

Network Working Group
Internet-Draft
Obsoletes: [5077](#), [5246](#), [5746](#) (if
approved)
Updates: [4492](#), [6066](#), [6961](#) (if approved)
Intended status: Standards Track
Expires: February 18, 2017

E. Rescorla
RTFM, Inc.
August 17, 2016

The Transport Layer Security (TLS) Protocol Version 1.3
draft-ietf-tls-tls13-15

Abstract

This document specifies version 1.3 of the Transport Layer Security (TLS) protocol. TLS allows client/server applications to communicate over the Internet in a way that is designed to prevent eavesdropping, tampering, and message forgery.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <http://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on February 18, 2017.

Copyright Notice

Copyright (c) 2016 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<http://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in [Section 4.e](#) of

the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

This document may contain material from IETF Documents or IETF Contributions published or made publicly available before November 10, 2008. The person(s) controlling the copyright in some of this material may not have granted the IETF Trust the right to allow modifications of such material outside the IETF Standards Process. Without obtaining an adequate license from the person(s) controlling the copyright in such materials, this document may not be modified outside the IETF Standards Process, and derivative works of it may not be created outside the IETF Standards Process, except to format it for publication as an RFC or to translate it into languages other than English.

Table of Contents

1.	Introduction	4
1.1.	Conventions and Terminology	5
1.2.	Major Differences from TLS 1.2	6
1.3.	Updates Affecting TLS 1.2	11
2.	Protocol Overview	11
2.1.	Incorrect DHE Share	14
2.2.	Resumption and Pre-Shared Key (PSK)	15
2.3.	Zero-RTT Data	17
3.	Presentation Language	18
3.1.	Basic Block Size	18
3.2.	Miscellaneous	19
3.3.	Vectors	19
3.4.	Numbers	20
3.5.	Enumerateds	20
3.6.	Constructed Types	21
3.6.1.	Variants	21
3.7.	Constants	23
4.	Handshake Protocol	23
4.1.	Key Exchange Messages	24
4.1.1.	Cryptographic Negotiation	25
4.1.2.	Client Hello	26
4.1.3.	Server Hello	28
4.1.4.	Hello Retry Request	29
4.2.	Hello Extensions	31
4.2.1.	Cookie	32
4.2.2.	Signature Algorithms	33
4.2.3.	Negotiated Groups	36
4.2.4.	Key Share	37
4.2.5.	Pre-Shared Key Extension	39
4.2.6.	Early Data Indication	41
4.2.7.	OCSF Status Extensions	44

4.2.8.	Encrypted Extensions	45
4.2.9.	Certificate Request	45
4.3.	Authentication Messages	47
4.3.1.	Certificate	49
4.3.2.	Certificate Verify	52
4.3.3.	Finished	54
4.4.	Post-Handshake Messages	55
4.4.1.	New Session Ticket Message	56
4.4.2.	Post-Handshake Authentication	57
4.4.3.	Key and IV Update	58
4.5.	Handshake Layer and Key Changes	59
5.	Record Protocol	59
5.1.	Record Layer	59
5.2.	Record Payload Protection	61
5.3.	Per-Record Nonce	63
5.4.	Record Padding	63
5.5.	Limits on Key Usage	64
6.	Alert Protocol	65
6.1.	Closure Alerts	66
6.2.	Error Alerts	67
7.	Cryptographic Computations	70
7.1.	Key Schedule	70
7.2.	Updating Traffic Keys and IVs	73
7.3.	Traffic Key Calculation	73
7.3.1.	Diffie-Hellman	74
7.3.2.	Elliptic Curve Diffie-Hellman	75
7.3.3.	Exporters	75
8.	Compliance Requirements	75
8.1.	MTI Cipher Suites	76
8.2.	MTI Extensions	76
9.	Security Considerations	77
10.	IANA Considerations	77
11.	References	80
11.1.	Normative References	80
11.2.	Informative References	83
Appendix A.	Protocol Data Structures and Constant Values	90
A.1.	Record Layer	90
A.2.	Alert Messages	90
A.3.	Handshake Protocol	92
A.3.1.	Key Exchange Messages	92
A.3.2.	Server Parameters Messages	96
A.3.3.	Authentication Messages	97
A.3.4.	Ticket Establishment	97
A.4.	Cipher Suites	98
A.4.1.	Unauthenticated Operation	99
Appendix B.	Implementation Notes	100
B.1.	API considerations for 0-RTT	100
B.2.	Random Number Generation and Seeding	100

B.3.	Certificates and Authentication	100
B.4.	Cipher Suite Support	100
B.5.	Implementation Pitfalls	101
B.6.	Client Tracking Prevention	102
Appendix C.	Backward Compatibility	102
C.1.	Negotiating with an older server	103
C.2.	Negotiating with an older client	104
C.3.	Zero-RTT backwards compatibility	104
C.4.	Backwards Compatibility Security Restrictions	105
Appendix D.	Overview of Security Properties	106
D.1.	Handshake	106
D.2.	Record Layer	108
Appendix E.	Working Group Information	109
Appendix F.	Contributors	109
Author's Address	113

1. Introduction

DISCLAIMER: This is a WIP draft of TLS 1.3 and has not yet seen significant security analysis.

RFC EDITOR: PLEASE REMOVE THE FOLLOWING PARAGRAPH The source for this draft is maintained in GitHub. Suggested changes should be submitted as pull requests at <https://github.com/tlswg/tls13-spec>.

Instructions are on that page as well. Editorial changes can be managed in GitHub, but any substantive change should be discussed on the TLS mailing list.

The primary goal of TLS is to provide a secure channel between two communicating peers. Specifically, the channel should provide the following properties.

- Authentication: The server side of the channel is always authenticated; the client side is optionally authenticated. Authentication can happen via asymmetric cryptography (e.g., RSA [RSA], ECDSA [ECDSA]) or a pre-shared symmetric key.
- Confidentiality: Data sent over the channel is not visible to attackers.
- Integrity: Data sent over the channel cannot be modified by attackers.

These properties should be true even in the face of an attacker who has complete control of the network, as described in [RFC3552]. See [Appendix D](#) for a more complete statement of the relevant security properties.

TLS consists of two primary components:

- A handshake protocol ([Section 4](#)) which authenticates the communicating parties, negotiates cryptographic modes and parameters, and establishes shared keying material. The handshake protocol is designed to resist tampering; an active attacker should not be able to force the peers to negotiate different parameters than they would if the connection were not under attack.
- A record protocol ([Section 5](#)) which uses the parameters established by the handshake protocol to protect traffic between the communicating peers. The record protocol divides traffic up into a series of records, each of which is independently protected using the traffic keys.

TLS is application protocol independent; higher-level protocols can layer on top of TLS transparently. The TLS standard, however, does not specify how protocols add security with TLS; the decisions on how to initiate TLS handshaking and how to interpret the authentication certificates exchanged are left to the judgment of the designers and implementors of protocols that run on top of TLS.

This document defines TLS version 1.3. While TLS 1.3 is not directly compatible with previous versions, all versions of TLS incorporate a versioning mechanism which allows clients and servers to interoperably negotiate a common version if one is supported.

1.1. Conventions and Terminology

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC 2119](#) [[RFC2119](#)].

The following terms are used:

client: The endpoint initiating the TLS connection.

connection: A transport-layer connection between two endpoints.

endpoint: Either the client or server of the connection.

handshake: An initial negotiation between client and server that establishes the parameters of their transactions.

peer: An endpoint. When discussing a particular endpoint, "peer" refers to the endpoint that is remote to the primary subject of discussion.

receiver: An endpoint that is receiving records.

sender: An endpoint that is transmitting records.

session: An association between a client and a server resulting from a handshake.

server: The endpoint which did not initiate the TLS connection.

1.2. Major Differences from TLS 1.2

(*) indicates changes to the wire protocol which may require implementations to update.

[draft-15](#)

- New negotiation syntax as discussed in Berlin (*)
- Require CertificateRequest.context to be empty during handshake (*)
- Forbid empty tickets (*)
- Forbid application data messages in between post-handshake messages from the same flight (*)
- Clean up alert guidance (*)
- Clearer guidance on what is needed for TLS 1.2.
- Guidance on 0-RTT time windows.
- Rename a bunch of fields.
- Remove old PRNG text.
- Explicitly require checking that handshake records not span key changes.

[draft-14](#)

- Allow cookies to be longer (*)

- Remove the "context" from EarlyDataIndication as it was undefined and nobody used it (*).
- Remove 0-RTT EncryptedExtensions and replace the ticket_age extension with an obfuscated version. Also necessitates a change to NewSessionTicket (*).
- Move the downgrade sentinel to the end of ServerHello.Random to accomodate tlsdate (*).
- Define ecdsa_shal (*).
- Allow resumption even after fatal alerts. This matches current practice.
- Remove non-closure warning alerts. Require treating unknown alerts as fatal.
- Make the rules for accepting 0-RTT less restrictive.
- Clarify 0-RTT backward-compatibility rules.
- Clarify how 0-RTT and PSK identities interact.
- Add a section describing the data limits for each cipher.
- Major editorial restructuring.
- Replace the Security Analysis section with a WIP draft.

[draft-13](#)

- Allow server to send SupportedGroups.
- Remove 0-RTT client authentication
- Remove (EC)DHE 0-RTT.
- Flesh out 0-RTT PSK mode and shrink EarlyDataIndication
- Turn PSK-resumption response into an index to save room
- Move CertificateStatus to an extension
- Extra fields in NewSessionTicket.
- Restructure key schedule and add a resumption_context value.

- Require DH public keys and secrets to be zero-padded to the size of the group.
- Remove the redundant length fields in KeyShareEntry.
- Define a cookie field for HRR.

[draft-12](#)

- Provide a list of the PSK cipher suites.
- Remove the ability for the ServerHello to have no extensions (this aligns the syntax with the text).
- Clarify that the server can send application data after its first flight (0.5 RTT data)
- Revise signature algorithm negotiation to group hash, signature algorithm, and curve together. This is backwards compatible.
- Make ticket lifetime mandatory and limit it to a week.
- Make the purpose strings lower-case. This matches how people are implementing for interop.
- Define exporters.
- Editorial cleanup

[draft-11](#)

- Port the CFRG curves & signatures work from RFC4492bis.
- Remove sequence number and version from additional_data, which is now empty.
- Reorder values in HkdfLabel.
- Add support for version anti-downgrade mechanism.
- Update IANA considerations section and relax some of the policies.
- Unify authentication modes. Add post-handshake client authentication.
- Remove early_handshake content type. Terminate 0-RTT data with an alert.

- Reset sequence number upon key change (as proposed by Fournet et al.)

draft-10

- Remove ClientCertificateTypes field from CertificateRequest and add extensions.
- Merge client and server key shares into a single extension.

draft-09

- Change to RSA-PSS signatures for handshake messages.
- Remove support for DSA.
- Update key schedule per suggestions by Hugo, Hoeteck, and Bjoern Tackmann.
- Add support for per-record padding.
- Switch to encrypted record ContentType.
- Change HKDF labeling to include protocol version and value lengths.
- Shift the final decision to abort a handshake due to incompatible certificates to the client rather than having servers abort early.
- Deprecate SHA-1 with signatures.
- Add MTI algorithms.

draft-08

- Remove support for weak and lesser used named curves.
- Remove support for MD5 and SHA-224 hashes with signatures.
- Update lists of available AEAD cipher suites and error alerts.
- Reduce maximum permitted record expansion for AEAD from 2048 to 256 octets.
- Require digital signatures even when a previous configuration is used.
- Merge EarlyDataIndication and KnownConfiguration.

- Change code point for server_configuration to avoid collision with server_hello_done.
- Relax certificate_list ordering requirement to match current practice.

draft-07

- Integration of semi-ephemeral DH proposal.
- Add initial 0-RTT support.
- Remove resumption and replace with PSK + tickets.
- Move ClientKeyShare into an extension.
- Move to HKDF.

draft-06

- Prohibit RC4 negotiation for backwards compatibility.
- Freeze & deprecate record layer version field.
- Update format of signatures with context.
- Remove explicit IV.

draft-05

- Prohibit SSL negotiation for backwards compatibility.
- Fix which MS is used for exporters.

draft-04

- Modify key computations to include session hash.
- Remove ChangeCipherSpec.
- Renumber the new handshake messages to be somewhat more consistent with existing convention and to remove a duplicate registration.
- Remove renegotiation.
- Remove point format negotiation.

draft-03

- Remove GMT time.
- Merge in support for ECC from [RFC 4492](#) but without explicit curves.
- Remove the unnecessary length field from the AD input to AEAD ciphers.
- Rename {Client,Server}KeyExchange to {Client,Server}KeyShare.
- Add an explicit HelloRetryRequest to reject the client's.

[draft-02](#)

- Increment version number.
- Rework handshake to provide 1-RTT mode.
- Remove custom DHE groups.
- Remove support for compression.
- Remove support for static RSA and DH key exchange.
- Remove support for non-AEAD ciphers.

1.3. Updates Affecting TLS 1.2

This document defines several changes that optionally affect implementations of TLS 1.2:

- A version downgrade protection mechanism is described in [Section 4.1.3](#).
- RSASSA-PSS signature schemes are defined in [Section 4.2.2](#).

An implementation of TLS 1.3 that also supports TLS 1.2 might need to include changes to support these changes even when TLS 1.3 is not in use. See the referenced sections for more details.

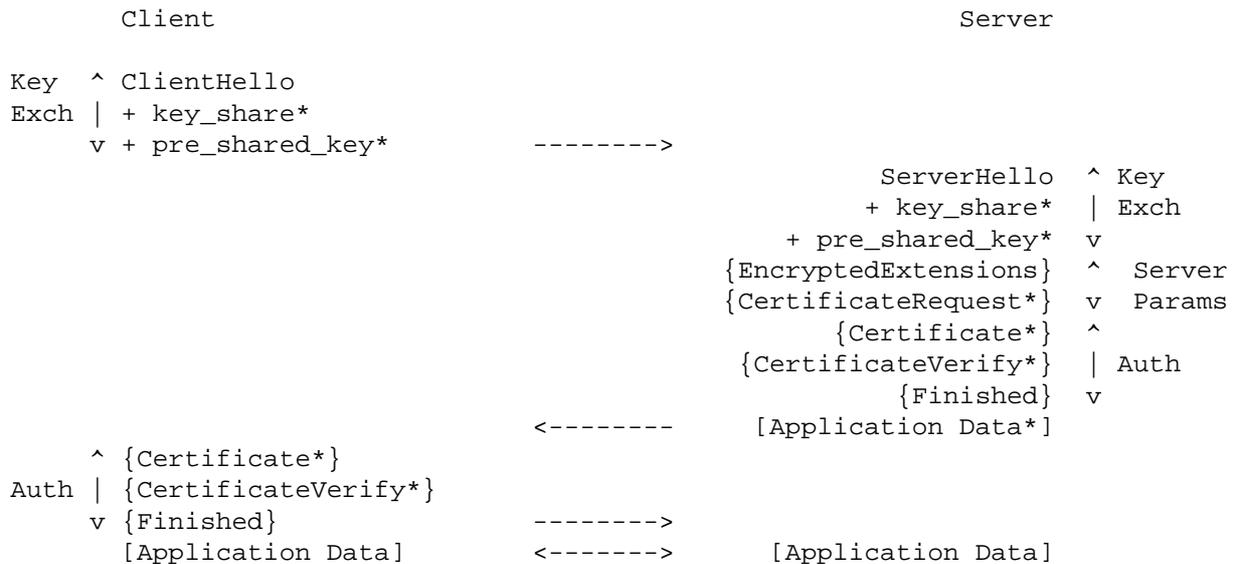
2. Protocol Overview

The cryptographic parameters of the session state are produced by the TLS handshake protocol. When a TLS client and server first start communicating, they agree on a protocol version, select cryptographic algorithms, optionally authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect application layer traffic.

TLS supports three basic key exchange modes:

- Diffie-Hellman (of both the finite field and elliptic curve varieties).
- A pre-shared symmetric key (PSK)
- A combination of a symmetric key and Diffie-Hellman

Figure 1 below shows the basic full TLS handshake:



+ Indicates extensions sent in the previously noted message.

* Indicates optional or situation-dependent messages that are not always sent.

{ } Indicates messages protected using keys derived from handshake_traffic_secret.

[] Indicates messages protected using keys derived from traffic_secret_N

Figure 1: Message flow for full TLS Handshake

The handshake can be thought of as having three phases, indicated in the diagram above.

- Key Exchange: Establish shared keying material and select the cryptographic parameters. Everything after this phase is encrypted.
- Server Parameters: Establish other handshake parameters. (whether the client is authenticated, application layer protocol support, etc.)
- Authentication: Authenticate the server (and optionally the client) and provide key confirmation and handshake integrity.

In the Key Exchange phase, the client sends the ClientHello (Section 4.1.2) message, which contains a random nonce (ClientHello.random), its offered protocol version, a list of symmetric cipher/HKDF hash pairs, some set of Diffie-Hellman key shares (in the "key_share" extension Section 4.2.4), one or more pre-shared key labels (in the "pre_shared_key" extension Section 4.2.5), or both, and potentially some other extensions.

The server processes the ClientHello and determines the appropriate cryptographic parameters for the connection. It then responds with its own ServerHello which indicates the negotiated connection parameters. [Section 4.1.3]. The combination of the ClientHello and the ServerHello determines the shared keys. If (EC)DHE key establishment is in use, then the ServerHello will contain a "key_share" extension with the server's ephemeral Diffie-Hellman share which MUST be in the same group as one of the client's shares. If PSK key establishment is in use, then the ServerHello will contain a "pre_shared_key" extension indicating which of the client's offered PSKs was selected. Note that implementations can use (EC)DHE and PSK together, in which case both extensions will be supplied.

The server then sends two messages to establish the Server Parameters:

EncryptedExtensions. responses to any extensions which are not required in order to determine the cryptographic parameters. [Section 4.2.8]

CertificateRequest. if certificate-based client authentication is desired, the desired parameters for that certificate. This message will be omitted if client authentication is not desired.

Finally, the client and server exchange Authentication messages. TLS uses the same set of messages every time that authentication is needed. Specifically:

Certificate. the certificate of the endpoint. This message is omitted if the server is not authenticating with a certificate. Note that if raw public keys [RFC7250] or the cached information extension [RFC7924] are in use, then this message will not contain a certificate but rather some other value corresponding to the server's long-term key. [Section 4.3.1]

CertificateVerify. a signature over the entire handshake using the public key in the Certificate message. This message is omitted if the server is not authenticating via a certificate. [Section 4.3.2]

Finished. a MAC (Message Authentication Code) over the entire handshake. This message provides key confirmation, binds the endpoint's identity to the exchanged keys, and in PSK mode also authenticates the handshake. [Section 4.3.3]

Upon receiving the server's messages, the client responds with its Authentication messages, namely Certificate and CertificateVerify (if requested), and Finished.

At this point, the handshake is complete, and the client and server may exchange application layer data. Application data **MUST NOT** be sent prior to sending the Finished message. Note that while the server may send application data prior to receiving the client's Authentication messages, any data sent at that point is, of course, being sent to an unauthenticated peer.

2.1. Incorrect DHE Share

If the client has not provided a sufficient "key_share" extension (e.g. it includes only DHE or ECDHE groups unacceptable or unsupported by the server), the server corrects the mismatch with a HelloRetryRequest and the client will need to restart the handshake with an appropriate "key_share" extension, as shown in Figure 2. If no common cryptographic parameters can be negotiated, the server will send a "handshake_failure" or "insufficient_security" fatal alert (see Section 6).

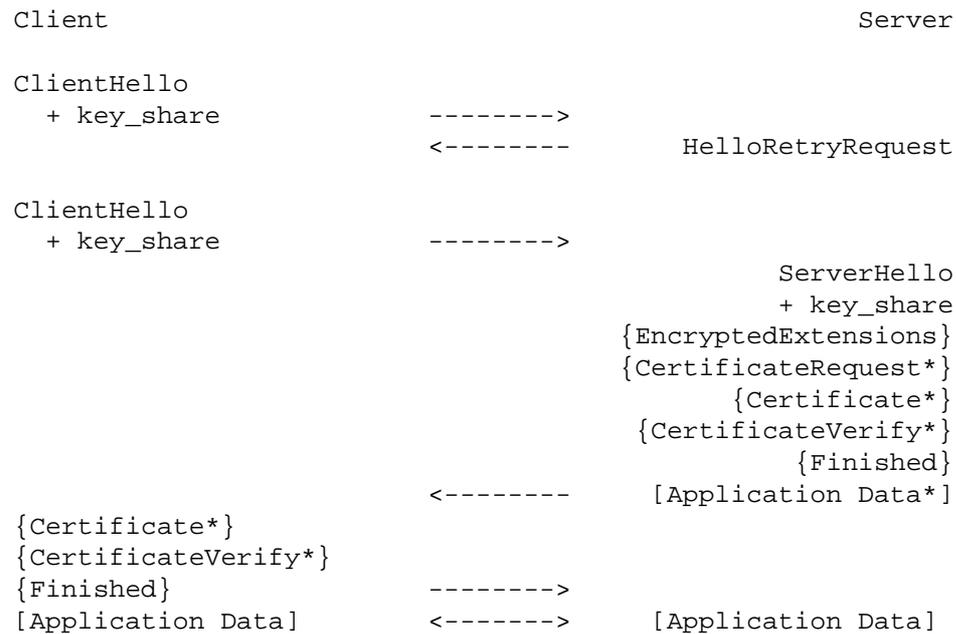


Figure 2: Message flow for a full handshake with mismatched parameters

Note: The handshake transcript includes the initial ClientHello/HelloRetryRequest exchange; it is not reset with the new ClientHello.

TLS also allows several optimized variants of the basic handshake, as described in the following sections.

2.2. Resumption and Pre-Shared Key (PSK)

Although TLS PSKs can be established out of band, PSKs can also be established in a previous session and then reused ("session resumption"). Once a handshake has completed, the server can send the client a PSK identity which corresponds to a key derived from the initial handshake (See [Section 4.4.1](#)). The client can then use that PSK identity in future handshakes to negotiate use of the PSK. If the server accepts it, then the security context of the new connection is tied to the original connection. In TLS 1.2 and below, this functionality was provided by "session IDs" and "session tickets" [[RFC5077](#)]. Both mechanisms are obsoleted in TLS 1.3.

PSKs can be used with (EC)DHE exchange in order to provide forward secrecy in combination with shared keys, or can be used alone, at the cost of losing forward secrecy.

Figure 3 shows a pair of handshakes in which the first establishes a PSK and the second uses it:



Figure 3: Message flow for resumption and PSK

As the server is authenticating via a PSK, it does not send a Certificate or a CertificateVerify. When a client offers resumption via PSK it SHOULD also supply a "key_share" extension to the server as well to allow the server to decline resumption and fall back to a full handshake, if needed. The server responds with a "pre_shared_key" extension to negotiate use of PSK key establishment and can (as shown here) respond with a "key_share" extension to do (EC)DHE key establishment, thus providing forward secrecy.

2.3. Zero-RTT Data

When resuming via a PSK with an appropriate ticket (i.e., one with the "allow_early_data" flag), clients can also send data on their first flight ("early data"). This data is encrypted solely under keys derived using the first offered PSK as the static secret. As shown in Figure 4, the Zero-RTT data is just added to the 1-RTT handshake in the first flight. The rest of the handshake uses the same messages.

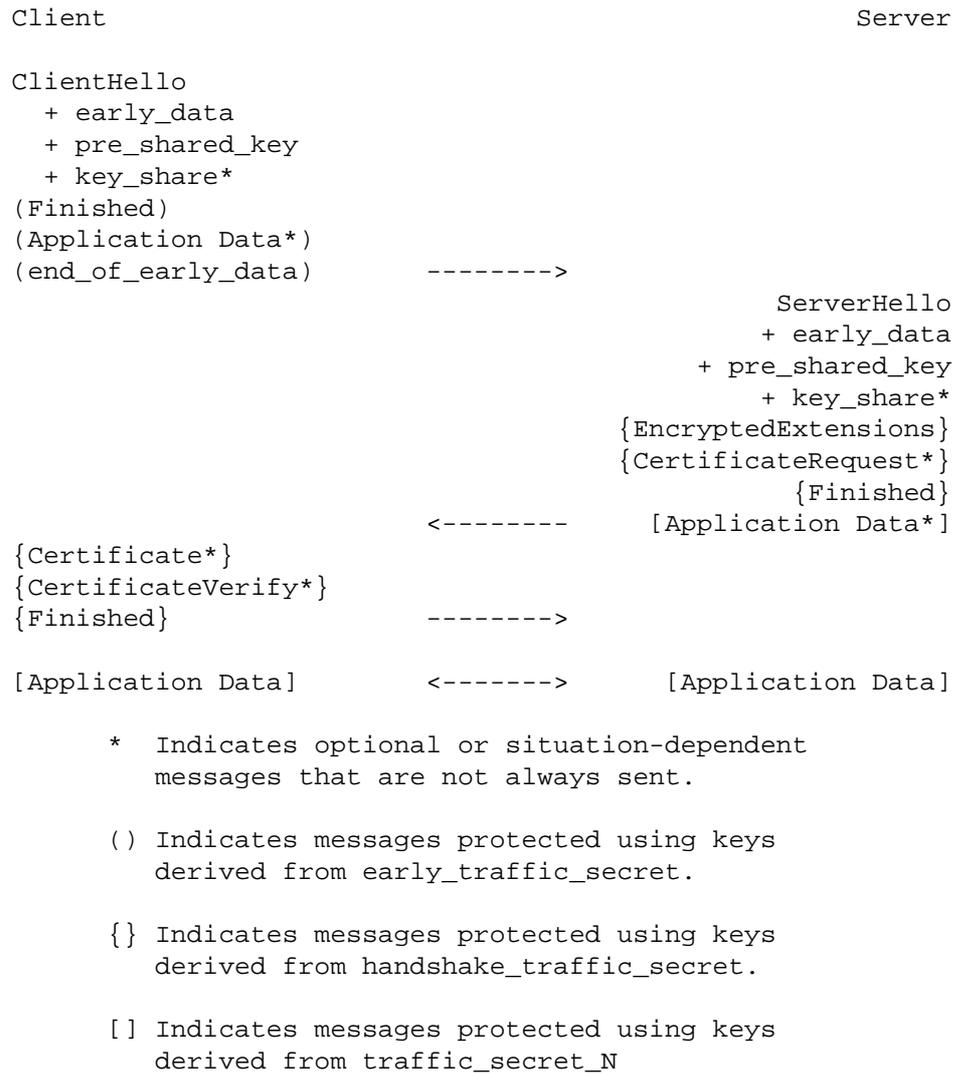


Figure 4: Message flow for a zero round trip handshake

[[OPEN ISSUE: Should it be possible to combine 0-RTT with the server authenticating via a signature <https://github.com/tlswg/tls13-spec/issues/443>]]

IMPORTANT NOTE: The security properties for 0-RTT data are weaker than those for other kinds of TLS data. Specifically:

1. This data is not forward secret, because it is encrypted solely with the PSK.
2. There are no guarantees of non-replay between connections. Unless the server takes special measures outside those provided by TLS, the server has no guarantee that the same 0-RTT data was not transmitted on multiple 0-RTT connections (See [Section 4.2.6.2](#) for more details). This is especially relevant if the data is authenticated either with TLS client authentication or inside the application layer protocol. However, 0-RTT data cannot be duplicated within a connection (i.e., the server will not process the same data twice for the same connection) and an attacker will not be able to make 0-RTT data appear to be 1-RTT data (because it is protected with different keys.)

The remainder of this document provides a detailed description of TLS.

3. Presentation Language

This document deals with the formatting of data in an external representation. The following very basic and somewhat casually defined presentation syntax will be used. The syntax draws from several sources in its structure. Although it resembles the programming language "C" in its syntax and XDR [[RFC4506](#)] in both its syntax and intent, it would be risky to draw too many parallels. The purpose of this presentation language is to document TLS only; it has no general application beyond that particular goal.

3.1. Basic Block Size

The representation of all data items is explicitly specified. The basic data block size is one byte (i.e., 8 bits). Multiple byte data items are concatenations of bytes, from left to right, from top to bottom. From the byte stream, a multi-byte item (a numeric in the example) is formed (using C notation) by:

```
value = (byte[0] << 8*(n-1)) | (byte[1] << 8*(n-2)) |  
        ... | byte[n-1];
```

This byte ordering for multi-byte values is the commonplace network byte order or big-endian format.

3.2. Miscellaneous

Comments begin with `/*` and end with `*/`.

Optional components are denoted by enclosing them in `"[]"` double brackets.

Single-byte entities containing uninterpreted data are of type `opaque`.

3.3. Vectors

A vector (single-dimensioned array) is a stream of homogeneous data elements. The size of the vector may be specified at documentation time or left unspecified until runtime. In either case, the length declares the number of bytes, not the number of elements, in the vector. The syntax for specifying a new type, `T'`, that is a fixed-length vector of type `T` is

```
T T'[n];
```

Here, `T'` occupies `n` bytes in the data stream, where `n` is a multiple of the size of `T`. The length of the vector is not included in the encoded stream.

In the following example, `Datum` is defined to be three consecutive bytes that the protocol does not interpret, while `Data` is three consecutive `Datum`, consuming a total of nine bytes.

```
opaque Datum[3];      /* three uninterpreted bytes */
Datum Data[9];        /* 3 consecutive 3 byte vectors */
```

Variable-length vectors are defined by specifying a subrange of legal lengths, inclusively, using the notation `<floor..ceiling>`. When these are encoded, the actual length precedes the vector's contents in the byte stream. The length will be in the form of a number consuming as many bytes as required to hold the vector's specified maximum (ceiling) length. A variable-length vector with an actual length field of zero is referred to as an empty vector.

```
T T'<floor..ceiling>;
```

In the following example, `mandatory` is a vector that must contain between 300 and 400 bytes of type `opaque`. It can never be empty. The actual length field consumes two bytes, a `uint16`, which is

sufficient to represent the value 400 (see [Section 3.4](#)). On the other hand, longer can represent up to 800 bytes of data, or 400 uint16 elements, and it may be empty. Its encoding will include a two-byte actual length field prepended to the vector. The length of an encoded vector must be an even multiple of the length of a single element (for example, a 17-byte vector of uint16 would be illegal).

```
opaque mandatory<300..400>;
    /* length field is 2 bytes, cannot be empty */
uint16 longer<0..800>;
    /* zero to 400 16-bit unsigned integers */
```

3.4. Numbers

The basic numeric data type is an unsigned byte (uint8). All larger numeric data types are formed from fixed-length series of bytes concatenated as described in [Section 3.1](#) and are also unsigned. The following numeric types are predefined.

```
uint8 uint16[2];
uint8 uint24[3];
uint8 uint32[4];
uint8 uint64[8];
```

All values, here and elsewhere in the specification, are stored in network byte (big-endian) order; the uint32 represented by the hex bytes 01 02 03 04 is equivalent to the decimal value 16909060.

Note that in some cases (e.g., DH parameters) it is necessary to represent integers as opaque vectors. In such cases, they are represented as unsigned integers (i.e., additional leading zero octets are not used even if the most significant bit is set).

3.5. Enumerateds

An additional sparse data type is available called enum. A field of type enum can only assume the values declared in the definition. Each definition is a different type. Only enumerateds of the same type may be assigned or compared. Every element of an enumerated must be assigned a value, as demonstrated in the following example. Since the elements of the enumerated are not ordered, they can be assigned any unique value, in any order.

```
enum { e1(v1), e2(v2), ... , en(vn) [[, (n)]] } Te;
```

An enumerated occupies as much space in the byte stream as would its maximal defined ordinal value. The following definition would cause one byte to be used to carry fields of type Color.

```
enum { red(3), blue(5), white(7) } Color;
```

One may optionally specify a value without its associated tag to force the width definition without defining a superfluous element.

In the following example, Taste will consume two bytes in the data stream but can only assume the values 1, 2, or 4.

```
enum { sweet(1), sour(2), bitter(4), (32000) } Taste;
```

The names of the elements of an enumeration are scoped within the defined type. In the first example, a fully qualified reference to the second element of the enumeration would be `Color.blue`. Such qualification is not required if the target of the assignment is well specified.

```
Color color = Color.blue;    /* overspecified, legal */  
Color color = blue;         /* correct, type implicit */
```

For enumerations that are never converted to external representation, the numerical information may be omitted.

```
enum { low, medium, high } Amount;
```

3.6. Constructed Types

Structure types may be constructed from primitive types for convenience. Each specification declares a new, unique type. The syntax for definition is much like that of C.

```
struct {  
    T1 f1;  
    T2 f2;  
    ...  
    Tn fn;  
} [[T]];
```

The fields within a structure may be qualified using the type's name, with a syntax much like that available for enumerations. For example, `T.f2` refers to the second field of the previous declaration. Structure definitions may be embedded.

3.6.1. Variants

Defined structures may have variants based on some knowledge that is available within the environment. The selector must be an enumerated type that defines the possible variants the structure defines. There must be a case arm for every element of the enumeration declared in

the select. Case arms have limited fall-through: if two case arms follow in immediate succession with no fields in between, then they both contain the same fields. Thus, in the example below, "orange" and "banana" both contain V2. Note that this is a new piece of syntax in TLS 1.2.

The body of the variant structure may be given a label for reference. The mechanism by which the variant is selected at runtime is not prescribed by the presentation language.

```
struct {
    T1 f1;
    T2 f2;
    ....
    Tn fn;
    select (E) {
        case e1: Te1;
        case e2: Te2;
        case e3: case e4: Te3;
        ....
        case en: Ten;
    } [[fv]];
} [[Tv]];
```

For example:

```
enum { apple, orange, banana } VariantTag;

struct {
    uint16 number;
    opaque string<0..10>; /* variable length */
} V1;

struct {
    uint32 number;
    opaque string[10]; /* fixed length */
} V2;

struct {
    select (VariantTag) { /* value of selector is implicit */
        case apple:
            V1; /* VariantBody, tag = apple */
        case orange:
        case banana:
            V2; /* VariantBody, tag = orange or banana */
    } variant_body; /* optional label on variant */
} VariantRecord;
```

3.7. Constants

Typed constants can be defined for purposes of specification by declaring a symbol of the desired type and assigning values to it.

Under-specified types (opaque, variable-length vectors, and structures that contain opaque) cannot be assigned values. No fields of a multi-element structure or vector may be elided.

For example:

```
struct {
    uint8 f1;
    uint8 f2;
} Example1;

Example1 ex1 = {1, 4}; /* assigns f1 = 1, f2 = 4 */
```

4. Handshake Protocol

The handshake protocol is used to negotiate the secure attributes of a session. Handshake messages are supplied to the TLS record layer, where they are encapsulated within one or more TLSPlaintext or TLSCiphertext structures, which are processed and transmitted as specified by the current active session state.

```
enum {
    client_hello(1),
    server_hello(2),
    new_session_ticket(4),
    hello_retry_request(6),
    encrypted_extensions(8),
    certificate(11),
    certificate_request(13),
    certificate_verify(15),
    finished(20),
    key_update(24),
    (255)
} HandshakeType;

struct {
    HandshakeType msg_type;      /* handshake type */
    uint24 length;             /* bytes in message */
    select (HandshakeType) {
        case client_hello:      ClientHello;
        case server_hello:      ServerHello;
        case hello_retry_request: HelloRetryRequest;
        case encrypted_extensions: EncryptedExtensions;
        case certificate_request: CertificateRequest;
        case certificate:        Certificate;
        case certificate_verify: CertificateVerify;
        case finished:           Finished;
        case new_session_ticket: NewSessionTicket;
        case key_update:         KeyUpdate;
    } body;
} Handshake;
```

Protocol messages MUST be sent in the order defined below (and shown in the diagrams in [Section 2](#)). Sending handshake messages in an unexpected order results in an "unexpected_message" fatal error. Unneeded handshake messages are omitted, however.

New handshake message types are assigned by IANA as described in [Section 10](#).

4.1. Key Exchange Messages

The key exchange messages are used to exchange security capabilities between the client and server and to establish the traffic keys used to protect the handshake and data.

4.1.1. Cryptographic Negotiation

TLS cryptographic negotiation proceeds by the client offering the following four sets of options in its ClientHello.

- A list of cipher suites which indicates the AEAD cipher/HKDF hash pairs which the client supports
- A "supported_group" ([Section 4.2.3](#)) extension which indicates the (EC)DHE groups which the client supports and a "key_share" ([Section 4.2.4](#)) extension which contains (EC)DHE shares for some or all of these groups
- A "signature_algorithms" ([Section 4.2.2](#)) extension which indicates the signature algorithms which the client can accept.
- A "pre_shared_key" ([Section 4.2.5](#)) extension which contains the identities of symmetric keys known to the client and the key exchange modes which each PSK supports.

If the server does not select a PSK, then the first three of these options are entirely orthogonal: the server independently selects a cipher suite, an (EC)DHE group and key share for key establishment, and a signature algorithm/certificate pair to authenticate itself to the client. If any of these parameters has no overlap between the client and server parameters, then the handshake will fail. If there is overlap in the "supported_group" extension but the client did not offer a compatible "key_share" extension, then the server will respond with a HelloRetryRequest ([Section 4.1.4](#)) message.

If the server selects a PSK, then the PSK will indicate which key establishment modes it can be used with (PSK alone or with (EC)DHE) and which authentication modes it can be used with (PSK alone or PSK with signatures). The server can then select those key establishment and authentication parameters to be consistent both with the PSK and the other extensions supplied by the client. Note that if the PSK can be used without (EC)DHE or without signatures, then non-overlap in either of these parameters need not be fatal.

The server indicates its selected parameters in the ServerHello as follows: If PSK is being used then the server will send a "pre_shared_key" extension indicating the selected key. If PSK is not being used, then (EC)DHE and certificate-based authentication are always used. When (EC)DHE is in use, the server will also provide a "key_share" extension. When authenticating via a certificate, the server will send an empty "signature_algorithms" extension in the ServerHello and will subsequently send Certificate ([Section 4.3.1](#)) and CertificateVerify ([Section 4.3.2](#)) messages.

If the server is unable to negotiate a supported set of parameters, it MUST return a "handshake_failure" alert and close the connection.

4.1.2. Client Hello

When this message will be sent:

When a client first connects to a server, it is required to send the ClientHello as its first message. The client will also send a ClientHello when the server has responded to its ClientHello with a HelloRetryRequest that selects cryptographic parameters that don't match the client's "key_share" extension. In that case, the client MUST send the same ClientHello (without modification) except:

- Including a new KeyShareEntry as the lowest priority share (i.e., appended to the list of shares in the "key_share" extension).
- Removing the EarlyDataIndication [Section 4.2.6](#) extension if one was present. Early data is not permitted after HelloRetryRequest.

If a server receives a ClientHello at any other time, it MUST send a fatal "unexpected_message" alert and close the connection.

Structure of this message:

```
struct {
    uint8 major;
    uint8 minor;
} ProtocolVersion;

struct {
    opaque random_bytes[32];
} Random;

uint8 CipherSuite[2];    /* Cryptographic suite selector */

struct {
    ProtocolVersion max_supported_version = { 3, 4 };    /* TLS v1.3 */
    Random random;
    opaque legacy_session_id<0..32>;
    CipherSuite cipher_suites<2..2^16-2>;
    opaque legacy_compression_methods<1..2^8-1>;
    Extension extensions<0..2^16-1>;
} ClientHello;
```

TLS allows extensions to follow the compression_methods field in an extensions block. The presence of extensions can be detected by

determining whether there are bytes following the `compression_methods` at the end of the ClientHello. Note that this method of detecting optional data differs from the normal TLS method of having a variable-length field, but it is used for compatibility with TLS before extensions were defined. As of TLS 1.3, all clients and servers will send at least one extension (at least "key_share" or "pre_shared_key").

`max_supported_version` The latest (highest valued) version of the TLS protocol offered by the client. This SHOULD be the same as the latest version supported. For this version of the specification, the version will be { 3, 4 }. (See [Appendix C](#) for details about backward compatibility.)

`random` 32 bytes generated by a secure random number generator. See [Appendix B](#) for additional information.

`legacy_session_id` Versions of TLS before TLS 1.3 supported a session resumption feature which has been merged with Pre-Shared Keys in this version (see [Section 2.2](#)). This field MUST be ignored by a server negotiating TLS 1.3 and SHOULD be set as a zero length vector (i.e., a single zero byte length field) by clients which do not have a cached session ID set by a pre-TLS 1.3 server.

`cipher_suites` This is a list of the symmetric cipher options supported by the client, specifically the record protection algorithm (including secret key length) and a hash to be used with HKDF, in descending order of client preference. If the list contains cipher suites the server does not recognize, support, or wish to use, the server MUST ignore those cipher suites, and process the remaining ones as usual. Values are defined in [Appendix A.4](#).

`legacy_compression_methods` Versions of TLS before 1.3 supported compression with the list of supported compression methods being sent in this field. For every TLS 1.3 ClientHello, this vector MUST contain exactly one byte set to zero, which corresponds to the "null" compression method in prior versions of TLS. If a TLS 1.3 ClientHello is received with any other value in this field, the server MUST generate a fatal "illegal_parameter" alert. Note that TLS 1.3 servers might receive TLS 1.2 or prior ClientHellos which contain other compression methods and MUST follow the procedures for the appropriate prior version of TLS.

`extensions` Clients request extended functionality from servers by sending data in the extensions field. The actual "Extension" format is defined in [Section 4.2](#).

In the event that a client requests additional functionality using extensions, and this functionality is not supplied by the server, the client MAY abort the handshake. Note that TLS 1.3 ClientHello messages MUST always contain extensions, and a TLS 1.3 server MUST respond to any TLS 1.3 ClientHello without extensions with a fatal "decode_error" alert. TLS 1.3 servers may receive TLS 1.2 ClientHello messages without extensions. If negotiating TLS 1.2, a server MUST check that the amount of data in the message precisely matches one of these formats; if not, then it MUST send a fatal "decode_error" alert.

After sending the ClientHello message, the client waits for a ServerHello or HelloRetryRequest message.

4.1.3. Server Hello

When this message will be sent:

The server will send this message in response to a ClientHello message when it was able to find an acceptable set of algorithms and the client's "key_share" extension was acceptable. If it is not able to find an acceptable set of parameters, the server will respond with a "handshake_failure" fatal alert.

Structure of this message:

```
struct {
    ProtocolVersion version;
    Random random;
    CipherSuite cipher_suite;
    Extension extensions<0..2^16-1>;
} ServerHello;
```

version This field contains the version of TLS negotiated for this session. Servers MUST select the lower of the highest supported server version and the version offered by the client in the ClientHello. In particular, servers MUST accept ClientHello messages with versions higher than those supported and negotiate the highest mutually supported version. For this version of the specification, the version is { 3, 4 }. (See [Appendix C](#) for details about backward compatibility.)

random This structure is generated by the server and MUST be generated independently of the ClientHello.random.

cipher_suite The single cipher suite selected by the server from the list in ClientHello.cipher_suites.

extensions A list of extensions. Note that only extensions offered by the client can appear in the server's list. In TLS 1.3, as opposed to previous versions of TLS, the server's extensions are split between the ServerHello and the EncryptedExtensions [Section 4.2.8](#) message. The ServerHello MUST only include extensions which are required to establish the cryptographic context. Currently the only such extensions are "key_share", "pre_shared_key", and "early_data". Clients MUST check the ServerHello for the presence of any forbidden extensions and if any are found MUST terminate the handshake with a "illegal_parameter" alert. In prior versions of TLS, the extensions field could be omitted entirely if not needed, similar to ClientHello. As of TLS 1.3, all clients and servers will send at least one extension (at least "key_share" or "pre_shared_key").

TLS 1.3 has a downgrade protection mechanism embedded in the server's random value. TLS 1.3 server implementations which respond to a ClientHello with a max_supported_version indicating TLS 1.2 or below MUST set the last eight bytes of their Random value to the bytes:

```
44 4F 57 4E 47 52 44 01
```

TLS 1.2 server implementations which respond to a ClientHello with a max_supported_version indicating TLS 1.1 or below SHOULD set the last eight bytes of their Random value to the bytes:

```
44 4F 57 4E 47 52 44 00
```

TLS 1.3 clients receiving a TLS 1.2 or below ServerHello MUST check that the last eight octets are not equal to either of these values. TLS 1.2 clients SHOULD also perform this check if the ServerHello indicates TLS 1.1 or below. If a match is found, the client MUST abort the handshake with a fatal "illegal_parameter" alert. This mechanism provides limited protection against downgrade attacks over and above that provided by the Finished exchange: because the ServerKeyExchange includes a signature over both random values, it is not possible for an active attacker to modify the randoms without detection as long as ephemeral ciphers are used. It does not provide downgrade protection when static RSA is used.

Note: This is an update to TLS 1.2 so in practice many TLS 1.2 clients and servers will not behave as specified above.

4.1.4. Hello Retry Request

When this message will be sent:

Servers send this message in response to a ClientHello message if they were able to find an acceptable set of algorithms and groups that are mutually supported, but the client's KeyShare did not contain an acceptable offer. If it cannot find such a match, it will respond with a fatal "handshake_failure" alert.

Structure of this message:

```
struct {
    ProtocolVersion server_version;
    NamedGroup selected_group;
    Extension extensions<0..2^16-1>;
} HelloRetryRequest;
```

`selected_group` The mutually supported group the server intends to negotiate and is requesting a retried ClientHello/KeyShare for.

The version and extensions fields have the same meanings as their corresponding values in the ServerHello. The server SHOULD send only the extensions necessary for the client to generate a correct ClientHello pair (currently no such extensions exist). As with ServerHello, a HelloRetryRequest MUST NOT contain any extensions that were not first offered by the client in its ClientHello.

Upon receipt of a HelloRetryRequest, the client MUST first verify that the `selected_group` field corresponds to a group which was provided in the "supported_groups" extension in the original ClientHello. It MUST then verify that the `selected_group` field does not correspond to a group which was provided in the "key_share" extension in the original ClientHello. If either of these checks fails, then the client MUST abort the handshake with a fatal "illegal_parameter" alert. Clients SHOULD also abort with "unexpected_message" in response to any second HelloRetryRequest which was sent in the same connection (i.e., where the ClientHello was itself in response to a HelloRetryRequest).

Otherwise, the client MUST send a ClientHello with an updated KeyShare extension to the server. The client MUST append a new KeyShareEntry for the group indicated in the `selected_group` field to the groups in its original KeyShare.

Upon re-sending the ClientHello and receiving the server's ServerHello/KeyShare, the client MUST verify that the selected NamedGroup matches that supplied in the HelloRetryRequest and MUST abort the connection with a fatal "illegal_parameter" alert if it does not.

4.2. Hello Extensions

The extension format is:

```
struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;

enum {
    supported_groups(10),
    signature_algorithms(13),
    key_share(40),
    pre_shared_key(41),
    early_data(42),
    cookie(44),
    (65535)
} ExtensionType;
```

Here:

- "extension_type" identifies the particular extension type.
- "extension_data" contains information specific to the particular extension type.

The initial set of extensions is defined in [RFC6066]. The list of extension types is maintained by IANA as described in [Section 10](#).

An extension type MUST NOT appear in the ServerHello or HelloRetryRequest unless the same extension type appeared in the corresponding ClientHello. If a client receives an extension type in ServerHello or HelloRetryRequest that it did not request in the associated ClientHello, it MUST abort the handshake with an "unsupported_extension" fatal alert.

Nonetheless, "server-oriented" extensions may be provided within this framework. Such an extension (say, of type x) would require the client to first send an extension of type x in a ClientHello with empty extension_data to indicate that it supports the extension type. In this case, the client is offering the capability to understand the extension type, and the server is taking the client up on its offer.

When multiple extensions of different types are present in the ClientHello or ServerHello messages, the extensions MAY appear in any order. There MUST NOT be more than one extension of the same type.

Finally, note that extensions can be sent both when starting a new session and when in resumption-PSK mode. A client that requests session resumption does not in general know whether the server will accept this request, and therefore it SHOULD send the same extensions as it would send normally.

In general, the specification of each extension type needs to describe the effect of the extension both during full handshake and session resumption. Most current TLS extensions are relevant only when a session is initiated: when an older session is resumed, the server does not process these extensions in ClientHello, and does not include them in ServerHello. However, some extensions may specify different behavior during session resumption. [[TODO: update this and the previous paragraph to cover PSK-based resumption.]]

There are subtle (and not so subtle) interactions that may occur in this protocol between new features and existing features which may result in a significant reduction in overall security. The following considerations should be taken into account when designing new extensions:

- Some cases where a server does not agree to an extension are error conditions, and some are simply refusals to support particular features. In general, error alerts should be used for the former, and a field in the server extension response for the latter.
- Extensions should, as far as possible, be designed to prevent any attack that forces use (or non-use) of a particular feature by manipulation of handshake messages. This principle should be followed regardless of whether the feature is believed to cause a security problem. Often the fact that the extension fields are included in the inputs to the Finished message hashes will be sufficient, but extreme care is needed when the extension changes the meaning of messages sent in the handshake phase. Designers and implementors should be aware of the fact that until the handshake has been authenticated, active attackers can modify messages and insert, remove, or replace extensions.

4.2.1. Cookie

```
struct {  
    opaque cookie<0..2^16-1>;  
} Cookie;
```

Cookies serve two primary purposes:

- Allowing the server to force the client to demonstrate reachability at their apparent network address (thus providing a

measure of DoS protection). This is primarily useful for non-connection-oriented transports (see [\[RFC6347\]](#) for an example of this).

- Allowing the server to offload state to the client, thus allowing it to send a HelloRetryRequest without storing any state. The server does this by pickling that post-ClientHello hash state into the cookie (protected with some suitable integrity algorithm).

When sending a HelloRetryRequest, the server MAY provide a "cookie" extension to the client (this is an exception to the usual rule that the only extensions that may be sent are those that appear in the ClientHello). When sending the new ClientHello, the client MUST echo the value of the extension. Clients MUST NOT use cookies in subsequent connections.

4.2.2. Signature Algorithms

The client uses the "signature_algorithms" extension to indicate to the server which signature algorithms may be used in digital signatures. Clients which desire the server to authenticate via a certificate MUST send this extension. If a server is authenticating via a certificate and the client has not sent a "signature_algorithms" extension then the server MUST close the connection with a fatal "missing_extension" alert (see [Section 8.2](#)).

Servers which are authenticating via a certificate MUST indicate so by sending the client an empty "signature_algorithms" extension.

The "extension_data" field of this extension contains a "supported_signature_algorithms" value:

```
enum {
    /* RSASSA-PKCS1-v1_5 algorithms */
    rsa_pkcs1_sha1 (0x0201),
    rsa_pkcs1_sha256 (0x0401),
    rsa_pkcs1_sha384 (0x0501),
    rsa_pkcs1_sha512 (0x0601),

    /* ECDSA algorithms */
    ecdsa_secp256r1_sha256 (0x0403),
    ecdsa_secp384r1_sha384 (0x0503),
    ecdsa_secp521r1_sha512 (0x0603),

    /* RSASSA-PSS algorithms */
    rsa_pss_sha256 (0x0700),
    rsa_pss_sha384 (0x0701),
    rsa_pss_sha512 (0x0702),

    /* EdDSA algorithms */
    ed25519 (0x0703),
    ed448 (0x0704),

    /* Reserved Code Points */
    private_use (0xFE00..0xFFFF),
    (0xFFFF)
} SignatureScheme;

SignatureScheme supported_signature_algorithms<2..2^16-2>;
```

Note: This enum is named "SignatureScheme" because there is already a "SignatureAlgorithm" type in TLS 1.2, which this replaces. We use the term "signature algorithm" throughout the text.

Each SignatureScheme value lists a single signature algorithm that the client is willing to verify. The values are indicated in descending order of preference. Note that a signature algorithm takes as input an arbitrary-length message, rather than a digest. Algorithms which traditionally act on a digest should be defined in TLS to first hash the input with a specified hash function and then proceed as usual. The code point groups listed above have the following meanings:

RSASSA-PKCS1-v1_5 algorithms Indicates a signature algorithm using RSASSA-PKCS1-v1_5 [RFC3447] with the corresponding hash algorithm as defined in [SHS]. These values refer solely to signatures which appear in certificates (see Section 4.3.1.1) and are not defined for use in signed TLS handshake messages.

ECDSA algorithms Indicates a signature algorithm using ECDSA [[ECDSA](#)], the corresponding curve as defined in ANSI X9.62 [[X962](#)] and FIPS 186-4 [[DSS](#)], and the corresponding hash algorithm as defined in [[SHS](#)]. The signature is represented as a DER-encoded [[X690](#)] ECDSA-Sig-Value structure.

RSASSA-PSS algorithms Indicates a signature algorithm using RSASSA-PSS [[RFC3447](#)] with MGF1. The digest used in the mask generation function and the digest being signed are both the corresponding hash algorithm as defined in [[SHS](#)]. When used in signed TLS handshake messages, the length of the salt **MUST** be equal to the length of the digest output. This codepoint is defined for use with TLS 1.2 as well as TLS 1.3. A server uses RSASSA-PSS signatures with RSA cipher suites.

EdDSA algorithms Indicates a signature algorithm using EdDSA as defined in [[I-D.irtf-cfrg-eddsa](#)] or its successors. Note that these correspond to the "PureEdDSA" algorithms and not the "prehash" variants. A server uses EdDSA signatures with ECDSA cipher suites.

`rsa_pkcs1_sha1`, `dsa_sha1`, and `ecdsa_sha1` **SHOULD NOT** be offered. Clients offering these values for backwards compatibility **MUST** list them as the lowest priority (listed after all other algorithms in the `supported_signature_algorithms` vector). TLS 1.3 servers **MUST NOT** offer a SHA-1 signed certificate unless no valid certificate chain can be produced without it (see [Section 4.3.1.1](#)).

The signatures on certificates that are self-signed or certificates that are trust anchors are not validated since they begin a certification path (see [[RFC5280](#)], [Section 3.2](#)). A certificate that begins a certification path **MAY** use a signature algorithm that is not advertised as being supported in the "signature_algorithms" extension.

Note that TLS 1.2 defines this extension differently. TLS 1.3 implementations willing to negotiate TLS 1.2 **MUST** behave in accordance with the requirements of [[RFC5246](#)] when negotiating that version. In particular:

- TLS 1.2 ClientHellos **MAY** omit this extension.
- In TLS 1.2, the extension contained hash/signature pairs. The pairs are encoded in two octets, so SignatureScheme values have been allocated to align with TLS 1.2's encoding. Some legacy pairs are left unallocated. These algorithms are deprecated as of TLS 1.3. They **MUST NOT** be offered or negotiated by any

implementation. In particular, MD5 [SLOTH] and SHA-224 MUST NOT be used.

- ECDSA signature schemes align with TLS 1.2's ECDSA hash/signature pairs. However, the old semantics did not constrain the signing curve. If TLS 1.2 is negotiated, implementations MUST be prepared to accept a signature that uses any curve that they advertised in the "supported_groups" extension.
- Implementations that advertise support for RSASSA-PSS (which is mandatory in TLS 1.3), MUST be prepared to accept a signature using that scheme even when TLS 1.2 is negotiated.

4.2.3. Negotiated Groups

When sent by the client, the "supported_groups" extension indicates the named groups which the client supports for key exchange, ordered from most preferred to least preferred.

Note: In versions of TLS prior to TLS 1.3, this extension was named "elliptic_curves" and only contained elliptic curve groups. See [RFC4492] and [I-D.ietf-tls-negotiated-ff-dhe]. This extension was also used to negotiate ECDSA curves. Signature algorithms are now negotiated independently (see Section 4.2.2).

The "extension_data" field of this extension contains a "NamedGroupList" value:

```
enum {
    /* Elliptic Curve Groups (ECDHE) */
    secp256r1 (23), secp384r1 (24), secp521r1 (25),
    x25519 (29), x448 (30),

    /* Finite Field Groups (DHE) */
    ffdhe2048 (256), ffdhe3072 (257), ffdhe4096 (258),
    ffdhe6144 (259), ffdhe8192 (260),

    /* Reserved Code Points */
    ffdhe_private_use (0x01FC..0x01FF),
    ecdhe_private_use (0xFE00..0xFEFF),
    (0xFFFF)
} NamedGroup;

struct {
    NamedGroup named_group_list<1..2^16-1>;
} NamedGroupList;
```

Elliptic Curve Groups (ECDHE) Indicates support of the corresponding named curve. Note that some curves are also recommended in ANSI X9.62 [X962] and FIPS 186-4 [DSS]. Others are recommended in [RFC7748]. Values 0xFE00 through 0xFEFF are reserved for private use.

Finite Field Groups (DHE) Indicates support of the corresponding finite field group, defined in [I-D.ietf-tls-negotiated-ff-dhe]. Values 0x01FC through 0x01FF are reserved for private use.

Items in `named_group_list` are ordered according to the client's preferences (most preferred choice first).

As of TLS 1.3, servers are permitted to send the "supported_groups" extension to the client. If the server has a group it prefers to the ones in the "key_share" extension but is still willing to accept the ClientHello, it SHOULD send "supported_groups" to update the client's view of its preferences. Clients MUST NOT act upon any information found in "supported_groups" prior to successful completion of the handshake, but MAY use the information learned from a successfully completed handshake to change what groups they offer to a server in subsequent connections.

4.2.4. Key Share

The "key_share" extension contains the endpoint's cryptographic parameters.

Clients MAY send an empty `client_shares` vector in order to request group selection from the server at the cost of an additional round trip. (see Section 4.1.4)

```
struct {
    NamedGroup group;
    opaque key_exchange<1..2^16-1>;
} KeyShareEntry;
```

`group` The named group for the key being exchanged. Finite Field Diffie-Hellman [DH] parameters are described in Section 4.2.4.1; Elliptic Curve Diffie-Hellman parameters are described in Section 4.2.4.2.

`key_exchange` Key exchange information. The contents of this field are determined by the specified group and its corresponding definition. Endpoints MUST NOT send empty or otherwise invalid `key_exchange` values for any reason.

The "extension_data" field of this extension contains a "KeyShare" value:

```
struct {
    select (role) {
        case client:
            KeyShareEntry client_shares<0..2^16-1>;

        case server:
            KeyShareEntry server_share;
    }
} KeyShare;
```

client_shares A list of offered KeyShareEntry values in descending order of client preference. This vector MAY be empty if the client is requesting a HelloRetryRequest. The ordering of values here SHOULD match that of the ordering of offered support in the "supported_groups" extension.

server_share A single KeyShareEntry value that is in the same group as one of the client's shares.

Clients offer an arbitrary number of KeyShareEntry values, each representing a single set of key exchange parameters. For instance, a client might offer shares for several elliptic curves or multiple FFDHE groups. The key_exchange values for each KeyShareEntry MUST be generated independently. Clients MUST NOT offer multiple KeyShareEntry values for the same group. Clients MUST NOT offer any KeyShareEntry values for groups not listed in the client's "supported_groups" extension. Servers MAY check for violations of these rules and MAY abort the connection with a fatal "illegal_parameter" alert if one is violated.

If using (EC)DHE key establishment, servers offer exactly one KeyShareEntry. This value MUST correspond to the KeyShareEntry value offered by the client that the server has selected for the negotiated key exchange. Servers MUST NOT send a KeyShareEntry for any group not indicated in the "supported_groups" extension.

[[TODO: Recommendation about what the client offers. Presumably which integer DH groups and which curves.]]

4.2.4.1. Diffie-Hellman Parameters

Diffie-Hellman [DH] parameters for both clients and servers are encoded in the opaque key_exchange field of a KeyShareEntry in a KeyShare structure. The opaque value contains the Diffie-Hellman

public value ($Y = g^X \bmod p$), encoded as a big-endian integer, padded with zeros to the size of p in bytes.

Note: For a given Diffie-Hellman group, the padding results in all public keys having the same length.

Peers SHOULD validate each other's public key Y by ensuring that $1 < Y < p-1$. This check ensures that the remote peer is properly behaved and isn't forcing the local system into a small subgroup.

4.2.4.2. ECDHE Parameters

ECDHE parameters for both clients and servers are encoded in the the opaque `key_exchange` field of a `KeyShareEntry` in a `KeyShare` structure.

For `secp256r1`, `secp384r1` and `secp521r1`, the contents are the byte string representation of an elliptic curve public value following the conversion routine in [Section 4.3.6](#) of ANSI X9.62 [X962].

Although X9.62 supports multiple point formats, any given curve MUST specify only a single point format. All curves currently specified in this document MUST only be used with the uncompressed point format (the format for all ECDH functions is considered uncompressed).

For `x25519` and `x448`, the contents are the byte string inputs and outputs of the corresponding functions defined in [RFC7748], 32 bytes for `x25519` and 56 bytes for `x448`.

Note: Versions of TLS prior to 1.3 permitted point negotiation; TLS 1.3 removes this feature in favor of a single point format for each curve.

4.2.5. Pre-Shared Key Extension

The `"pre_shared_key"` extension is used to indicate the identity of the pre-shared key to be used with a given handshake in association with PSK key establishment (see [RFC4279] for background).

The `"extension_data"` field of this extension contains a `"PreSharedKeyExtension"` value:

```
enum { psk_ke(0), psk_dhe_ke(1), (255) } PskKeyExchangeModes;
enum { psk_auth(0), psk_sign_auth(1), (255) } PskAuthenticationModes;

opaque psk_identity<0..2^16-1>;

struct {
    PskKeMode ke_modes<1..255>;
    PskAuthMode auth_modes<1..255>;
    opaque identity<0..2^16-1>;
} PskIdentity;

struct {
    select (Role) {
        case client:
            psk_identity identities<2..2^16-1>;

        case server:
            uint16 selected_identity;
    }
} PreSharedKeyExtension;
```

identities A list of the identities (labels for keys) that the client is willing to negotiate with the server. If sent alongside the "early_data" extension (see [Section 4.2.6](#)), the first identity is the one used for 0-RTT data.

selected_identity The server's chosen identity expressed as a (0-based) index into the identities in the client's list.

Each PSK offered by the client also indicates the authentication and key exchange modes with which the server can use it, with each list being in the order of the client's preference, with most preferred first.

PskKeyExchangeModes have the following meanings:

psk_ke PSK-only key establishment. In this mode, the server **MUST** not supply a "key_share" value.

psk_dhe_ke PSK key establishment with (EC)DHE key establishment. In this mode, the client and servers **MUST** supply "key_share" values as described in [Section 4.2.4](#).

PskAuthenticationModes have the following meanings:

psk_auth PSK-only authentication. In this mode, the server **MUST NOT** supply either a Certificate or CertificateVerify message. [TODO: Add a signing mode.]

In order to accept PSK key establishment, the server sends a "pre_shared_key" extension with the selected identity. Clients MUST verify that the server's selected_identity is within the range supplied by the client and that the "key_share" and "signature_algorithms" extensions are consistent with the indicated ke_modes and auth_modes values. If these values are not consistent, the client MUST generate an "illegal_parameter" alert and close the connection.

If the server supplies an "early_data" extension, the client MUST verify that the server selected the first offered identity. If any other value is returned, the client MUST generate a fatal "unknown_psk_identity" alert and close the connection.

Note that although 0-RTT data is encrypted with the first PSK identity, the server MAY fall back to 1-RTT and select a different PSK identity if multiple identities are offered.

4.2.6. Early Data Indication

When PSK resumption is used, the client can send application data in its first flight of messages. If the client opts to do so, it MUST supply an "early_data" extension as well as the "pre_shared_key" extension.

The "extension_data" field of this extension contains an "EarlyDataIndication" value:

```
struct {
    select (Role) {
        case client:
            uint32 obfuscated_ticket_age;

        case server:
            struct {};
    }
} EarlyDataIndication;
```

obfuscated_ticket_age The time since the client learned about the server configuration that it is using, in milliseconds. This value is added modulo 2^{32} to with the "ticket_age_add" value that was included with the ticket, see [Section 4.4.1](#). This addition prevents passive observers from correlating sessions unless tickets are reused. Note: because ticket lifetimes are restricted to a week, 32 bits is enough to represent any plausible age, even in milliseconds.

A server MUST validate that the `ticket_age` is within a small tolerance of the time since the ticket was issued (see [Section 4.2.6.2](#)).

The parameters for the 0-RTT data (symmetric cipher suite, ALPN, etc.) are the same as those which were negotiated in the connection which established the PSK. The PSK used to encrypt the early data MUST be the first PSK listed in the client's "pre_shared_key" extension.

0-RTT messages sent in the first flight have the same content types as their corresponding messages sent in other flights (handshake, application_data, and alert respectively) but are protected under different keys. After all the 0-RTT application data messages (if any) have been sent, an "end_of_early_data" alert of type "warning" is sent to indicate the end of the flight. 0-RTT MUST always be followed by an "end_of_early_data" alert.

A server which receives an "early_data" extension can behave in one of two ways:

- Ignore the extension and return no response. This indicates that the server has ignored any early data and an ordinary 1-RTT handshake is required.
- Return an empty extension, indicating that it intends to process the early data. It is not possible for the server to accept only a subset of the early data messages.

In order to accept early data, the server server MUST have accepted a PSK cipher suite and selected the the first key offered in the client's "pre_shared_key" extension. In addition, it MUST verify that the following values are consistent with those negotiated in the connection during which the ticket was established.

- The TLS version number, AEAD algorithm, and the hash for HKDF.
- The selected ALPN [[RFC7443](#)] value, if any.
- The `server_name` [[RFC6066](#)] value provided by the client, if any.

Future extensions MUST define their interaction with 0-RTT.

If any of these checks fail, the server MUST NOT respond with the extension and must discard all the remaining first flight data (thus falling back to 1-RTT). If the client attempts a 0-RTT handshake but the server rejects it, it will generally not have the 0-RTT record protection keys and must instead trial decrypt each record with the

1-RTT handshake keys until it finds one that decrypts properly, and then pick up the handshake from that point.

If the server chooses to accept the "early_data" extension, then it MUST comply with the same error handling requirements specified for all records when processing early data records. Specifically, decryption failure of any 0-RTT record following an accepted "early_data" extension MUST produce a fatal "bad_record_mac" alert as per [Section 5.2](#).

If the server rejects the "early_data" extension, the client application MAY opt to retransmit the data once the handshake has been completed. TLS stacks SHOULD not do this automatically and client applications MUST take care that the negotiated parameters are consistent with those it expected. For example, if the ALPN value has changed, it is likely unsafe to retransmit the original application layer data.

4.2.6.1. Processing Order

Clients are permitted to "stream" 0-RTT data until they receive the server's Finished, only then sending the "end_of_early_data" alert. In order to avoid deadlock, when accepting "early_data", servers MUST process the client's Finished and then immediately send the ServerHello, rather than waiting for the client's "end_of_early_data" alert.

4.2.6.2. Replay Properties

As noted in [Section 2.3](#), TLS provides a limited mechanism for replay protection for data sent by the client in the first flight.

The "obfuscated_ticket_age" parameter in the client's "early_data" extension SHOULD be used by servers to limit the time over which the first flight might be replayed. A server can store the time at which it sends a session ticket to the client, or encode the time in the ticket. Then, each time it receives an "early_data" extension, it can subtract the base value and check to see if the value used by the client matches its expectations.

The ticket age (the value with "ticket_age_add" subtracted) provided by the client will be shorter than the actual time elapsed on the server by a single round trip time. This difference is comprised of the delay in sending the NewSessionTicket message to the client, plus the time taken to send the ClientHello to the server. For this reason, a server SHOULD measure the round trip time prior to sending the NewSessionTicket message and account for that in the value it saves.

To properly validate the ticket age, a server needs to save at least two items:

- The time that the server generated the session ticket and the estimated round trip time can be added together to form a baseline time.
- The "ticket_age_add" parameter from the NewSessionTicket is needed to recover the ticket age from the "obfuscated_ticket_age" parameter.

There are several potential sources of error that make an exact measurement of time difficult. Variations in client and server clocks are likely to be minimal, outside of gross time corrections. Network propagation delays are most likely causes of a mismatch in legitimate values for elapsed time. Both the NewSessionTicket and ClientHello messages might be retransmitted and therefore delayed, which might be hidden by TCP.

A small allowance for errors in clocks and variations in measurements is advisable. However, any allowance also increases the opportunity for replay. In this case, it is better to reject early data and fall back to a full 1-RTT handshake than to risk greater exposure to replay attacks. In common network topologies for browser clients, small allowances on the order of ten seconds are reasonable. Clock skew distributions are not symmetric, so the optimal tradeoff may involve an asymmetric replay window.

4.2.7. OCSP Status Extensions

[RFC6066] and [RFC6961] provide extensions to negotiate the server sending OCSP responses to the client. In TLS 1.2 and below, the server sends an empty extension to indicate negotiation of this extension and the OCSP information is carried in a CertificateStatus message. In TLS 1.3, the server's OCSP information is carried in an extension in EncryptedExtensions. Specifically: The body of the "status_request" or "status_request_v2" extension from the server MUST be a CertificateStatus structure as defined in [RFC6066] and [RFC6961] respectively.

Note: This means that the certificate status appears prior to the certificates it applies to. This is slightly anomalous but matches the existing behavior for SignedCertificateTimestamps [RFC6962], and is more easily extensible in the handshake state machine.

4.2.8. Encrypted Extensions

When this message will be sent:

In all handshakes, the server MUST send the EncryptedExtensions message immediately after the ServerHello message. This is the first message that is encrypted under keys derived from handshake_traffic_secret.

Meaning of this message:

The EncryptedExtensions message contains any extensions which should be protected, i.e., any which are not needed to establish the cryptographic context.

The same extension types MUST NOT appear in both the ServerHello and EncryptedExtensions. If the same extension appears in both locations, the client MUST rely only on the value in the EncryptedExtensions block. All server-sent extensions other than those explicitly listed in [Section 4.1.3](#) or designated in the IANA registry MUST only appear in EncryptedExtensions. Extensions which are designated to appear in ServerHello MUST NOT appear in EncryptedExtensions. Clients MUST check EncryptedExtensions for the presence of any forbidden extensions and if any are found MUST terminate the handshake with an "illegal_parameter" alert.

Structure of this message:

```
struct {  
    Extension extensions<0..2^16-1>;  
} EncryptedExtensions;
```

extensions A list of extensions.

4.2.9. Certificate Request

When this message will be sent:

A server which is authenticating with a certificate can optionally request a certificate from the client. This message, if sent, will follow EncryptedExtensions.

Structure of this message:

```
opaque DistinguishedName<1..2^16-1>;

struct {
    opaque certificate_extension_oid<1..2^8-1>;
    opaque certificate_extension_values<0..2^16-1>;
} CertificateExtension;

struct {
    opaque certificate_request_context<0..2^8-1>;
    SignatureScheme
    supported_signature_algorithms<2..2^16-2>;
    DistinguishedName certificate_authorities<0..2^16-1>;
    CertificateExtension certificate_extensions<0..2^16-1>;
} CertificateRequest;
```

`certificate_request_context` An opaque string which identifies the certificate request and which will be echoed in the client's Certificate message. The `certificate_request_context` MUST be unique within the scope of this connection (thus preventing replay of client CertificateVerify messages). Within the handshake, this field MUST be empty.

`supported_signature_algorithms` A list of the signature algorithms that the server is able to verify, listed in descending order of preference. Any certificates provided by the client MUST be signed using a signature algorithm found in `supported_signature_algorithms`.

`certificate_authorities` A list of the distinguished names [X501] of acceptable certificate authorities, represented in DER-encoded [X690] format. These distinguished names may specify a desired distinguished name for a root CA or for a subordinate CA; thus, this message can be used to describe known roots as well as a desired authorization space. If the `certificate_authorities` list is empty, then the client MAY send any certificate that meets the rest of the selection criteria in the CertificateRequest, unless there is some external arrangement to the contrary.

`certificate_extensions` A list of certificate extension OIDs [RFC5280] with their allowed values, represented in DER-encoded [X690] format. Some certificate extension OIDs allow multiple values (e.g. Extended Key Usage). If the server has included a non-empty `certificate_extensions` list, the client certificate MUST contain all of the specified extension OIDs that the client recognizes. For each extension OID recognized by the client, all of the specified values MUST be present in the client certificate (but the certificate MAY have other values as well). However, the client MUST ignore and skip any unrecognized certificate extension

OIDs. If the client has ignored some of the required certificate extension OIDs, and supplied a certificate that does not satisfy the request, the server MAY at its discretion either continue the session without client authentication, or terminate the session with a fatal `unsupported_certificate` alert. PKIX RFCs define a variety of certificate extension OIDs and their corresponding value types. Depending on the type, matching certificate extension values are not necessarily bitwise-equal. It is expected that TLS implementations will rely on their PKI libraries to perform certificate selection using certificate extension OIDs. This document defines matching rules for two standard certificate extensions defined in [RFC5280]:

- o The Key Usage extension in a certificate matches the request when all key usage bits asserted in the request are also asserted in the Key Usage certificate extension.
- o The Extended Key Usage extension in a certificate matches the request when all key purpose OIDs present in the request are also found in the Extended Key Usage certificate extension. The special `anyExtendedKeyUsage` OID MUST NOT be used in the request.

Separate specifications may define matching rules for other certificate extensions.

Note: It is a fatal `unexpected_message` alert for an anonymous server to request client authentication.

4.3. Authentication Messages

As discussed in Section 2, TLS uses a common set of messages for authentication, key confirmation, and handshake integrity: `Certificate`, `CertificateVerify`, and `Finished`. These messages are always sent as the last messages in their handshake flight. The `Certificate` and `CertificateVerify` messages are only sent under certain circumstances, as defined below. The `Finished` message is always sent as part of the Authentication block.

The computations for the Authentication messages all uniformly take the following inputs:

- The certificate and signing key to be used.
- A Handshake Context based on the hash of the handshake messages
- A base key to be used to compute a MAC key.

Based on these inputs, the messages then contain:

Certificate The certificate to be used for authentication and any supporting certificates in the chain. Note that certificate-based client authentication is not available in the 0-RTT case.

CertificateVerify A signature over the value Hash(Handshake Context + Certificate) + Hash(resumption_context) See [Section 4.4.1](#) for the definition of resumption_context.

Finished A MAC over the value Hash(Handshake Context + Certificate + CertificateVerify) + Hash(resumption_context) using a MAC key derived from the base key.

Because the CertificateVerify signs the Handshake Context + Certificate and the Finished MACs the Handshake Context + Certificate + CertificateVerify, this is mostly equivalent to keeping a running hash of the handshake messages (exactly so in the pure 1-RTT cases). Note, however, that subsequent post-handshake authentications do not include each other, just the messages through the end of the main handshake.

The following table defines the Handshake Context and MAC Base Key for each scenario:

Mode	Handshake Context	Base Key
0-RTT	ClientHello	early_traffic_secret
1-RTT (Server)	ClientHello ... later of EncryptedExtensions/CertificateRequest	handshake_traffic_secret
1-RTT (Client)	ClientHello ... ServerFinished	handshake_traffic_secret
Post-Handshake	ClientHello ... ClientFinished + CertificateRequest	traffic_secret_0

Note: The Handshake Context for the last three rows does not include any 0-RTT handshake messages, regardless of whether 0-RTT is used.

4.3.1. Certificate

When this message will be sent:

The server **MUST** send a Certificate message whenever the agreed-upon key exchange method uses certificates for authentication (this includes all key exchange methods defined in this document except PSK).

The client **MUST** send a Certificate message if and only if server has requested client authentication via a CertificateRequest message (Section 4.2.9). If the server requests client authentication but no suitable certificate is available, the client **MUST** send a Certificate message containing no certificates (i.e., with the "certificate_list" field having length 0).

Meaning of this message:

This message conveys the endpoint's certificate chain to the peer.

Structure of this message:

```
opaque ASN1Cert<1..2^24-1>;

struct {
    opaque certificate_request_context<0..2^8-1>;
    ASN1Cert certificate_list<0..2^24-1>;
} Certificate;
```

certificate_request_context If this message is in response to a CertificateRequest, the value of certificate_request_context in that message. Otherwise, in the case of server authentication this field **SHALL** be zero length.

certificate_list This is a sequence (chain) of certificates. The sender's certificate **MUST** come first in the list. Each following certificate **SHOULD** directly certify one preceding it. Because certificate validation requires that trust anchors be distributed independently, a certificate that specifies a trust anchor **MAY** be omitted from the chain, provided that supported peers are known to possess any omitted certificates.

Note: Prior to TLS 1.3, "certificate_list" ordering required each certificate to certify the one immediately preceding it, however some implementations allowed some flexibility. Servers sometimes send both a current and deprecated intermediate for transitional purposes, and others are simply configured incorrectly, but these cases can nonetheless be validated properly. For maximum compatibility, all

implementations SHOULD be prepared to handle potentially extraneous certificates and arbitrary orderings from any TLS version, with the exception of the end-entity certificate which MUST be first.

The server's certificate list MUST always be non-empty. A client will send an empty certificate list if it does not have an appropriate certificate to send in response to the server's authentication request.

4.3.1.1. Server Certificate Selection

The following rules apply to the certificates sent by the server:

- The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC5081]).
- The server's end-entity certificate's public key (and associated restrictions) MUST be compatible with the selected authentication algorithm (currently RSA or ECDSA).
- The certificate MUST allow the key to be used for signing (i.e., the digitalSignature bit MUST be set if the Key Usage extension is present) with a signature scheme indicated in the client's "signature_algorithms" extension.
- The "server_name" and "trusted_ca_keys" extensions [RFC6066] are used to guide certificate selection. As servers MAY require the presence of the "server_name" extension, clients SHOULD send this extension, when applicable.

All certificates provided by the server MUST be signed by a signature algorithm that appears in the "signature_algorithms" extension provided by the client, if they are able to provide such a chain (see Section 4.2.2). Certificates that are self-signed or certificates that are expected to be trust anchors are not validated as part of the chain and therefore MAY be signed with any algorithm.

If the server cannot produce a certificate chain that is signed only via the indicated supported algorithms, then it SHOULD continue the handshake by sending the client a certificate chain of its choice that may include algorithms that are not known to be supported by the client. This fallback chain MAY use the deprecated SHA-1 hash algorithm only if the "signature_algorithms" extension provided by the client permits it. If the client cannot construct an acceptable chain using the provided certificates and decides to abort the handshake, then it MUST send an "unsupported_certificate" alert message and close the connection.

If the server has multiple certificates, it chooses one of them based on the above-mentioned criteria (in addition to other criteria, such as transport layer endpoint, local configuration and preferences).

4.3.1.2. Client Certificate Selection

The following rules apply to certificates sent by the client:

In particular:

- The certificate type MUST be X.509v3 [RFC5280], unless explicitly negotiated otherwise (e.g., [RFC5081]).
- If the `certificate_authorities` list in the certificate request message was non-empty, one of the certificates in the certificate chain SHOULD be issued by one of the listed CAs.
- The certificates MUST be signed using an acceptable signature algorithm, as described in Section 4.2.9. Note that this relaxes the constraints on certificate-signing algorithms found in prior versions of TLS.
- If the `certificate_extensions` list in the certificate request message was non-empty, the end-entity certificate MUST match the extension OIDs recognized by the client, as described in Section 4.2.9.

Note that, as with the server certificate, there are certificates that use algorithm combinations that cannot be currently used with TLS.

4.3.1.3. Receiving a Certificate Message

In general, detailed certificate validation procedures are out of scope for TLS (see [RFC5280]). This section provides TLS-specific requirements.

If the server supplies an empty Certificate message, the client MUST terminate the handshake with a fatal "decode_error" alert.

If the client does not send any certificates, the server MAY at its discretion either continue the handshake without client authentication, or respond with a fatal "handshake_failure" alert. Also, if some aspect of the certificate chain was unacceptable (e.g., it was not signed by a known, trusted CA), the server MAY at its discretion either continue the handshake (considering the client unauthenticated) or send a fatal alert.

Any endpoint receiving any certificate signed using any signature algorithm using an MD5 hash MUST send a "bad_certificate" alert message and close the connection. SHA-1 is deprecated and therefore NOT RECOMMENDED. All endpoints are RECOMMENDED to transition to SHA-256 or better as soon as possible to maintain interoperability with implementations currently in the process of phasing out SHA-1 support.

Note that a certificate containing a key for one signature algorithm MAY be signed using a different signature algorithm (for instance, an RSA key signed with an ECDSA key).

Endpoints that reject certification paths due to use of a deprecated hash MUST send a fatal "bad_certificate" alert message before closing the connection.

4.3.2. Certificate Verify

When this message will be sent:

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate and also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a Certificate (i.e., when the Certificate message is non-empty). When sent, this message MUST appear immediately after the Certificate Message and immediately prior to the Finished message.

Structure of this message:

```
struct {
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} CertificateVerify;
```

The algorithm field specifies the signature algorithm used (see [Section 4.2.2](#) for the definition of this field). The signature is a digital signature using that algorithm that covers the hash output described in [Section 4.3](#) namely:

$$\text{Hash}(\text{Handshake Context} + \text{Certificate}) + \text{Hash}(\text{resumption_context})$$

In TLS 1.3, the digital signature process takes as input:

- A signing key

RSASSA-PSS algorithm, regardless of whether RSASSA-PKCS1-v1_5 algorithms appear in "signature_algorithms". SHA-1 MUST NOT be used in any signatures in CertificateVerify. All SHA-1 signature algorithms in this specification are defined solely for use in legacy certificates, and are not valid for CertificateVerify signatures.

Note: When used with non-certificate-based handshakes (e.g., PSK), the client's signature does not cover the server's certificate directly, although it does cover the server's Finished message, which transitively includes the server's certificate when the PSK derives from a certificate-authenticated handshake. [PSK-FINISHED] describes a concrete attack on this mode if the Finished is omitted from the signature. It is unsafe to use certificate-based client authentication when the client might potentially share the same PSK/key-id pair with two different endpoints. In order to ensure this, implementations MUST NOT mix certificate-based client authentication with pure PSK modes (i.e., those where the PSK was not derived from a previous non-PSK handshake).

4.3.3. Finished

When this message will be sent:

The Finished message is the final message in the authentication block. It is essential for providing authentication of the handshake and of the computed keys.

Meaning of this message:

Recipients of Finished messages MUST verify that the contents are correct. Once a side has sent its Finished message and received and validated the Finished message from its peer, it may begin to send and receive application data over the connection.

The key used to compute the finished message is computed from the Base key defined in [Section 4.3](#) using HKDF (see [Section 7.1](#)). Specifically:

```
client_finished_key =
    HKDF-Expand-Label(BaseKey, "client finished", "", Hash.Length)

server_finished_key =
    HKDF-Expand-Label(BaseKey, "server finished", "", Hash.Length)
```

Structure of this message:

```
struct {  
    opaque verify_data[Hash.length];  
} Finished;
```

The `verify_data` value is computed as follows:

```
verify_data =  
    HMAC(finished_key, Hash(  
        Handshake Context +  
        Certificate* +  
        CertificateVerify*  
    ) +  
    Hash(resumption_context)  
)
```

* Only included if present.

Where HMAC [RFC2104] uses the Hash algorithm for the handshake. As noted above, the HMAC input can generally be implemented by a running hash, i.e., just the handshake hash at this point.

In previous versions of TLS, the `verify_data` was always 12 octets long. In the current version of TLS, it is the size of the HMAC output for the Hash used for the handshake.

Note: Alerts and any other record types are not handshake messages and are not included in the hash computations.

Any records following a 1-RTT Finished message MUST be encrypted under the application traffic key. In particular, this includes any alerts sent by the server in response to client Certificate and CertificateVerify messages.

4.4. Post-Handshake Messages

TLS also allows other messages to be sent after the main handshake. These messages use a handshake content type and are encrypted under the application traffic key.

Handshake messages sent after the handshake MUST NOT be interleaved with other record types. That is, if a message is split over two or more handshake records, there MUST NOT be any other records between them.

4.4.1. New Session Ticket Message

At any time after the server has received the client Finished message, it MAY send a NewSessionTicket message. This message creates a pre-shared key (PSK) binding between the ticket value and the following two values derived from the resumption master secret:

```

resumption_psk = HKDF-Expand-Label(
    resumption_secret,
    "resumption psk", "", Hash.Length)

resumption_context = HKDF-Expand-Label(
    resumption_secret,
    "resumption context", "", Hash.Length)

```

The client MAY use this PSK for future handshakes by including the ticket value in the "pre_shared_key" extension in its ClientHello (Section 4.2.5). Servers MAY send multiple tickets on a single connection, either immediately after each other or after specific events. For instance, the server might send a new ticket after post-handshake authentication in order to encapsulate the additional client authentication state. Clients SHOULD attempt to use each ticket no more than once, with more recent tickets being used first. For handshakes that do not use a resumption_psk, the resumption_context is a string of Hash.Length zeroes. [[Note: this will not be safe if/when we add additional server signatures with PSK: OPEN ISSUE <https://github.com/tlswg/tls13-spec/issues/558>]]

Any ticket MUST only be resumed with a cipher suite that is identical to that negotiated connection where the ticket was established.

```

enum { (65535) } TicketExtensionType;

struct {
    TicketExtensionType extension_type;
    opaque extension_data<1..2^16-1>;
} TicketExtension;

struct {
    uint32 ticket_lifetime;
    PskKeMode ke_modes<1..255>;
    PskAuthMode auth_modes<1..255>;
    opaque ticket<1..2^16-1>;
    TicketExtension extensions<0..2^16-2>;
} NewSessionTicket;

```

ke_modes The key exchange modes with which this ticket can be used in descending order of server preference.

`auth_modes` The authentication modes with which this ticket can be used in descending order of server preference.

`ticket_lifetime` Indicates the lifetime in seconds as a 32-bit unsigned integer in network byte order from the time of ticket issuance. Servers **MUST NOT** use any value more than 604800 seconds (7 days). The value of zero indicates that the ticket should be discarded immediately. Clients **MUST NOT** cache session tickets for longer than 7 days, regardless of the `ticket_lifetime`. It **MAY** delete the ticket earlier based on local policy. A server **MAY** treat a ticket as valid for a shorter period of time than what is stated in the `ticket_lifetime`.

`ticket` The value of the ticket to be used as the PSK identifier. The ticket itself is an opaque label. It **MAY** either be a database lookup key or a self-encrypted and self-authenticated value. [Section 4 of \[RFC5077\]](#) describes a recommended ticket construction mechanism.

`ticket_extensions` A set of extension values for the ticket. Clients **MUST** ignore unrecognized extensions.

This document defines one ticket extension, "ticket_early_data_info"

```
struct {  
    uint32 ticket_age_add;  
} TicketEarlyDataInfo;
```

This extension indicates that the ticket may be used to send 0-RTT data ([Section 4.2.6](#)). It contains one value:

`ticket_age_add` A randomly generated 32-bit value that is used to obscure the age of the ticket that the client includes in the "early_data" extension. The client-side ticket age is added to this value modulo 2^{32} to obtain the value that is transmitted by the client.

4.4.2. Post-Handshake Authentication

The server is permitted to request client authentication at any time after the handshake has completed by sending a `CertificateRequest` message. The client **SHOULD** respond with the appropriate Authentication messages. If the client chooses to authenticate, it **MUST** send `Certificate`, `CertificateVerify`, and `Finished`. If it declines, it **MUST** send a `Certificate` message containing no certificates followed by `Finished`.

Note: Because client authentication may require prompting the user, servers MUST be prepared for some delay, including receiving an arbitrary number of other messages between sending the CertificateRequest and receiving a response. In addition, clients which receive multiple CertificateRequests in close succession MAY respond to them in a different order than they were received (the certificate_request_context value allows the server to disambiguate the responses).

4.4.3. Key and IV Update

```
struct {} KeyUpdate;
```

The KeyUpdate handshake message is used to indicate that the sender is updating its sending cryptographic keys. This message can be sent by the server after sending its first flight and the client after sending its second flight. Implementations that receive a KeyUpdate message prior to receiving a Finished message as part of the 1-RTT handshake MUST generate a fatal "unexpected_message" alert. After sending a KeyUpdate message, the sender SHALL send all its traffic using the next generation of keys, computed as described in [Section 7.2](#). Upon receiving a KeyUpdate, the receiver MUST update their receiving keys and if they have not already updated their sending state up to or past the then current receiving generation MUST send their own KeyUpdate prior to sending any other messages. This mechanism allows either side to force an update to the entire connection. Note that implementations may receive an arbitrary number of messages between sending a KeyUpdate and receiving the peer's KeyUpdate because those messages may already be in flight.

Note that if implementations independently send their own KeyUpdates and they cross in flight, this only results in an update of one generation; when each side receives the other side's update it just updates its receive keys and notes that the generations match and thus no send update is needed.

Note that the side which sends its KeyUpdate first needs to retain its receive traffic keys (though not the traffic secret) for the previous generation of keys until it receives the KeyUpdate from the other side.

Both sender and receiver MUST encrypt their KeyUpdate messages with the old keys. Additionally, both sides MUST enforce that a KeyUpdate with the old key is received before accepting any messages encrypted with the new key. Failure to do so may allow message truncation attacks.

4.5. Handshake Layer and Key Changes

Handshake messages MUST NOT span key changes. Because the ServerHello, Finished, and KeyUpdate messages signal a key change, upon receiving these messages a receiver MUST verify that the end of these messages aligns with a record boundary; if not, then it MUST send a fatal "unexpected_message" alert.

5. Record Protocol

The TLS record protocol takes messages to be transmitted, fragments the data into manageable blocks, protects the records, and transmits the result. Received data is decrypted and verified, reassembled, and then delivered to higher-level clients.

TLS records are typed, which allows multiple higher level protocols to be multiplexed over the same record layer. This document specifies three content types: handshake, application data, and alert. Implementations MUST NOT send record types not defined in this document unless negotiated by some extension. If a TLS implementation receives an unexpected record type, it MUST send an "unexpected_message" alert. New record content type values are assigned by IANA in the TLS Content Type Registry as described in [Section 10](#).

Application data messages are carried by the record layer and are fragmented and encrypted as described below. The messages are treated as transparent data to the record layer.

5.1. Record Layer

The TLS record layer receives uninterpreted data from higher layers in non-empty blocks of arbitrary size.

The record layer fragments information blocks into TLSPlaintext records carrying data in chunks of 2^{14} bytes or less. Message boundaries are not preserved in the record layer (i.e., multiple messages of the same ContentType MAY be coalesced into a single TLSPlaintext record, or a single message MAY be fragmented across several records). Alert messages ([Section 6](#)) MUST NOT be fragmented across records.

```
enum {
    alert(21),
    handshake(22),
    application_data(23)
    (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion legacy_record_version = { 3, 1 }; /* TLS v1.x */
    uint16 length;
    opaque fragment[TLSPplaintext.length];
} TLSPplaintext;
```

type The higher-level protocol used to process the enclosed fragment.

legacy_record_version This value MUST be set to { 3, 1 } for all records. This field is deprecated and MUST be ignored for all purposes.

length The length (in bytes) of the following TLSPplaintext.fragment. The length MUST NOT exceed 2^{14} .

fragment The data being transmitted. This value transparent and treated as an independent block to be dealt with by the higher-level protocol specified by the type field.

This document describes TLS Version 1.3, which uses the version { 3, 4 }. The version value 3.4 is historical, deriving from the use of { 3, 1 } for TLS 1.0 and { 3, 0 } for SSL 3.0. In order to maximize backwards compatibility, the record layer version identifies as simply TLS 1.0. Endpoints supporting other versions negotiate the version to use by following the procedure and requirements in [Appendix C](#).

Implementations MUST NOT send zero-length fragments of Handshake or Alert types, even if those fragments contain padding. Zero-length fragments of Application data MAY be sent as they are potentially useful as a traffic analysis countermeasure.

When record protection has not yet been engaged, TLSPplaintext structures are written directly onto the wire. Once record protection has started, TLSPplaintext records are protected and sent as described in the following section.

5.2. Record Payload Protection

The record protection functions translate a `TLSPplaintext` structure into a `TLSCiphertext`. The deprotection functions reverse the process. In TLS 1.3 as opposed to previous versions of TLS, all ciphers are modeled as "Authenticated Encryption with Additional Data" (AEAD) [RFC5116]. AEAD functions provide a unified encryption and authentication operation which turns plaintext into authenticated ciphertext and back again. Each encrypted record consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

```
struct {
    opaque content[TLSPplaintext.length];
    ContentType type;
    uint8 zeros[length_of_padding];
} TLSInnerPlaintext;

struct {
    ContentType opaque_type = application_data(23); /* see fragment.type */
    ProtocolVersion legacy_record_version = { 3, 1 }; /* TLS v1.x */
    uint16 length;
    opaque encrypted_record[length];
} TLSCiphertext;
```

`content` The cleartext of `TLSPplaintext.fragment`.

`type` The content type of the record.

`zeros` An arbitrary-length run of zero-valued bytes may appear in the cleartext after the type field. This provides an opportunity for senders to pad any TLS record by a chosen amount as long as the total stays within record size limits. See [Section 5.4](#) for more details.

`opaque_type` The outer `opaque_type` field of a `TLSCiphertext` record is always set to the value 23 (`application_data`) for outward compatibility with middleboxes accustomed to parsing previous versions of TLS. The actual content type of the record is found in `fragment.type` after decryption.

`legacy_record_version` The `legacy_record_version` field is identical to `TLSPplaintext.legacy_record_version` and is always { 3, 1 }. Note that the handshake protocol including the `ClientHello` and `ServerHello` messages authenticates the protocol version, so this value is redundant.

length The length (in bytes) of the following `TLSCiphertext.fragment`, which is the sum of the lengths of the content and the padding, plus one for the inner content type. The length MUST NOT exceed $2^{14} + 256$. An endpoint that receives a record that exceeds this length MUST generate a fatal "record_overflow" alert.

encrypted_record The AEAD encrypted form of the serialized `TLSPayload` structure.

AEAD ciphers take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check, as described in [Section 2.1 of \[RFC5116\]](#). The key is either the `client_write_key` or the `server_write_key`, the nonce is derived from the sequence number (see [Section 5.3](#)) and the `client_write_iv` or `server_write_iv`, and the additional data input is empty (zero length). Derivation of traffic keys is defined in [Section 7.3](#).

The plaintext is the concatenation of `TLSPayload.fragment`, `TLSPayload.type`, and any padding bytes (zeros).

The AEAD output consists of the ciphertext output by the AEAD encryption operation. The length of the plaintext is greater than `TLSPayload.length` due to the inclusion of `TLSPayload.type` and however much padding is supplied by the sender. The length of the AEAD output will generally be larger than the plaintext, but by an amount that varies with the AEAD cipher. Since the ciphers might incorporate padding, the amount of overhead could vary with different lengths of plaintext. Symbolically,

```
AEADEncrypted =  
    AEAD-Encrypt(write_key, nonce, plaintext of fragment)
```

In order to decrypt and verify, the cipher takes as input the key, nonce, and the `AEADEncrypted` value. The output is either the plaintext or an error indicating that the decryption failed. There is no separate integrity check. That is:

```
plaintext of fragment =  
    AEAD-Decrypt(write_key, nonce, AEADEncrypted)
```

If the decryption fails, a fatal "bad_record_mac" alert MUST be generated.

An AEAD cipher MUST NOT produce an expansion of greater than 255 bytes. An endpoint that receives a record from its peer with `TLSCiphertext.length` larger than $2^{14} + 256$ octets MUST generate a fatal "record_overflow" alert. This limit is derived from the

maximum TLSPlaintext length of 2^{14} octets + 1 octet for ContentType + the maximum AEAD expansion of 255 octets.

5.3. Per-Record Nonce

A 64-bit sequence number is maintained separately for reading and writing records. Each sequence number is set to zero at the beginning of a connection and whenever the key is changed.

The sequence number is incremented after reading or writing each record. The first record transmitted under a particular set of traffic keys record key MUST use sequence number 0.

Sequence numbers do not wrap. If a TLS implementation would need to wrap a sequence number, it MUST either rekey ([Section 4.4.3](#)) or terminate the connection.

The length of the per-record nonce (`iv_length`) is set to $\max(8 \text{ bytes}, N_MIN)$ for the AEAD algorithm (see [\[RFC5116\] Section 4](#)). An AEAD algorithm where `N_MAX` is less than 8 bytes MUST NOT be used with TLS. The per-record nonce for the AEAD construction is formed as follows:

1. The 64-bit record sequence number is padded to the left with zeroes to `iv_length`.
2. The padded sequence number is XORed with the static `client_write_iv` or `server_write_iv`, depending on the role.

The resulting quantity (of length `iv_length`) is used as the per-record nonce.

Note: This is a different construction from that in TLS 1.2, which specified a partially explicit nonce.

5.4. Record Padding

All encrypted TLS records can be padded to inflate the size of the TLSCipherText. This allows the sender to hide the size of the traffic from an observer.

When generating a TLSCiphertext record, implementations MAY choose to pad. An unpadded record is just a record with a padding length of zero. Padding is a string of zero-valued bytes appended to the ContentType field before encryption. Implementations MUST set the padding octets to all zeros before encrypting.

Application Data records may contain a zero-length fragment.content if the sender desires. This permits generation of plausibly-sized

cover traffic in contexts where the presence or absence of activity may be sensitive. Implementations MUST NOT send Handshake or Alert records that have a zero-length `fragment.content`.

The padding sent is automatically verified by the record protection mechanism: Upon successful decryption of a `TLSCiphertext.fragment`, the receiving implementation scans the field from the end toward the beginning until it finds a non-zero octet. This non-zero octet is the content type of the message. This padding scheme was selected because it allows padding of any encrypted TLS record by an arbitrary size (from zero up to TLS record size limits) without introducing new content types. The design also enforces all-zero padding octets, which allows for quick detection of padding errors.

Implementations MUST limit their scanning to the cleartext returned from the AEAD decryption. If a receiving implementation does not find a non-zero octet in the cleartext, it should treat the record as having an unexpected `ContentType`, sending an "unexpected_message" alert.

The presence of padding does not change the overall record size limitations - the full fragment plaintext may not exceed 2^{14} octets.

Selecting a padding policy that suggests when and how much to pad is a complex topic, and is beyond the scope of this specification. If the application layer protocol atop TLS has its own padding, it may be preferable to pad application_data TLS records within the application layer. Padding for encrypted handshake and alert TLS records must still be handled at the TLS layer, though. Later documents may define padding selection algorithms, or define a padding policy request mechanism through TLS extensions or some other means.

5.5. Limits on Key Usage

There are cryptographic limits on the amount of plaintext which can be safely encrypted under a given set of keys. [AEAD-LIMITS] provides an analysis of these limits under the assumption that the underlying primitive (AES or ChaCha20) has no weaknesses. Implementations SHOULD do a key update [Section 4.4.3](#) prior to reaching these limits.

For AES-GCM, up to $2^{24.5}$ full-size records may be encrypted on a given connection while keeping a safety margin of approximately 2^{-57} for Authenticated Encryption (AE) security. For ChaCha20/Poly1305, the record sequence number will wrap before the safety limit is reached.

6. Alert Protocol

One of the content types supported by the TLS record layer is the alert type. Like other messages, alert messages are encrypted as specified by the current connection state.

Alert messages convey the severity of the message (warning or fatal) and a description of the alert. Warning-level messages are used to indicate orderly closure of the connection (see [Section 6.1](#)). Upon receiving a warning-level alert, the TLS implementation SHOULD indicate end-of-data to the application and, if appropriate for the alert type, send a closure alert in response.

Fatal-level messages are used to indicate abortive closure of the connection (See [Section 6.2](#)). Upon receiving a fatal-level alert, the TLS implementation SHOULD indicate an error to the application and MUST NOT allow any further data to be sent or received on the connection. Servers and clients MUST forget keys and secrets associated with a failed connection. Stateful implementations of session tickets (as in many clients) SHOULD discard tickets associated with failed connections.

All the alerts listed in [Section 6.2](#) MUST be sent as fatal and MUST be treated as fatal regardless of the AlertLevel in the message. Unknown alert types MUST be treated as fatal.

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
    close_notify(0),
    end_of_early_data(1),
    unexpected_message(10),
    bad_record_mac(20),
    record_overflow(22),
    handshake_failure(40),
    bad_certificate(42),
    unsupported_certificate(43),
    certificate_revoked(44),
    certificate_expired(45),
    certificate_unknown(46),
    illegal_parameter(47),
    unknown_ca(48),
    access_denied(49),
    decode_error(50),
    decrypt_error(51),
    protocol_version(70),
    insufficient_security(71),
    internal_error(80),
    inappropriate_fallback(86),
    user_canceled(90),
    missing_extension(109),
    unsupported_extension(110),
    certificate_unobtainable(111),
    unrecognized_name(112),
    bad_certificate_status_response(113),
    bad_certificate_hash_value(114),
    unknown_psk_identity(115),
    (255)
} AlertDescription;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

6.1. Closure Alerts

The client and the server must share knowledge that the connection is ending in order to avoid a truncation attack. Failure to properly close a connection does not prohibit a session from being resumed.

`close_notify` This alert notifies the recipient that the sender will not send any more messages on this connection. Any data received after a closure **MUST** be ignored.

`end_of_early_data` This alert is sent by the client to indicate that all 0-RTT `application_data` messages have been transmitted (or none will be sent at all) and that this is the end of the flight. This alert **MUST** be at the warning level. Servers **MUST NOT** send this alert and clients receiving it **MUST** terminate the connection with an `unexpected_message` alert.

`user_canceled` This alert notifies the recipient that the sender is canceling the handshake for some reason unrelated to a protocol failure. If a user cancels an operation after the handshake is complete, just closing the connection by sending a `close_notify` is more appropriate. This alert **SHOULD** be followed by a `close_notify`. This alert is generally a warning.

Either party **MAY** initiate a close by sending a `close_notify` alert. Any data received after a closure alert is ignored. If a transport-level close is received prior to a `close_notify`, the receiver cannot know that all the data that was sent has been received.

Each party **MUST** send a `close_notify` alert before closing the write side of the connection, unless some other fatal alert has been transmitted. The other party **MUST** respond with a `close_notify` alert of its own and close down the connection immediately, discarding any pending writes. The initiator of the close need not wait for the responding `close_notify` alert before closing the read side of the connection.

If the application protocol using TLS provides that any data may be carried over the underlying transport after the TLS connection is closed, the TLS implementation must receive the responding `close_notify` alert before indicating to the application layer that the TLS connection has ended. If the application protocol will not transfer any additional data, but will only close the underlying transport connection, then the implementation **MAY** choose to close the transport without waiting for the responding `close_notify`. No part of this standard should be taken to dictate the manner in which a usage profile for TLS manages its data transport, including when connections are opened or closed.

Note: It is assumed that closing a connection reliably delivers pending data before destroying the transport.

6.2. Error Alerts

Error handling in the TLS Handshake Protocol is very simple. When an error is detected, the detecting party sends a message to its peer. Upon transmission or receipt of a fatal alert message, both parties immediately close the connection. Whenever an implementation

encounters a condition which is defined as a fatal alert, it MUST send the appropriate alert prior to closing the connection. All alerts defined in this section below, as well as all unknown alerts are universally considered fatal as of TLS 1.3 (see [Section 6](#)).

The following error alerts are defined:

`unexpected_message` An inappropriate message was received. This alert should never be observed in communication between proper implementations.

`bad_record_mac` This alert is returned if a record is received which cannot be deprotected. Because AEAD algorithms combine decryption and verification, this alert is used for all deprotection failures. This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

`record_overflow` A TLSCiphertext record was received that had a length more than $2^{14} + 256$ bytes, or a record decrypted to a TLSPlaintext record with more than 2^{14} bytes. This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

`handshake_failure` Reception of a "handshake_failure" alert message indicates that the sender was unable to negotiate an acceptable set of security parameters given the options available.

`bad_certificate` A certificate was corrupt, contained signatures that did not verify correctly, etc.

`unsupported_certificate` A certificate was of an unsupported type.

`certificate_revoked` A certificate was revoked by its signer.

`certificate_expired` A certificate has expired or is not currently valid.

`certificate_unknown` Some other (unspecified) issue arose in processing the certificate, rendering it unacceptable.

`illegal_parameter` A field in the handshake was out of range or inconsistent with other fields.

`unknown_ca` A valid certificate chain or partial chain was received, but the certificate was not accepted because the CA certificate could not be located or couldn't be matched with a known, trusted CA.

`access_denied` A valid certificate or PSK was received, but when access control was applied, the sender decided not to proceed with negotiation.

`decode_error` A message could not be decoded because some field was out of the specified range or the length of the message was incorrect. This alert should never be observed in communication between proper implementations, except when messages were corrupted in the network.

`decrypt_error` A handshake cryptographic operation failed, including being unable to correctly verify a signature or validate a Finished message.

`protocol_version` The protocol version the peer has attempted to negotiate is recognized but not supported. (see [Appendix C](#))

`insufficient_security` Returned instead of "handshake_failure" when a negotiation has failed specifically because the server requires ciphers more secure than those supported by the client.

`internal_error` An internal error unrelated to the peer or the correctness of the protocol (such as a memory allocation failure) makes it impossible to continue.

`inappropriate_fallback` Sent by a server in response to an invalid connection retry attempt from a client. (see [\[RFC7507\]](#))

`missing_extension` Sent by endpoints that receive a hello message not containing an extension that is mandatory to send for the offered TLS version. [\[\[TODO: IANA Considerations.\]\]](#)

`unsupported_extension` Sent by endpoints receiving any hello message containing an extension known to be prohibited for inclusion in the given hello message, including any extensions in a ServerHello not first offered in the corresponding ClientHello.

`certificate_unobtainable` Sent by servers when unable to obtain a certificate from a URL provided by the client via the "client_certificate_url" extension [\[RFC6066\]](#).

`unrecognized_name` Sent by servers when no server exists identified by the name provided by the client via the "server_name" extension [\[RFC6066\]](#).

`bad_certificate_status_response` Sent by clients when an invalid or unacceptable OCSP response is provided by the server via the "status_request" extension [\[RFC6066\]](#). This alert is always fatal.

`bad_certificate_hash_value` Sent by servers when a retrieved object does not have the correct hash provided by the client via the `"client_certificate_url"` extension [RFC6066].

`unknown_psk_identity` Sent by servers when PSK key establishment is desired but no acceptable PSK identity is provided by the client. Sending this alert is OPTIONAL; servers MAY instead choose to send a `"decrypt_error"` alert to merely indicate an invalid PSK identity.

New Alert values are assigned by IANA as described in [Section 10](#).

7. Cryptographic Computations

In order to begin connection protection, the TLS Record Protocol requires specification of a suite of algorithms, a master secret, and the client and server random values.

7.1. Key Schedule

The TLS handshake establishes one or more input secrets which are combined to create the actual working keying material, as detailed below. The key derivation process makes use of the HKDF-Extract and HKDF-Expand functions as defined for HKDF [RFC5869], as well as the functions defined below:

```
HKDF-Expand-Label(Secret, Label, HashValue, Length) =
    HKDF-Expand(Secret, HkdfLabel, Length)
```

Where `HkdfLabel` is specified as:

```
struct HkdfLabel
{
    uint16 length = Length;
    opaque label<9..255> = "TLS 1.3, " + Label;
    opaque hash_value<0..255> = HashValue;
};
```

```
Derive-Secret(Secret, Label, Messages) =
    HKDF-Expand-Label(Secret, Label,
        Hash(Messages) +
        Hash(resumption_context), Hash.Length)
```

The Hash function and the HKDF hash are the cipher suite hash function. `Hash.Length` is its output length.

Given a set of `n` `InputSecrets`, the final "master secret" is computed by iteratively invoking HKDF-Extract with `InputSecret_1`,

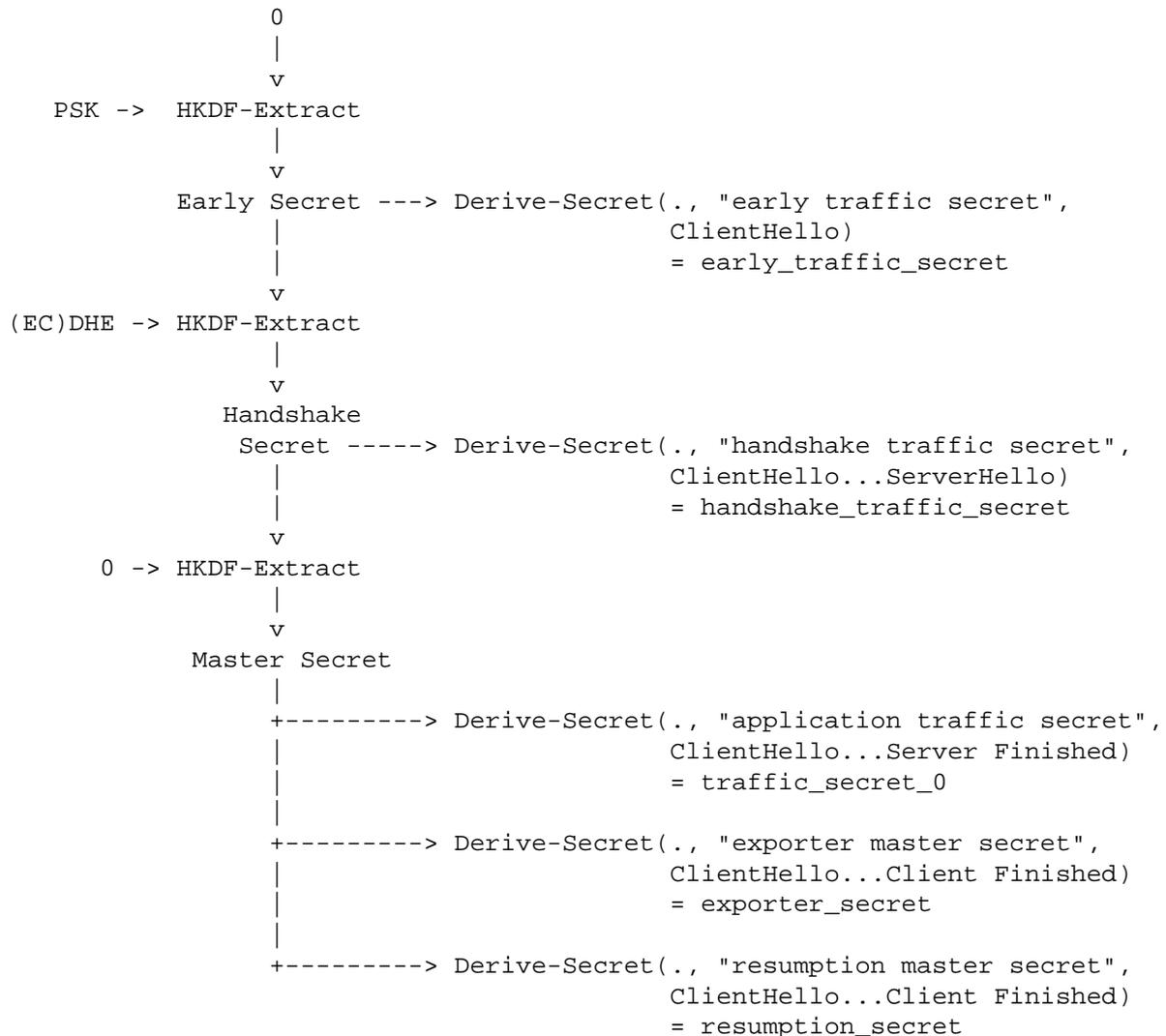
InputSecret_2, etc. The initial secret is simply a string of zeroes as long as the size of the Hash that is the basis for the HKDF. Concretely, for the present version of TLS 1.3, secrets are added in the following order:

- PSK
- (EC)DHE shared secret

This produces a full key derivation schedule shown in the diagram below. In this diagram, the following formatting conventions apply:

- HKDF-Extract is drawn as taking the Salt argument from the top and the IKM argument from the left.
- Derive-Secret's Secret argument is indicated by the arrow coming in from the left. For instance, the Early Secret is the Secret for generating the early_traffic_secret.

Note that the 0-RTT Finished message is not included in the Derive-Secret operation.



The general pattern here is that the secrets shown down the left side of the diagram are just raw entropy without context, whereas the secrets down the right side include handshake context and therefore can be used to derive working keys without additional context. Note that the different calls to `Derive-Secret` may take different Messages arguments, even with the same secret. In a 0-RTT exchange, `Derive-Secret` is called with four distinct transcripts; in a 1-RTT only exchange with three distinct transcripts.

If a given secret is not available, then the 0-value consisting of a string of `Hash.length` zeroes is used. Note that this does not mean skipping rounds, so if PSK is not in use Early Secret will still be `HKDF-Extract(0, 0)`.

7.2. Updating Traffic Keys and IVs

Once the handshake is complete, it is possible for either side to update its sending traffic keys using the KeyUpdate handshake message defined in [Section 4.4.3](#). The next generation of traffic keys is computed by generating `traffic_secret_N+1` from `traffic_secret_N` as described in this section then re-deriving the traffic keys as described in [Section 7.3](#).

The next-generation `traffic_secret` is computed as:

```
traffic_secret_N+1 = HKDF-Expand-Label(  
    traffic_secret_N,  
    "application traffic secret", "", Hash.Length)
```

Once `traffic_secret_N+1` and its associated traffic keys have been computed, implementations SHOULD delete `traffic_secret_N`. Once the directional keys are no longer needed, they SHOULD be deleted as well.

7.3. Traffic Key Calculation

The traffic keying material is generated from the following input values:

- A secret value
- A phase value indicating the phase of the protocol the keys are being generated for
- A purpose value indicating the specific value being generated
- The length of the key

The keying material is computed using:

```
key = HKDF-Expand-Label(Secret,  
    phase + " ", " + purpose,  
    "",  
    key_length)
```

The following table describes the inputs to the key calculation for each class of traffic keys:

Record Type	Secret	Phase
0-RTT Handshake	early_traffic_secret	"early handshake key expansion"
0-RTT Application	early_traffic_secret	"early application data key expansion"
Handshake	handshake_traffic_secret	"handshake key expansion"
Application Data	traffic_secret_N	"application data key expansion"

The following table indicates the purpose values for each type of key:

Key Type	Purpose
client_write_key	"client write key"
server_write_key	"server write key"
client_write_iv	"client write iv"
server_write_iv	"server write iv"

All the traffic keying material is recomputed whenever the underlying Secret changes (e.g., when changing from the handshake to application data keys or upon a key update).

7.3.1. Diffie-Hellman

A conventional Diffie-Hellman computation is performed. The negotiated key (Z) is converted to byte string by encoding in big-endian, padded with zeros up to the size of the prime. This byte string is used as the shared secret, and is used in the key schedule as specified above.

Note that this construction differs from previous versions of TLS which remove leading zeros.

7.3.2. Elliptic Curve Diffie-Hellman

For `secp256r1`, `secp384r1` and `secp521r1`, ECDH calculations (including parameter and key generation as well as the shared secret calculation) are performed according to [IEEE1363] using the ECKAS-DH1 scheme with the identity map as key derivation function (KDF), so that the shared secret is the x-coordinate of the ECDH shared secret elliptic curve point represented as an octet string. Note that this octet string (Z in IEEE 1363 terminology) as output by FE2OSP, the Field Element to Octet String Conversion Primitive, has constant length for any given field; leading zeros found in this octet string MUST NOT be truncated.

(Note that this use of the identity KDF is a technicality. The complete picture is that ECDH is employed with a non-trivial KDF because TLS does not directly use this secret for anything other than for computing other secrets.)

ECDH functions are used as follows:

- The public key to put into the `KeyShareEntry.key_exchange` structure is the result of applying the ECDH function to the secret key of appropriate length (into scalar input) and the standard public basepoint (into u-coordinate point input).
- The ECDH shared secret is the result of applying ECDH function to the secret key (into scalar input) and the peer's public key (into u-coordinate point input). The output is used raw, with no processing.

For X25519 and X448, see [RFC7748].

7.3.3. Exporters

[RFC5705] defines keying material exporters for TLS in terms of the TLS PRF. This document replaces the PRF with HKDF, thus requiring a new construction. The exporter interface remains the same, however the value is computed as:

```
HKDF-Expand-Label(exporter_secret,  
                  label, context_value, key_length)
```

8. Compliance Requirements

8.1. MTI Cipher Suites

In the absence of an application profile standard specifying otherwise, a TLS-compliant application **MUST** implement the TLS_AES_128_GCM_SHA256 cipher suite and **SHOULD** implement the TLS_AES_256_GCM_SHA384 and TLS_CHACHA20_POLY1305_SHA256 cipher suites.

A TLS-compliant application **MUST** support digital signatures with rsa_pkcs1_sha256 (for certificates), rsa_pss_sha256 (for CertificateVerify and certificates), and ecdsa_secp256r1_sha256. A TLS-compliant application **MUST** support key exchange with secp256r1 (NIST P-256) and **SHOULD** support key exchange with X25519 [RFC7748].

8.2. MTI Extensions

In the absence of an application profile standard specifying otherwise, a TLS-compliant application **MUST** implement the following TLS extensions:

- Signature Algorithms ("signature_algorithms"; [Section 4.2.2](#))
- Negotiated Groups ("supported_groups"; [Section 4.2.3](#))
- Key Share ("key_share"; [Section 4.2.4](#))
- Pre-Shared Key ("pre_shared_key"; [Section 4.2.5](#))
- Server Name Indication ("server_name"; [Section 3 of \[RFC6066\]](#))
- Cookie ("cookie"; [Section 4.2.1](#))

All implementations **MUST** send and use these extensions when offering applicable cipher suites:

- "signature_algorithms" is **REQUIRED** for certificate authenticated cipher suites.
- "supported_groups" and "key_share" are **REQUIRED** for DHE or ECDHE cipher suites.
- "pre_shared_key" is **REQUIRED** for PSK cipher suites.
- "cookie" is **REQUIRED** for all cipher suites.

When negotiating use of applicable cipher suites, endpoints **MUST** abort the connection with a "missing_extension" alert if the required extension was not provided. Any endpoint that receives any invalid

combination of cipher suites and extensions MAY abort the connection with a "missing_extension" alert, regardless of negotiated parameters.

Additionally, all implementations MUST support use of the "server_name" extension with applications capable of using it. Servers MAY require clients to send a valid "server_name" extension. Servers requiring this extension SHOULD respond to a ClientHello lacking a "server_name" extension with a fatal "missing_extension" alert.

Servers MUST NOT send the "signature_algorithms" extension; if a client receives this extension it MUST respond with a fatal "unsupported_extension" alert and close the connection.

9. Security Considerations

Security issues are discussed throughout this memo, especially in Appendices B, C, and D.

10. IANA Considerations

This document uses several registries that were originally created in [RFC4346]. IANA has updated these to reference this document. The registries and their allocation policies are below:

- TLS Cipher Suite Registry: Values with the first byte in the range 0-254 (decimal) are assigned via Specification Required [RFC2434]. Values with the first byte 255 (decimal) are reserved for Private Use [RFC2434]. IANA [SHALL add/has added] a "Recommended" column to the cipher suite registry. All cipher suites listed in [Appendix A.4](#) are marked as "Yes". All other cipher suites are marked as "No". IANA [SHALL add/has added] add a note to this column reading:

Cipher suites marked as "Yes" are those allocated via Standards Track RFCs. Cipher suites marked as "No" are not; cipher suites marked "No" range from "good" to "bad" from a cryptographic standpoint.

- TLS ContentType Registry: Future values are allocated via Standards Action [RFC2434].
- TLS Alert Registry: Future values are allocated via Standards Action [RFC2434].
- TLS HandshakeType Registry: Future values are allocated via Standards Action [RFC2434]. IANA [SHALL update/has updated] this

registry to rename item 4 from "NewSessionTicket" to "new_session_ticket".

This document also uses a registry originally created in [RFC4366]. IANA has updated it to reference this document. The registry and its allocation policy is listed below:

- TLS ExtensionType Registry: Values with the first byte in the range 0-254 (decimal) are assigned via Specification Required [RFC2434]. Values with the first byte 255 (decimal) are reserved for Private Use [RFC2434]. IANA [SHALL update/has updated] this registry to include the "key_share", "pre_shared_key", and "early_data" extensions as defined in this document.

IANA [shall update/has updated] this registry to include a "TLS 1.3" column with the following four values: "Client", indicating that the server shall not send them. "Clear", indicating that they shall be in the ServerHello. "Encrypted", indicating that they shall be in the EncryptedExtensions block, and "No" indicating that they are not used in TLS 1.3. This column [shall be/has been] initially populated with the values in this document. IANA [shall update/has updated] this registry to add a "Recommended" column. IANA [shall/has] initially populated this column with the values in the table below. This table has been generated by marking Standards Track RFCs as "Yes" and all others as "No".

Extension	Recommended	TLS 1.3
server_name [RFC6066]	Yes	Encrypted
max_fragment_length [RFC6066]	Yes	Encrypted
client_certificate_url [RFC6066]	Yes	Encrypted
trusted_ca_keys [RFC6066]	Yes	Encrypted
truncated_hmac [RFC6066]	Yes	No
status_request [RFC6066]	Yes	Encrypted
user_mapping [RFC4681]	Yes	Encrypted
client_authz [RFC5878]	No	Encrypted

server_authz [RFC5878]	No	Encrypted
cert_type [RFC6091]	Yes	Encrypted
supported_groups [RFC-ietf-tls-negotiated-ff-dhe]	Yes	Encrypted
ec_point_formats [RFC4492]	Yes	No
srp [RFC5054]	No	No
signature_algorithms [RFC5246]	Yes	Client
use_srtp [RFC5764]	Yes	Encrypted
heartbeat [RFC6520]	Yes	Encrypted
application_layer_protocol_negotiation [RFC7301]	Yes	Encrypted
status_request_v2 [RFC6961]	Yes	Encrypted
signed_certificate_timestamp [RFC6962]	No	Encrypted
client_certificate_type [RFC7250]	Yes	Encrypted
server_certificate_type [RFC7250]	Yes	Encrypted
padding [RFC7685]	Yes	Client
encrypt_then_mac [RFC7366]	Yes	No
extended_master_secret [RFC7627]	Yes	No
SessionTicket TLS [RFC4507]	Yes	No
renegotiation_info [RFC5746]	Yes	No
key_share [[this document]]	Yes	Clear
pre_shared_key [[this document]]	Yes	Clear

early_data [[this document]]	Yes	Encrypted
cookie [[this document]]	Yes	Encrypted/HelloRetryRequest

In addition, this document defines two new registries to be maintained by IANA

- TLS SignatureScheme Registry: Values with the first byte in the range 0-254 (decimal) are assigned via Specification Required [RFC2434]. Values with the first byte 255 (decimal) are reserved for Private Use [RFC2434]. This registry SHALL have a "Recommended" column. The registry [shall be/ has been] initially populated with the values described in Section 4.2.2. The following values SHALL be marked as "Recommended":
ecdsa_secp256r1_sha256, ecdsa_secp384r1_sha384, rsa_pss_sha256, rsa_pss_sha384, rsa_pss_sha512, ed25519.

11. References

11.1. Normative References

- [AES] National Institute of Standards and Technology, "Specification for the Advanced Encryption Standard (AES)", NIST FIPS 197, November 2001.
- [DH] Diffie, W. and M. Hellman, "New Directions in Cryptography", IEEE Transactions on Information Theory, V.IT-22 n.6 , June 1977.
- [I-D.irtf-cfrg-eddsa] Josefsson, S. and I. Liusvaara, "Edwards-curve Digital Signature Algorithm (EdDSA)", [draft-irtf-cfrg-eddsa-06](#) (work in progress), August 2016.
- [I-D.mattsson-tls-ecdhe-psk-aead] Mattsson, J. and D. Migault, "ECDHE_PSK with AES-GCM and AES-CCM Cipher Suites for Transport Layer Security (TLS)", [draft-mattsson-tls-ecdhe-psk-aead-05](#) (work in progress), April 2016.
- [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <<http://www.rfc-editor.org/info/rfc2104>>.

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [BCP 14](#), [RFC 2119](#), DOI 10.17487/RFC2119, March 1997, <<http://www.rfc-editor.org/info/rfc2119>>.
- [RFC2434] Narten, T. and H. Alvestrand, "Guidelines for Writing an IANA Considerations Section in RFCs", [RFC 2434](#), DOI 10.17487/RFC2434, October 1998, <<http://www.rfc-editor.org/info/rfc2434>>.
- [RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", [RFC 3447](#), DOI 10.17487/RFC3447, February 2003, <<http://www.rfc-editor.org/info/rfc3447>>.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", [RFC 5280](#), DOI 10.17487/RFC5280, May 2008, <<http://www.rfc-editor.org/info/rfc5280>>.
- [RFC5288] Salowey, J., Choudhury, A., and D. McGrew, "AES Galois Counter Mode (GCM) Cipher Suites for TLS", [RFC 5288](#), DOI 10.17487/RFC5288, August 2008, <<http://www.rfc-editor.org/info/rfc5288>>.
- [RFC5289] Rescorla, E., "TLS Elliptic Curve Cipher Suites with SHA-256/384 and AES Galois Counter Mode (GCM)", [RFC 5289](#), DOI 10.17487/RFC5289, August 2008, <<http://www.rfc-editor.org/info/rfc5289>>.
- [RFC5487] Badra, M., "Pre-Shared Key Cipher Suites for TLS with SHA-256/384 and AES Galois Counter Mode", [RFC 5487](#), DOI 10.17487/RFC5487, March 2009, <<http://www.rfc-editor.org/info/rfc5487>>.
- [RFC5705] Rescorla, E., "Keying Material Exporters for Transport Layer Security (TLS)", [RFC 5705](#), DOI 10.17487/RFC5705, March 2010, <<http://www.rfc-editor.org/info/rfc5705>>.
- [RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", [RFC 5869](#), DOI 10.17487/RFC5869, May 2010, <<http://www.rfc-editor.org/info/rfc5869>>.

- [RFC6066] Eastlake 3rd, D., "Transport Layer Security (TLS) Extensions: Extension Definitions", [RFC 6066](#), DOI 10.17487/RFC6066, January 2011, <<http://www.rfc-editor.org/info/rfc6066>>.
- [RFC6209] Kim, W., Lee, J., Park, J., and D. Kwon, "Addition of the ARIA Cipher Suites to Transport Layer Security (TLS)", [RFC 6209](#), DOI 10.17487/RFC6209, April 2011, <<http://www.rfc-editor.org/info/rfc6209>>.
- [RFC6367] Kanno, S. and M. Kanda, "Addition of the Camellia Cipher Suites to Transport Layer Security (TLS)", [RFC 6367](#), DOI 10.17487/RFC6367, September 2011, <<http://www.rfc-editor.org/info/rfc6367>>.
- [RFC6655] McGrew, D. and D. Bailey, "AES-CCM Cipher Suites for Transport Layer Security (TLS)", [RFC 6655](#), DOI 10.17487/RFC6655, July 2012, <<http://www.rfc-editor.org/info/rfc6655>>.
- [RFC6961] Pettersen, Y., "The Transport Layer Security (TLS) Multiple Certificate Status Request Extension", [RFC 6961](#), DOI 10.17487/RFC6961, June 2013, <<http://www.rfc-editor.org/info/rfc6961>>.
- [RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", [RFC 6979](#), DOI 10.17487/RFC6979, August 2013, <<http://www.rfc-editor.org/info/rfc6979>>.
- [RFC7251] McGrew, D., Bailey, D., Campagna, M., and R. Dugal, "AES-CCM Elliptic Curve Cryptography (ECC) Cipher Suites for TLS", [RFC 7251](#), DOI 10.17487/RFC7251, June 2014, <<http://www.rfc-editor.org/info/rfc7251>>.
- [RFC7443] Patil, P., Reddy, T., Salgueiro, G., and M. Petit-Huguenin, "Application-Layer Protocol Negotiation (ALPN) Labels for Session Traversal Utilities for NAT (STUN) Usages", [RFC 7443](#), DOI 10.17487/RFC7443, January 2015, <<http://www.rfc-editor.org/info/rfc7443>>.
- [RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", [RFC 7748](#), DOI 10.17487/RFC7748, January 2016, <<http://www.rfc-editor.org/info/rfc7748>>.

- [RFC7905] Langley, A., Chang, W., Mavrogiannopoulos, N., Strombergson, J., and S. Josefsson, "ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)", RFC 7905, DOI 10.17487/RFC7905, June 2016, <<http://www.rfc-editor.org/info/rfc7905>>.
- [SHS] National Institute of Standards and Technology, U.S. Department of Commerce, "Secure Hash Standard", NIST FIPS PUB 180-4, March 2012.
- [X690] ITU-T, "Information technology - ASN.1 encoding Rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)", ISO/IEC 8825-1:2002, 2002.
- [X962] ANSI, "Public Key Cryptography For The Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI X9.62, 1998.

11.2. Informative References

- [AEAD-LIMITS] Luykx, A. and K. Paterson, "Limits on Authenticated Encryption Use in TLS", 2016, <<http://www.isg.rhul.ac.uk/~kp/TLS-AEbounds.pdf>>.
- [BBFKZG16] Bhargavan, K., Brzuska, C., Fournet, C., Kohlweiss, M., Zanella-Beguelin, S., and M. Green, "Downgrade Resilience in Key-Exchange Protocols", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2016 , 2016.
- [CHSV16] Cremers, C., Horvat, M., Scott, S., and T. van der Merwe, "Automated Analysis and Verification of TLS 1.3: 0-RTT, Resumption and Delayed Authentication", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2016 , 2016.
- [CK01] Canetti, R. and H. Krawczyk, "Analysis of Key-Exchange Protocols and Their Use for Building Secure Channels", Proceedings of Eurocrypt 2001 , 2001.
- [DOW92] Diffie, W., van Oorschot, P., and M. Wiener, "'Authentication and authenticated key exchanges'", Designs, Codes and Cryptography , n.d..

- [DSS] National Institute of Standards and Technology, U.S. Department of Commerce, "Digital Signature Standard, version 4", NIST FIPS PUB 186-4, 2013.
- [ECDSA] American National Standards Institute, "Public Key Cryptography for the Financial Services Industry: The Elliptic Curve Digital Signature Algorithm (ECDSA)", ANSI ANS X9.62-2005, November 2005.
- [FGSW16] Fischlin, M., Guenther, F., Schmidt, B., and B. Warinschi, "Key Confirmation in Key Exchange: A Formal Treatment and Implications for TLS 1.3", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2016 , 2016.
- [FI06] Finney, H., "Bleichenbacher's RSA signature forgery based on implementation error", August 2006, <<https://www.ietf.org/mail-archive/web/openpgp/current/msg00999.html>>.
- [GCM] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST Special Publication 800-38D, November 2007.
- [I-D.ietf-tls-negotiated-ff-dhe]
Gillmor, D., "Negotiated Finite Field Diffie-Hellman Ephemeral Parameters for TLS", [draft-ietf-tls-negotiated-ff-dhe-10](#) (work in progress), June 2015.
- [IEEE1363]
IEEE, "Standard Specifications for Public Key Cryptography", IEEE 1363 , 2000.
- [LXZFH16] Li, X., Xu, J., Feng, D., Zhang, Z., and H. Hu, "Multiple Handshakes Security of TLS 1.3 Candidates", Proceedings of IEEE Symposium on Security and Privacy (Oakland) 2016 , 2016.
- [PKCS6] RSA Laboratories, "PKCS #6: RSA Extended Certificate Syntax Standard, version 1.5", November 1993.
- [PKCS7] RSA Laboratories, "PKCS #7: RSA Cryptographic Message Syntax Standard, version 1.5", November 1993.
- [PSK-FINISHED]
Cremers, C., Horvat, M., van der Merwe, T., and S. Scott, "Revision 10: possible attack if client authentication is allowed during PSK", 2015, <<https://www.ietf.org/mail-archive/web/tls/current/msg18215.html>>.

- [RFC0793] Postel, J., "Transmission Control Protocol", STD 7, [RFC 793](#), DOI 10.17487/RFC0793, September 1981, <<http://www.rfc-editor.org/info/rfc793>>.
- [RFC1948] Bellovin, S., "Defending Against Sequence Number Attacks", [RFC 1948](#), DOI 10.17487/RFC1948, May 1996, <<http://www.rfc-editor.org/info/rfc1948>>.
- [RFC3552] Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", [BCP 72](#), [RFC 3552](#), DOI 10.17487/RFC3552, July 2003, <<http://www.rfc-editor.org/info/rfc3552>>.
- [RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", [BCP 106](#), [RFC 4086](#), DOI 10.17487/RFC4086, June 2005, <<http://www.rfc-editor.org/info/rfc4086>>.
- [RFC4279] Eronen, P., Ed. and H. Tschofenig, Ed., "Pre-Shared Key Ciphersuites for Transport Layer Security (TLS)", [RFC 4279](#), DOI 10.17487/RFC4279, December 2005, <<http://www.rfc-editor.org/info/rfc4279>>.
- [RFC4302] Kent, S., "IP Authentication Header", [RFC 4302](#), DOI 10.17487/RFC4302, December 2005, <<http://www.rfc-editor.org/info/rfc4302>>.
- [RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", [RFC 4303](#), DOI 10.17487/RFC4303, December 2005, <<http://www.rfc-editor.org/info/rfc4303>>.
- [RFC4346] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.1", [RFC 4346](#), DOI 10.17487/RFC4346, April 2006, <<http://www.rfc-editor.org/info/rfc4346>>.
- [RFC4366] Blake-Wilson, S., Nystrom, M., Hopwood, D., Mikkelsen, J., and T. Wright, "Transport Layer Security (TLS) Extensions", [RFC 4366](#), DOI 10.17487/RFC4366, April 2006, <<http://www.rfc-editor.org/info/rfc4366>>.
- [RFC4492] Blake-Wilson, S., Bolyard, N., Gupta, V., Hawk, C., and B. Moeller, "Elliptic Curve Cryptography (ECC) Cipher Suites for Transport Layer Security (TLS)", [RFC 4492](#), DOI 10.17487/RFC4492, May 2006, <<http://www.rfc-editor.org/info/rfc4492>>.

- [RFC4506] Eisler, M., Ed., "XDR: External Data Representation Standard", STD 67, [RFC 4506](#), DOI 10.17487/RFC4506, May 2006, <<http://www.rfc-editor.org/info/rfc4506>>.
- [RFC4507] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", [RFC 4507](#), DOI 10.17487/RFC4507, May 2006, <<http://www.rfc-editor.org/info/rfc4507>>.
- [RFC4681] Santesson, S., Medvinsky, A., and J. Ball, "TLS User Mapping Extension", [RFC 4681](#), DOI 10.17487/RFC4681, October 2006, <<http://www.rfc-editor.org/info/rfc4681>>.
- [RFC5054] Taylor, D., Wu, T., Mavrogiannopoulos, N., and T. Perrin, "Using the Secure Remote Password (SRP) Protocol for TLS Authentication", [RFC 5054](#), DOI 10.17487/RFC5054, November 2007, <<http://www.rfc-editor.org/info/rfc5054>>.
- [RFC5077] Salowey, J., Zhou, H., Eronen, P., and H. Tschofenig, "Transport Layer Security (TLS) Session Resumption without Server-Side State", [RFC 5077](#), DOI 10.17487/RFC5077, January 2008, <<http://www.rfc-editor.org/info/rfc5077>>.
- [RFC5081] Mavrogiannopoulos, N., "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication", [RFC 5081](#), DOI 10.17487/RFC5081, November 2007, <<http://www.rfc-editor.org/info/rfc5081>>.
- [RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", [RFC 5116](#), DOI 10.17487/RFC5116, January 2008, <<http://www.rfc-editor.org/info/rfc5116>>.
- [RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", [RFC 5246](#), DOI 10.17487/RFC5246, August 2008, <<http://www.rfc-editor.org/info/rfc5246>>.
- [RFC5746] Rescorla, E., Ray, M., Dispensa, S., and N. Oskov, "Transport Layer Security (TLS) Renegotiation Indication Extension", [RFC 5746](#), DOI 10.17487/RFC5746, February 2010, <<http://www.rfc-editor.org/info/rfc5746>>.
- [RFC5763] Fischl, J., Tschofenig, H., and E. Rescorla, "Framework for Establishing a Secure Real-time Transport Protocol (SRTP) Security Context Using Datagram Transport Layer Security (DTLS)", [RFC 5763](#), DOI 10.17487/RFC5763, May 2010, <<http://www.rfc-editor.org/info/rfc5763>>.

- [RFC5764] McGrew, D. and E. Rescorla, "Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP)", [RFC 5764](#), DOI 10.17487/RFC5764, May 2010, <<http://www.rfc-editor.org/info/rfc5764>>.
- [RFC5878] Brown, M. and R. Housley, "Transport Layer Security (TLS) Authorization Extensions", [RFC 5878](#), DOI 10.17487/RFC5878, May 2010, <<http://www.rfc-editor.org/info/rfc5878>>.
- [RFC5929] Altman, J., Williams, N., and L. Zhu, "Channel Bindings for TLS", [RFC 5929](#), DOI 10.17487/RFC5929, July 2010, <<http://www.rfc-editor.org/info/rfc5929>>.
- [RFC6091] Mavrogiannopoulos, N. and D. Gillmor, "Using OpenPGP Keys for Transport Layer Security (TLS) Authentication", [RFC 6091](#), DOI 10.17487/RFC6091, February 2011, <<http://www.rfc-editor.org/info/rfc6091>>.
- [RFC6176] Turner, S. and T. Polk, "Prohibiting Secure Sockets Layer (SSL) Version 2.0", [RFC 6176](#), DOI 10.17487/RFC6176, March 2011, <<http://www.rfc-editor.org/info/rfc6176>>.
- [RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", [RFC 6347](#), DOI 10.17487/RFC6347, January 2012, <<http://www.rfc-editor.org/info/rfc6347>>.
- [RFC6520] Seggelmann, R., Tuexen, M., and M. Williams, "Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension", [RFC 6520](#), DOI 10.17487/RFC6520, February 2012, <<http://www.rfc-editor.org/info/rfc6520>>.
- [RFC6962] Laurie, B., Langley, A., and E. Kasper, "Certificate Transparency", [RFC 6962](#), DOI 10.17487/RFC6962, June 2013, <<http://www.rfc-editor.org/info/rfc6962>>.
- [RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", [RFC 7230](#), DOI 10.17487/RFC7230, June 2014, <<http://www.rfc-editor.org/info/rfc7230>>.
- [RFC7250] Wouters, P., Ed., Tschofenig, H., Ed., Gilmore, J., Weiler, S., and T. Kivinen, "Using Raw Public Keys in Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", [RFC 7250](#), DOI 10.17487/RFC7250, June 2014, <<http://www.rfc-editor.org/info/rfc7250>>.

- [RFC7301] Friedl, S., Popov, A., Langley, A., and E. Stephan, "Transport Layer Security (TLS) Application-Layer Protocol Negotiation Extension", RFC 7301, DOI 10.17487/RFC7301, July 2014, <<http://www.rfc-editor.org/info/rfc7301>>.
- [RFC7366] Gutmann, P., "Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)", RFC 7366, DOI 10.17487/RFC7366, September 2014, <<http://www.rfc-editor.org/info/rfc7366>>.
- [RFC7465] Popov, A., "Prohibiting RC4 Cipher Suites", RFC 7465, DOI 10.17487/RFC7465, February 2015, <<http://www.rfc-editor.org/info/rfc7465>>.
- [RFC7568] Barnes, R., Thomson, M., Pironti, A., and A. Langley, "Deprecating Secure Sockets Layer Version 3.0", RFC 7568, DOI 10.17487/RFC7568, June 2015, <<http://www.rfc-editor.org/info/rfc7568>>.
- [RFC7627] Bhargavan, K., Ed., Delignat-Lavaud, A., Pironti, A., Langley, A., and M. Ray, "Transport Layer Security (TLS) Session Hash and Extended Master Secret Extension", RFC 7627, DOI 10.17487/RFC7627, September 2015, <<http://www.rfc-editor.org/info/rfc7627>>.
- [RFC7685] Langley, A., "A Transport Layer Security (TLS) ClientHello Padding Extension", RFC 7685, DOI 10.17487/RFC7685, October 2015, <<http://www.rfc-editor.org/info/rfc7685>>.
- [RFC7924] Santesson, S. and H. Tschofenig, "Transport Layer Security (TLS) Cached Information Extension", RFC 7924, DOI 10.17487/RFC7924, July 2016, <<http://www.rfc-editor.org/info/rfc7924>>.
- [RSA] Rivest, R., Shamir, A., and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", Communications of the ACM v. 21, n. 2, pp. 120-126., February 1978.
- [SIGMA] Krawczyk, H., "SIGMA: the 'SIGn-and-MAC' approach to authenticated Di e-Hellman and its use in the IKE protocols", Proceedings of CRYPTO 2003 , 2003.
- [SLOTH] Bhargavan, K. and G. Leurent, "Transcript Collision Attacks: Breaking Authentication in TLS, IKE, and SSH", Network and Distributed System Security Symposium (NDSS 2016) , 2016.

- [SSL2] Hickman, K., "The SSL Protocol", February 1995.
- [SSL3] Freier, A., Karlton, P., and P. Kocher, "The SSL 3.0 Protocol", November 1996.
- [TIMING] Boneh, D. and D. Brumley, "Remote timing attacks are practical", USENIX Security Symposium, 2003.
- [X501] "Information Technology - Open Systems Interconnection - The Directory: Models", ITU-T X.501, 1993.

11.3. URIs

- [1] <mailto:tls@ietf.org>

Appendix A. Protocol Data Structures and Constant Values

This section describes protocol types and constants. Values listed as `_RESERVED` were used in previous versions of TLS and are listed here for completeness. TLS 1.3 implementations **MUST NOT** send them but might receive them from older TLS implementations.

A.1. Record Layer

```
enum {
    invalid_RESERVED(0),
    change_cipher_spec_RESERVED(20),
    alert(21),
    handshake(22),
    application_data(23)
    (255)
} ContentType;

struct {
    ContentType type;
    ProtocolVersion legacy_record_version = { 3, 1 }; /* TLS v1.x */
    uint16 length;
    opaque fragment[TLSPplaintext.length];
} TLSPplaintext;

struct {
    opaque content[TLSPplaintext.length];
    ContentType type;
    uint8 zeros[length_of_padding];
} TLSInnerPlaintext;

struct {
    ContentType opaque_type = application_data(23); /* see fragment.type */
    ProtocolVersion legacy_record_version = { 3, 1 }; /* TLS v1.x */
    uint16 length;
    opaque encrypted_record[length];
} TLSCiphertext;
```

A.2. Alert Messages

```
enum { warning(1), fatal(2), (255) } AlertLevel;

enum {
    close_notify(0),
    end_of_early_data(1),
    unexpected_message(10),
    bad_record_mac(20),
    decryption_failed_RESERVED(21),
    record_overflow(22),
    decompression_failure_RESERVED(30),
    handshake_failure(40),
    no_certificate_RESERVED(41),
    bad_certificate(42),
    unsupported_certificate(43),
    certificate_revoked(44),
    certificate_expired(45),
    certificate_unknown(46),
    illegal_parameter(47),
    unknown_ca(48),
    access_denied(49),
    decode_error(50),
    decrypt_error(51),
    export_restriction_RESERVED(60),
    protocol_version(70),
    insufficient_security(71),
    internal_error(80),
    inappropriate_fallback(86),
    user_canceled(90),
    no_renegotiation_RESERVED(100),
    missing_extension(109),
    unsupported_extension(110),
    certificate_unobtainable(111),
    unrecognized_name(112),
    bad_certificate_status_response(113),
    bad_certificate_hash_value(114),
    unknown_psk_identity(115),
    (255)
} AlertDescription;

struct {
    AlertLevel level;
    AlertDescription description;
} Alert;
```

A.3. Handshake Protocol

```
enum {
    hello_request_RESERVED(0),
    client_hello(1),
    server_hello(2),
    new_session_ticket(4),
    hello_retry_request(6),
    encrypted_extensions(8),
    certificate(11),
    server_key_exchange_RESERVED(12),
    certificate_request(13),
    server_hello_done_RESERVED(14),
    certificate_verify(15),
    client_key_exchange_RESERVED(16),
    finished(20),
    key_update(24),
    (255)
} HandshakeType;

struct {
    HandshakeType msg_type;      /* handshake type */
    uint24 length;              /* bytes in message */
    select (HandshakeType) {
        case client_hello:      ClientHello;
        case server_hello:      ServerHello;
        case hello_retry_request: HelloRetryRequest;
        case encrypted_extensions: EncryptedExtensions;
        case certificate_request: CertificateRequest;
        case certificate:        Certificate;
        case certificate_verify: CertificateVerify;
        case finished:           Finished;
        case new_session_ticket: NewSessionTicket;
        case key_update:         KeyUpdate;
    } body;
} Handshake;
```

A.3.1. Key Exchange Messages

```
struct {
    uint8 major;
    uint8 minor;
} ProtocolVersion;

struct {
    opaque random_bytes[32];
} Random;
```

```
uint8 CipherSuite[2];    /* Cryptographic suite selector */

struct {
    ProtocolVersion max_supported_version = { 3, 4 };    /* TLS v1.3 */
    Random random;
    opaque legacy_session_id<0..32>;
    CipherSuite cipher_suites<2..2^16-2>;
    opaque legacy_compression_methods<1..2^8-1>;
    Extension extensions<0..2^16-1>;
} ClientHello;

struct {
    ProtocolVersion version;
    Random random;
    CipherSuite cipher_suite;
    Extension extensions<0..2^16-1>;
} ServerHello;

struct {
    ProtocolVersion server_version;
    NamedGroup selected_group;
    Extension extensions<0..2^16-1>;
} HelloRetryRequest;

struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;

enum {
    supported_groups(10),
    signature_algorithms(13),
    key_share(40),
    pre_shared_key(41),
    early_data(42),
    cookie(44),
    (65535)
} ExtensionType;

struct {
    NamedGroup group;
    opaque key_exchange<1..2^16-1>;
} KeyShareEntry;

struct {
    select (role) {
        case client:
            KeyShareEntry client_shares<0..2^16-1>;
    }
}
```

```
        case server:
            KeyShareEntry server_share;
    }
} KeyShare;

enum { psk_ke(0), psk_dhe_ke(1), (255) } PskKeyExchangeModes;
enum { psk_auth(0), psk_sign_auth(1), (255) } PskAuthenticationModes;

opaque psk_identity<0..2^16-1>;

struct {
    PskKeMode ke_modes<1..255>;
    PskAuthMode auth_modes<1..255>;
    opaque identity<0..2^16-1>;
} PskIdentity;

struct {
    select (Role) {
        case client:
            psk_identity identities<2..2^16-1>;

        case server:
            uint16 selected_identity;
    }
} PreSharedKeyExtension;

struct {
    select (Role) {
        case client:
            uint32 obfuscated_ticket_age;

        case server:
            struct {};
    }
} EarlyDataIndication;
```

A.3.1.1. Cookie Extension

```
struct {
    opaque cookie<0..2^16-1>;
} Cookie;
```

A.3.1.2. Signature Algorithm Extension

```
enum {
    /* RSASSA-PKCS1-v1_5 algorithms */
    rsa_pkcs1_sha1 (0x0201),
    rsa_pkcs1_sha256 (0x0401),
    rsa_pkcs1_sha384 (0x0501),
    rsa_pkcs1_sha512 (0x0601),

    /* ECDSA algorithms */
    ecdsa_secp256r1_sha256 (0x0403),
    ecdsa_secp384r1_sha384 (0x0503),
    ecdsa_secp521r1_sha512 (0x0603),

    /* RSASSA-PSS algorithms */
    rsa_pss_sha256 (0x0700),
    rsa_pss_sha384 (0x0701),
    rsa_pss_sha512 (0x0702),

    /* EdDSA algorithms */
    ed25519 (0x0703),
    ed448 (0x0704),

    /* Reserved Code Points */
    dsa_sha1_RESERVED (0x0202),
    dsa_sha256_RESERVED (0x0402),
    dsa_sha384_RESERVED (0x0502),
    dsa_sha512_RESERVED (0x0602),
    ecdsa_sha1_RESERVED (0x0203),
    obsolete_RESERVED (0x0000..0x0200),
    obsolete_RESERVED (0x0204..0x0400),
    obsolete_RESERVED (0x0404..0x0500),
    obsolete_RESERVED (0x0504..0x0600),
    obsolete_RESERVED (0x0604..0x06FF),
    private_use (0xFE00..0xFFFF),
    (0xFFFF)
} SignatureScheme;

SignatureScheme supported_signature_algorithms<2..2^16-2>;
```

A.3.1.3. Supported Groups Extension

```
enum {
    /* Elliptic Curve Groups (ECDHE) */
    obsolete_RESERVED (1..22),
    secp256r1 (23), secp384r1 (24), secp521r1 (25),
    obsolete_RESERVED (26..28),
    x25519 (29), x448 (30),

    /* Finite Field Groups (DHE) */
    ffdhe2048 (256), ffdhe3072 (257), ffdhe4096 (258),
    ffdhe6144 (259), ffdhe8192 (260),

    /* Reserved Code Points */
    ffdhe_private_use (0x01FC..0x01FF),
    ecdhe_private_use (0xFE00..0xFEFF),
    obsolete_RESERVED (0xFF01..0xFF02),
    (0xFFFF)
} NamedGroup;

struct {
    NamedGroup named_group_list<1..2^16-1>;
} NamedGroupList;
```

Values within "obsolete_RESERVED" ranges were used in previous versions of TLS and MUST NOT be offered or negotiated by TLS 1.3 implementations. The obsolete curves have various known/theoretical weaknesses or have had very little usage, in some cases only due to unintentional server configuration issues. They are no longer considered appropriate for general use and should be assumed to be potentially unsafe. The set of curves specified here is sufficient for interoperability with all currently deployed and properly configured TLS implementations.

A.3.1.4. Deprecated Extensions

The following extensions are no longer applicable to TLS 1.3, although TLS 1.3 clients MAY send them if they are willing to negotiate them with prior versions of TLS. TLS 1.3 servers MUST ignore these extensions if they are negotiating TLS 1.3: truncated_hmac [RFC6066], srp [RFC5054], encrypt_then_mac [RFC7366], extended_master_secret [RFC7627], SessionTicket [RFC5077], and renegotiation_info [RFC5746].

A.3.2. Server Parameters Messages

```
struct {
    Extension extensions<0..2^16-1>;
} EncryptedExtensions;

opaque DistinguishedName<1..2^16-1>;

struct {
    opaque certificate_extension_oid<1..2^8-1>;
    opaque certificate_extension_values<0..2^16-1>;
} CertificateExtension;

struct {
    opaque certificate_request_context<0..2^8-1>;
    SignatureScheme
        supported_signature_algorithms<2..2^16-2>;
    DistinguishedName certificate_authorities<0..2^16-1>;
    CertificateExtension certificate_extensions<0..2^16-1>;
} CertificateRequest;
```

A.3.3. Authentication Messages

```
opaque ASN1Cert<1..2^24-1>;

struct {
    opaque certificate_request_context<0..2^8-1>;
    ASN1Cert certificate_list<0..2^24-1>;
} Certificate;

struct {
    SignatureScheme algorithm;
    opaque signature<0..2^16-1>;
} CertificateVerify;

struct {
    opaque verify_data[Hash.length];
} Finished;
```

A.3.4. Ticket Establishment

```

enum { (65535) } TicketExtensionType;

struct {
    TicketExtensionType extension_type;
    opaque extension_data<1..2^16-1>;
} TicketExtension;

struct {
    uint32 ticket_lifetime;
    PskKeMode ke_modes<1..255>;
    PskAuthMode auth_modes<1..255>;
    opaque ticket<1..2^16-1>;
    TicketExtension extensions<0..2^16-2>;
} NewSessionTicket;

```

A.4. Cipher Suites

A symmetric cipher suite defines the pair of the AEAD cipher and hash function to be used with HKDF. Cipher suites follow the naming convention: Cipher suite names follow the naming convention:

```
CipherSuite TLS13_CIPHER_HASH = VALUE;
```

Component	Contents
TLS	The string "TLS"
CIPHER	The symmetric cipher used for record protection
HASH	The hash algorithm used with HKDF
VALUE	The two byte ID assigned for this cipher suite

The "CIPHER" component commonly has sub-components used to designate the cipher name, bits, and mode, if applicable. For example, "AES_256_GCM" represents 256-bit AES in the GCM mode of operation.

Cipher Suite Name	Value	Specification
TLS_AES_128_GCM_SHA256	{0x13,0x01}	[This RFC]
TLS_AES_256_GCM_SHA384	{0x13,0x02}	[This RFC]
TLS_CHACHA20_POLY1305_SHA256	{0x13,0x03}	[This RFC]
TLS_AES_128_CCM_SHA256	{0x13,0x04}	[This RFC]
TLS_AES_128_CCM_8_SHA256	{0x13,0x05}	[This RFC]

Although TLS 1.3 uses the same cipher suite space as previous versions of TLS, TLS 1.3 cipher suites are defined differently, only specifying the symmetric ciphers, and cannot it be used for TLS 1.2. Similarly, TLS 1.2 and lower cipher suites cannot be used with TLS 1.3.

New cipher suite values are assigned by IANA as described in [Section 10](#).

A.4.1. Unauthenticated Operation

Previous versions of TLS offered explicitly unauthenticated cipher suites based on anonymous Diffie-Hellman. These cipher suites have been deprecated in TLS 1.3. However, it is still possible to negotiate cipher suites that do not provide verifiable server authentication by several methods, including:

- Raw public keys [[RFC7250](#)].
- Using a public key contained in a certificate but without validation of the certificate chain or any of its contents.

Either technique used alone is are vulnerable to man-in-the-middle attacks and therefore unsafe for general use. However, it is also possible to bind such connections to an external authentication mechanism via out-of-band validation of the server's public key, trust on first use, or channel bindings [[RFC5929](#)]. [[NOTE: TLS 1.3 needs a new channel binding definition that has not yet been defined.]] If no such mechanism is used, then the connection has no protection against active man-in-the-middle attack; applications **MUST NOT** use TLS in such a way absent explicit configuration or a specific application profile.

Appendix B. Implementation Notes

The TLS protocol cannot prevent many common security mistakes. This section provides several recommendations to assist implementors.

B.1. API considerations for 0-RTT

0-RTT data has very different security properties from data transmitted after a completed handshake: it can be replayed. Implementations SHOULD provide different functions for reading and writing 0-RTT data and data transmitted after the handshake, and SHOULD NOT automatically resend 0-RTT data if it is rejected by the server.

B.2. Random Number Generation and Seeding

TLS requires a cryptographically secure pseudorandom number generator (PRNG). In most cases, the operating system provides an appropriate facility such as `/dev/urandom`, which should be used absent other (performance) concerns. It is generally preferable to use an existing PRNG implementation in preference to crafting a new one, and many adequate cryptographic libraries are already available under favorable license terms. Should those prove unsatisfactory, [RFC4086] provides guidance on the generation of random values.

B.3. Certificates and Authentication

Implementations are responsible for verifying the integrity of certificates and should generally support certificate revocation messages. Certificates should always be verified to ensure proper signing by a trusted Certificate Authority (CA). The selection and addition of trusted CAs should be done very carefully. Users should be able to view information about the certificate and root CA.

B.4. Cipher Suite Support

TLS supports a range of key sizes and security levels, including some that provide no or minimal security. A proper implementation will probably not support many cipher suites. Applications SHOULD also enforce minimum and maximum key sizes. For example, certification paths containing keys or signatures weaker than 2048-bit RSA or 224-bit ECDSA are not appropriate for secure applications. See also [Appendix C.4](#).

B.5. Implementation Pitfalls

Implementation experience has shown that certain parts of earlier TLS specifications are not easy to understand, and have been a source of interoperability and security problems. Many of these areas have been clarified in this document, but this appendix contains a short list of the most important things that require special attention from implementors.

TLS protocol issues:

- Do you correctly handle handshake messages that are fragmented to multiple TLS records (see [Section 5.1](#))? Including corner cases like a ClientHello that is split to several small fragments? Do you fragment handshake messages that exceed the maximum fragment size? In particular, the certificate and certificate request handshake messages can be large enough to require fragmentation.
- Do you ignore the TLS record layer version number in all TLS records? (see [Appendix C](#))
- Have you ensured that all support for SSL, RC4, EXPORT ciphers, and MD5 (via the "signature_algorithm" extension) is completely removed from all possible configurations that support TLS 1.3 or later, and that attempts to use these obsolete capabilities fail correctly? (see [Appendix C](#))
- Do you handle TLS extensions in ClientHello correctly, including unknown extensions or omitting the extensions field completely?
- When the server has requested a client certificate, but no suitable certificate is available, do you correctly send an empty Certificate message, instead of omitting the whole message (see [Section 4.3.1.2](#))?
- When processing the plaintext fragment produced by AEAD-Decrypt and scanning from the end for the ContentType, do you avoid scanning past the start of the cleartext in the event that the peer has sent a malformed plaintext of all-zeros?
- When processing a ClientHello containing a version of { 3, 5 } or higher, do you respond with the highest common version of TLS rather than requiring an exact match? Have you ensured this continues to be true with arbitrarily higher version numbers? (e.g. { 4, 0 }, { 9, 9 }, { 255, 255 })

- Do you properly ignore unrecognized cipher suites ([Section 4.1.2](#)), hello extensions ([Section 4.2](#)), named groups ([Section 4.2.3](#)), and signature algorithms ([Section 4.2.2](#))?

Cryptographic details:

- What countermeasures do you use to prevent timing attacks against RSA signing operations [[TIMING](#)]?
- When verifying RSA signatures, do you accept both NULL and missing parameters? Do you verify that the RSA padding doesn't have additional data after the hash value? [[FI06](#)]
- When using Diffie-Hellman key exchange, do you correctly preserve leading zero bytes in the negotiated key (see [Section 7.3.1](#))?
- Does your TLS client check that the Diffie-Hellman parameters sent by the server are acceptable, (see [Section 4.2.4.1](#))?
- Do you use a strong and, most importantly, properly seeded random number generator (see [Appendix B.2](#)) when generating Diffie-Hellman private values, the ECDSA "k" parameter, and other security-critical values? It is RECOMMENDED that implementations implement "deterministic ECDSA" as specified in [[RFC6979](#)].
- Do you zero-pad Diffie-Hellman public key values to the group size (see [Section 4.2.4.1](#))?

B.6. Client Tracking Prevention

Clients SHOULD NOT reuse a session ticket for multiple connections. Reuse of a session ticket allows passive observers to correlate different connections. Servers that issue session tickets SHOULD offer at least as many session tickets as the number of connections that a client might use; for example, a web browser using HTTP/1.1 [[RFC7230](#)] might open six connections to a server. Servers SHOULD issue new session tickets with every connection. This ensures that clients are always able to use a new session ticket when creating a new connection.

[Appendix C](#). Backward Compatibility

The TLS protocol provides a built-in mechanism for version negotiation between endpoints potentially supporting different versions of TLS.

TLS 1.x and SSL 3.0 use compatible ClientHello messages. Servers can also handle clients trying to use future versions of TLS as long as

the ClientHello format remains compatible and the client supports the highest protocol version available in the server.

Prior versions of TLS used the record layer version number for various purposes. (TLSPlaintext.legacy_record_version & TLSCiphertext.legacy_record_version) As of TLS 1.3, this field is deprecated and its value MUST be ignored by all implementations. Version negotiation is performed using only the handshake versions. (ClientHello.max_supported_version & ServerHello.version) In order to maximize interoperability with older endpoints, implementations that negotiate the use of TLS 1.0-1.2 SHOULD set the record layer version number to the negotiated version for the ServerHello and all records thereafter.

For maximum compatibility with previously non-standard behavior and misconfigured deployments, all implementations SHOULD support validation of certification paths based on the expectations in this document, even when handling prior 'TLS versions' handshakes. (see [Section 4.3.1.1](#))

TLS 1.2 and prior supported an "Extended Master Secret" [[RFC7627](#)] extension which digested large parts of the handshake transcript into the master secret. Because TLS 1.3 always hashes in the transcript up to the server CertificateVerify, implementations which support both TLS 1.3 and earlier versions SHOULD indicate the use of the Extended Master Secret extension in their APIs whenever TLS 1.3 is used.

C.1. Negotiating with an older server

A TLS 1.3 client who wishes to negotiate with such older servers will send a normal TLS 1.3 ClientHello containing { 3, 4 } (TLS 1.3) in ClientHello.max_supported_version. If the server does not support this version it will respond with a ServerHello containing an older version number. If the client agrees to use this version, the negotiation will proceed as appropriate for the negotiated protocol. A client resuming a session SHOULD initiate the connection using the version that was previously negotiated.

Note that 0-RTT data is not compatible with older servers. See [Appendix C.3](#).

If the version chosen by the server is not supported by the client (or not acceptable), the client MUST send a "protocol_version" alert message and close the connection.

If a TLS server receives a ClientHello containing a version number greater than the highest version supported by the server, it **MUST** reply according to the highest version supported by the server.

Some legacy server implementations are known to not implement the TLS specification properly and might abort connections upon encountering TLS extensions or versions which it is not aware of. Interoperability with buggy servers is a complex topic beyond the scope of this document. Multiple connection attempts may be required in order to negotiate a backwards compatible connection, however this practice is vulnerable to downgrade attacks and is **NOT RECOMMENDED**.

C.2. Negotiating with an older client

A TLS server can also receive a ClientHello containing a version number smaller than the highest supported version. If the server wishes to negotiate with old clients, it will proceed as appropriate for the highest version supported by the server that is not greater than ClientHello.max_supported_version. For example, if the server supports TLS 1.0, 1.1, and 1.2, and max_supported_version is TLS 1.0, the server will proceed with a TLS 1.0 ServerHello. If the server only supports versions greater than max_supported_version, it **MUST** send a "protocol_version" alert message and close the connection.

Note that earlier versions of TLS did not clearly specify the record layer version number value in all cases (TLSPlaintext.legacy_record_version). Servers will receive various TLS 1.x versions in this field, however its value **MUST** always be ignored.

C.3. Zero-RTT backwards compatibility

0-RTT data is not compatible with older servers. An older server will respond to the ClientHello with an older ServerHello, but it will not correctly skip the 0-RTT data and fail to complete the handshake. This can cause issues when a client attempts to use 0-RTT, particularly against multi-server deployments. For example, a deployment could deploy TLS 1.3 gradually with some servers implementing TLS 1.3 and some implementing TLS 1.2, or a TLS 1.3 deployment could be downgraded to TLS 1.2.

A client that attempts to send 0-RTT data **MUST** fail a connection if it receives a ServerHello with TLS 1.2 or older. A client that attempts to repair this error **SHOULD NOT** send a TLS 1.2 ClientHello, but instead send a TLS 1.3 ClientHello without 0-RTT data.

To avoid this error condition, multi-server deployments SHOULD ensure a uniform and stable deployment of TLS 1.3 without 0-RTT prior to enabling 0-RTT.

C.4. Backwards Compatibility Security Restrictions

If an implementation negotiates use of TLS 1.2, then negotiation of cipher suites also supported by TLS 1.3 SHOULD be preferred, if available.

The security of RC4 cipher suites is considered insufficient for the reasons cited in [RFC7465]. Implementations MUST NOT offer or negotiate RC4 cipher suites for any version of TLS for any reason.

Old versions of TLS permitted the use of very low strength ciphers. Ciphers with a strength less than 112 bits MUST NOT be offered or negotiated for any version of TLS for any reason.

The security of SSL 2.0 [SSL2] is considered insufficient for the reasons enumerated in [RFC6176], and MUST NOT be negotiated for any reason.

Implementations MUST NOT send an SSL version 2.0 compatible CLIENT-HELLO. Implementations MUST NOT negotiate TLS 1.3 or later using an SSL version 2.0 compatible CLIENT-HELLO. Implementations are NOT RECOMMENDED to accept an SSL version 2.0 compatible CLIENT-HELLO in order to negotiate older versions of TLS.

Implementations MUST NOT send or accept any records with a version less than { 3, 0 }.

The security of SSL 3.0 [SSL3] is considered insufficient for the reasons enumerated in [RFC7568], and MUST NOT be negotiated for any reason.

Implementations MUST NOT send a ClientHello.max_supported_version or ServerHello.version set to { 3, 0 } or less. Any endpoint receiving a Hello message with ClientHello.max_supported_version or ServerHello.version set to { 3, 0 } MUST respond with a "protocol_version" alert message and close the connection.

Implementations MUST NOT use the Truncated HMAC extension, defined in Section 7 of [RFC6066], as it is not applicable to AEAD ciphers and has been shown to be insecure in some scenarios.

Appendix D. Overview of Security Properties

[[TODO: This section is still a WIP and needs a bunch more work.]]

A complete security analysis of TLS is outside the scope of this document. In this section, we provide an informal description the desired properties as well as references to more detailed work in the research literature which provides more formal definitions.

We cover properties of the handshake separately from those of the record layer.

D.1. Handshake

The TLS handshake is an Authenticated Key Exchange (AKE) protocol which is intended to provide both one-way authenticated (server-only) and mutually authenticated (client and server) functionality. At the completion of the handshake, each side outputs its view on the following values:

- A "session key" (the master secret) from which can be derived a set of working keys.
- A set of cryptographic parameters (algorithms, etc.)
- The identities of the communicating parties.

We assume that the attacker has complete control of the network in between the parties [RFC3552]. Even under these conditions, the handshake should provide the properties listed below. Note that these properties are not necessarily independent, but reflect the protocol consumers' needs.

Establishing the same session key. The handshake needs to output the same session key on both sides of the handshake, provided that it completes successfully on each endpoint (See [CK01]; defn 1, part 1).

Secrecy of the session key. The shared session key should be known only to the communicating parties, not to the attacker (See [CK01]; defn 1, part 2). Note that in a unilaterally authenticated connection, the attacker can establish its own session keys with the server, but those session keys are distinct from those established by the client.

Peer Authentication. The client's view of the peer identity should reflect the server's identity. If the client is authenticated,

the server's view of the peer identity should match the client's identity.

Uniqueness of the session key: Any two distinct handshakes should produce distinct, unrelated session keys

Downgrade protection. The cryptographic parameters should be the same on both sides and should be the same as if the peers had been communicating in the absence of an attack (See [BBFKZG16]; defns 8 and 9}).

Forward secret If the long-term keying material (in this case the signature keys in certificate-based authentication modes or the PSK in PSK-(EC)DHE modes) are compromised after the handshake is complete, this does not compromise the security of the session key (See [DOW92]).

Protection of endpoint identities. The server's identity (certificate) should be protected against passive attackers. The client's identity should be protected against both passive and active attackers.

Informally, the signature-based modes of TLS 1.3 provide for the establishment of a unique, secret, shared, key established by an (EC)DHE key exchange and authenticated by the server's signature over the handshake transcript, as well as tied to the server's identity by a MAC. If the client is authenticated by a certificate, it also signs over the handshake transcript and provides a MAC tied to both identities. [SIGMA] describes the analysis of this type of key exchange protocol. If fresh (EC)DHE keys are used for each connection, then the output keys are forward secret.

The PSK and resumption-PSK modes bootstrap from a long-term shared secret into a unique per-connection short-term session key. This secret may have been established in a previous handshake. If PSK-(EC)DHE modes are used, this session key will also be forward secret. The resumption-PSK mode has been designed so that the resumption master secret computed by connection N and needed to form connection N+1 is separate from the traffic keys used by connection N, thus providing forward secrecy between the connections.

For all handshake modes, the Finished MAC (and where present, the signature), prevents downgrade attacks. In addition, the use of certain bytes in the random nonces as described in Section 4.1.3 allows the detection of downgrade to previous TLS versions.

As soon as the client and the server have exchanged enough information to establish shared keys, the remainder of the handshake

is encrypted, thus providing protection against passive attackers. Because the server authenticates before the client, the client can ensure that it only reveals its identity to an authenticated server. Note that implementations must use the provided record padding mechanism during the handshake to avoid leaking information about the identities due to length.

The 0-RTT mode of operation generally provides the same security properties as 1-RTT data, with the two exceptions that the 0-RTT encryption keys do not provide full forward secrecy and that the server is not able to guarantee full uniqueness of the handshake (non-replayability) without keeping potentially undue amounts of state. See [Section 4.2.6](#) for one mechanism to limit the exposure to replay.

The reader should refer to the following references for analysis of the TLS handshake [[CHSV16](#)] [[FGSW16](#)] [[LXZFH16](#)].

D.2. Record Layer

The record layer depends on the handshake producing a strong session key which can be used to derive bidirectional traffic keys and nonces. Assuming that is true, and the keys are used for no more data than indicated in [Section 5.5](#) then the record layer should provide the following guarantees:

Confidentiality. An attacker should not be able to determine the plaintext contents of a given record.

Integrity. An attacker should not be able to craft a new record which is different from an existing record which will be accepted by the receiver.

Order protection/non-replayability An attacker should not be able to cause the receiver to accept a record which it has already accepted or cause the receiver to accept record N+1 without having first processed record N. [[TODO: If we merge in DTLS to this document, we will need to update this guarantee.]]

Length concealment. Given a record with a given external length, the attacker should not be able to determine the amount of the record that is content versus padding.

Forward security after key change. If the traffic key update mechanism described in [Section 4.4.3](#) has been used and the previous generation key is deleted, an attacker who compromises the endpoint should not be able to decrypt traffic encrypted with the old key.

Informally, TLS 1.3 provides these properties by AEAD-protecting the plaintext with a strong key. AEAD encryption [RFC5116] provides confidentiality and integrity for the data. Non-replayability is provided by using a separate nonce for each record, with the nonce being derived from the record sequence number (Section 5.3), with the sequence number being maintained independently at both sides thus records which are delivered out of order result in AEAD deprotection failures.

The plaintext protected by the AEAD function consists of content plus variable-length padding. Because the padding is also encrypted, the attacker cannot directly determine the length of the padding, but may be able to measure it indirectly by the use of timing channels exposed during record processing (i.e., seeing how long it takes to process a record). In general, it is not known how to remove this type of channel because even a constant time padding removal function will then feed the content into data-dependent functions.

Generation N+1 keys are derived from generation N keys via a key derivation function Section 7.2. As long as this function is truly one way, it is not possible to compute the previous keys after a key change (forward secrecy). However, TLS does not provide security for data which is sent after the traffic secret is compromised, even after a key update (backward secrecy); systems which want backward secrecy must do a fresh handshake and establish a new session key with an (EC)DHE exchange.

The reader should refer to the following references for analysis of the TLS record layer.

[Appendix E.](#) Working Group Information

The discussion list for the IETF TLS working group is located at the e-mail address tls@ietf.org [1]. Information on the group and information on how to subscribe to the list is at <https://www.ietf.org/mailman/listinfo/tls>

Archives of the list can be found at: <https://www.ietf.org/mail-archive/web/tls/current/index.html>

[Appendix F.](#) Contributors

- Martin Abadi
University of California, Santa Cruz
abadi@cs.ucsc.edu
- Christopher Allen (co-editor of TLS 1.0)
Alacrity Ventures

ChristopherA@AlacrityManagement.com

- Steven M. Bellovin
Columbia University
smb@cs.columbia.edu
- David Benjamin
Google
davidben@google.com
- Benjamin Beurdouche
- Karthikeyan Bhargavan (co-author of [RFC7627])
INRIA
karthikeyan.bhargavan@inria.fr
- Simon Blake-Wilson (co-author of [RFC4492])
BCI
sblakewilson@bcisse.com
- Nelson Bolyard (co-author of [RFC4492])
Sun Microsystems, Inc.
nelson@bolyard.com
- Ran Canetti
IBM
canetti@watson.ibm.com
- Pete Chown
Skygate Technology Ltd
pc@skygate.co.uk
- Antoine Delignat-Lavaud (co-author of [RFC7627])
INRIA
antoine.delignat-lavaud@inria.fr
- Tim Dierks (co-editor of TLS 1.0, 1.1, and 1.2)
Independent
tim@dierks.org
- Taher Elgamal
Securify
taher@securify.com
- Pasi Eronen
Nokia
pasi.eronen@nokia.com

- Cedric Fournet
Microsoft
fournet@microsoft.com
- Anil Gangolli
anil@busybuddha.org
- David M. Garrett
- Vipul Gupta (co-author of [\[RFC4492\]](#))
Sun Microsystems Laboratories
vipul.gupta@sun.com
- Chris Hawk (co-author of [\[RFC4492\]](#))
Corriente Networks LLC
chris@corriente.net
- Kipp Hickman
- Alfred Hoenes
- David Hopwood
Independent Consultant
david.hopwood@blueyonder.co.uk
- Subodh Iyengar
Facebook
subodh@fb.com
- Daniel Kahn Gillmor
ACLU
dkg@fifthhorseman.net
- Phil Karlton (co-author of SSL 3.0)
- Paul Kocher (co-author of SSL 3.0)
Cryptography Research
paul@cryptography.com
- Hugo Krawczyk
IBM
hugo@ee.technion.ac.il
- Adam Langley (co-author of [\[RFC7627\]](#))
Google
agl@google.com
- Ilari Liusvaara

- Independent
ilariliusvaara@welho.com
- Jan Mikkelsen
Transactionware
janm@transactionware.com
 - Bodo Moeller (co-author of [[RFC4492](#)])
Google
bodo@openssl.org
 - Erik Nygren
Akamai Technologies
erik+ietf@nygren.org
 - Magnus Nystrom
RSA Security
magnus@rsasecurity.com
 - Alfredo Pironti (co-author of [[RFC7627](#)])
INRIA
alfredo.pironti@inria.fr
 - Andrei Popov
Microsoft
andrei.popov@microsoft.com
 - Marsh Ray (co-author of [[RFC7627](#)])
Microsoft
maray@microsoft.com
 - Robert Relyea
Netscape Communications
relyea@netscape.com
 - Kyle Rose
Akamai Technologies
krose@krose.org
 - Jim Roskind
Netscape Communications
jar@netscape.com
 - Michael Sabin
 - Dan Simon
Microsoft, Inc.
dansimon@microsoft.com

- Nick Sullivan
CloudFlare Inc.
nick@cloudflare.com
- Bjoern Tackmann
University of California, San Diego
btackmann@eng.ucsd.edu
- Martin Thomson
Mozilla
mt@mozilla.com
- Filippo Valsorda
CloudFlare Inc.
filippo@cloudflare.com
- Tom Weinstein
- Hoeteck Wee
Ecole Normale Superieure, Paris
hoeteck@alum.mit.edu
- Tim Wright
Vodafone
timothy.wright@vodafone.com

Author's Address

Eric Rescorla
RTFM, Inc.

E-Mail: ekr@rtfm.com