape in the mid-1990s. The first publicly released protocol was SSL v2 in February 1995 [355]. A redesign, aiming to fix several security flaws, was published in November 1996 (and published as a historical RFC in August 2011 [283]). In January 1999, the Internet Engineering Task Force published the TLS 1.0 protocol [249], which made minor changes to SSL v3. In April 2006, the IETF published TLS 1.1 [250], primarily making changes in how the CBC block cipher encryption mode worked. TLS 1.2 was published in August 2008 [251], incorporating changes to the PRF, support for authenticated encryption with additional data, and more precise negotiation of algorithms. There are more than 45 additional RFCs that describe additional behaviour or functionality; notable ones include the specification of elliptic curve cryptography [106] and pre-shared keys [269], the addition of new ciphers such as AES [213], extensions to the protocol specification [112], and deprecation of old algorithms [55, 619].

TLS is used to protect many applications. It is most familiar to many when it is used to protect web traffic transmitted over the Hypertext Transport Protocol (HTTP). In this context, called HTTPS, an SSL/TLS connection is established (typically on TCP port 443, different from port 80 for the unsecured website), and then

HTTP data is transmitted across the secured connection. TLS can also be used to protect email transport protocols (IMAP and POP, for clients to download messages from mail servers, and SMTP, for delivery of outgoing messages), as well as file transfer (FTP). In these contexts, the unsecured connection is ‘upgraded’ to a secure connection: first, the normal unsecured connection is established, and then a special command (such as ‘STARTTLS’) is used to activate TLS, at which point a TLS connection will be established, and all subsequent data will be transmitted over the TLS connection.

6.3 Protocol Structure

The TLS protocol consists of several subprotocols; for cryptographic purposes, the two most important subprotocols are the *handshake protocol* and the *record layer protocol*. In the handshake protocol, the client and server agree on a set of cryptographic parameters, called a *ciphersuite*, exchange authentication credentials, establish a shared secret, perform explicit authentication, and derive keys for bulk encryption and message authentication. The record layer protocol provides delivery of all messages in TLS, including handshake protocol messages and application data, but in particular the record layer protocol optionally protects messages using authentication and encryption. There is an additional *alert protocol*, which is used to notify peers about errors or to close the connection. The stacking of these layers is shown in Fig. 6.1. The exact cryptographic algorithms used in both the handshake and the record layer protocol depend on which ciphersuite the parties have negotiated.

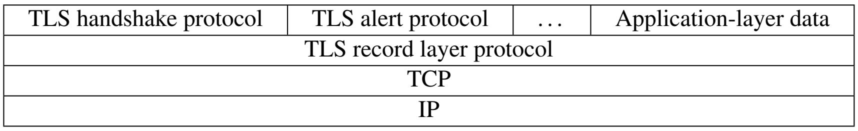


Fig. 6.1: Layering of TLS subprotocols and the TCP/IP stack

TLS can provide entity authentication using long-term public keys using the X.509 public key infrastructure [223]. Authentication can be either mutual or only server-to-client. Parties generate long-term secret key/public key pairs, then submit their public key and a proof of possession (usually a signature using the key) via a certificate-signing request to certification authority (CA). The CA verifies the certificate-signing request and the identity of the requesting party, then issues a certificate containing the party’s identity (in the case of servers, this is the server’s fully qualified domain name), and public key, as well as additional fields such as the period of validity and the purposes for which the certificate can be used; the certificate is signed by the CA using its long-term key. Web browsers typically have 150 or more certificates for 80 or more commercial and governmental root CAs installed by default; however, many root CAs allow independent subordinate CAs to also issue certificates, so the exact number of certificate issuers that are trusted by default by browsers is unknown even to browser vendors. Taking into account these subordinate issuers, over 650 certificate issuers trusted by default by major browsers have been observed in Internet-wide surveys of TLS server certificates [266].

6.3.1 Handshake Protocol

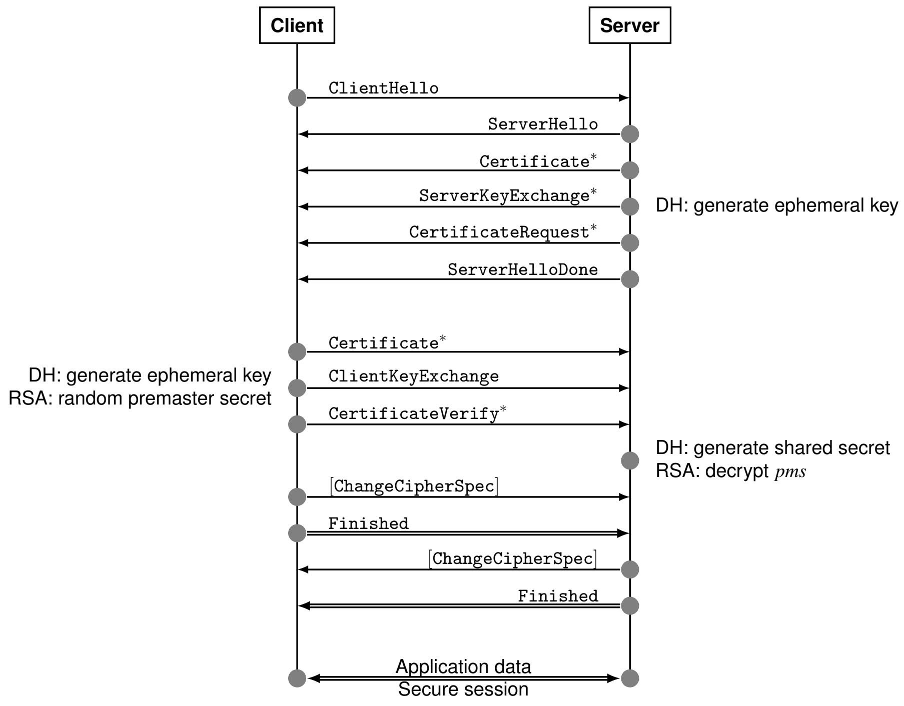
To first establish a TLS connection, a client and a server run the full TLS handshake protocol. As shown in Protocol 6.1, the TLS handshake protocol proceeds as follows.

1. The client initiates the connection by sending a `ClientHello` message, which contains a nonce as well as the client's list of preferred ciphersuites.
2. The server responds with several packets together:
   - `ServerHello` message, containing a nonce and the chosen ciphersuite;
   - `Certificate` message, if server authentication is being used;
   - `ServerKeyExchange` message containing the server's ephemeral Diffie–Hellman value, if the ciphersuite calls for it;
   - `CertificateRequest` message, if the server is asking for client authentication.
3. The client verifies the server's certificate, then responds with:
   - `Certificate` message, if client authentication is being used;
   - `ClientKeyExchange` message, which the parties use to compute the session key;
   - `CertificateVerify` message, if client authentication is being used.

   At this point, the client computes a *premaster secret*, which is the raw shared secret. Using the premaster secret and the client and server nonces, the client then computes the *master* secret, from which it derives four record-layer session keys: two keys for bulk encryption (client-to-server and server-to-client), and two keys for message authentication (client-to-server and server-to-client).
4. The client sends a `ChangeCipherSpec` message, which indicates that all further messages will be sent encrypted using the record layer protocol. Finally, the client sends (encrypted by the record layer) a `Finished` message, containing a key confirmation value.
5. The server computes the premaster secret and master secret, then derives the session keys. It verifies the client authentication, if any, then verifies the `Finished` message it has received from the client.
6. The server sends a `ChangeCipherSpec` message, which similarly indicates that all further messages will be sent encrypted using the record layer protocol. Finally, the server sends (encrypted by the record layer) its own `Finished` message for key confirmation.

This concludes the handshake protocol, and application data can now be sent using the record layer protocol.

If the client and server have previously established a session, they can perform an abbreviated handshake for *session resumption*. In this case, the message flow is as



* denotes messages that may not be present in all ciphersuites.
[...] denotes messages that are sent over the TLS alert protocol.
Single arrows denote plaintext flows; double arrows denote encrypted flows.

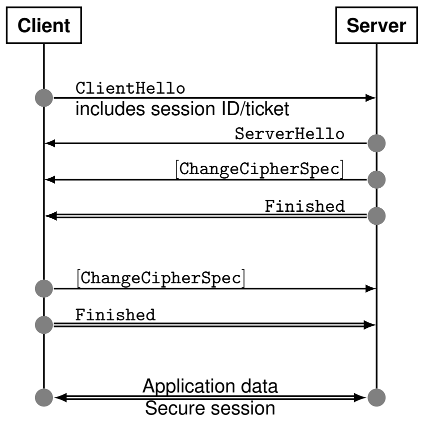
Protocol 6.1: TLS $\leq$ 1.2 handshake protocol – full handshake

shown in Protocol 6.2. The abbreviated handshake allows re-establishment of a TLS record layer in a single round trip, rather than two round trips as in the full handshake protocol; it also avoids some expensive public key operations.

We now examine each message of the handshake protocol in detail.

`ClientHello`. With this message, the client initiates the connection. The primary purpose of the message is to convey the client's parameter preferences. The message includes:

- a protocol version field, indicating the maximum supported version of SSL or TLS;
- the `client_random` value, a 32-byte nonce;
- the `session_id` of a previous session, if session resumption is being used;
- a list of the client's supported cryptographic parameters in order of preference;
- a list of the client's supported compression methods in order of preference; and



Protocol 6.2: TLS $\leq$ 1.2 handshake protocol – abbreviated handshake

- any optional extensions, indicating additional information.

TLS 1.0 and TLS 1.1 were updated to support extensions. Frequently used extensions include the server name indication extension, which helps a TLS server hosting multiple domains to pick the appropriate certificate to send to the client; the elliptic curves extension, indicating which elliptic curves the client supports; and the renegotiation indication extension, which help protects against the renegotiation attack (see Sect. 6.11.2).

`ServerHello`. With this message, the server indicates which parameters have been chosen. The message includes:

- a protocol version field, indicating which protocol version will be used for this connection;
- the `server_random` value, a 32-byte nonce;
- the `session_id` of this session, for the purposes of future session resumption;
- the chosen ciphersuite;
- the chosen compression method; and
- any optional extensions.

The `client_random` and `server_random` nonces serve to provide freshness or liveness guarantees and prevent replay attacks.

`Certificate` (server). For all server-authenticated ciphersuites which involve certificates, the server also sends a message containing its certificate. This message is structured as a chain of X.509 certificates: the first certificate is the server's certificate, and each subsequent certificate in the chain is the certificate of the certification authority that signed the preceding certificate. The last (root) certificate may be omitted, as the client must have that certificate installed already in order to trust it. Upon receiving this message, the client validates the certificate chain by (a) checking that the subject of the server's certificate matches the domain name of the server with which it is communicating, (b) verifying the signature of each certificate under the public key of the next certificate in the chain, and (c) checking the validity period of the certificate. Optionally, the client may check the revocation status of the certificate using either *certificate revocation lists* (CRLs) or the *online certificate status protocol* (OCSP). Servers can include a recent OCSP response in the handshake in a process known as *OCSP stapling*.

`ServerKeyExchange`. This message is sent if the chosen ciphersuite calls for the server to send an ephemeral public key, which is the case for ephemeral Diffie–Hellman ciphersuites, but not for static Diffie–Hellman or RSA key transport ciphersuites. For signed-Diffie–Hellman ciphersuites, the structure of this message is:

- the server Diffie–Hellman parameters and ephemeral public key:
  - for finite-field Diffie–Hellman, this contains the prime modulus p, the generator g, and the server's ephemeral public key $Y \equiv g^y \mod p$;
  - for elliptic curve Diffie–Hellman, this contains the chosen elliptic curve, either as a named curve or as the explicit parameters (field, curve, generator, order, cofactor), and the server's ephemeral public key $Y = yP$; and
- the server's signature over the `client_random`, `server_random`, and Diffie–Hellman parameters.

In SSL v3, the `ServerKeyExchange` message would be sent for RSA key transport ciphersuites if the server's long-term key was solely a signature key. In TLS 1.0, this was removed, except for export ciphersuites where the server's long-term key was not sufficiently long, and as of TLS 1.1 was removed entirely.

`CertificateRequest`. The server sends this message if it requires the client to authenticate itself using a certificate. The message includes:

- a list of supported client certificate types;
- a list of supported signature types supported for certificate verification; and
- a list of acceptable certificate authorities.

`Certificate` (client). If the server has sent a `CertificateRequest` message, then the client responds with its certificate; this message has the same structure as the server's `Certificate` message. Upon receipt of this message, the server verifies the validity of the client's certificate as above.

`ClientKeyExchange`. In the full handshake, this message is always sent by the client to establish a premaster secret. The structure of the message depends on the ciphersuite chosen.

- For RSA-key-transport-based ciphersuites, this message contains the encrypted premaster secret. In particular, the client chooses a random 46-byte value, and, together with the two bytes of `client_version`, these 48 bytes comprise the *premaster secret*. The client encrypts the premaster secret under the server's RSA public key (from the server's certificate) using PKCS#1v1.5 encryption. These ciphersuites do not provide forward secrecy.
- For finite-field or elliptic curve ephemeral Diffie–Hellman ciphersuites, this message contains the client's ephemeral public key. In this case the premaster secret is the Diffie–Hellman shared secret. These ciphersuites provide forward secrecy.

- For static Diffie–Hellman ciphersuites, this message is empty.

Upon receipt of this message, the server derives the premaster secret. In the case of RSA key transport, the server must carefully implement the PKCS#1v1.5 decryption process to avoid a side-channel leak (see Sect. 6.9.1). Both parties derive the master secret from the premaster secret as specified in the subsequent subsection. Finally, encryption and authentication keys are derived from the master secret.

`CertificateVerify`. This message is sent if the client has sent a certificate. The message is computed as the client's RSA, DSA, or ECDSA signature on all handshake messages it has sent and received up to, but not including, this message. Upon receipt of this message, the server verifies the signature.

`ChangeCipherSpec` (client). The client sends this single-byte message to the server, indicating that all future messages it sends will be encrypted and authenticated. Note that, for networking reasons, this message is technically not part of the handshake protocol but a separate subprotocol.

`Finished` (client). This message is sent by the client to verify that the key exchange and entity authentication were successful. Since it includes authentication of the handshake messages, from a cryptographic perspective the `Finished` message allows the other party to verify that its peer had the same view of the handshake and that no downgrade attacks occurred. The message contains

$$PRF\left(ms, label \| H(handshake)\right),$$

where $label =$ "client finished" and *handshake* is the transcript of all handshake messages sent and received by the party; H is the hash function specified by the ciphersuite. Upon receipt of this message, the server compares the received value with its own computed value. If the values match, the server *accepts*.

`ChangeCipherSpec` (server). The server sends this single-byte message to the client indicating that all future messages it sends will be encrypted and authenticated.

`Finished` (server). The server sends a `Finished` message similar to the client's one above, but computed with $label =$ "server finished". Upon receipt of this message, the client compares the received value with its own computed value. If the values match, the client *accepts*.

Cryptographic Computations in the Handshake Protocol

PRF. In TLS 1.0 and 1.1, the pseudo-random function *PRF* is computed as

$$PRF(secret, label, seed) = P_{\mathrm{MD5}}(S_1, label\|seed) \oplus P_{\mathrm{SHA\text{-}1}}(S_2, label\|seed),$$

where S_1 and S_2 are the first and last $|secret|/2$ bytes of *secret* (possibly with a shared middle byte), and

$$\begin{aligned} P_H(secret, seed) = {} & \mathrm{HMAC}_H(secret, A(1)\|seed) \\ & \| \ \mathrm{HMAC}_H(secret, A(2)\|seed) \ \| \ \ldots \end{aligned}$$

for $A(0) = seed$, $A(i) = \text{HMAC}_H(secret, A(i-1))$. In TLS 1.2, the PRF is computed as

$$PRF(secret, label, seed) = P_H(secret, label \| seed)$$

where H is a hash function defined by the ciphersuite; for most ciphersuites defined by TLS 1.2, H is SHA-256.

Master secret. In SSL v3, the 48-byte master secret *ms* is computed from the premaster secret *pms* as

$$ms \leftarrow f(\text{“A”}) \| f(\text{“BB”}) \| f(\text{“CCC”}),$$

where

$$f(\ell) \leftarrow \text{MD5}(pms \| \text{SHA-1}(\ell \| pms \| \texttt{client\_random} \| \texttt{server\_random})).$$

In TLS 1.0 and onward, the 48-byte master secret *ms* is computed from the premaster secret *pms* as

$$ms \leftarrow PRF(pms, \text{“master secret”}, \texttt{client\_random} \| \texttt{server\_random}),$$

where *PRF* is defined as above.

Encryption and authentication keys. In TLS 1.0 and higher, the parties derive encryption and authentication keys from the master secret as follows. First, they compute a sufficiently long value

$$k \leftarrow PRF(ms, \text{“key expansion”} \| \texttt{server\_random} \| \texttt{client\_random})$$

and then partition k into the client-to-server MAC write key, the server-to-client MAC write key, the client-to-server encryption key, the server-to-client encryption key, and, if required, the client-to-server and server-to-client encryption initialisation vectors (IVs). Note that keys are derived slightly different in SSL v3, and also for weakened export ciphersuites.

6.3.2 Record Layer Protocol

A plaintext packet in the record layer protocol consists of:

- a *content type*, which specifies what type of data the payload contains; this may be a handshake message, a `ChangeCipherSpec` message, an alert (error) message, or application data;
- the *version* of TLS used;
- the *length* of the payload, not more than 2^{14};
- the payload data.

Once the plaintext packet is assembled, its payload is *compressed* using whichever compression algorithm was negotiated during the handshake. Supported compression algorithms include the null algorithm (i.e. no compression) and the `DEFLATE` algorithm.

The compressed payload is finally encrypted and authenticated based on the ciphersuite in use. There are three different approaches.

Stream-cipher-based ciphersuites. First, an authentication tag is computed using a message authentication code as

$$\mathsf{MAC}(\texttt{msg\_write\_key}, sn \| \texttt{content\_type} \| \texttt{version} \| len \| m),$$

where `msg_write_key` is the client-to-server or server-to-client MAC write key as appropriate, *sn* is the sequence number of the packet, *m* is the (optionally compressed) payload, and *len* is the length of the payload. The payload is concatenated with the MAC tag and then that plaintext is encrypted using the stream cipher in question. RC4 was the most widely used stream cipher in TLS; since it does not use an initialisation vector, the internal RC4 state persists across packets.

Block-cipher-based ciphersuites. TLS specifies the use of the cipher-block chaining (CBC) mode of operation for block ciphers. First, an initialisation vector (IV) is chosen for the packet. The computation of the IV depends on the TLS version: for SSL v3 and TLS 1.0, the IV of the first packet is the IV derived in the handshake protocol, while the IV of each subsequent packet is the last ciphertext block of the previous packet. For TLS 1.1 and higher, the IV is chosen at random. Next, a MAC is computed as described in the stream-cipher item above. The (optionally compressed) payload and MAC are concatenated, then padded with sufficient bytes to bring the length up to a multiple of the block length; all padding bytes should be the same and should be the padding length value (for example, if 12 bytes of padding are needed, then every byte of padding contains the value $\texttt{0x0C} = 12$). This data is then encrypted using the block cipher in question in CBC mode. This sequence of operations is sometimes referred to as 'MAC-then-encode-then-encrypt' (MEE). The most widely supported block cipher is AES; the use of DES is deprecated, and the use of Triple-DES is no longer recommended.

Authenticated-encryption-based ciphersuites. TLS 1.2 added support for *authenticated encryption with associated data* (AEAD). In these modes, a single algorithm is used for encryption and authentication of packets. The 'additional data' field is used to convey unencrypted data unauthentically; in the case of TLS, the additional data is the same as the non-message data in the MAC computation in stream-cipher-based ciphersuites: the sequence number, content type, version, and plaintext length. There are two supported block cipher modes of operation that provide AEAD: *counter mode with CBC-MAC* (CCM) and *Galois/counter mode* (GCM), both generally implemented using AES. Since these modes provide encryption and authentication using a single algorithm, only the encryption keys, not the MAC keys, are required.

6.4 Additional Functionality

In addition to the core cryptography task of mutual entity authentication and establishment of a secure channel, the TLS protocol performs several other operations, which are described in this section.

6.4.1 Compression

The TLS record layer protocol allows application data to be compressed. In the ClientHello and ServerHello messages, the parties negotiate whether or not to enable the DEFLATE compression algorithm [246, 362] (a combination of the LZ77 algorithm and Huffman coding, widely used in the ZIP and gzip formats).

The record layer protocol splits application data up into *fragments* of at most 2^{14} bytes. If compression is enabled, then each plaintext fragment is independently compressed using the DEFLATE algorithm before being encrypted. The receiver reverses the process, decrypting and then decompressing.

Because the amount of compression on a plaintext yields information about the amount of redundancy in the plaintext, TLS compression acts as a side channel, leaking some information about the plaintext content. This leads to a successful adaptive chosen plaintext attack against TLS that allows an attacker to recover a secret value (such as an HTTP cookie) by exploiting differences in the amount of compression. Details of the CRIME attack and related attacks appear in Sect. 6.11.3.

6.4.2 Session Resumption

Session resumption allows a client and server to use an *abbreviated handshake* to resume a previously established session. This saves both communication—using 1.5 round trips instead of 2.5—and computation, as generally no expensive public key operations are needed in session resumption.

There are two distinct mechanisms for session resumption.

- *Session IDs* [251, Sect. F.1.4]. When the initial session is established, the server includes in its ServerHello message a session identifier. The server stores the session's parameters and master secret in its cache. To resume a session, the client replays that session identifier in its ClientHello message. The server looks up the session in its cache. If the session is found, the server immediately sends the Finished message calculated using the stored master secret.
- *Session tickets* [649]. When the initial session is established, the server includes an additional handshake message just prior to ChangeCipherSpec: the NewSessionTicket message contains the server's state (including its master secret), encrypted and authenticated under a symmetric key known only to the server. To resume a sesion, the client replays that session ticket in its ClientHello message. The server decrypts the encrypted state and verifies its integrity; if it passes, the server immediately sends the Finished message, calculated using the master secret from the session ticket.

The main difference between session IDs and session tickets is about who stores state: with session IDs, the server stores state for each connection; with session tickets, the client stores the server's state for it. Session tickets reduce storage requirements for the server; as well, a collection of load-balancing servers can easily resume each others' sessions simply by sharing session ticket encryption keys, rather than needing to pool session ID states.

In 2014, Bhargavan *et al.* [96] discovered the *triple handshake attack*, in which an attacker performs a man-in-the-middle attack over three successive handshakes (with various combinations of session resumption and renegotiation), eventually leading to a successful client impersonation attack. Among the causes of the attack is the fact that TLS session resumption does not cryptographically bind the previous session's handshake to the new session's handshake. Details of the attack and countermeasures appear in Sect. 6.11.5.

6.4.3 Renegotiation

Renegotiation allows two parties who are using an existing TLS session to run a new handshake; upon completion of the new handshake, the session uses the newly established encryption and authentication keys for communication. Renegotiation allows parties to either (a) obtain a fresh session key, (b) change cryptographic parameters (such as to negotiate a new ciphersuite), or (c) change authentication credentials. The renegotiated handshake is run *inside* the existing encrypted record layer.

A principal use case of TLS renegotiation is client identity privacy. In the handshake protocol, the client sends its certificate in plaintext. If the client does not wish to reveal its identity over a public channel, it can instead run the first handshake anonymously, then renegotiate using its long-term credential; since the handshake messages for the renegotiation are transmitted within the existing record layer, the transmission of the client certificate is encrypted and the client has privacy of its identity.

Either the client or the server can initiate renegotiation at any time after a session is established. The client triggers renegotiation simply by sending a new `ClientHello` message. The server triggers renegotiation by sending a `HelloRequest` message which asks the client to send a new `ClientHello` message. Renegotiation is supported in SSL v3 and higher.

In 2009, Ray and Dispensa [623] described an attack against some applications supporting TLS renegotiation. As a consequence of the attack, two countermeasures were standardised. Details of the attack and countermeasures appear in Sect. 6.11.2.

6.5 Variants

Versions. There are currently six distinct versions of SSL/TLS. The Secure Sockets Layer (SSL) protocol version 2 was published by Hickman at Netscape in February 1995 (version 1.0 was never released publicly). Owing to security flaws in SSL v2, the protocol was completely reworked, and Netscape released SSL v3 in 1996 [283]. The Internet Engineering Task Force (IETF) made minor changes to SSL v3 and standardised it as the Transport Layer Security (TLS) protocol 1.0 in 1999. This protocol was revised in 2006 to TLS 1.1, particularly with changes in the use of initialisation vectors and padding in CBC mode encryption to protect against attacks identified by Möller [564] and Bard [53]. TLS 1.2 was published in 2008, and replaced the ad hoc MD-5 + SHA-1 pseudo-random function with a ciphersuite-negotiated hash function, usually SHA-256, added new encryption modes for AES, namely Galois/counter mode (GCM) and CBC in counter mode (CCM), and incorporated functionality from various additional RFCs into the core standard. From 2014-2018, the next version of TLS was under development by the IETF, and was standardised at TLS 1.3 in August 2018. TLS 1.3 constitutes a major revision of the TLS protocol; see Sect. 6.14 for more information.

Ciphersuites. The IETF has standardised more than 320 ciphersuites,[1] each of which specifies a valid combination of the following cryptographic algorithms. Note that up to and including TLS 1.2, cryptographic algorithms cannot be negotiated independently, so not all combinations of the following algorithms are possible. (TLS 1.3 provides à la carte negotiation of each component individually.)

- Key exchange:
  - RSA key transport;
  - ephemeral finite-field or elliptic curve Diffie–Hellman;
  - none (null).
- Entity authentication:
  - RSA key transport;
  - RSA digital signatures;
  - finite-field (DSA) or elliptic curve (ECDSA) digital signatures;
  - static finite-field or elliptic curve Diffie–Hellman key exchange;
  - pre-shared keys;
  - password authentication using Secure Remote Password (SRP) protocol;
  - none (null).
- Bulk encryption:
  - RC4 or ChaCha20;
  - RC2, DES, Triple-DES, IDEA, or SEED in CBC mode;
  - AES-128 or AES-256 in CBC, CCM, or GCM mode;
  - ARIA or CAMELLIA in CBC or GCM mode;
  - none (null).
- Message authentication:
  - HMAC-MD5, HMAC-SHA1, HMAC-SHA256, or HMAC-SHA384;
  - AES-128, AES-256, ARIA, or CAMELLIA in GCM mode;
  - Poly1305;
  - none (null).

SSL v2 and v3, and TLS 1.0 also contained *export* ciphersuites: US export regulations in the 1990s restricted the export of cryptographic software or hardware with keys larger than 40 bits (for symmetric cryptography) or 512 bits (for RSA and finite-field Diffie–Hellman). As a result, 'export' ciphersuites were standardised in which the final encryption keys were derived from subkeys that were

[1] https://www.iana.org/assignments/tls-parameters/tls-parameters.xhtml

only 40 (or 512) bits long. Regulations were relaxed starting in 1996, simplifying the export of commercial and open-source cryptography software. Export ciphersuites were removed from TLS in version 1.1, although some software still supports operations involving export-sized keys either directly or indirectly; see Sect. 6.9.5 for resulting attacks.

Datagram TLS. SSL/TLS are designed to work over a stream-oriented network protocol, namely the Transmission Control Protocol (TCP) which provides reliable, in-order delivery of packets. Datagram TLS (DTLS) is designed to work over datagram protcools, such as the User Datagram Protocol (UDP) and others which do not guarantee delivery or ordering of packets, and are used in a variety of applications, including simple query–response protocols such as domain name lookups (DNS) and media-streaming protocols such as the Real-time Transport Protocol (RTP). DTLS provides confidentiality and protection against message tampering and forgery for such applications. One notable distinction between TLS and DTLS is that DTLS generally does not terminate the session upon an error. There are currently two versions of DTLS: DTLS 1.0 [627] (based on TLS 1.1) and DTLS 1.2 [628] (based on TLS 1.2), as well as several standards which specify how to use DTLS to protect various higher-level protocols.

Extensions. After the publication of TLS 1.0 in 1999, the TLS extension mechanism was published [111, 264] which allows the client and server to send extensions on the `ClientHello` and `ServerHello` messages for various purposes. Extensions were incorporated into the main specification in TLS 1.2. Some of the functionality provided by extensions includes the client telling the server which of several domains it is trying to connect to, using the *server name indication* (SNI) extension; negotiation of elliptic curves and point formats; a list of certificate authorities trusted by the client; and a *heartbeat* extension which provides a keep-alive functionality.

Keying-material exporters. RFC 5705 [624] provides a mechanism to obtain keying material derived from the master secret that can be used for any purpose by an application. (Here the term 'exporters' is not related to 'export'-grade ciphersuites.) This mechanism uses the TLS PRF with distinct labels to derive export keys that are computationally independent from the TLS record-layer encryption and MAC keys. Among other purposes, this formalises how the Extensible Authentication Protocol (EAP) obtains keying material from TLS.

Implementation-specific behaviour. Various implementations have bugs that are exhibited in certain configurations. As a result, many TLS implementations have optional code that tries to work around bugs in other implementations.

6.6 Implementations

There are several important SSL/TLS implementations.

OpenSSL (`http://www.openssl.org`) is a leading open source implementation, licensed under a BSD-like licence. OpenSSL is split into two libraries:

libcrypto, which contains the core cryptographic algorithms, and libssl, which contains an implementation of SSL versions 2 and 3, TLS versions 1.0 to 1.3, and DTLS 1.0 and 1.2. OpenSSL is generally included as a standard library on most Unix-based operating systems. It is used by the Apache (via the mod_ssl module) and nginx web servers, and many command-line Unix tools, such as curl and wget. In 2014, several forks of OpenSSL appeared in response to software vulnerabilities in OpenSSL; prominent forks included LibReSSL (http://www.libressl.org) and Google's BoringSSL (https://boringssl.googlesource.com/).

GnuTLS (http://www.gnutls.org) is an open source implementation of SSL and TLS, licensed under the GNU Lesser General Public License v2.1. GnuTLS depends on the Nettle library (http://www.lysator.liu.se/~nisse/nettle) for its cryptographic algorithms. GnuTLS is used by a variety of GNU-licensed software.

Network Security Services (NSS) (https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS) is an open source implementation of SSL/TLS, licensed under the Mozilla Public License. Originally part of the Netscape Navigator web browser, NSS is used in the Mozilla Firefox browser and other Mozilla software, Google's Chrome browser, the Opera browser, and commercial software from RedHat, Oracle, and AOL, among others. NSS can be used with Apache via the mod_nss module.

Bouncy Castle (http://www.bouncycastle.org) and **Java Secure Socket Extension (JSSE)** are prominent open source Java implementations of SSL/TLS.

Microsoft and Apple each have their own implementation of SSL/TLS, named *SChannel* and *SecureTransport*, respectively, which ship with their desktop and mobile operating systems, and are, in particular, used by their web browsers Internet Explorer/Edge and Safari, respectively.

6.7 Security Analyses

The earliest work on the security of SSL, by Wagner and Schneier [724], examined various characteristics of SSL v3 and identified certain weaknesses. Since then, a systematic study of the security of TLS has been carried out in two lines of work: using provable security and using formal methods.

6.7.1 Provable Security

Historically, the TLS handshake and record layer protocols have been analysed separately in the provable-security setting.

As demonstrated throughout this book, there is a vast literature on authenticated key exchange protocols and their security. Chapter 2 details security models for AKE protocols, starting with the Bellare–Rogaway model and continuing with the Canetti–Krawczyk and eCK models. Central to all of those models is that a session key should be indistinguishable from random: in the security definitions, the task of the adversary is to determine whether it has been given the real session key from a target session, or a random string of the same length. Unfortunately, a technicality of TLS prevents it from being proven secure in standard AKE models: the final messages of the TLS handshake protocol (the `Finished` messages) are sent over the encrypted record layer; an adversary who is given a challenge real-or-random session key can try to decrypt the ciphertexts to see if they decrypt correctly, thereby distinguishing real session keys from random ones.

Initial work on the provable security of the TLS handshake protocol looked at a *truncated* handshake protocol in which the `Finished` messages were either omitted or transmitted unencrypted. Jonsson and Kaliski [401] showed that a truncated version of the TLS handshake using RSA key transport is secure under an RSA-based assumption. Subsequently, Morrissey, Smart, and Warinschi [568] showed that a truncated version of the TLS handshake using either RSA key-transport or signed Diffie–Hellman is secure in the random oracle model; their approach was modular, in the sense that they first showed that the premaster secret agreement is secure in a weaker, one-way sense, then that this implied that the master secret is secure in the traditional AKE sense. Gajek *et al.* [288] showed that TLS is a secure, unauthenticated channels protocol in the universal composability setting, but this analysis provided only confidentiality of messages, not authentication of endpoints.

The record layer protocol was also investigated. Krawczyk [451] analysed a variant of the MAC-then-encode-then-encrypt approach using CBC mode encryption in the TLS record layer and showed that it achieved ciphertext indistinguishability (IND-CPA) as well as a certain form of ciphertext unforgeability, weaker than ciphertext integrity. Paterson, Ristenpart, and Shrimpton [605] identified a distinguishing attack against TLS when variable-length padding and short MACs were used, defined a stronger security notion, called *length-hiding authenticated encryption* (LHAE), and showed that the record layer protocol for TLS 1.1 and TLS 1.2, in CBC mode, satisfies this property.

In 2012, the first provable security analysis of a full, unaltered TLS ciphersuite was presented by Jager, Kohlar, Schäge, and Schwenk [392]. They defined a new security model for authenticated and confidential channel establishment (ACCE) protocols. Instead of simply establishing a key, an ACCE protocol establishes a secure channel which can then be used for communication. Formally, the model effectively combines the Bellare–Rogaway model for authenticated key exchange with the Paterson *et al.* model for length-hiding authenticated encryption; this overcomes the aforementioned problem involving the encrypted `Finished` messages that prevented TLS from being proven a secure authenticated key exchange protocol. Originally the model only addressed mutual authentication, but it has subsequently been extended to include the server-only authentication mode that is more widely used in practice.

The original ACCE paper of Jager *et al.* [392] showed that signed finite-field Diffie–Hellman TLS ciphersuites are secure ACCE protocols, assuming that the signature scheme is existentially unforgeable under chosen message attack, certain Diffie–Hellman problems are hard, and the bulk encryption scheme is a secure stateful length-hiding authenticated encryption scheme. Subsequently, several works have analysed a variety of other ciphersuites, including RSA key transport and static Diffie–Hellman [442, 456], and pre-shared keys [489].

The ACCE model has also been extended to consider additional functionality. Giesen, Kohlar, and Stebila [304] extended the ACCE model to formalise the notion of renegotiation security in light of the renegotiation attack (described in Sect. 6.11.2). Bergsma *et al.* [89] considered multi-ciphersuite ACCE security as a result of the cross-protocol attack (described in Sect. 6.10.2).

There are also several alternative approaches to the ACCE model for proving security of TLS. Brzuska *et al.* [164] used a modular approach in which the key exchange is composed with the authenticated encryption scheme, under an additional constraint where the key from the key exchange has been shown to be 'suitable for' this type of composition.

Bhargavan *et al.* [98] implemented several TLS ciphersuites using a programming language that allows for formal verification that the code of the implementation meets provable security properties, and were thus able to derive security results for ciphersuites based on RSA key transport, signed Diffie–Hellman, and static Diffie–Hellman. Related work by several of the same authors [99] examined the TLS handshake in detail, deriving provable security results (supported by formal verification using the F# functional programming language and the EasyCrypt proof verification tool) for handshakes involving RSA key transport, signed Diffie–Hellman, and static Diffie–Hellman using a compositional approach that provides for security even with limited agility of long-term signing keys, i.e. when the same long-term keys are used in several ciphersuites.

A modular analysis of TLS using the recent *constructive cryptography* formalism [528] (related to *abstract cryptography*) has also been given [443].

6.7.2 Formal Methods

SSL and TLS have also been evaluated using formal methods, which use a variety of approaches in logic and theoretical computer science to give precise specifications of desirable protocol behaviour, and either demonstrate that the protocol achieves these goals or demonstrate a flaw. This approach often, but not always, involves automated tools. Three main classes of automated tools are model checkers (a.k.a. finite state analysis), which enumerate all reachable states in an execution to see if an undesirable state is reached; verifiers, which check some human-specified argument (e.g. a proof) against a formal specification; and theorem provers, which construct and verify a proof of a certain property, or alternatively sometimes find a counterexample.

Model Checking. Mitchell *et al.* [562] performed finite state analysis (model checking) using the Murφ tool. They applied the tool to a sequence of protocols, starting from an approximation of the SSL v2 handshake, and culminating in an approximation of the SSL v3 handshake; they simplified, among other things, the derivation of keys from the premaster secret. The tool successfully identified the main weaknesses in SSL v2 that motivated SSL v3. For (the approximation to) SSL v3, the tool found two downgrade attacks and a weakness in resumption, but no other flaws. The model checking was run with some constraints, namely that it involved two clients, one server, no more than two simultaneous open sessions per server, and no more than one resumption per session.

Theorem Provers. Paulson [606] gave machine-checked proofs of TLS handshake authentication and session key security in both full and abbreviated handshakes using the Isabelle theorem prover, assuming idealised cryptographic functions; this 'inductive' approach reasons about an inductively defined set of protocol traces built from the protocol specification and adversary actions. Ogata and Futatsugi [588] gave an analysis of the TLS handshake using the OTS/CafeOBJ tool for equation reasoning, showing in the Dolev–Yao model (which assumes perfect cryptography and abstracts away the details of cryptographic operations) that the premaster secret is secure and that TLS handshake messages are authenticated.

Logic-Based Proofs. In 2005, He *et al.* [353] gave an analysis of the IEEE 802.11i protocol (including an analysis of TLS as a subprotocol) using a protocol composition logic (PCL). For TLS, their arguments showed that the TLS handshake messages are authenticated and that the premaster secret is secure in the face of a symbolic adversary. Kamil and Lowe [410] analysed the TLS handshake and record layer protocols in the strand spaces model, which uses the Dolev–Yao model. Their results included findings that the premaster secret key is secure, the handshake is authenticated, the record layer provides two authenticated streams, and those streams also have confidentiality.

Formal Methods Applied to Concrete Implementations. Formal verification tools have also been applied to concrete implementations of SSL/TLS. Jürjens [405] verified a Java implementation of SSL using an automated theorem prover for first-order logic, showing authentication of the handshake and secrecy of the session key in the Dolev–Yao model; the analysis works on the control-flow graph of the protocol execution. Chaki and Datta [186] applied the Aspier framework to perform a symbolic evaluation of the handshake authentication and session key secrecy of the TLS handshake as implemented in the OpenSSL library, in configurations of up to three clients and servers. Bhargavan *et al.* [97] gave an implementation of a simple yet compatible subset of TLS in the F# programming language, and used the CryptoVerif tool to provide symbolic and computational cryptographic security results. This was the first in a series of results by this set of authors; their later results implement additional levels of complexity and move from symbolic to computational results, and are noted in the section on provable security of TLS above. These results are part of the miTLS project (`http://www.mitls.org/`).

6.8 Attacks: Overview

Because of its importance, TLS has been subject to much study and analysis, and many attacks have been found on the protocol and on TLS implementations. A few attacks date from the early days of SSL/TLS in the late 1990s (such as Bleichenbacher's attack and version downgrade attacks). In the early 2000s, some theoretical weaknesses were identified, but no realised attacks were publicised. Starting in 2009, many major attacks on various aspects of TLS were revealed, necessitating revisions to standards and coordinated release of patches. Many of these recent practical attacks are actually realisations of ideas from Wagner and Schneier's seminal analysis of SSL v3 [724], or other theoretical work from the same era.

Table 6.2 lists attacks against various aspects of SSL/TLS since its inception. The table is organised into several sections, depending on what aspect of SSL/TLS the attack is against. These include weaknesses in core cryptographic algorithms, attacks arising from how cryptographic components are used in TLS ciphersuites, attacks on non-cryptographic functionality in TLS, attacks on libraries implementing TLS, attacks on HTTP-based applications using TLS, and attacks on protocols using TLS.

RFC 7457 [666] summarises some of these attacks, as do Wikipedia's page on TLS and surveys by Meyer and Schwenk [552] and Clark and van Oorschot [218]. The Trustworthy Internet Movement's monthly SSL Pulse survey [621] reports statistics on the TLS configuration of popular websites, such as ciphersuite usage and vulnerability to known attacks.

The remainder of this chapter describes many of these attacks in more detail.

6.9 Attacks: Core Cryptography

The first class of attacks that we look at exploit properties of the cryptographic algorithms used in TLS. This includes both public-key primitives used in the TLS handshake and symmetric-key primitives used in the record layer.

6.9.1 Bleichenbacher's Attack on PKCS#1v1.5 RSA Key Transport

TLS ciphersuites using RSA key transport do not use 'schoolbook' RSA encryption; instead, they rely on the PKCS#1v1.5 standard for RSA encryption [408]. Bleichenbacher [118] discovered an attack in which access to a certain type of error information in PKCS#1 decryption can enable recovery of the secret key. It should be noted that Bleichenbacher's attack applies only to PKCS#1v1.5, not later versions of the PKCS#1 standard. However, RSA key transport in SSL/TLS from SSL v3 to TLS 1.2 uses PKCS#1v1.5.

For RSA encryption, the receiver generates a public key/secret key pair by picking two large, distinct secret primes p and q, computing the RSA modulus $n \leftarrow pq$, and picking values e, d such that $ed \equiv 1 \mod \phi(n)$, where $\phi(n) = (p-1)(q-1)$; the receiver's public key is (n, e) and the private key is d.

In 'schoolbook' RSA encryption, the sender represents a message M as an integer $m \mod n$, then computes the ciphertext as $c \leftarrow m^e \mod n$. The receiver can decrypt this by computing $m \leftarrow c^d \mod n$ and converting the integer m back into the message M. While this does indeed provide confidentiality, it does not protect against malleability. For example, $c^2 \mod n$ is the ciphertext corresponding to the

Table 6.2: Known attacks on SSL/TLS. * denotes a theoretical basis for a later practical attack

Target	Attack name	Year	References
Core cryptography			
RSA PKCS#1v1.5 decryption	Side channel – Bleichenbacher	1998*, 2014	[118]*, [553]
DES	Weakness – brute force	1998	[265]
MD5	Weakness – collisions	2005	[482]
RC4	Weakness – biases	2000*, 2013	[280, 515]*, [28]
RSA export keys	FREAK	2015	[92]
Diffie–Hellman export keys	Logjam	2015	[21]
RSA-MD5 signatures	SLOTH	2016	[101]
Triple-DES	Sweet32	2011*, 2016	[635]*, [100]
Crypto usage in ciphersuites			
CBC mode encryption	BEAST	2002*, 2011	[564]*, [260]
Diffie–Hellman parameters	Cross-protocol attack	1996*, 2012	[724]*, [529]
MAC–encode–encrypt padding	Lucky 13	2013	[29]
CBC mode encryption + padding	POODLE	2014	[565]
TLS protocol functionality			
Support for old versions	Jager *et al.*, DROWN	2015, 2016	[46, 393]
Negotiation	Downgrade to weak crypto	1996, 2015	[21, 92, 724]
Termination	Truncation attack	2007, 2013	[88, 684]
Renegotiation	Renegotiation attack	2009	[623]
Compression	CRIME, BREACH, HEIST	2002*, 2012,16	[422]*, [620, 632, 720]
Session resumption	Triple-handshake attack	2014	[96]
Implementation – libraries			
OpenSSL – RSA	Side channel	2005, 2007	[20, 608]
Debian OpenSSL	Weak RNG	2008	[710]
OpenSSL – elliptic curve	Side channel	2011–14	[161, 162, 756]
Apple – certificate validation	goto fail;	2014	[476]
OpenSSL – Heartbeat extension	Heartbleed	2014	[220]
Multiple – certificate validation	Frankencerts	2014	[160]
NSS – RSA PKCS#1v1.5 signatures	BERserk (Bleichenbacher)	2006*, 2014	[276]*, [473]
Multiple – state machine	CCS injection, SMACK	2014, 2015	[92, 483]
Implementation – HTTP-based applications			
Netscape	Weak RNG	1996	[309]
Multiple – certificate validation	'The most dangerous code...'	2012	[302]
Application-level protocols			
HTTP	SSL stripping	2009	[524]
HTTP server virtual hosts	Virtual host confusion	2014	[239]
IMAP/POP/FTP	STARTTLS command injection	2011	[722]

message m^2. One mechanism for preventing ciphertext malleability is to impose formatting and padding requirements on the message; the homomorphic property of RSA encryption would not a priori preserve that formatting requirement.

PKCS#1v1.5 encryption pads the message M as follows. Let k be the length in bytes of the RSA modulus n. The padded format of M is

$$m = 00||02||PS||00||M,$$

where 00 and 02 are single bytes, and PS is a padding string composed of $k-3-|M| \geq 8$ non-zero bytes. Once M is converted into m as above, the ciphertext is $m^e \bmod n$. During decryption of a ciphertext c, the receiver again computes $m \leftarrow c^d \bmod n$, then checks to see if m is *PKCS-conforming*, meaning that (a) the first byte $m_1 = 00$, (b) the second byte $m_2 = 02$, (c) bytes m_3 to m_{10} are all non-zero, and (d) at least one of the bytes from m_{11} to m_k is 00. If m is PKCS-conforming, then M is extracted. A ciphertext is said to be PKCS-conforming if its decryption is.

In Bleichenbacher's attack [118], we assume that an attacker has access to an oracle which indicates whether a given ciphertext is PKCS-conforming. An attacker can use such an oracle to decrypt a target ciphertext in a relatively small number of oracle queries; for a 1024-bit RSA modulus, around 2^{20} queries to the oracle suffice. The attack proceeds as follows. Let c_0 be the challenge ciphertext, which corresponds to PKCS#1v1.5 plaintext m_0 (and message M_0). The attacker chooses a small value s and computes $c' \leftarrow c_0 s^e \mod n$, then submits c' to the PKCS-conformance-checking oracle. If the ciphertext is PKCS-conforming, then the attacker knows that the first two bytes of $m_0 s \mod n$ are 00 and 02. Mathematically, this means that $2B \leq m_0 s \bmod n < 3B$, where $B = 2^{8(k-2)}$. The attacker repeats this many times with different s values, recording each s such that $m_0 s \mod n$ is in the required range. The adversary can then use this information to narrow the range of values for m_0. For a detailed description of the range-narrowing process and an evaluation of the success probability, see [118], and improvements by Bardou *et al.* [54].

Straightforward implementations of SSL v3 enabled the attacker to obtain a PKCS-conformance-checking oracle using a timing attack. In particular, the pseudocode of early implementations was as follows.

```
1. Compute $m \leftarrow c^d \mod n$.
2. If $m$ is not PKCS-conforming, then reject.
3. Otherwise, do additional cryptographic processing based on $m$ (e.g. strong
   authentication).
4. If authentication fails, then reject; otherwise accept.
```

Since the strong-authentication operations in step 3 above require some non-trivial amount of time to perform, there is a timing mismatch between a reject in step 2 and a reject in step 4, allowing an adversary to learn whether the ciphertext was PKCS-conforming or not.

SSL v3 actually imposes some additional padding constraints of its own, with the effect that the 46-byte premaster secret PMS is padded to

$$m = 00||02||PS||00||03||00||PMS,$$

and hence the padding string *PS* is always of the same length (for the same RSA key size k). Imposing this additional 'SSL-conformance' check that the bytes $03\|00$ always appear in the correct place increases the number of oracle queries to around 2^{40}, making the attack harder but still feasible. Klíma *et al.* [434] proposed a variant of the attack and tested the attack in the wild, finding that nearly two-thirds of tested web servers were vulnerable at the time.

TLS versions 1.0–1.2 still use PKCS#1v1.5 encryption for RSA key transport, but impose additional requirements on implementations [251, Sect. 7.4.7.1]. In particular, before RSA decryption, implementations generate a random 48-byte string. If the decryption does not conform to the PKCS and SSL formatting and version requirements, processing continues, but using the randomly generated 48-byte string as the premaster secret rather than the decrypted value. In other words, the handshake protocol continues regardless of whether the ciphertext was PKCS- and SSL-conforming or not. Rejection only happens when authentication fails in the processing of the `Finished` message; since the attacker has forged a ciphertext, it expects that authentication will fail, and thus learns nothing about the PKCS conformance of the forged ciphertext.

Bleichenbacher's attack continues to plague TLS (and other) implementations. Specifically related to TLS implementations, Meyer *et al.* [553] successfully applied Bleichenbacher's attack against timing side channels in OpenSSL, JSSE, and Cavium, as well as an error message side channel in JSSE. Jager *et al.* [393] showed that even though TLS 1.3 (draft 10) does not include RSA key transport, and thus does not rely on PKCS#1v1.5 encryption, Bleichenbacher's attack on older versions of TLS at servers which use the same RSA certificate for TLS 1.3 can potentially lead to impersonation attacks on TLS 1.3 servers.

In 2016, Aviram *et al.* [46] presented the DROWN (Decrypting RSA using Obsolete and Weakened eNcryption) attack, which works against servers that use the same RSA key with SSL 2 and modern versions of TLS, up to TLS 1.2. They identified a Bleichenbacher RSA padding oracle in the SSL 2 protocol, which can then be used to decrypt TLS 1.2 ciphertexts. To decrypt at 2048-bit RSA TLS 1.2 ciphertext, their attack requires an attacker to observe 1,000 TLS handshakes, initiate 40,000 SSL 2 connections, and do 2^{50} offline work. They found that some 33% of publicly accessible HTTPS servers were vulnerable to this attack.

6.9.2 Bleichenbacher's Attack on PKCS#1v1.5 RSA Signature Verification

In 2006, Bleichenbacher [276] identified another vulnerability involving PKCS# v1.5, this time in how implementations parse data during signature verification. Bleichenbacher observed that the OpenPGP implementation of signature verification did not correctly parse the padding of the hash value after exponentiation with the public exponent. Bleichenbacher's 2006 attack required that the RSA public exponent be 3 and that the RSA modulus be quite large (e.g. 3072 bits); a subsequent improvement by Kühn *et al.* [460] allowed for smaller moduli and demonstrated how to use the attack against CA, intermediate, or server certificates used by TLS web browsers, including a specific attack against the NSS library. This technique was again applicable in a 2014 attack, called the ‘BERserk’ attack, on the ASN.1 parsing of padding in PKCS#1v1.5 RSA signature verification in the NSS library, independently discovered by Delignat-Lavaud and Intel Security [473].

6.9.3 Weaknesses in DES, Triple-DES, MD5, and SHA-1

DES. The Data Encryption Standard (DES) was supported for encryption in CBC mode in SSL v2, SSL v3, TLS 1.0, and TLS 1.1. DES has an effective key size of 56 bits, far below the level needed for security against attackers today. Indeed, in 1998, the Electronic Frontier Foundation (EFF) built a DES cracker for under US$250,000 that could perform an exhaustive key search in less than 3 days [265], and in 2006 Kumar *et al.* [461] built an FPGA-based machine for under US$10,000 that performs an exhaustive key search in less than 9 days. DES ciphersuites were not included in TLS 1.2, and their use in earlier versions was recommended to be discontinued [268].

Triple-DES. TLS versions up to 1.2 still support Triple-DES (three-key encrypt–decrypt–encrypt) in CBC mode, which has an effective security of 112 bits. No significant weaknesses are known in the core Triple-DES function, and, in guidance dated January 2012, NIST continued to allow the use of Triple-DES for encryption of ‘sensitive unclassified’ data. However, CBC mode is known to suffer from collision attacks [635] which require a number of ciphertext blocks up to the birthday bound on the block size, so block ciphers with small block sizes are at risk. Triple-DES has the same block size as DES: 64 bits. Bhargavan and Leurent [100] observed that the amount of ciphertext required to carry out such an attack against Triple-DES is only 32 GB, and showed how to use the attack to recover secret session cookies from HTTPS traffic encrypted using Triple-DES; this was called the ‘Sweet32’ attack. The guidance following the attack was to disable Triple-DES ciphersuites.

MD5. The MD5 hash function is used in several places in various versions of TLS. In TLS 1.0 and TLS 1.1, the following apply.

1. Ciphersuites using RSA signatures sign the concatenation of the MD5 and SHA-1 hashes of the `ServerKeyExchange` message.
2. The TLS PRF combines outputs from HMAC-MD5 and HMAC-SHA-1.
3. The message authentication code can be HMAC-MD5 or HMAC-SHA-1.
4. X.509 certificates can be signed using RSA with MD5 hashes or SHA-1 hashes.

In TLS 1.2, the following apply.

1. The MAC can be HMAC-MD5 or HMAC-SHA-1.
2. X.509 certificates can be signed using RSA with MD5 hashes or SHA-1 hashes.

The collision resistance of MD5 is completely broken. Following Wang *et al.*’s initial discovery of a collision in MD5 [730], Lenstra and de Weger [482] constructed two different X.509 certificates containing identical signatures using a 'meaningful' MD5 collision; Stevens, Lenstra, and de Weger [696] later constructed colliding X.509 certificates, one of which was a normal certificate (that they got signed by a commercial CA) and one of which included flags for acting as a certificate authority, thus giving them the ability to issue fraudulent certificates that would be browser-trusted. This means that use of 4 in the list above in TLS 1.0 and TLS 1.1 and use of 2 for TLS 1.2 are insecure.

Conventional wisdom would be that the other uses of MD5 in TLS mentioned above do not immediately become insecure owing to the failure of collision resistance: signatures involving the concatenation of the MD5 and SHA-1 hashes remain secure if SHA-1 is collision resistant (theoretical attacks are known, but no collision has been published to date); and HMAC can remain secure both for message authentication and as a PRF under weaker conditions than collision resistance [68]. Despite this, these particular uses of MD5 in TLS are still problematic. In 2016, Bhargavan and Leurent [101] identified a class of "transcript collision" attacks (which they called the SLOTH attack). Their immediately practical attacks (48 core hours of parallelisable computation) included impersonation of clients or servers using RSA-MD5 certificates for authentication, and the breaking of `tls-unique` channel binding (which relies on HMAC-MD5 truncated to 96 bits); they also projected the complexity of finding a collision in the concatenated MD5 and SHA-1 hashes of the handshake transcript. At the time of the SLOTH disclosure, many software packages, and 32% of TLS servers, accepted RSA-MD5 signatures, although commercial CAs have not been issuing for RSA-MD5 certificates for many years.

SHA-1. The SHA-1 hash algorithm is also used in various places in TLS. The projected complexity of finding a SHA-1 hash collision is between $2^{60.3}$ and $2^{65.3}$ operations [695], no collisions were publicly known as of October 2015. Major browser vendors (including Google, Microsoft, and Mozilla) and CAs are transitioning to SHA-256; CAs have been required to not issue SHA-1 certificates since January 2016, and browsers have been treating SHA-1 certificates as insecure since January 2017.

6.9.4 RC4 Biases

In many TLS ciphersuites, the RC4 stream cipher is used for bulk encryption. RC4 was for many years viewed as preferred compared with DES (RC4 having larger keys), Triple-DES (RC4 being faster and easier to implement) and initial implementations of AES (RC4 again being faster than early software-only implementations of AES). In December 2018, Qualys SSL Labs' SSL Pulse [621] reported that 15.6% of popular web servers had RC4 support enabled, and 1.6% would negotiate an RC4-based ciphersuite even with modern browsers.

It has long been known that the RC4 stream cipher is less than ideal in a variety of ways. Sen Gupta *et al.* [662] provided an excellent survey of the history of RC4 cryptanalysis over the years, including the existence of weak keys, the partial recovery of the initial key from the internal state and from the keystream, and observed biases in the keystream.

In 2013, AlFardan *et al.* [28] demonstrated two nearly practical plaintext recovery attacks on RC4 as used in TLS based on prior known biases and some novel improvements. In their attack scenario, a fixed plaintext is encrypted using RC4 many times in succession, using the same or independent RC4 keystreams.

Their first attack, based on single-byte bias, is against the initial 256 bytes of the RC4 keystream: if the same plaintext is encrypted many times under distinct RC4 keys, then plaintext bytes can be recovered. In TLS, the keystream is used to encrypt the last handshake message (the 36-byte `Finished` message) which changes each session, but the next 220 bytes are application plaintext: a careful attacker may be able to cause a client to send the same request many times. In HTTP, cookies may appear in the first 220 bytes (though browsers often send more HTTP headers before the cookies); in the IMAP protocol for email collection, the user's password typically appears in the first 220 bytes. The attack works based on statistically averaging many ciphertexts, considering known biases in the first 256 bytes of RC4 keystreams. In AlFardan *et al.*'s experiments, the first 40 bytes of TLS application data were each recovered with a success rate of over 50% per byte using 2^{26} sessions, and all 220 bytes of application data with a success rate of over 96% per byte using 2^{32} sessions.

These attacks on RC4 have subsequently been improved. In March 2015, Garman *et al.* [292] demonstrated proof-of-concept password recovery attacks on certain application-layer protocols (BasicAuth and IMAP) when using RC4 ciphersuites in TLS. Their attacks achieved a significant success rate using 2^{26} ciphertexts. The main advantage comes from assuming the secret password is not distributed uniformly, instead using knowledge of typical password distributions, for example based on leaked corpuses of passwords [126].

It has been previously suggested [555] that the first few hundred bytes of the RC4 keystream should be dropped owing to known biases; however, the TLS standards do not provide a mechanism for doing so. AlFardan *et al.* gave a second attack, based on double-byte biases, that allows recovery of plaintext bytes anywhere in the ciphertext, not just the first 256 bytes. It is based on biases in pairs of bytes in the RC4 keystream. In their experiments, 16 consecutive bytes could be recovered with a 100% success rate using statistical analysis of 13×2^{30} ciphertexts. Vanhoef and Piessens [721] identified new biases and exploited them to be able to recover a 16-byte cookie with a 94% success rate using a novel statistical analysis of 9×2^{27} ciphertexts.

After the BEAST attack [260] against CBC mode in TLS in 2011, it was recommended that RC4 be adopted as the preferred ciphersuite. After the attacks by AlFardan *et al.*, it was recommended that RC4 be avoided, and that CBC mode with a certain countermeasure be used or, preferably, that the Galois/counter mode of authenticated encryption in TLS 1.2 be used where supported. In February 2015, the IETF TLS working group approved an RFC to prohibit the use of RC4 in all versions of TLS [619].

6.9.5 Weak RSA and Diffie–Hellman: FREAK and Logjam Attacks

As noted above, early versions of SSL included support for *export ciphersuites* which used shorter keys, as required by US export regulations. For RSA encryption and finite-field Diffie–Hellman key exchange, this meant the use of 512-bit (or shorter) keys. This key size would not be considered secure by modern standards, as RSA keys of this size could be factored in 1999. With the lapsing of these particular export regulations, modern TLS clients do not offer export ciphersuites. Nonetheless, many TLS implementations still include code supporting either export ciphersuites directly, or smaller keys indirectly. This old code led to the discovery of two major vulnerabilities in 2015.

Beurdouche *et al.* [92] discovered the so-called FREAK ("Factoring RSA Export Keys") attack, which exploits a state machine vulnerability in OpenSSL, Microsoft's SChannel library, and Apple's Secure Transport library, to trick a client and server into using an export-grade RSA key as long as the server supports RSA export ciphersuites, even if the client does not.

The FREAK attack works as follows. It requires a vulnerable client and a server that supports export-grade RSA key transport ciphersuites. First, the client sends a `ClientHello` message with a (non-export) RSA ciphersuite supported. The man-in-the-middle (MITM) replaces the supported ciphersuites with an export RSA ciphersuite and forwards it to the server. When the server responds with a `ServerHello` message for the RSA export ciphersuite, the MITM replaces it with a standard RSA ciphersuite and forwards it to the client. The server will also send its RSA certificate (which may be, for example, 2048 bits) as well as a `ServerKeyExchange` message containing an export-grade (e.g. 512-bit) ephemeral RSA public key. The MITM forwards these along to the client. The client *should* reject at this point, since no honest server running a non-export RSA ciphersuite would send a `ServerKeyExchange` message, but, owing to state machine bugs, several client implementations did not reject, and instead proceeded to use the ephemeral RSA public key for key transport. In particular, these clients would respond with a `ClientKeyExchange` message which contained the random premaster secret encrypted under the export-grade ephemeral RSA public key. The MITM needs to factor the RSA public key, derive all secrets, and then use these secrets to forge `Finished` message MAC tags to trick the client and server into accepting the altered transcripts. While factoring a 512-bit RSA key is possible, it would still take weeks with 2015 technology; however, many implementations will cache ephemeral RSA public keys, since RSA key generation is costly, so it may be possible to carry out the attack using sufficient pre-computation. Affected implementations were patched prior to the release of the paper.

Later in 2015, Adrian *et al.* [21] discovered the Logjam attack, which similarly allows a MITM attacker to downgrade the cryptography used in TLS, this time downgrading ciphersuites using finite-field Diffie–Hellman for forward secrecy to export-grade (or lower!) keys. The attack applied against all major browsers (Internet Explorer, Chrome, Firefox, Opera, and Safari). (Interestingly, Safari accepted finite-field Diffie–Hellman groups as small as 16 bits.) Using Internet-wide scanning tools, the researchers identified that 8.4% of the Alexa Top 1 Million HTTPS domains allowed export-grade ephemeral Diffie–Hellman key exchange.

The Logjam attack works as follows. It requires a vulnerable client and a server which supports export-grade finite-field ephemeral Diffie–Hellman (‘`DHE_EXPORT`’) ciphersuites. First, the client sends a `ClientHello` message with a (non-export) finite-field Diffie–Hellman ciphersuite supported. The MITM replaces the supported ciphersuites with an export-grade finite-field Diffie–Hellman ciphersuite and forwards it to the server. When the server responds with a `ServerHello` message for the export DH ciphersuite, the MITM replaces it with a standard Diffie–Hellman ciphersuite and forwards it to the client. The server also sends its export-grade ephemeral Diffie–Hellman key in a `ServerKeyExchange` message, which the MITM forwards to the client. A vulnerable client implementation will accept the server's export-grade ephemeral Diffie–Hellman public key even though the client and server did not explicitly negotiate an export ciphersuite, and respond with its ephemeral Diffie–Hellman public key. The MITM needs to compute the shared Diffie–Hellman secret, derive all secrets, and then use these secrets to forge `Finished` message MAC tags to trick the client and server into accepting the altered transcripts.

Two implementation details make it feasible for the attacker to complete the attack. First, 92% of web servers supporting `DHE_EXPORT` ciphersuites use one of two standardised 512-bit finite-field groups. While each server will use a distinct ephemeral public key from this group, the number field sieve discrete logarithm algorithm can be structured to make use of pre-computation for the group that is independent of the specific discrete logarithm being computed. Second, many web servers, including Apache, nginx, and Microsoft's SChannel, reuse server ephemeral Diffie–Hellman keys. On Adrian *et al.*'s computing cluster, with 1 week of pre-computation for each group, computing individual 512-bit discrete logarithms took on average 70 seconds. The paper includes a discussion about the potential of state-level attackers using greater computing power for 768-bit and even 1024-bit finite-field Diffie–Hellman key exchange, and suggests that, based on the Snowden documents, the National Security Agency may be employing this technique to decrypt some TLS connections as well as virtual private networking connections established using IPsec's Internet Key Exchange (IKE) protocol with a specific 1024-bit finite-field Diffie–Hellman group.

One notable characteristic of both the FREAK and the Logjam attacks is that they exploit the fact that in SSL and TLS up to version 1.2, authentication of the transcript is done using a MAC rather than a signature, and the MAC is computed using (a key derived from) the master secret, *the security of which is itself negotiated inside the protocol.* (Note that the server's signature is only over the nonces and the raw ephemeral public key, and excludes the negotiation in the `ClientHello`/`ServerHello` messages.) In other words, the key used to authenticate the transcript and show that a downgrade attack has not occurred (i.e. that the parties agree on the `ClientHello`/`ServerHello` messages) is being downgraded prior to authentication. TLS 1.3 aims to avoid this type of attack by using the long-term public key to sign the entire transcript.

6.10 Attacks: Crypto Usage in Ciphersuites

We next examine attacks in which the cryptographic algorithms in the TLS ciphersuite interact in an unfortunate way with the other protocol components.

6.10.1 BEAST Adaptive Chosen Plaintext Attack and POODLE

The use of block ciphers (DES, Triple-DES, AES) in cipher block chaining (CBC) mode in SSL v3 and TLS 1.0 is vulnerable to an adaptive chosen plaintext attack owing to the manner in which initialisation vectors for different requests in the same SSL/TLS connection are set. This vulnerability was observed by Möller in 2002 [564] and Bard in 2004 [53], but neither was able to demonstrate an actual attack. In 2011, Rizzo and Duong demonstrated a working attack, which they called the BEAST attack (a 'backronym' for 'Browser Exploit Against SSL/TLS') that allowed them to recover short secret strings in known locations in HTTP plaintexts [260, 625, 650].

CBC mode is one of several block cipher modes that allow an arbitrary-length message m to be split into blocks $m_1 \| m_2 \| m_3 \| \dots \| m_n$ and encrypted by a fixed-length cipher. In CBC mode, the first block of plaintext, m_1, is XORed with an initialisation vector iv, then encrypted using the key k:

$$c_1 \leftarrow \{m_1 \oplus iv\}_k.$$

Subsequent plaintext blocks are encrypted in a similar way, except that the initialisation vector is replaced with the previous ciphertext block:

$$c_i \leftarrow \{m_i \oplus c_{i-1}\}_k.$$

In SSL and TLS, the initialisation vector is derived from the master secret using the PRF. In many applications, including HTTPS, the same SSL/TLS connection is used for multiple requests. For example, in a web browser, the SSL/TLS connection will be established for the first request to a server, and then subsequent requests (including page resources such as images, JavaScript, applets, and later HTML pages) may all be sent over the same connection. SSL v3 and TLS 1.0 do *not* choose a new initialisation vector for each subsequent request within the same TLS connection: instead, they simply continue cipher-block chaining, using the last ciphertext block of the previous request as the initialisation vector for the next request.

The attack allows an adversary to test whether a particular guess at a plaintext block has a particular value. Suppose the adversary has observed the transmission of ciphertext $c_1 \| c_2 \| \dots \| c_n$ and wishes to determine whether plaintext block m_j equals some guessed plaintext m^*. Now, the adversary knows that the next plaintext block sent will be encrypted with initialisation vector c_n. The adversary directs the user to send the next plaintext, block $m_{n+1} = c_{j-1} \oplus c_n \oplus m^*$; the user will compute the ciphertext

$$c_{n+1} = \{m_{n+1} \oplus c_n\}_k = \{c_{j-1} \oplus c_n \oplus m^* \oplus c_n\}_k = \{c_{j-1} \oplus m^*\}_k.$$

If $m_j = m^*$, then $c_{n+1} = c_j$, allowing the adversary to verify whether or not its guess of m^* for plaintext block j was correct.

The above attack allows the adversary to verify guesses of a single block one at a time. In AES, for example, a single block is 128 bits long; an adversary trying to guess a whole secret block could require up to 2^{128} operations, as much work as a brute force key recovery attack. However, Rizzo and Duong showed how an attacker who can inject plaintext near the beginning of a request can adjust how the plaintext is broken across block boundaries, isolating just a single unknown byte at a time. For example, suppose the attacker can cause the victim to make an HTTP request to a given URL, and that the cookie (which contains the secret value the attacker is targeting) comes immediately after, in a request like this:

$$\underbrace{\texttt{GET /}\hookleftarrow\texttt{Cookie: s=}}_{16}\underbrace{\texttt{1234567890123456}}_{16}$$

If the plaintext is encrypted in blocks of 16 bytes (128 bits), notice that the first block of 16 bytes is entirely known to the adversary, but the second block of 16 bytes is entirely unknown. If the adversary can change the URL that the victim requests, it can control where the block boundary falls. For example, if the adversary causes the victim to request a 15-character URL such as `abcdeabcdeabcde`, then the block boundaries fall like this:

$$\underbrace{\texttt{GET /abcdeabcdea}}_{16}\underbrace{\texttt{bcde}\hookleftarrow\texttt{Cookie: s=1}}_{16}\underbrace{\texttt{234567890123456}}_{15}$$

The second block (`bcde`↩`Cookie: s=1`) now contains only a single unknown byte, and thus can be found by carrying out the above attack using only 256 plaintext guesses. Once that byte is found, the adversary can shift the boundary again for the next unknown byte:

$$\underbrace{\texttt{GET /abcdeabcdea}}_{16}\underbrace{\texttt{bcd}\hookleftarrow\texttt{Cookie: s=12}}_{16}\underbrace{\texttt{34567890123456}}_{14}$$

and so on, allowing it to recover a k-byte secret with just $256k$ guesses.

To carry out this attack, the adversary needs to be able to observe ciphertexts, know the format of requests from the victim and be able to cause the victim to make multiple requests with the same secret, while being able to inject plaintext to control where the block boundary falls and to test guesses. Modern web browsers have a variety of communication technologies that have the potential to allow the adversary such powers, such as asynchronous Javascript requests and HTML5 WebSockets, as well as APIs in plugin technologies such as Java's URLConnection API and Silverlight's WebClient API. In most cases, the browser enforces a same-origin policy—preventing scripts from one domain sending data to another domain—that make it difficult to carry out the attack. Rizzo and Duong made use of a zero-day exploit in Java that bypassed the same-origin policy restrictions; this exploit has subsequently been patched, and to date no other means of carrying out the BEAST attack has been exhibited.

One of the major changes introduced in TLS 1.1 was the use of explicit IVs in CBC mode ciphers to prevent this attack. This suffices as a countermeasure to the BEAST attack. However, deployment of web browsers and servers supporting TLS 1.1 was slow; in early 2012, just after the BEAST attack, less than 1% of SSL-enabled websites surveyed by SSL Pulse supported TLS 1.1 or TLS 1.2. As a result, the immediate recommendation at the time of the BEAST attack was to switch to ciphersuites using the RC4 stream cipher; this recommendation has since been rescinded owing to subsequent attacks on RC4 described below. Browsers have since been updated to do so-called $1/n-1$ record splitting: the first block in each new message contains only a single byte of plaintext, the second block contains the next $n-1$ bytes (here n is the block size in bytes), and then the remaining blocks are full size. This has the effect of randomizing the IV in a (relatively) backward-compatible way [475].

In 2014, Möller, Duong, and Kotowicz [565] described a more powerful form of the BEAST attack against CBC-mode encryption SSL v3 in an attack which they called the POODLE (Padding Oracle On Downgraded Legacy Encryption) attack. In CBC-mode encryption in SSL v3, plaintext is concatenated with the MAC over the plaintext, then padding is applied to pad it out to a multiple of the block length L. The padding must be between 1 and L bytes, and the last byte of the last block is the length of the padding. There are no requirements on the content of the padding, and it is not covered by the MAC, so the integrity of the padding is not verified when decrypting.

The POODLE attack allows an attacker to decrypt a secret value inside the plaintext as follows. The attack is simpler if we assume the full last block is padding, which can be achieved easily by an attacker who has partial control over the plaintext. The attacker takes the ciphertext block it is trying to decrypt and uses it as the final ciphertext block. The decryption algorithm then XORs this with the previous block of ciphertext, which is known to the adversary. If the last byte of the result is the same as the expected padding length, then the block will decrypt successfully, otherwise an error will occur. Thus, with a 1-in-256 chance, the attacker can learn the last byte of one block of plaintext. Once one byte of the plaintext has been learnt, an attacker with partial chosen plaintext abilities can request ciphertexts with an additional byte, shifting the next byte of the target secret into the last byte of a block, and repeat.

While the POODLE attack can be defeated with a clever record-splitting technique similar to the $1/n-1$ record-splitting technique as described for the BEAST attack above, the preferred technique is to simply avoid use of SSL v3. Since some servers still support only SSL v3, some clients continue to offer SSL v3 support, to which an attacker could try to cause clients to downgrade. A recent new feature of TLS, the `TLS_FALLBACK_SCSV` signalling-ciphersuite value, can prevent downgrade attacks and thus protect clients from the POODLE attack [474].

6.10.2 Cross-Protocol Attack on Diffie–Hellman Parameters

As previously identified, TLS supports many different ciphersuites, and most deployments also support several ciphersuites. For example, a standard server might support ciphersuites using RSA key transport, RSA-signed finite-field ephemeral Diffie–Hellman, and RSA-signed elliptic curve ephemeral Diffie–Hellman, using AES encryption in CBC or Galois/counter mode, and with either SHA-1 or SHA-256 hash functions. However, in almost all installations, each server has only a single RSA long-term key, which is reused across all supported ciphersuites. In fact, popular web servers such as Apache only allow the server administrator to install a single key for each long-term key algorithm. Reuse of keying material—also known as *key agility*—can sometimes result in a vulnerability, either because secrets used in one algorithm may leak information when used in another algorithm, or because data encrypted/authenticated by one protocol may be unintentionally useful in another protocol.

Recall from Sect. 6.3.1 that, in ciphersuites where authentication is based on digital signatures, the server signs the `ServerKeyExchange` message. The contents of the `ServerKeyExchange` message depend on the type of key exchange method and are shown in Fig. 6.2. For ephemeral Diffie–Hellman key exchange (either finite-field or elliptic curve), the `ServerKeyExchange` message includes the description of the group as well as the server's ephemeral public key. In RSA key transport ciphersuites in SSL v3 and the export ciphersuites of TLS 1.0, the server may also send an RSA public key for encryption.

Wagner and Schneier [724] identified potential cross-protocol attacks in SSL v3 in 1996. They observed that the data structure to be signed—either `ServerRSA-Params` or `ServerDHParams`—does not explicitly indicate what the type of the data structure is; instead, the data type is inferred by each party based on the negotiated ciphersuite. Wagner and Schneier hypothesised that it may be possible for an attacker to convince a client to misinterpret one type of data structure as another.

In 2012, Mavrogiannopoulos *et al.* [529] noted that, while the exact attack of Wagner and Schneier would not work owing to a subtlety in how TLS packets were processed, a similar attack could be executed. In 2006, support for elliptic curve Diffie–Hellman key exchange had been added to TLS [106]; in the relevant ciphersuites, the `ServerKeyExchange` message contains the description of the elliptic curve parameters and the server's ephemeral public key point. The specification allows curves to be specified either as one of several predefined *named curves* using a single index value, or as an *explicit curve* by specifying the prime or irreducible polynomial, curve equation coefficients, base point, and group order. When explicit curves are used, Mavrogiannopoulos *et al.* observed that it is possible to construct a `ServerECDHParams` data structure that is also a valid `ServerDHParams` data structure. Moreover, with careful choices of the curve parameters, a non-trival proportion (around 1 in 2^{40}) of elliptic curve ephemeral public keys will, when the `ServerECDHParams` data structure is interpreted as a `ServerDHParams` structure, result in a modulus that is sufficiently smooth to allow efficient factoring, enabling the attacker to compute the premaster secret and impersonate the server in the con-

```
struct {
  select (KeyExchangeAlgorithm):
    case rsa:
      ServerRSAParams params;
      Signature signed_params;
    case dhe_dss, dhe_rsa:
      ServerDHParams params;
      Signature signed_params;
    case ec_diffie_hellman:
      ServerECDHParams params;
      Signature signed_params;
} ServerKeyExchange
```

Note that `case rsa` only applies in some scenarios of SSL v3 and TLS 1.0.

```
struct {
  opaque rsa_modulus;
  opaque rsa_exponent;
} ServerRSAParams;

struct {
  opaque dh_p<1..2^16-1>;
  opaque dh_g<1..2^16-1>;
  opaque dh_Ys<1..2^16-1>;
} ServerDHParams;

struct {
  ECCurveType curve_type = 1;
  opaque      prime_p <1..2^8-1>;
  ECCurve     curve;
  ECPoint     base;
  opaque      order <1..2^8-1>;
  opaque      cofactor <1..2^8-1>;
  opaque      point <1..2^8-1>;
} ServerECDHParams;
```

Fig. 6.2: `ServerKeyExchange` data structures for signed ciphersuites in TLS

nection. Fortunately, as of this writing no major TLS server implementations support explicit elliptic curve parameters, so the attack remains theoretical. Motivated by this attack, Bergsma *et al.* [89] extended the ACCE security notion to consider multi-ciphersuite security and characterised the security of certain subsets of TLS ciphersuites with long-term key reuse across ciphersuites.

6.10.3 Lucky 13 Attack on MAC-Then-Encode-Then-Encrypt

The use of CBC mode for block ciphers in the TLS record layer follows a MAC-then-encode-then-encrypt pattern: a MAC is computed over the plaintext, then the plaintext and MAC tag are padded to a multiple of the block length, then the plaintext+MAC+padding is encrypted. The padding must have a specified format. In order to prevent one type of timing attack, the TLS standards require that a MAC check still be performed even if the padding is invalid. However, it becomes unclear as to which data the MAC should be computed over, so the RFC recommends processing as if there was a zero-length pad. In contrast, no processing of a validly padded and encrypted message would ever be performed with a zero-length pad. This difference opens up the possibility of a small timing side channel in the form of a padding oracle attack. The RFCs indicated it was "not believed to be large enough to be exploitable, due to the large block size of existing MACs and the small size of the timing signal" [251, Sect. 6.2.3.2].

AlFardan and Paterson [29] demonstrated how to exploit this small timing side channel, including practical attacks on open source implementations. The name 'Lucky 13' comes from what they describe as a "fortuitous alignment of various factors such as the size of MAC tags, the block cipher's block size, and the number of header bytes". Ciphertexts with invalid padding will result in an error message appearing at a slightly different time than for ciphertexts with valid padding but an invalid MAC tag.

In their most general attack on TLS in OpenSSL, an attacker on the same LAN segment is able to recover a full plaintext block using roughly 2^{23} sessions, provided that the same plaintext is sent in multiple sessions. More specific variants are more effective. It is possible to use the attack technique to distinguish the encryptions of two chosen messages in just a few sessions. Partial plaintext recovery attacks against TLS and DTLS are possible using fewer sessions. Attacks against DTLS are more effective: since DTLS is non-reliable, errors in DTLS are not fatal to the session, so it is much more practical to carry out repeated queries against the session. All major open source TLS implementations were vulnerable to attack.

One countermeasure proposed was to remove the timing side channel using a careful implementation, although the authors of the attack cautioned that it would be difficult to remove all related timing side channels, especially in DTLS implementations. Adding random timing delays was noted to be relatively ineffective, adding only a small increase in statistical uncertainty that could be averaged out with additional samples. The alternatives proposed were to switch away from CBC mode, either to RC4 or to authenticated encryption modes. The recommendation to switch to RC4 was with the caveat that RC4 was known to have some statistical weaknesses, and the same authors (plus a few new co-authors) subsequently demonstrated practical attacks against RC4-based ciphersuites, invalidating that recommendation (see Sect. 6.9). Authenticated encryption modes are only supported in TLS 1.2, leaving lower versions stuck with a difficult choice. An option to switch to pad-then-encrypt-then-MAC was standardised in 2014 [339] but does not seem to be widely implemented as of the time of writing.

6.11 Attacks: Protocol Functionality

This section examines attacks which arise owing to the extensive range of functionality provided in TLS. Unfortunately, flexibility which can be useful to applications can sometimes also open up opportunities for attackers.

6.11.1 Downgrade Attacks

All existing versions of SSL and TLS have compatible packet formats and in most applications run over the same network ports, but have different security properties, so there is the risk that devices which support multiple versions of SSL/TLS may be subject to a downgrade attack in which they are tricked into using a lower version than they both support.

Let us briefly recall the version negotiation mechanism. In the `ClientHello` message, the client sends the highest version it supports. In the `ServerHello` message, the server responds with the highest version it supports that the client also supports.

The risk of downgrade attacks in SSL/TLS was identified as early as Wagner and Schneier's 1996 paper [724]. SSL v2 was vulnerable to a ciphersuite rollback attack: since the ciphersuite negotiation was not authenticated in SSL v2, an active attacker could replace one party's list of supported ciphersuites with the weakest mutually supported ciphersuite (for example, an export ciphersuite or one with a weak algorithm).

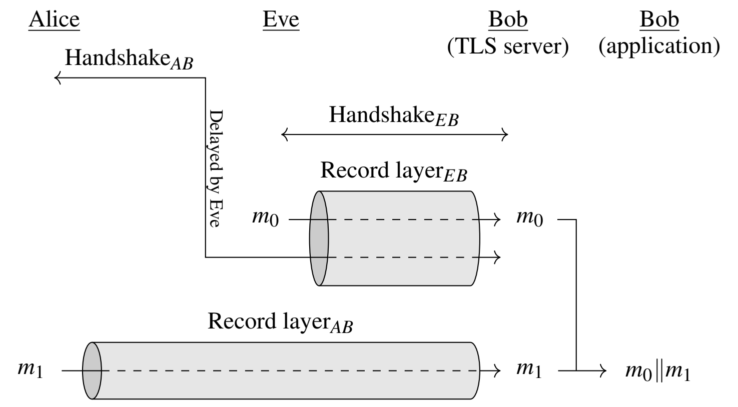
SSL v3 introduced a countermeasure to this attack: the complete handshake transcript was authenticated using a MAC with (a key derived from) the shared session key. This was designed to protect both version negotiation and ciphersuite negotiation. While this provides immediate protection against the naive downgrade attack on SSL v2 noted above, it is still flawed: the security of the version and ciphersuite negotiation mechanism comes from mutual authentication of the transcript, the transcript is authenticated *using the shared session key*, and the algorithm for computing this is determined by the ciphersuite and version being authenticated. This cyclic dependency places negotiation security at risk: a man-in-the-middle attacker who can compute the shared session key in real time can potentially disrupt negotiation undetectably. The FREAK and Logjam attacks, discussed in detail in Sect. 6.9.5, are examples of downgrade attacks that depend on this cyclic dependency of authentication negotiation using a key derived from the negotiated ciphersuite.

If strong ciphersuites are used, defeating attacks like FREAK and Logjam, then authentication of the transcript (which includes the version negotiation) theoretically ensures the parties are not subject to a version downgrade attack. Unfortunately, this is not the case in practice. Many implementations implement a further version negotiation mechanism called *fallback*, sometimes called the *downgrade dance*. Since some flawed TLS server implementations respond with a failure message when confronted with a `ClientHello` containing versions higher than they support, TLS clients will retry the handshake with the next lower version, sidestepping the in-protocol version negotiation mechanism. Unfortunately, this downgrade is easy for an attacker to trigger: the failure message is not authenticated, so a man-in-the-middle attacker can spoof the failure message and trigger a downgrade. Originally, there was no link between the original request and the fallback request. In particular, there was no mechanism for the client to indicate to the server that it was attempting a downgraded handshake owing to a failure message, and thus the client and server would not detect the downgrade attack. In April 2015, the IETF standardised the TLS Fallback Signalling Ciphersuite Value (SCSV) [563]: when trying a lower version during a fallback, the client can send a flag indicating it is doing so (for engineering reasons, this flag is implemented as a special 'dummy' ciphersuite in the list of supported ciphersuites, hence the term 'signalling ciphersuite value'). A server that also supports SCSV can thus detect when a downgrade attack is occurring.

Cryptographic modelling of negotiation and downgrade resistance has been presented by Dowling and Stebila [258] and Bhargavan *et al.* [94].

6.11.2 Renegotiation Attack

Recall from Sect. 6.4.3 that renegotiation allows two parties who are using an existing TLS session to run a new handshake; upon completion of the new handshake, the session switches to the newly established encryption and authentication keys for



Attack 6.1: Ray and Dispensa's attack on TLS renegotiation

communication. In November 2009, Ray and Dispensa [623] described a man-in-the-middle attack that exploits how certain TLS-reliant applications—such as HTTP over TLS—process data across renegotiations. The attack is as shown in Attack 6.1.

The attacker Eve observes Alice attempting to establish a TLS session with Bob. Eve delays Alice's initial `ClientHello` and instead establishes her own TLS session with Bob, then transmits a message m_0 over that record layer. Then Eve passes Alice's initial `ClientHello` to Bob over the Eve–Bob record layer. Bob views this as a valid renegotiation and responds accordingly; Eve relays the handshake messages between Alice and Bob, which serve to establish a new record layer between Alice and Bob to which Eve has no access. Alice then transmits a message m_1 over the Alice–Bob record layer.

This is not, strictly speaking, an attack on TLS but on how some applications process TLS-protected data. It results from some applications, including many implementations of HTTPS [623] and SMTPS, concatenating m_0 and m_1 and treating them as coming from the same party in the same context. For example, if Eve sends the HTTP request

$$m_0 = \text{“GET /orderPizza?deliverTo=123-Fake-St} \hookleftarrow \text{X-Ignore-This: ”}$$

and Alice sends the HTTP request

$$m_1 = \text{“GET /orderPizza?deliverTo=456-Real-St} \hookleftarrow \text{Cookie: Account=111A2B”}$$

(where $\hookleftarrow$ denotes a new-line character), then the concatenated request (across multiple lines for readability) is

$$\begin{aligned} m_0 \| m_1 = \text{“}&\text{GET /orderPizza?deliverTo=123-Fake-St} \hookleftarrow \\ &\text{X-Ignore-This: GET /orderPizza?deliverTo=456-Real-St} \hookleftarrow \\ &\text{Cookie: Account=111A2B”} \end{aligned}$$

The ‘X-Ignore-This:’ prefix is an invalid HTTP header. Since this header, without a new-line character, is concatenated with the first line of Alice’s request, Bob’s application receives a full HTTP header with an unknown header name, so this line is ignored. However, the following line, Alice’s account cookie, is still processed. Eve is able to have the pizza delivered to herself but paid for by Alice’s account.

It should be noted that Ray and Dispensa’s attack works for both the server-only authentication and the mutual authentication modes of TLS: the use of client certificates does not in general prevent the attack.

The immediate recommendation due to this attack was to disable renegotiation except in cases where it was essential. The IETF TLS working group developed RFC 5746 [629] to provide countermeasures to this attack, with the goal of applicability to all versions of TLS and SSL v3. Two countermeasures were standardised:

Renegotiation Information Extension (RIE). This countermeasure essentially provides handshake recognition, confirming that both parties have the same view of the previous handshake when renegotiating. With this countermeasure, each client or server always includes a renegotiation information extension in its respective `ClientHello` or `ServerHello` message. This extension contains one of three values. If the party is not renegotiating, then it includes a fixed ‘empty’ string, which denotes that the party supports and understands the renegotiation extension, but that party is in fact not renegotiating. If the party is renegotiating, then it includes the handshake/key confirmation value from the previous handshake: the client sends the previous client verification data value while the server sends the concatenation of the previous client verification data and server verification data values. Intuitively, by including the verification data from the previous handshake, the parties can be assured that they have the same view of the previous handshake, and thus the attack in Fig. 6.1 is avoided.

Signalling Ciphersuite Value (SCSV). This countermeasure was designed to avoid interoperability problems with TLS 1.0 and SSL v3 implementations that did not gracefully ignore extension data at the end of `ClientHello` and `ServerHello` messages. Here, the client puts an additional value in its initial handshake—an extra, fixed, distinguished ciphersuite value (byte codes `0x00,0xFF`) included in its ciphersuite list—to indicate that it knows how to securely renegotiate. Old servers will ignore this extra value; new servers will recognise that the client supports secure renegotiation, and the server will use the RIE in the remainder of the session. In other words, the only difference between SCSV and RIE is in the `ClientHello` message of the initial handshake: with RIE, the client sends an empty extension, whereas with SCSV the client sends a distinguished value in the list of supported ciphersuites.

These countermeasures were quickly implemented in most TLS libraries, web browsers, and web servers. Giesen *et al.* [304] extended the ACCE model presented in Sect. 2.8.3 to formalise the notion of renegotiation security, and proved that these countermeasures provided renegotiation security for TLS.

6.11.3 Compression-Related Attacks: CRIME, BREACH

As described in Sect. 6.4.1, the TLS record layer can optionally perform compression of all application data. The *Compression Ratio Info-leak Made Easy* (CRIME) attack, presented by Rizzo and Duong in 2012 [632] makes use of a side channel based on the length of the compressed plaintext, first identified by Kelsey in 2002 [422], to extract secret values such as HTTP cookies from encrypted application data.

In 2002, Kelsey [422] described at a general level the various side channels that exist when plaintext is compressed prior to encryption; in particular, the length of the compressed plaintext compared with the original plaintext acts as a side channel, leaking some information about the amount of redundancy in the plaintext. One of the most powerful attacks described by Kelsey was an adaptive chosen input attack. Suppose the adversary is given access to an oracle that behaves as follows. When the adversary inputs a message x, the oracle outputs

$$\mathcal{O}_{a,b,S}(x) = \mathsf{Enc}(\mathsf{Comp}(a\|x\|b\|S)),$$

where Enc denotes encryption, Comp denotes compression, a and b are fixed public strings, and S is the secret the adversary is trying to recover. The basic idea of the attack is that, when the attacker's input x overlaps with S, the compressed string will be slightly shorter than when x does not overlap with S. The attacker works adaptively character by character.

Rizzo and Duong [632] showed how to use Kelsey's attack against TLS compression in the context of HTTP. On the web, we have a web browser making HTTP GET requests to an HTTPS website. In addition to the URL, the browser will send a header containing the authentication cookie; all of this is encrypted using TLS. For example, to visit `https://server.com/index.html`, the browser might send

$$\mathsf{Enc}(\mathsf{Comp}(\text{“}\texttt{GET /index.html}\hookleftarrow\texttt{Cookie: secret=31415926}\text{”})).$$

If the attacker can cause the user to visit arbitrary URLs on the target website `server.com`, then the browser acts as an oracle in Kelsey's adaptive chosen input attack above. In particular, if the attacker can cause the browser to visit the URL `https://server.com/`*url*, then the browser will send

$$\mathsf{Enc}(\mathsf{Comp}(\text{“}\texttt{GET /}\mathit{url}\hookleftarrow\texttt{Cookie: secret=31415926}\text{”})).$$

The attacker now iterates through different values of *url* (`secret=1`, `secret=2`, ...):

$$\mathsf{Enc}(\mathsf{Comp}(\text{“}\texttt{GET /secret=1}\hookleftarrow\texttt{Cookie: secret=31415926}\text{”})),$$
$$\mathsf{Enc}(\mathsf{Comp}(\text{“}\texttt{GET /secret=2}\hookleftarrow\texttt{Cookie: secret=31415926}\text{”})),$$
$$\mathsf{Enc}(\mathsf{Comp}(\text{“}\texttt{GET /secret=3}\hookleftarrow\texttt{Cookie: secret=31415926}\text{”})).$$

The third value will compress slightly more than the others, because of the greater overlap between `secret=3` and the cookie `secret=31415926`. The attacker now knows that the first character of the secret is `3`. The attacker now adaptively queries for the next character:

$$\mathsf{Enc}(\mathsf{Comp}(\text{“}\texttt{GET /secret=31}\hookleftarrow\texttt{Cookie: secret=31415926}\text{”}))$$

$$\mathsf{Enc}(\mathsf{Comp}(\text{“}\texttt{GET /secret=32}\hookleftarrow\texttt{Cookie: secret=31415926}\text{”}))$$

and so on. The attacker can rapidly recover the secret. In particular, assuming no noise in the repeated requests, for a secret of n characters chosen from a c-character alphabet, the attacker can recover the secret with at most nc adaptive queries. Since the compressed plaintext is encrypted, some care is required when block encryption schemes are used to ensure that different-length compressed plaintexts lead to different-length ciphertexts, but this is possible using padding like in the BEAST attack.

The CRIME attack was quite effective: it worked regardless of the cipher used (AES or RC4) and only required the victim to visit once a URL controlled by the attacker, although it required the attacker be able to observe the size of the responses transmitted over the network. Both the Chrome and the Firefox browsers were vulnerable (since they supported TLS compression), but Internet Explorer was not (since it did not support compression). The attack was also effective against the SPDY protocol [86, 323], a modification of the HTTP protocol to improve performance. The only effective countermeasure against CRIME is to disable TLS compression. The TIME attack [67] extends the CRIME attack by using a timing side channel via JavaScript, relieving the attacker from the need to observe the responses as they are transmitted over the network. The HEIST attack [720] is another variant that removes the need for a man-in-the-middle network observer.

In 2013, Prado *et al.* [620] announced the BREACH (Browser Reconnaissance and Exfiltration via Aadaptive Compression of Hypertext) attack, which used a similar adaptive chosen input attack, this time against HTTP compression, allowing an attacker to recover secrets in the HTTP body (such as anti-cross-site-request-forgery tokens). The BREACH attack works independently of TLS compression. Despite the attack, HTTP compression remains widespread.

6.11.4 Termination Attack

TLS includes an alert protocol which allows a party to signal a communication error to its peer, or to signal that it is terminating the connection. Each party is required to send a `close_notify` alert immediately before closing the write side of its connection, and the receiving party should respond with a `close_notify` alert and then terminate the connection. This functionality was introduced in SSL v3. Originally, a connection that had been terminated using `close_notify` could not be resumed using session resumption, but this behaviour was permitted starting in TLS 1.1.

The purpose of the `close_notify` alert is to ensure that each party's application agrees on the data sent and received, and in particular to avoid *truncation attacks*, in which an attacker drops some data as well as the `close_notify` alert. Unfortunately, several attacks have been identified involving how data is passed from SSL/TLS implementations to relying applications in the context of termination/truncation.

Berbecaru and Lioy [88] showed how various SSL v3 and TLS 1.0 implementations deal inconsistently with truncated data. For example, consider a web server that is transmitting an image using HTTPS to a web browser (Mozilla Firefox v1.5.0.7 in their tests), and a man-in-the-middle who cancels various parts of the transmission. They observed that if the MITM cancelled part of the data record encrypting the image, but allowed subsequent records including the `close_notify` to be transmitted, the browser would report it could not display the image because of errors, the expected behaviour. However, if the MITM cancelled part of the data record encrypting the image and *all* subsequent records, including the `close_notify` alert, the web browser would display the truncated image to the user. The SSL library did not transmit a warning to the application data that the received data was incomplete, and correspondingly the web browser did not abort in error, despite receiving an HTTP response that was shorter than indicated in the HTTP `Content-Length` header.

Smyth and Pironti [684] showed how to exploit this type of flaw in more complex modern web applications. For example, a user browsing a variety of Google properties (Gmail, Search, etc.) has sessions established with each service, even though a single sign-on is used. When the user clicks ‘Log out’, the user’s session must be logged out with each service. Smyth and Pironti observed that this is sometimes achieved using parallel logout requests to each service, followed by display of a success page. In their study, Smyth and Pironti showed how an attacker who can truncate/drop selected TLS packets can cause some sessions to remain active even after the user has clicked log out and received the success page.

6.11.5 Triple Handshake Attack

As noted in Sect. 6.4, TLS allows parties to create new sessions from existing ones. *Session resumption* uses an abbreviated handshake to start a new session from a prior session. *Renegotiation* performs a new, full handshake inside an existing session, allowing parties to change ciphersuites or authentication credentials, or obtain fresh keying material.

Bhargavan *et al.* [96] discovered a *triple handshake attack*, which allows an attacker to confuse a client into thinking it is connected to a different server; the attack takes advantage of flaws in session resumption and renegotiation. The attack proceeds in three steps. In the first step of the attack, the client establishes a TLS session using RSA key transport with the attacker; the attacker establishes a session with the target server using the same nonces and premaster secret. In the second step, the client uses session resumption with the attacker to resume its session; the attacker relays the session resumption handshake messages to the target server. Since the relevant cryptographic values are the same (specifically, the premaster secret), the session resumption succeeds, and the client now has a (resumed) session, but with the target server. The attacker injects a message to the target server, which it can do because it still knows the session key of the resumed session. In the final step, a renegotiation takes place between the client and the server, the result of which is a renegotiated session between the client and the target server which the adversary is not able to access. However, some applications concatenate the data sent *by the attacker before the final step* and the data sent *by the client after the final step* as coming from the same party (this is similar to the renegotiation attack in Sect. 6.11.2).

The primary reason that the triple handshake attack is successful is that session resumption depends only on the previous session's master secret, rather than the whole handshake. The principal proposed countermeasure is to bind the master secret to the full handshake, rather than just the nonces and premaster secret, and was standardised in RFC 7627 [102].

6.12 Attacks: Implementations

Software libraries implementing TLS, like other pieces of software, unfortunately contain bugs which often result in vulnerabilities. While we do not aim to provide an extensive overview of software-related TLS vulnerabilities, we will briefly mention a few notable ones. These subsections examine attacks resulting from bad random number generators and bad certificate validation code in software libraries.

Software vulnerabilities, by their nature, usually affect only a particular implementation. OpenSSL, being a widely used open source TLS implementation, is a frequent target of vulnerability research, but all TLS implementations have had flaws.

6.12.1 Side Channel Attacks

Several flaws in OpenSSL have been identified related to side channels in processor microarchitectures. Percival [608] successfully recovered an RSA private key from OpenSSL owing to a cache-timing side channel in its implementation of the Chinese Remainder Theorem. Acıiçmez *et al.* [20] identified a further vulnerability in RSA private key operations in OpenSSL due to a simple branch prediction analysis attack. Yarom and Benger [756] subsequently identified a cache-timing side channel that also allowed for recovery of signing keys when ECDSA over binary fields was used.

Brumley and Tuveri [162] discovered a timing side channel in OpenSSL's implementation of elliptic curve arithmetic over binary fields. The exploit allows an attacker to recover the ECDSA private key from a TLS server; the authors demonstrated the exploit over a loopback network interface.

Brumley *et al.* [161] discovered how to exploit a fault/error side-channel in OpenSSL's implementation of elliptic curve point multiplication for prime fields. Using an adaptive attack against repeated use of a NIST-P256 prime field elliptic curve private key, the attack recovers the private key in just 663 queries. The attack requires reuse of the private key, which is the case with static ECDH TLS ciphersuites and with ephemeral ECDH TLS ciphersuites where the server re-uses the ephemeral key as an optimisation technique. While reuse of ephemeral keys is optional, it is apparently the default behaviour of OpenSSL.

Most of the above attacks led to updated versions of OpenSSL and reports in the Common Vulnerabilities and Exposures (CVE) database. Although supported by OpenSSL, few applications actually make use of elliptic curves over binary fields, and, in particular, no commercial certificate authority to date has issued X.509 certificates for ECDSA public keys over binary fields, so the practical impact of the binary ECDSA vulnerabilities was minimal.

6.12.2 TLS-Specific Implementation Flaws

Researchers at Google and Codenomicon [220] independently identified a buffer over-read vulnerability in OpenSSL's implementation of the TLS heartbeat extension [661], allowing an attacker to read up to 64 KB of additional memory from the server, memory which could potentially contain private keys, passwords, or other sensitive information. The bug was given the moniker 'Heartbleed' and even got its own website and logo (`http://heartbleed.com`). Unlike the other attacks above, this bug requires no cryptographic expertise to discover or exploit. Turnkey exploits were readily developed and deployed, and researchers were successfully able to remotely recover signing private keys from test servers. Using the Zmap tool [262], researchers identified that approximately 17% (around half a million) of TLS-enabled websites were vulnerable at the time of discovery; 2 months later, 1.5% of the 800,000 most popular TLS-enabled websites remained vulnerable [484]. The Heartbleed vulnerability attracted widespread public attention.

Another specific class of implementation errors is related to processing messages in the correct order. Kikuchi [483] discovered a flaw in OpenSSL where it would accept `ChangeCipherSpec` messages at any point in time, rather than only immediately prior to the transmission of the `Finished` messages. If the `ChangeCipherSpec` message is accepted earlier, the server will use an empty master secret to compute the session keys. It is unclear how to exploit this weakness to defeat cryptographic protections, but it is still an implementation flaw. In the same vein, Beurdouche *et al.* [92] identified several more state machine attacks (which they named 'SMACK'). They discovered that many TLS implementations do not correctly implement the TLS state machine: the various TLS versions, extensions, authentication modes, and key exchange methods may lead to different sequences of messages and processing, and failure to use the correct sequence in every context can lead to vulnerabilities.

6.12.3 Certificate Validation

A notable class of implementation errors is related to certificate validation, either in TLS implementations directly or in applications or higher-level libraries that make use of TLS implementations.

OpenSSL releases prior to 0.9.8j failed to correctly validate some types of malformed signatures in a certificate chain, allowing an attacker to bypass validation [595]. Apple's implementation of SSL/TLS in its Secure Transport library for Mac OS X and iOS had a bug in which certificate validation always succeeded owing to a spurious `goto fail` statement in the code, giving the bug its name [476].

Georgiev *et al.* [302] reported SSL certificate validation errors in a variety of non-browser applications and libraries, typically related to incorrect parameters used in libraries. For example, Amazon provided third-party PHP developers with a Flexible Payments Service library allowing them to receive payments. The library made use of the command-line program cURL for HTTPS connections. The library attempted to enable hostname verification in cURL by setting an option `CURLOPT_SSL_VERIFYHOST` = `true`. However, the correct value should have been 2 rather than `true`, and the effect was that certificate validation was disabled, allowing an attacker to carry out an impersonation attack. Similar mistakes involving hostname verification, certificate revocation, and certificate chain validation were observed by these authors in a variety of libraries using a variety of programming languages and applications.

In 2014, Brubaker *et al.* [160] reported on the results of testing the certificate validation logic of a variety of client and server TLS implementation using *frankencerts*, described as “synthetic certificates that randomly mutated from parts of real certificates and thus include unusual combinations of extensions and constraints”. When one implementation accepted a frankencert while another did not, this identified a discrepancy in certificate validation logic meriting further investigation. Overall, Brubaker *et al.* found 208 discrepancies between popular implementations, many of which led to significant vulnerabilities, such as the ability to act as a rogue CA, to create certificates not signed by a trusted CA, to accept certificates not intended for authentication, or to trigger user interface indicators that provide inaccurate warnings.

6.12.4 Bad Random Number Generators

The security of any TLS implementation depends crucially on the quality of the random number generator and seed used for the picking of long-term private keys and per-session private keys. Several prominent SSL/TLS implementations have had flaws in how their random number generators are seeded, leading to exploitable attacks.

In 1996, Goldberg and Wagner [309] determined, by reverse engineering, that the pseudo-random number generator (PRNG) in one of the earliest web browsers, Netscape Navigator v1, was poorly seeded. They found that the seed to the PRNG in Netscape was constructed from the process ID (pid), parent process ID (ppid), and the time in seconds and microseconds. The pid and ppid can be easily determined by another user running on the same system; for a remote attacker, there are at most 2^{27} possible pid/ppid value pairs, and often the pid and ppid values are considerably smaller. A local or network attacker can determine the time in seconds based on network observations, leaving at most 1,000,000 (approximately 2^{20}) values to test for the microseconds. Thus, for an attacker, there are at most 47 bits of entropy in the seed of the PRNG and hence at most 47 bits of randomness in the secret keys. Netscape Navigator version 1.22 and higher improved the seeding of the PRNG to avoid this problem, but the flaw still serves as a significant real-world example of the challenges of random number generation.

In 2008, Bello [710] discovered a flaw in the random number generator used by the OpenSSL library package distributed on the Debian operating system. The flaw was introduced in 2006 when Debian maintainers commented out a couple of lines of code in OpenSSL when updating the package for Debian; the flaw was not present in the original OpenSSL source code. The effect of the code change was to remove almost all of the entropy added to the PRNG seed, leaving the sole randomness as the process ID. Taking into account the effects of different processor architectures, there are just 294,912 keys that can possibly be generated by the vulnerable versions of Debian OpenSSL [761]. Because the OpenSSL library is used in many applications, this flaw affected not only TLS, but also SSH and OpenVPN (for virtual private networking). In a survey of 50,000 SSL servers [761] on the day of vulnerability disclosure in 2009, approximately 700 (1.4%) were using a weak key; 6 months later, 30% of those Debian weak keys were still in use. In a later, large-scale survey of 12.8 million TLS servers and 10.2 million SSH servers published in 2012 [354], Heninger *et al.* found a very small percentage (0.03%) of TLS servers using Debian weak keys, though a non-trivial percentage (0.52%) of SSH servers were still using Debian weak keys.

6.13 Attacks: Other

Finally, there are a few other attacks which do not fit into the categories explored above.

6.13.1 Application-Level Protocols

The most common use of TLS is in combination with the Hypertext Transport Protocol (HTTP). Because much web traffic still runs over HTTP, rather than HTTPS, security vulnerabilities can arise owing to how HTTP and HTTPS interact.

One common flow on the web is for users to initially browse a website using plaintext (HTTP) communication. At some point, such as when they click ‘login’ or try to purchase something, they are redirected to a secure site, using an HTTPS address specified in the web page’s source code. In 2009, Marlinspike [524] demonstrated an HTTPS ‘stripping’ attack using a new tool called ‘sslstrip’. A man-in-the-middle attacker observing an unencrypted HTTP connection alters the source code of responses from the server to replace HTTPS URLs with HTTP URLs: thus, when the user clicks the ‘login’ button and enters her password, the password is transmitted unencrypted.

To prevent SSL stripping, the HTTP Strict Transport Security (HSTS) mechanism was created [360]. HSTS allows a web server to tell a client to automatically use HTTPS for all pages on a certain domain for a certain period of time, even if the URL is given as HTTP. This is a trust-on-first-use mechanism, so it requires that the client make at least one non-adversary-controlled connection to the server. Despite HSTS, HTTPS stripping can still be achieved by an attacker who rewrites links in plaintext pages to alternative domain names that are not covered by an HSTS policy [523].

HTTP servers that serve multiple sites—such as content distribution networks—have also been vulnerable to ‘virtual host confusion’ attacks [239]. These attacks depend on a complex interaction between TLS and application characteristics, so we omit the details; the root cause of the attacks is that servers have many different ‘identities’ at different levels of the network stack (IP, TCP, TLS, HTTP), and they do not always match the identity that is cryptographically authenticated by TLS.

Other application-level protocols also use TLS, but in a different way from HTTPS. Whereas HTTPS runs on a separate TCP port from HTTP and immediately begins with a TLS handshake, other applications, such as IMAP, POP3, XMPP, NNTP, IRC, and FTP, run the secure and plaintext versions on the same port: the parties start with a plaintext communication, then use a protocol-specific command, often called ‘STARTTLS’, to initiate a TLS handshake for secure communication. Venema and Orlando [722] discovered vulnerabilities in many applications’ use of STARTTLS that allowed an attacker to inject commands in the plaintext portion of the protocol that would be executed during the encrypted portion of the protocol.

6.13.2 Certificate Authority Breaches and Related Flaws

Since authentication in TLS uses certificates, users’ security can be undermined by mistakes or by attacks on certificate authorities. In some applications, TLS clients may be configured to accept certificates from a single CA or a very small number of CAs (see Sect. 6.12.3 for cases when this should—but does not—happen).

In many settings, however, TLS clients are configured to accept certificates from many CAs. On the public web, mainstream web browsers ship with 150–200 root CA certificates that are trusted by default, issued by some 80+ organizations. Some of these root CAs issue web server certificates directly or via an intermediate CA run by the same company, but many root CAs also issue certificates for subordinate CAs run by other organizations, who then also issue web server certificates directly or via intermediate CAs. For example, at the time of writing, an HTTPS connection to `eprint.iacr.org` returned a certificate issued by ‘RapidSSL SHA256 CA - G3’, whose certificate was issued by ‘GeoTrust Global CA’. The subordinate CA, RapidSSL, is a separate organization from the root CA, GeoTrust.

An Internet-wide scan by the Electronic Frontier Foundation’s SSL Observatory project in 2010 found more than 650 organizations that were trusted by default by major browsers as CAs [266]. In general, any one of these organizations has the ability to issue a browser-trusted certificate for any domain name, without geographic or other restrictions. As a result, there are many potential points of failure in the CA ecosystem. The CA/Browser Forum is a consortium of CAs and browser vendors who work together to agree on guidelines for the operation of CAs and the criteria for root CA inclusion in browsers and operating systems.

There have been several incidents involving CAs mistakenly issuing certificates. Among the most dramatic was the DigiNotar breach of 2011 [282]. DigiNotar was a Dutch CA that issued certificates under two main CAs, one for public websites (‘DigiNotar Root CA’) and one for the Dutch government. The certificate for ‘DigiNotar Root CA’ was built in to all major browsers. In July 2011, DigiNotar mistakenly issued a wildcard certificate for `*.google.com`. This certificate was used in Iran in August 2011 to conduct man-in-the-middle attacks against users of Google services. Subsequently, it was discovered that DigiNotar had issued more than 500 other fraudulent certificates, including ones for Yahoo!, Mozilla, WordPress, and the Tor project. All major browser vendors issued updates that removed 'DigiNotar Root CA' from their browser's list of trusted CAs. DigiNotar declared bankruptcy in September 2011.

The DigiNotar breach was detected owing to Google Chrome's use of *certificate pinning*. At the time, Google Chrome had hard-coded the fact that Google services would use only certificates from certain CAs, so Chrome users were in fact protected against the man-in-the-middle attack on Google services, while other users would not have been. This ad hoc approach has been standardised as HTTP Public Key Pinning [272], which allows web servers to specify that future connections to that same domain will be secured using a certain certificate or CA. This limits the ability of an attacker to make use of a fraudulently issued certificate in the event of a CA breach, though HTTP Public Key Pinning is a trust-on-first-use mechanism, meaning it only provides security if the *initial* connection between the client and the server does not involve a fraudulently issued certificate.

System administrators and local software can also modify the browser or operating system's list of trusted root CAs. This can be done for legitimate reasons, to support a company's internal PKI or to enable an HTTPS proxy to inspect downloads for viruses, but can also be done maliciously. Some malware is known to do this to enable further attacks. In 2015, it was discovered that Lenovo computers were shipping with a piece of advertising software called Superfish which added a root CA certificate to computers so that it could act as an HTTP/HTTPS proxy and inject advertising into web pages. Unfortunately, the software used the same root CA certificate on all computers, and, moreover, the corresponding private key was also included in the software, so anyone who could reverse engineer the software could recover the root CA key and impersonate any web server to any user with the Superfish software installed [40]. Interestingly, public key pinning would not protect against this attack, as browsers are configured to allow locally installed certificates to override pins.

6.14 TLS Version 1.3

In 2014, the Internet Engineering Task Force began a multi-year process to develop the next version of TLS, now called TLS version 1.3. In August 2018, TLS 1.3 was standardised in RFC 8446 [626]. There were several motivations for the development of TLS 1.3:

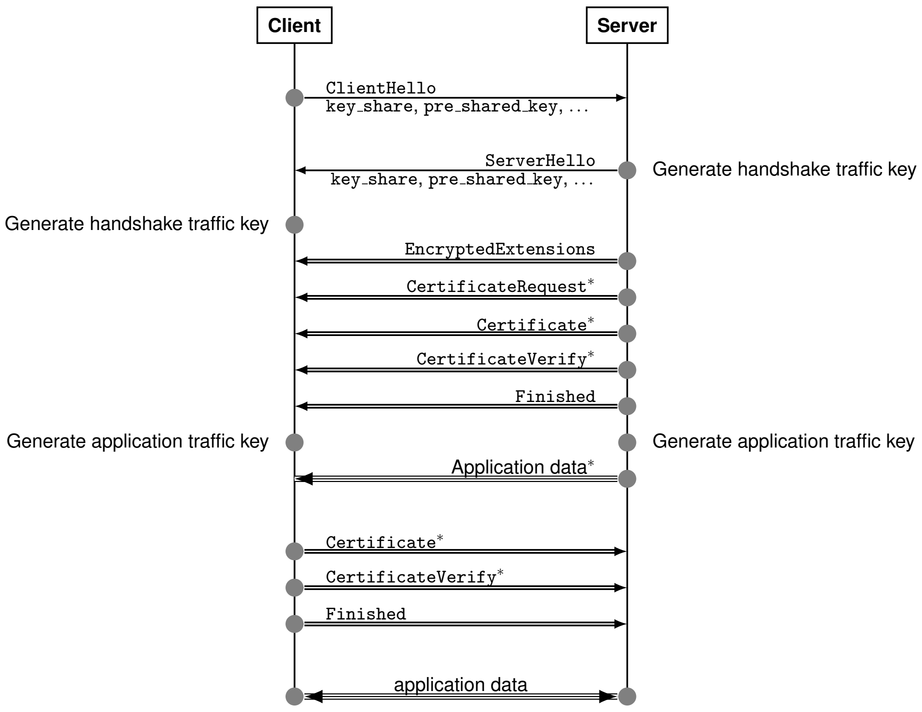
1. to deprecate old cryptographic algorithms and switch to modern algorithms and modes of operation;
2. to encrypt parts of the handshake to enhance privacy, in part as a response to concerns about mass surveillance;
3. to reduce the latency of connection establishment by providing modes with fewer round trips;

4. and to make general changes to cryptographic and other operations of the protocol, including simplifying protocol logic.

To achieve the first goal, many cryptographic algorithms that had been in use in TLS 1.2 and earlier versions were removed from TLS 1.3. The symmetric encryption and integrity algorithms in the record layer protocol had been the cause of several vulnerabilities; TLS 1.3 removed the MAC-then-encode-then-encrypt mode of operation, and focused on provably secure authenticated encryption schemes, mandating AES in Galois/counter mode, and optionally ChaCha20 with Poly1305. In terms of public key cryptography, static Diffie–Hellman ciphersuites were removed, as was key exchange based on RSA key transport; RSA-based digital signatures within TLS must use the PSS algorithm, rather than PKCS #1 v1.5 (although certificates may still be signed using PKCS #1 v1.5). All asymmetric key exchange algorithms in TLS 1.3 are based on ephemeral Diffie–Hellman to provide forward secrecy. New elliptic curve algorithms are available, including the use of Curve25519 in key exchange (X25519) and signatures (Ed25519), and some finite-field and elliptic curve groups have been removed.

To achieve the second goal of enhancing privacy, the flow of messages in the handshake algorithm was substantially altered. Most notably, unauthenticated ephemeral key exchange happens in the first two flows of the protocol, after which the parties establish a temporary 'handshake traffic key' which is used to encrypt the remainder of the handshake, including the transmission of certificates, before finally switching to the 'application traffic key' which is used to encrypt application data in the record layer protocol. Protocol 6.3 shows the handshake protocol for a full handshake in TLS 1.3. The client sends one or more ephemeral keys in the `key_share` extension of the `ClientHello` message. Since the parties have at this point not yet negotiated algorithms, the client is simply guessing which ephemeral key exchange algorithms might be acceptable to the server; if the server does not support any of the algorithms offered by the client, the server can send a `HelloRetryRequest` message with a list of algorithms to ask the client to retry.

To reduce latency of connection establishment, a new zero-round-trip (0-RTT) mode of operation is available when pre-shared keys are being used for session establishment (which includes session resumption). This allows the client to send application data immediately on its first flow to the server, rather than having to wait until receiving a response from the server. In 0-RTT mode, the client derives an early application traffic key from the pre-shared key, and uses this to encrypt its first application flow, which it sends along with the `ClientHello`. The client and server can still optionally negotiate a forward secure ephemeral shared secret in this mode, but that ephemeral secret will only apply to subsequent application data flows, not the first one from the client to the server. Protocol 6.4 shows the 0-RTT handshake mode. One concern regarding 0-RTT mode is that the first client-to-server flow can be replayed; the TLS 1.3 document discusses some potential mitigations for servers to prevent replay attacks, such as keeping state and using application-level protections. TLS 1.3 has no explicit session resumption mode; instead, a 'resumption master se-


Client
Server
ClientHello
key_share, pre_shared_key, ...
ServerHello
key_share, pre_shared_key, ...
Generate handshake traffic key
Generate handshake traffic key
EncryptedExtensions
CertificateRequest*
Certificate*
CertificateVerify*
Finished
Generate application traffic key
Generate application traffic key
Application data*
Certificate*
CertificateVerify*
Finished
application data

* denotes messages that may not be present in all ciphersuites.
Single lines denote plaintext flows; double lines denote encrypted flows using the handshake traffic key; triple lines denote encrypted flows using the application traffic key.

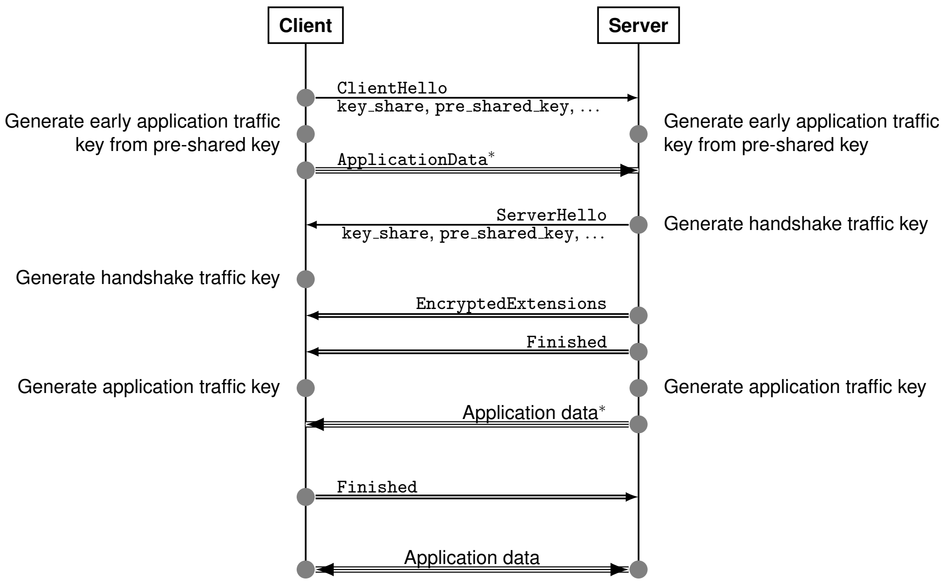
Protocol 6.3: TLS 1.3 handshake protocol – full handshake

cret' can be exported from any session, and this value can be used as a pre-shared key in a subsequent PSK or 0-RTT handshake.

Finally, many other changes have been made throughout the protocol, some cryptographic, some not. The key derivation function is now HKDF [454], and there is a new key derivation schedule that incorporates each new piece of keying material and generates all necessary traffic keys. The handshake state machine has been reorganized, and the `ChangeCipherSpec` messages have been removed. Digital signatures in the `CertificateVerify` message now cover the entire handshake transcript. Ciphersuite negotiation has been made modular: each cryptographic component (authenticated encryption algorithm, digital signature algorithm, key exchange algorithm) is negotiated separately.

All major browser and library vendors have committed to support TLS 1.3, and as of December 2018, support is available in the Chrome and Firefox browsers, and the OpenSSL and NSS libraries.

One notable aspect of the development of TLS 1.3 has been the early and ongoing involvement of the academic community. The core cryptographic design of TLS 1.3


Client
Server
ClientHello
key_share, pre_shared_key, ...
Generate early application traffic key from pre-shared key
Generate early application traffic key from pre-shared key
ApplicationData*
ServerHello
key_share, pre_shared_key, ...
Generate handshake traffic key
Generate handshake traffic key
EncryptedExtensions
Finished
Generate application traffic key
Generate application traffic key
Application data*
Finished
Application data

* denotes messages that may not be present in all ciphersuites.
Single lines denote plaintext flows; double lines denote encrypted flows using the handshake traffic key; triple lines denote encrypted flows using the early or regular application traffic key.

Protocol 6.4: TLS 1.3 handshake protocol – pre-shared key handshake with early application data ('zero-round-trip')

was heavily influenced by the OPTLS protocol of Krawczyk and Wee [457]. There were a variety of academic papers published analyzing and commenting on various aspects of drafts of TLS 1.3, including results using provable security [257, 457, 487], constructive cryptography [444], and formal methods [230], and the miTLS team's efforts of using formal methods combined with a verified implementation [92, 95]. A workshop called 'TLS 1.3: Ready or Not? (TRON)' was held in February 2016 that brought together academic researchers, industry professionals, and members of the IETF TLS working group to exchange knowledge. In late 2016, Paterson and van der Merwe [604] published an account of the TLS 1.3 standardization effort.