TLS 1.3 for engineers: An exploration of the TLS 1.3 specification and Oracle’s Java implementation

Ben Smyth
Crypto Stream Ltd., UK

January 22, 2019
Copyright and licensing

The listings in this manuscript are subject to the following copyright and licensing notice:

Copyright (c) 1996, 2003, 2015, 2018, Oracle and/or its affiliates. All rights reserved. This code is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License version 2 only, as published by the Free Software Foundation. Oracle designates this particular file as subject to the “Classpath” exception as provided by Oracle in the LICENSE file that accompanied this code.

This code is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License version 2 for more details (a copy is included in the LICENSE file that accompanied this code).

You should have received a copy of the GNU General Public License version 2 along with this work; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA.

Please contact Oracle, 500 Oracle Parkway, Redwood Shores, CA 94065 USA or visit www.oracle.com if you need additional information or have any questions.

The LICENSE file is available from the following URL: http://hg.openjdk.java.net/jdk/jdk11/file/1ddf9a99e4ad/LICENSE
Contents

1 Introduction 5

2 Handshake protocol 6
  2.1 ClientHello .............................................................. 7
  2.2 ServerHello ............................................................... 11
    2.2.1 HelloRetryRequest .................................................. 19
  2.3 Key establishment ....................................................... 22
    2.3.1 Transcript hash ..................................................... 22
    2.3.2 Key derivation ....................................................... 25
    2.3.3 Traffic keys .......................................................... 29
  2.4 Server parameters: EncryptedExtensions ............................ 37
  2.5 Authentication ........................................................... 40
    2.5.1 Certificate and CertificateVerify .................................. 40
    2.5.2 Finished ............................................................... 41
  2.6 Further features ........................................................ 48
    2.6.1 NewSessionTicket ................................................... 48
    2.6.2 KeyUpdate ........................................................... 57
    2.6.3 Early data ............................................................ 58

3 Record protocol 58
  3.1 TLSPlaintext .............................................................. 58
  3.2 TLSCiphertext ............................................................ 58

A Extensions 59

B Alert protocol 62

C Client authentication: CertificateRequest message 63
## Listings

|   | Description                                                                                           | Page |
|---|--------------------------------------------------------------------------------------------------------|------|
| 1 | ClientHello.ClientHelloMessage defines ClientHello                                                  | 8    |
| 2 | ClientHello.ClientHelloKickstartProducer produces ClientHello                                       | 9    |
| 3 | ClientHello.T13ClientHelloConsumer consumes generic ClientHello                                     | 12   |
| 4 | ClientHello.T13ClientHelloConsumer consumes ClientHello (cont.)                                     | 13   |
| 5 | ServerHello.ServerHelloMessage defines ServerHello/HelloRetryRequest                               | 15   |
| 6 | ServerHello.ServerHelloMessage defines ServerHello/HelloRetryRequest (cont.)                        | 16   |
| 7 | ServerHello.T13ServerHelloProducer produces ServerHello                                              | 17   |
| 8 | ServerHello.T13ServerHelloProducer produces ServerHello (cont.)                                     | 18   |
| 9 | ServerHello.T13ServerHelloConsumer consumes generic ServerHello/HelloRetryRequest                  | 20   |
| 10| ServerHello.T13ServerHelloConsumer consumes ServerHello/HelloRetryRequest (cont.)                  | 21   |
| 11| ServerHello.T13HelloRetryRequestProducer produces HelloRetryRequest                                | 23   |
| 12| ServerHello.T13HelloRetryRequestProducer produces HelloRetryRequest (cont.)                        | 24   |
| 13| ServerHello.ServerHelloConsumer consumes HelloRetryRequest                                         | 24   |
| 14| HandshakeHash supports transcript hashes                                                             | 26   |
| 15| HelloCookieManager.T13HelloCookieManager modifies transcript hashes                                 | 27   |
| 16| HKDF implements function HKDF-Extract                                                               | 31   |
| 17| SSLSecretDerivation implements function Derive-Secret                                                | 32   |
| 18| SSLSecretDerivation.SecretSchedule implements function Derive-Secret (cont.)                       | 33   |
| 19| ServerHello.setUpPskKD derives Early Secret over a pre-shared key                                   | 33   |
| 20| HKDF implements function HKDF-Expand                                                                 | 35   |
| 21| SSLTrafficKeyDerivation.T13TrafficKeyDerivation derives traffic keys                                | 36   |
| 22| ServerHello.T13ServerHelloProducer deriving keys                                                   | 38   |
| 23| ServerHello.T13ServerHelloProducer deriving keys (cont.)                                            | 39   |
| 24| CertificateMessage.T13CertificateMessage defines Certificate                                        | 41   |
| 25| CertificateMessage.T13CertificateProducer produces Certificate                                     | 42   |
| 26| CertificateMessage.T13CertificateProducer produces Certificate (cont.)                             | 43   |
| 27| CertificateMessage.T13CertificateConsumer consumes Certificate                                      | 44   |
| 28| CertificateVerify .T13CertificateVerifyMessage defines CertificateVerify                              | 45   |
| 29| CertificateVerify .T13CertificateVerifyMessage defines CertificateVerify (cont.)                   | 46   |
| 30| CertificateVerify .T13CertificateVerifyProducer produces CertificateVerify                           | 47   |
| 31| CertificateVerify .T13CertificateVerifyConsumer consumes CertificateVerify                           | 47   |
| 32| Finished.FinishedMessage defines Finished                                                            | 49   |
| 33| Finished.T13VerifyDataGenerator defines Finished (cont.)                                             | 50   |
| 34| Finished.T13FinishedProducer produces server-side Finished                                           | 51   |
| 35| Finished.T13FinishedProducer produces server-side Finished (cont.)                                 | 52   |
| 36| Finished.T13FinishedConsumer consumes server-generated Finished                                     | 53   |
| 37| Finished.T13FinishedConsumer consumes server-generated Finished (cont.)                            | 54   |
| 38| Finished.T13FinishedProducer produces client-side Finished                                           | 55   |
| 39| Finished.T13FinishedConsumer consumes client-generated Finished                                     | 56   |
| 40| Finished.T13FinishedConsumer consumes client-generated Finished (cont.)                            | 58   |
| 41| SSLSocketOutputRecord writes outgoing records                                                        | 60   |
| 42| OutputRecord writes outgoing records                                                                 | 61   |
| 43| Enum SSLExtension                                                                                   | 64   |
| 44| SSLExtensions                                                                                       | 65   |
| 45| SSLExtensions (cont.)                                                                               | 66   |
1 Introduction

The Internet delivered in excess of forty terabytes per second in 2017 (Cisco, 2018), and over half of today’s Internet traffic is encrypted (Sandvine, 2018); enabling trade worth trillions of dollars (Statista, 2017). Yet, the underlying encryption technology is only understood by a select few. This manuscript broadens understanding by exploring TLS, an encryption technology used to protect application layer communication (including HTTP, FTP and SMTP traffic), and by examining Oracle’s Java implementation. We focus on the most recent TLS release, namely, version 1.3, which is defined by RFC 8446.

TLS is a protocol that establishes a channel between an initiating client and a interlocutory server (also known as endpoints and peers), for the purpose of enabling:

Authentication. An endpoint’s belief of their peer’s identity is correct.

Confidentiality. Communication over an established channel is only visible to endpoints.

Integrity. Communication over an established channel is received-as-sent, or tampering is detected.

These properties should hold even in the presence of an adversary that has complete control of the underlying network, i.e., an adversary that may read, modify, drop, and inject messages.

The TLS protocol commences with a handshake, wherein cryptographic primitives and parameters are negotiated, and shared (traffic) keys are established. Moreover, the handshake includes unilateral authentication of the server. (Mutual authentication of both the client and the server is also possible.) The handshake results in a channel which uses the negotiated cryptography and parameters, along with a shared key, to protect communication.

The handshake does not require any prior knowledge: A shared key may be derived from secrets established using Diffie-Hellman key exchange over finite fields (DHE) or elliptic curves (ECDHE). Alternatively, such a shared key may be derived from a secret pre-shared key (PSK), which endpoints establish out-of-band or during a previous connection. (Shared keys are combined with nonces to ensure they are always unique, regardless of whether secrets have been previously used.) The former achieves forward secrecy – i.e., confidentiality is preserved even if long-term keying material is compromised after the handshake, as long as (EC)DHE secrets are erased – whereas the latter does not. The two key exchange modes can be combined, using PSK with (EC)DHE key exchange, to achieve forward secrecy with pre-shared keys.

The handshake is itself a protocol (summarised in Figure 1). It is commenced by the client sending a ClientHello message, comprising: a nonce; offered protocol versions, symmetric ciphers, and hash functions; offered Diffie-Hellman key shares, pre-shared key labels, or both; and details of any extended functionality. The protocol proceeds with the server receiving the client’s message, establishing mutually acceptable cryptographic primitives and parameters, and responding with a ServerHello message, containing: a nonce; selected protocol version, symmetric cipher, and hash function; and a Diffie-Hellman key share, a selected pre-shared key label, or both. (The server may respond with a HelloRetryRequest message, if the offered key shares are unsuitable.) Once the client receives the server’s message, a shared (handshake traffic) key can be established to enable confidentiality and integrity for the remainder of the handshake protocol. In particular, that shared key is used to protect an EncryptedExtensions message, sent by the server to the client, which may detail extended functionality.

The handshake protocol concludes with unilateral authentication of the server. (Client authentication is also possible.) For (EC)DHE-only key exchange, after sending the EncryptedExtensions message, the server sends a Certificate message, containing a certificate (or some other suitable material corresponding to the server’s long-term, private key), and a CertificateVerify message, containing a signature (using the private key corresponding to the public key in the certificate) over a hash of the handshake protocol’s transcript (i.e., a concatenation of each handshake message, e.g., ClientHello, ServerHello, EncryptedExtensions, and Certificate, in
2 HANDSHAKE PROTOCOL

Figure 1: A client initiates the handshake protocol by sending a ClientHello (CH) message. After sending that message, the client waits for a ServerHello (SH) message followed by an EncryptedExtensions (EE) message, or a HelloRetryRequest (HRR) message. An EncryptedExtensions message might be followed by a CertificateRequest (CR) message (requesting client authentication). Moreover, for certificate-based server authentication, the client waits for a Certificate (CT) message followed by a CertificateVerify (CV) message. The handshake protocol concludes upon an exchange of Finished (FIN) messages from each of the client and server. (We omit the client’s Finished message for brevity.) The client’s Finished message may be preceded by client generated Certificate and CertificateVerify messages, when client authentication is requested. (We omit those messages for brevity.)

Beyond the handshake protocol, TLS defines a record protocol which writes handshake protocol messages (and application data, as well as error messages) to the transport layer, after adding headers and, where necessary, protecting messages.

Contribution and structure. We explore the TLS handshake (§2) and record (§3) protocols, as defined by RFC 8446[1] moreover, we examine Oracle’s Java implementation[2], namely, JDK 11 package sun.security.ssl.

2 Handshake protocol

A client initiates the handshake protocol from an initial context detailing the client’s expectations, e.g., willingness to use particular cryptographic primitives and parameters. A server participates with a similar initial context. Those contexts evolve during the handshake protocol, to reach agreement on cryptographic primitives and parameters, along with shared keys. (The protocol may abort if the endpoints cannot reach agreement.)

Client and server contexts are implemented by classes ClientHandshakeContext and ServerHandshakeContext, respectively, that share parent HandshakeContext (which implements empty interface ConnectionContext). Those classes are both parameterised by instances of classes SSLContextImpl (with parent SSLContext) and TransportContext (which also implements empty interface ConnectionContext), which define initial contexts.

[1] https://tools.ietf.org/html/rfc8446
[2] http://hg.openjdk.java.net/jdk/jdk11/file/1ddf9a99e4ad/
2 HANDSHAKE PROTOCOL

Table 1: Symmetric cipher suites defined by a value identifying an AEAD algorithm and a hash function. Suites are named in the format TLS_AEAD_HASH, where AEAD and HASH are replaced by the corresponding algorithm and function names.

| Name                                          | Value |
|-----------------------------------------------|-------|
| TLS_AES_128_GCM_SHA256                        | 0x1301|
| TLS_AES_256_GCM_SHA384                        | 0x1302|
| TLS_CHACHA20_POLY1305_SHA256                  | 0x1303|
| TLS_AES_128_CCM_SHA256                        | 0x1304|
| TLS_AES_128_CCM_8_SHA256                      | 0x1305|

2.1 ClientHello

The handshake protocol is initiated by a ClientHello message, comprising the following fields:

- **legacy_version**: Constant 0x0303. (Previous versions of TLS used this field for the client’s highest offered protocol version. In TLS 1.3, protocol versions are offered in an extension, as explained below.)

- **random**: A 32 byte nonce.

- **legacy_session_id**: A zero-length vector, except to resume an earlier pre-TLS 1.3 session or for “compatibility mode.” (Previous versions of TLS used this field for “session resumption.” In TLS 1.3, that feature has been merged with pre-shared keys.)

- **cipher_suites**: A list of offered symmetric cipher suites in descending order of client preference, where a suite defines a value identifying an Authenticated Encryption with Associated Data (AEAD) algorithm and a hash function (Table 1).

- **legacy_compression_methods**: Constant 0x00. (Previous versions of TLS used this field to list supported compression methods. In TLS 1.3, this feature has been removed.)

- **extensions**: A list of extensions, where an extension comprises a name along with associated data. The list must contain at least extension supported_versions associated with a list of offered protocol versions in descending order of client preference, minimally including constant 0x0304, denoting TLS 1.3.

Legacy fields legacy_version, legacy_session_id, and legacy_compression_methods are included for backwards compatibility.

The ClientHello message is implemented by class ClientHello.ClientHelloMessage (Listing 1). Instances of that class are produced by class ClientHello.ClientHelloKickstartProducer (Listing 2), which is instantiated as static constant ClientHello.kickstartProducer. That constant is used by method SSLHandshake.kickstart.

The primary goal of the handshake protocol is to establish a channel that protects communication using one of the symmetric cipher suites offered by the client and a key shared between the endpoints. That key is derived from (secret) client and server key shares for (EC)DHE key exchange, from a (secret) pre-shared key for PSK-only key exchange, or by a combination of key shares and a pre-shared key for PSK with (EC)DHE key exchange. The desired key exchange mode determines which extensions to include: For (EC)DHE, extensions supported_groups and key_share are included; for PSK-only, extensions pre_shared_key and psk_key_exchange_modes must be included, and extensions supported_groups and key_share may be included to allow the server to decline resumption and fall back to a full handshake; and for PSK with (EC)DHE, all four of the aforementioned extensions are included. Those extensions are associated with data:
2 HANDSHAKE PROTOCOL

Listing 1: Class ClientHello.ClientHelloMessage defines the six fields of a ClientHello message (Lines 74–81) and constructors to instantiate them from parameters (Lines 85–106) or an input buffer (Lines 160–200). The former constructor does not populate the extensions field (and a call to method SSLExtensions.produce, Listing 46, is required), whereas the latter may (Line 195–196). Method send (Lines 312–316) writes those fields to an output stream, using method sendCore (Lines 318–328) to write all fields except the extensions field, which is written by method SSLExtensions.send (Listing 46, Lines 293–307).
static final SSLProducer kickstartProducer =
    new ClientHelloKickstartProducer();

private static final
class ClientHelloKickstartProducer implements SSLProducer {
    public byte[] produce(ConnectionContext context) throws IOException {
        ClientHandshakeContext chc = (ClientHandshakeContext)context;
        SessionId sessionId = SSLSessionImpl.nullSession.getSessionId();
        List<CipherSuite> cipherSuites = chc.activeCipherSuites;
        SSLSessionContextImpl ssci = (SSLSessionContextImpl)
            chc.sslContext.engineGetClientSessionContext();
        SSLSessionImpl session = ssci.get(
            chc.conContext.transport.getPeerHost(),
            chc.conContext.transport.getPeerPort());
        CipherSuite sessionSuite = null;
        if (session != null) {
            sessionSuite = session.getSuite();
            cipherSuites = Arrays.asList(sessionSuite);
            chc.isResumption = true;
            chc.resumingSession = session;
        }
        clientHelloVersion = ProtocolVersion.TLS12;
        ClientHelloMessage chm = new ClientHelloMessage(chc,
            clientHelloVersion.id, sessionId, cipherSuites,
            chc.sslContext.getSecureRandom());
        // cache the client random number for further using
        chc.clientHelloRandom = chm.clientRandom;
        chc.clientHelloVersion = clientHelloVersion.id;
        // Produce extensions for ClientHello handshake message.
        SSLExtension[] extTypes = chc.sslConfig.getEnabledExtensions(
            SSLHandshake.CLIENT_HELLO, chc.activeProtocols);
        chm.extensions.produce(chc, extTypes);
        // Output the handshake message.
        chm.write(chc.handshakeOutput);
        chc.handshakeOutput.flush();
        // Reserve the initial ClientHello message for the follow on
        // cookie exchange if needed.
        chc.initialClientHelloMsg = chm;
        // What’s the expected response?
        chc.handshakeConsumers.put(
            SSLHandshake.SERVER_HELLO.id, SSLHandshake.SERVER_HELLO);
        // The handshake message has been delivered.
        return null;
    }
}

Listing 2: Class ClientHello.ClientHelloKickstartProducer defines method produce which instantiates a ClientHello message (Lines 619–621), populates the extension field for the active context (Lines 628–630), writes the ClientHello message to an output stream (Lines 637–638), and prepares the client’s active context for the server’s response (Lines 624–625, 642 & 645–646). The ClientHello message parametrises legacy_session_id as a zero-length byte array (Line 407); cipher_suites as the list of available cipher suites, for (EC)DHE-only key exchange (Line 410), or as a list containing the cipher suite used by the previous session, for PSK-based key exchange (Line 542); and legacy_version as constant 0x0303 (Line 615). (Prior versions of TLS are supported by the class and constants other than 0x0303 may be assigned to legacy_version. We omit those details for brevity.) The output stream is written-to using method ClientHello.ClientHelloMessage.write, defined by parent class SSLHandshake.HandshakeMessage, which in turn uses method ClientHello.ClientHelloMessage.send (Listing 1).
supported_groups and key_share: A list of offered Diffie-Hellman groups for key exchange (supported_groups) and key shares for some or all of those groups (key_share), in descending order of client preference. Groups may be selected over finite fields or elliptic curves. A key share for a particular group must be listed in the same order that the group is listed. However, a key share for a particular group may be omitted, even when a key share for a less preferred group is present. This situation could arise when a group is new or lacking support, making key shares for such groups redundant and wasteful. An empty vector of key shares can be used to request group selection from the server. (Servers respond with HelloRetryRequest messages when no key share is offered for the server selected group.)

pre_shared_key and psk_key_exchange_modes: A list of offered pre-shared key identifiers (pre_shared_key) and a key exchange mode for each (psk_key_exchange_modes). (Further details on extension pre_shared_key appear in Section 2.6.1 after NewSessionTicket messages – which establish pre-shared keys for subsequent connections – are introduced.) Key exchange modes include PSK-only (psk_ke) and PSK with (EC)DHE (psk_dhe_ke). Extension pre_shared_key must be the last extension in the ClientHello message. (Other extensions may appear in any order.)

A further goal of the handshake protocol is unilateral authentication of the server, which for (EC)DHE key exchange mode is achieved by inclusion of extensions signature_algorithms and signature_algorithms_cert (for PSK-only and PSK with (EC)DHE, authentication is derived from the Finished message), and associated data:

signature_algorithms and signature_algorithms_cert: A list of accepted signature algorithms in descending order of client preference for CertificateVerify messages (signature_algorithms) and Certificate messages (signature_algorithms_cert). (Extension signature_algorithms_cert may be omitted in favour of extension signature_algorithms, when accepted algorithms for Certificate and CertificateVerify messages coincide. In such cases, algorithms listed by extension signature_algorithms apply to certificates too.)

Additional extensions exist and may be included in ClientHello messages. (Appendix A lists all extensions.)

A ClientHello message is consumed by the server: The server first checks that the message is a TLS 1.3 ClientHello message, which is achieved by checking that extension supported_versions is present and that constant 0x0304 is the first listed preference. (The ClientHello message format is backward compatible with previous versions of TLS, hence, the message might need to be processed by a prior version of TLS. Those details are beyond the scope of this manuscript.) The server may also check that field legacy_version is set to constant 0x0303 and field legacy_session_id is set to a zero-length vector. Moreover, the server checks field legacy_compression_methods is set to constant 0x00 and aborts with an illegal_parameter alert if the check fails. Next, the server selects an acceptable cipher suite from field cipher_suites, disregarding suites that are not recognised, unsupported, or otherwise unacceptable, and aborting with a handshake_failure or an insufficient_security alert if no mutually acceptable cipher suite exists. Finally, the server processes any remaining extensions:

supported_groups and key_share: The server selects an acceptable group from the list; aborting with a missing_extension alert if extension supported_groups is present and extension key_share is absent, or vice versa; aborting with a handshake_failure or an insufficient_security alert if extension supported_groups is absent and extension key_share is present.
alert if no mutually acceptable group exists; and responding with a HelloRetryRequest message if extension key_share does not offer a key share for the selected group.

**pre_shared_key and psk_key_exchange_modes**: The server selects an acceptable key identifier from the list, disregarding unknown identifiers, aborting with an illegal_parameter alert if extension pre_shared_key is not the last extension in the ClientHello message, and aborting if extension pre_shared_key is present without psk_key_exchange_modes. The server also selects a key exchange mode. If no mutually acceptable key identifier exists and extensions supported_groups and key_share are present, then the server should perform a non-PSK handshake.

**signature_algorithms and signature_algorithms_cert**: The server selects acceptable signature algorithms for CertificateVerify and Certificate messages.

Any unrecognised extensions are ignored and the server aborts with a missing_extension alert if extension pre_shared_key is absent as-is either extension supported_groups, signature_algorithms, or both. (Alerts are formally defined by the TLS specification, as discussed in Appendix B.)

Consumption is implemented by class ClientHello.ClientHelloConsumer (Listing 3). That class checks the presence of extension supported_versions, to determine whether the message is a TLS 1.3 ClientHello message, and the remainder of the message is processed by class ClientHello.T13ClientHelloConsumer (Listings 4 & 5), if it is a TLS 1.3 message. Consumption of the ClientHello message may result in the server aborting or responding with either a ServerHello or HelloRetryRequest message.

### 2.2 ServerHello

A server that is able to successfully consume a ClientHello message responds with a ServerHello message, comprising fields legacy_version, random, and extensions as per the ClientHello message and the following fields:

- **legacy_session_id_echo**: The contents of ClientHello.legacy_session_id.
- **cipher_suite**: The cipher suite selected by the server from ClientHello.cipher_suites.
- **legacy_compression_method**: Constant 0x00.

Legacy fields are included for backwards compatibility.

The ServerHello message is implemented by class ServerHello.ServerHelloMessage (Listings 6 & 7). Instances of that class are produced by class ServerHello.T13ServerHelloProducer (Listings 8 & 9), which is instantiated as static constant ServerHello.t13HandshakeProducer. That constant is used indirectly – via class SSL-Handshake.SERVER_HELLO – to produce ServerHello messages in class ClientHello.T13ClientHelloConsumer (Listing 5).

In addition to mandatory extension supported_versions, message ServerHello must include additional extensions depending on the key exchange mode: For ECDHE/DHE, extension key_share is included in association with the server’s key share, which must be in the group selected by the server from ClientHello.supported_groups; for PSK-only, extension pre_shared_key is included in association with the pre-shared key identifier selected by the server.
static final SSLConsumer handshakeConsumer =
    new ClientHelloConsumer();

private static final class ClientHelloConsumer implements SSLConsumer {
    public void consume(ConnectionContext context,
        ByteBuffer message) throws IOException {
        ServerHandshakeContext shc = (ServerHandshakeContext)context;
        SSLExtension[] enabledExtensions =
            shc.sslConfig.getEnabledExtensions(
                SSLHandshake.CLIENT_HELLO);
        ClientHelloMessage chm =
            new ClientHelloMessage(shc, message, enabledExtensions);
        shc.clientHelloVersion = chm.clientVersion;
        onClientHello(shc, chm);
    }

    private void onClientHello(ServerHandshakeContext context,
        ClientHelloMessage clientHello) throws IOException {
        SSLExtension[] extTypes =
            new SSLExtension[] {
                SSLExtension.CH_SUPPORTED_VERSIONS
            };
        clientHello.extensions.consumeOnLoad(context, extTypes);
        ProtocolVersion negotiatedProtocol;
        CHSupportedVersionsSpec svs =
            (CHSupportedVersionsSpec)context.handshakeExtensions.get(
                SSLExtension.CH_SUPPORTED_VERSIONS);
        if (svs != null) {
            negotiatedProtocol =
                negotiateProtocol(context, svs.requestedProtocols);
        } else {
            negotiatedProtocol =
                negotiateProtocol(context, clientHello.clientVersion);
        }
        context.negotiatedProtocol = negotiatedProtocol;
        if (negotiatedProtocol.useTLS13PlusSpec()) {
            t13HandshakeConsumer.consume(context, clientHello);
        } else {
            t12HandshakeConsumer.consume(context, clientHello);
        }
    }

    private ProtocolVersion negotiateProtocol(
        ServerHandshakeContext context,
        int[] clientSupportedVersions) throws SSLEception {
        // The client supported protocol versions are present in client
        // preference order. This implementation chooses to use the server
        // preference of protocol versions instead.
        for (ProtocolVersion spv : context.activeProtocols) {
            for (int cpv : clientSupportedVersions) {
                if (spv.id == cpv) {
                    return spv;
                }
            }
        }
    }
}

Listing 3: Class ClientHello.ClientHelloConsumer defines method consume to instantiate a (generic) ClientHello message from an input buffer (Lines 785–786); update the server’s active context to include the client’s offered versions (Lines 800–803), indirectly using method SupportedVersionsExtension.CHSupportedVersionsConsumer.consume, which calls context.handshakeExtensions.put(CH_SUPPORTED_VERSIONS, spec), where parameter spec is a byte array encoding of extension supported_versions; select the first server preferred version that the client offered (Lines 810–811 & 880–892); and update the active context to include that selected version preference as the negotiated protocol (Line 816). Further processing is deferred (Line 831) to class ClientHello.T13ClientHelloConsumer (Listing 4).
```java
private static final HandshakeConsumer t13HandshakeConsumer =
    new T13ClientHelloConsumer();

private static final class T13ClientHelloConsumer implements HandshakeConsumer {
    public void consume(ConnectionContext context ,
        HandshakeMessage message) throws IOException {
        ServerHandshakeContext shc = (ServerHandshakeContext)context ;
        ClientHelloMessage clientHello = (ClientHelloMessage)message ;
        // Check and launch the "psk_key_exchange_modes" and
        // "pre_shared_key" extensions first, which will reset the
        // resuming session, no matter the extensions present or not.
        shc.isResumption = true ;
        SSLExtension[] extTypes = new SSLExtension[] {;
            SSLExtension.PSK_KEY.Exchange_Modes,
            SSLExtension.CH_PRE_SHARED_KEY
        };
        clientHello.extensions.consumeOnLoad(shc , extTypes );
        // Check and launch ClientHello extensions other than
        // "psk_key_exchange_modes" , "pre_shared_key" , "protocol_version"
        // and "key_share" extensions .
        extTypes = shc.sslConfig.getExclusiveExtensions(
            SSLHandshake.CLIENT_HELLO, Arrays.asList(
                SSLExtension.PSK_KEY.Exchange_Modes,
                SSLExtension.CH_PRE_SHARED_KEY,
                SSLExtension.CH_SUPPORTED_VERSIONS));
        clientHello.extensions.consumeOnLoad(shc , extTypes );
        if (!shc.handshakeProducers.isEmpty()) {
            // Should be HelloRetryRequest producer.
            goHelloRetryRequest(shc , clientHello );
        } else {
            goServerHello(shc , clientHello );
        }
    }
```

Listing 4: Class ClientHello.T13ClientHelloConsumer defines method consume to process incoming (TLS 1.3) ClientHello messages (further to processing shown in Listing 3). The method updates the server's active context to include any pre-shared key identifiers and key exchange modes offered by the client (Lines 1101–1105), indirectly using the consume method of classes PskKeyExchangeModesExtension, PskKeyExchangeModesConsumer and PreSharedKeyExtension.CHPreSharedKeyConsumer; updates the active context to include any further (enabled) extensions (Lines 1113–1119), excluding those that have already been added to the active context, namely, extensions supported_versions, pre_shared_key, and psk_key_exchange_modes; and proceeds by producing either a HelloRetryRequest message if extension key_share does not offer a key share for the server selected group (method KeyShareExtension.CHKeyShareConsumer.consume may add a producer for HelloRetryRequest messages which ensures !shc.handshakeProducers.isEmpty() holds) or a ServerHello message otherwise (Lines 1121–1126).
private void goHelloRetryRequest(ServerHandshakeContext shc, 
ClientHelloMessage clientHello) throws IOException {
    HandshakeProducer handshakeProducer = 
        shc.handshakeProducers.remove(SSLHandshake.HELLO_RETRY_REQUEST.id);
    handshakeProducer.produce(shc, clientHello);
}

private void goServerHello(ServerHandshakeContext shc, 
ClientHelloMessage clientHello) throws IOException {
    shc.clientHelloRandom = clientHello.clientRandom;
    if (!shc.conContext.isNegotiated) {
        shc.conContext.protocolVersion = shc.negotiatedProtocol;
        shc.conContext.outputRecord.setVersion(shc.negotiatedProtocol);
    }

    // update the responders
    // Only ServerHello/HelloRetryRequest producer, which adds
    // more responders later.
    shc.handshakeProducers.put(SSLHandshake.SERVER_HELLO.id, 
        SSLHandshake.SERVER_HELLO);
    SSLHandshake[] probableHandshakeMessages = new SSLHandshake[] {
        SSLHandshake.SERVER_HELLO,
        SSLHandshake.ENCRYPTED_EXTENSIONS,
        SSLHandshake.CERTIFICATE_REQUEST,
        SSLHandshake.CERTIFICATE,
        SSLHandshake.CERTIFICATE_VERIFY,
        SSLHandshake.FINISHED
    };
    for (SSLHandshake hs : probableHandshakeMessages) {
        HandshakeProducer handshakeProducer = 
            shc.handshakeProducers.remove(hs.id);
        if (handshakeProducer != null) {
            handshakeProducer.produce(shc, clientHello);
        }
    }
}

Listing 5: Class ClientHello.T13ClientHelloConsumer (continued from Listing 4) defines methods goHelloRetryRequest to produce a HelloRetryRequest message and goServerHello to produce a ServerHello message. The latter method prepares the server’s active context for the client’s response (Lines 1154 & 1159–1162); updates the active context to include a producer for ServerHello messages (Lines 1168–1169); constructs an array of producers that servers might use during the handshake protocol, namely, producers for messages ServerHello, EncryptedExtensions, CertificateRequest, Certificate, CertificateVerify, and Finished, in the order that they might be used (Lines 1171–1180); and uses those producers to produce messages when the active context includes the producer (Lines 1185–1191). Since a ServerHello message producer is included, a ServerHello message is always produced, using method ServerHello.T13ServerHelloProducer.produce (Listing 8). That method adds producers for EncryptedExtensions and Finished messages (Listing 8, Lines 560–563), since those messages must be sent. Other producers may also be added.
static final class ServerHelloMessage extends HandshakeMessage {
    final ProtocolVersion serverVersion; // TLS 1.3 legacy
    final RandomCookie serverRandom;
    final SessionId sessionId; // TLS 1.3 legacy
    final CipherSuite cipherSuite;
    final byte compressionMethod; // TLS 1.3 legacy
    final SSLExtensions extensions;

    // The HelloRetryRequest producer needs to use the ClientHello message
    // for cookie generation. Please don’t use this field for other
    // purpose unless it is really necessary.
    final ClientHelloMessage clientHello;

    // Reserved for HelloRetryRequest consumer. Please don’t use this
    // field for other purpose unless it is really necessary.
    final ByteBuffer handshakeRecord;

    ServerHelloMessage(HandshakeContext context,
                      ProtocolVersion serverVersion, SessionId sessionId,
                      CipherSuite cipherSuite, RandomCookie serverRandom,
                      ClientHelloMessage clientHello) {
        super(context);

        this.serverVersion = serverVersion;
        this.serverRandom = serverRandom;
        this.sessionId = sessionId;
        this.cipherSuite = cipherSuite;
        this.compressionMethod = 0x00; // Don’t support compression.
        this.extensions = new SSLExtensions(this);

        // Reserve the ClientHello message for cookie generation.
        this.clientHello = clientHello;

        // The handshakeRecord field is used for HelloRetryRequest consumer
        // only. It’s fine to set it to null for generating side of the
        // ServerHello/HelloRetryRequest message.
        this.handshakeRecord = null;
    }
}

Listing 6: Class ServerHello.ServerHelloMessage defines the six fields of a ServerHello message (Lines 86–91), two additional fields for production and consumption of a HelloRetryRequest message (Lines 96 & 100), and constructors to instantiate those fields from parameters (Lines 102–122) or an input buffer (Listing 7). The former constructor does not populate the extensions field, whereas the latter may (Listing 7 Line 173–174).
`ServerHelloMessage(HandshakeContext context, ByteBuffer m) throws IOException {
    super(context);
    // Reserve for HelloRetryRequest consumer if needed.
    this.handshakeRecord = m.duplicate();
    byte major = m.get();
    byte minor = m.get();
    this.serverVersion = ProtocolVersion.valueOf(major, minor);
    this.serverRandom = new RandomCookie(m);
    this.sessionId = new SessionId(Record.getBytes8(m));
    int cipherSuiteId = Record.getInt16(m);
    this.cipherSuite = CipherSuite.valueOf(cipherSuiteId);
    this.compressionMethod = m.get();
    SSLExtension[] supportedExtensions;
    if (serverRandom.isHelloRetryRequest()) {
        supportedExtensions = context.sslConfig.getEnabledExtensions(
            SSLHandshake.HELLO_RETRY_REQUEST);
    } else {
        supportedExtensions = context.sslConfig.getEnabledExtensions(
            SSLHandshake.SERVER_HELLO);
    }
    if (m.hasRemaining()) {
        this.extensions = new SSLExtensions(this, m, supportedExtensions);
    } else {
        this.extensions = new SSLExtensions(this);
    }
    // The clientHello field is used for HelloRetryRequest producer
    // only. It’s fine to set it to null for receiving side of
    // ServerHello/HelloRetryRequest message.
    this.clientHello = null; // not used, let it be null;
}
public void send(HandshakeOutStream hos) throws IOException {
    hos.putInt8(serverVersion.major);
    hos.putInt8(serverVersion.minor);
    hos.write(serverRandom.randomBytes);
    hos.putBytes8(sessionId.getId());
    hos.putInt8((cipherSuite.id >> 8) & 0xFF);
    hos.putInt8(cipherSuite.id & 0xff);
    hos.putInt8(compressionMethod);
    extensions.send(hos); // In TLS 1.3, use of certain
    // extensions is mandatory.
}
Listing 7: Class ServerHello.ServerHelloMessage (continued from Listing 6) defines a constructor which instantiates ServerHello or HelloRetryRequest messages from an input buffer (Lines 124-183), and method send to write such messages to an output stream, using method SSLExtensions.send to write the extensions field (Listing 46 Lines 293–307).
static final HandshakeProducer t13HandshakeProducer = new T13ServerHelloProducer();

private static final class T13ServerHelloProducer implements HandshakeProducer {
    public byte[] produce(ConnectionContext context, HandshakeMessage message) throws IOException {
        ServerHandshakeContext shc = (ServerHandshakeContext)context;
        ClientHelloMessage clientHello = (ClientHelloMessage)message;

        // If client hasn't specified a session we can resume, start a new one and choose its cipher suite and compression options, unless new session creation is disabled for this connection!
        if (shc.isResumption || shc.resumingSession == null) {
            SSLSessionImpl session = new SSLSessionImpl(shc, CipherSuite.C_NULL);
            shc.handshakeSession = session;
            // consider the handshake extension impact
            SSLExtension[] enabledExtensions = shc.sslConfig.getEnabledExtensions(
                SSLHandshake.CLIENT_HELLO, shc.negotiatedProtocol);
            clientHello.extensions.consumeOnTrade(shc, enabledExtensions);

            // negotiate the cipher suite.
            CipherSuite cipherSuite = chooseCipherSuite(shc, clientHello);
            shc.negotiatedCipherSuite = cipherSuite;
            shc.handshakeSession.setSuite(cipherSuite);
        } else {
            shc.handshakeSession = shc.resumingSession;
            // consider the handshake extension impact
            SSLExtension[] enabledExtensions = shc.sslConfig.getEnabledExtensions(
                SSLHandshake.CLIENT_HELLO, shc.negotiatedProtocol);
            clientHello.extensions.consumeOnTrade(shc, enabledExtensions);
            shc.negotiatedCipherSuite = shc.resumingSession.getSuite();

            setUpPskKD(shc, shc.resumingSession.consumePreSharedKey().get());
        }

        // update the responders
        shc.handshakeProducers.put(SSLHandshake.ENCRYPTED_EXTENSIONS.id, SSLHandshake.ENCRYPTED_EXTENSIONS);
        shc.handshakeProducers.put(SSLHandshake.FINISHED.id, SSLHandshake.FINISHED);
    }

Listing 8: Class ServerHello.T13ServerHelloProducer defines method produce to write a ServerHello message to an output stream. Prior to instantiating such a message, the server's active context is updated to include extensions – in particular, signature_algorithms, signature_algorithms_cert, and pre_shared_key – that may impact the ServerHello message (Lines 519–522 or 539–542). Moreover, the active context is updated to include a producer for EncryptedExtensions and Finished messages (Lines 560–563). Code for writing the ServerHello message appears in Listing 9.
// Generate the ServerHello handshake message.
ServerHelloMessage shm = new ServerHelloMessage(shc,
    ProtocolVersion.TLS12, // use legacy version
    clientHello.sessionId, // echo back
    shc.negotiatedCipherSuite,
    new RandomCookie(shc),
    clientHello);
shc.serverHelloRandom = shm.serverRandom;

// Produce extensions for ServerHello handshake message.
SSLExtension[] serverHelloExtensions =
    shc.sslConfig.getEnabledExtensions(
        SSLHandshake.SERVER_HELLO, shc.negotiatedProtocol);
shm.extensions.produce(shc, serverHelloExtensions);

// Output the handshake message.
shm.write(shc.handshakeOutput);
shc.handshakeOutput.flush();
// The handshake message has been delivered.
return null;
}

private static CipherSuite chooseCipherSuite(
    ServerHandshakeContext shc,
    ClientHelloMessage clientHello) throws IOException {
    List<CipherSuite> preferred;
    List<CipherSuite> proposed;
    if (shc.sslConfig.preferLocalCipherSuites) {
        preferred = shc.activeCipherSuites;
        proposed = clientHello.cipherSuites;
    } else {
        preferred = clientHello.cipherSuites;
        proposed = shc.activeCipherSuites;
    }
    CipherSuite legacySuite = null;
    for (CipherSuite cs : preferred) {
        if (!HandshakeContext.isNegotiable(
            proposed, shc.negotiatedProtocol, cs)) {
            continue;
        }
    return cs;
    }
    return null;
}
from ClientHello.pre_shared_key; and for PSK with (EC)DHE, both of those extensions are included. Additional extensions are sent separately in the EncryptedExtensions message.

A ServerHello message is consumed by the client: The client first checks that the message is a TLS 1.3 ServerHello message, which is achieved by checking that extension supported_versions is present and that constant 0x0304 is the first listed preference. Next, the client checks whether the server’s nonce (random) is a special value (defined by constant RandomCookie.IrrRandomBytes) indicating that the ServerHello message is a HelloRetryRequest message and should be processed as such (§2.2.1). The client also checks whether the server selected protocol version (supported_versions) is amongst those offered (ClientHello.supported_versions) and is at least version 1.3, whether the server selected cipher suite (cipher_suite) is amongst those offered (ClientHello.cipher_suites), and whether field legacy_session_id_echo matches ClientHello.legacy_session_id, aborting with an illegal_parameter alert if any check fails. Finally, the client processes any remaining extensions:

pre_shared_key: The client checks whether the server-selected key identifier is amongst those offered by the client and that extension key_share is present if the offered key exchange mode for that identifier is PSK with (EC)DHE, aborting with an illegal_parameter alert if either check fails.

Consumption is implemented by class ServerHello.ServerHelloConsumer (Listing 10). That class checks the presence of extension supported_versions, to determine whether the message is a TLS 1.3 ServerHello message, and the remainder of the message is processed by ServerHello.T13ServerHelloConsumer (Listing 11), if it is a TLS 1.3 message.

2.2.1 HelloRetryRequest

A server that consumes a ClientHello message, without a share for the server-selected group, responds with a HelloRetryRequest message. That message is an instance of a ServerHello message, with field random set to a special constant value. In addition to mandatory extension supported_versions, message HelloRetryRequest should include extension key_share to indicate the server-selected group. (The server should defer producing a key share for this group until the client’s response is received.) The server may also include extension cookie associated with some data:

cookie: Some server-specific data for purposes including, but not limited to, first, offloading state (required to construct transcripts) to the client, by storing the hash of the ClientHello message in the cookie (with suitable integrity protection); and, secondly, DoS protection, by forcing the client to demonstrate reachability of their network address.

A HelloRetryRequest message is consumed by the client, which performs the checks specified for ServerHello messages (above), additionally aborting with an illegal_parameter alert if the server-selected group is not amongst those offered (ClientHello.supported_groups) or a key share for that group was already offered, or aborting with an an unexpected_message alert if a HelloRetryRequest message was already received in the same connection. A client that is able to successfully consume a HelloRetryRequest message responds with their original ClientHello message, replacing the key shares in extension key_share with a single key share from the server-selected group and including a copy of extension cookie and associated data if the extension

---

5 For convenience, HelloRetryRequest and ServerHello messages are distinctly named (in the specification), despite HelloRetryRequest messages being instances of ServerHello messages. It follows that a ServerHello message might be confused for a HelloRetryRequest message, but this only occurs with probability \( \frac{1}{2^{256}} \), hence, confusion will not occur in practice.

6 Extension key_share is associated with key shares for ClientHello messages and a single key share for ServerHello messages, whereas the extension is associated with the server-selected group for HelloRetryRequest messages. Hence, data structures associated with extension key_share vary between messages.
Listing 10: Class ServerHello.ServerHelloConsumer defines method consume to instantiate a (generic) ServerHello message from an input buffer (Line 864) and processes the message as a HelloRetryRequest (Line 870) or a ServerHello message (Line 872). The latter updates the client’s active context to include the server’s selected version (Lines 933–936), using method SupportedVersionsExtension.SHSupportedVersionsConsumer.consume, which calls chc.handshakeExtensions.get(SH_SUPPORTED_VERSIONS, spec), and checks whether that version was offered by the client (Lines 949), aborting if it was not (Lines 950–953) and, otherwise, updating the active context to include that version as the negotiated protocol (Lines 956–960). (Variable serverVersion, Lines 943–944, cannot be null for (TLS 1.3) HelloRetryRequest nor ServerHello messages.) Further processing is deferred (Line 984) to class ServerHello.T13ServerHelloConsumer (Listing [11]).
private static final HandshakeConsumer t13HandshakeConsumer =
new T13ServerHelloConsumer();

private static final class T13ServerHelloConsumer implements HandshakeConsumer {

public void consume(ConnectionContext context,
HandshakeMessage message) throws IOException {
ClientHandshakeContext chc = (ClientHandshakeContext)context;
ServerHelloMessage serverHello = (ServerHelloMessage)message;

chc.negotiatedCipherSuite = serverHello.cipherSuite;
chc.serverHelloRandom = serverHello.serverRandom;
SSLExtension[] extTypes = chc.sslConfig.getEnabledExtensions(SSLHandshake.SERVER_HELLO);
serverHello.extensions.consumeOnLoad(chc, extTypes);
if (!chc.isResumption) {
    chc.handshakeSession = new SSLSessionImpl(chc,
        chc.negotiatedCipherSuite,
        serverHello.sessionId);
} else {
    Optional<SecretKey> psk =
        chc.resumingSession.consumePreSharedKey();
    chc.handshakeSession = chc.resumingSession;
    setUpPskKD(chc, psk.get());
}

// update the consumers and producers
chc.handshakeConsumers.put(
    SSLHandshake.ENCRYPTED_EXTENSIONS.id ,
    SSLHandshake.ENCRYPTED_EXTENSIONS);
chc.handshakeConsumers.put(
    SSLHandshake.CERTIFICATE_REQUEST.id ,
    SSLHandshake.CERTIFICATE_REQUEST);
chc.handshakeConsumers.put(
    SSLHandshake.CERTIFICATE.id ,
    SSLHandshake.CERTIFICATE);
chc.handshakeConsumers.put(
    SSLHandshake.CERTIFICATE_VERIFY.id ,
    SSLHandshake.CERTIFICATE_VERIFY);
chc.handshakeConsumers.put(
    SSLHandshake.FINISHED.id ,
    SSLHandshake.FINISHED);
}

Listing 11: Class ServerHello.T13ServerHelloConsumer defines method consume to process incoming (TLS 1.3) ServerHello or HelloRetryRequest messages (further to processing shown in Listing 10). The method updates the client’s active context to include the server’s selected cipher suite as the negotiated suite (Line 1190), extensions, including pre_shared_key (Lines 1200–1202), and additional session information (Lines 1203–1231). (The client also updates the active context to include new keying material, Section 2.3.) Moreover, the active context is made ready to received further server messages (Lines 1330–1356).
appeared in the HelloRetryRequest message. Moreover, the client should remove any pre-shared key identifiers that are incompatible with the server-selected cipher suite. That ClientHello message is consumed by the server (§2.1) and the server responds with a ServerHello message, which must contain the previously selected cipher suite, namely, HelloRetryRequest.CipherSuite. The ServerHello message is consumed by the client as described above, additionally aborting with an illegal_parameter alert if the server-selected cipher suite differs from the previous server-selected cipher suite (HelloRetryRequest.CipherSuite), if extension supported_versions is associated with a list of offered protocol versions that differ from the previous list (HelloRetryRequest.supported_versions), or if the server’s key share does not belong to the previous server-selected group (HelloRetryRequest.key_share).

The HelloRetryRequest message is implemented by class ServerHello.ServerHelloMessage (Listings 6-7). Instances of that class are produced by class ServerHello.T13ServerHelloProducer (Listings 12), which is instantiated as static constant ServerHello.t13HandshakeProducer. That constant is used indirectly – via class SSLHandshake.HelloRetryRequest — to produce HelloRetryRequest messages in class ClientHello.T13ClientHelloConsumer (Listing 5). Consumption is implemented by class ServerHello.ServerHelloConsumer (Listing 10 & 13). That class checks the presence of extension supported_versions, to determine whether the message is a TLS 1.3 HelloRetryRequest message, and the remainder of the message is processed by class ServerHello.T13HelloRetryRequestConsumer (Listing 14), if it is a TLS 1.3 message. Successful consumption results in transmission of a further ClientHello message (which is consumed by class ClientHello.T13ClientHelloConsumer, Listings 4 & 5), with any cookie extension being indirectly processed – via class CookieExtension – by class HelloCookieManager.

2.3 Key establishment

Once a ServerHello message has been sent, a shared (handshake traffic) key can be established, and that key can be used to enable confidentiality and integrity for the remainder of the handshake protocol, which includes the subsequent EncryptedExtensions message (§2.4). The initial shared key is derived by application of a key derivation function, known as a HMAC-based Extract-and-Expand Key Derivation Function (HKDF), which applies the negotiated hash function to the handshake protocol’s transcript and either the negotiated pre-shared key, the negotiated (EC)DHE key, or both. Further shared (application traffic) keys can be established similarly, to protect additional data, including application data. Since transcripts include client- and server-generated nonces, shared (traffic) keys are always unique, regardless of whether the pre-shared key (or for that matter (EC)DHE key shares) are used for multiple connections.

2.3.1 Transcript hash

A protocol’s transcript concatenates each of the protocol’s messages, in the order that they were sent, including message headers (namely, type and length fields), but excluding record-layer headers. To capture a transcript hash (i.e., a hash of a transcript), we introduce function Transcript-Hash such that

\[
\text{Transcript-Hash}(M_1, \ldots, M_n) = \text{Hash}(M_1 \parallel \cdots \parallel M_n)
\]

for handshake protocol messages \(M_1, \ldots, M_n\) (sent in that order), where \(\text{Hash}\) is the negotiated hash function and \(\parallel\) denotes concatenation, except when messages \(M_1\) and \(M_2\) are ClientHello and HelloRetryRequest messages, respectively. In that case, \(M_1\) is replaced by \(M'_1\) in the hash, i.e.,

\[
\text{Transcript-Hash}(M_1, \ldots, M_n) = \text{Hash}(M'_1 \parallel M_2 \parallel \cdots \parallel M_n),
\]
static final HandshakeProducer hrrHandshakeProducer =
    new T13HelloRetryRequestProducer();

private static final class T13HelloRetryRequestProducer implements HandshakeProducer {
    public byte[] produce(ConnectionContext context, HandshakeMessage message)
        throws IOException {
        ServerHandshakeContext shc = (ServerHandshakeContext) context;
        ClientHelloMessage clientHello = (ClientHelloMessage) message;
        // negotiate the cipher suite.
        CipherSuite cipherSuite = T13ServerHelloProducer.chooseCipherSuite(shc, clientHello);
        ServerHelloMessage hhrm = new ServerHelloMessage(shc,
            ProtocolVersion.TLS12, // use legacy version
            clientHello.sessionId, // echo back
            cipherSuite,
            RandomCookie.hrrRandom,
            clientHello);
        shc.negotiatedCipherSuite = cipherSuite;
        // Produce extensions for HelloRetryRequest handshake message.
        SSLExtension[] serverHelloExtensions =
            shc.sslConfig.getEnabledExtensions(SSLHandshake.HELLO_RETRY_REQUEST, shc.negotiatedProtocol);
        hhrm.extensions.produce(shc, serverHelloExtensions);
        // Output the handshake message.
        hhrm.write(shc.handshakeOutput);
        shc.handshakeOutput.flush();
        // Stateless, shall we clean up the handshake context as well?
        shc.handshakeHash.finish(); // forgot about the handshake hash
        shc.handshakeExtensions.clear();
        // What's the expected response?
        shc.handshakeConsumers.put(
            SSLHandshake.CLIENT_HELLO.id, SSLHandshake.CLIENT_HELLO);
        // The handshake message has been delivered.
        return null;
    }
}

Listing 12: Class ServerHello.T13HelloRetryRequestProducer defines method produce to instantiate a HelloRetryRequest message, i.e., a ServerHello message with field random set to a special constant value (Lines 754–760), populate the extension field for the active context (Lines 767–770), write the message to an output stream (Lines 777–778), and prepare the server’s active context for the client’s response (Lines 762 & 785–786).
private void onHelloRetryRequest(ClientHandshakeContext chc, ServerHelloMessage helloRetryRequest) throws IOException {
    SSLExtension[] extTypes = new SSLExtension[] {
        SSLExtension.HRR_SUPPORTED_VERSIONS
    };  
    helloRetryRequest.extensions.consumeOnLoad(chc, extTypes);
    ProtocolVersion serverVersion;
    SHSupportedVersionsSpec svs = (SHSupportedVersionsSpec)chc.handshakeExtensions.get(
        SSLExtension.HRR_SUPPORTED_VERSIONS);
    serverVersion = // could be null
    ProtocolVersion.valueOf(svs.selectedVersion);
    if (!chc.activeProtocols.contains(serverVersion)) {
        chc.conContext.fatal(Alert.PROTOCOL_VERSION,
            "The selected protocol version is not accepted by client preferences" +
            chc.activeProtocols);
    }
    chc.negotiatedProtocol = serverVersion;
    t13HrrHandshakeConsumer.consume(chc, helloRetryRequest);
}

Listing 13: Class ServerHello.ServerHelloConsumer (omitted from Listing 10) defines method onHelloRetryRequest to consume a (generic) HelloRetryRequest message. Similarly to method ServerHello.ServerHelloConsumer.onHelloServer (Listing 10), the client’s active context is updated to include the server’s selected version (Lines 886–892), aborting if that version was not offered by the client (Lines 897–902). Further processing is deferred (Line 924) to class ServerHello.T13HelloRetryRequestConsumer (Listing 14).

private static final HandshakeConsumer t13HrrHandshakeConsumer =
    new T13HelloRetryRequestConsumer();
public void consume(ConnectionContext context, HandshakeMessage message) throws IOException {
    ClientHandshakeContext chc = (ClientHandshakeContext)context;
    ServerHelloMessage helloRetryRequest = (ServerHelloMessage)message;
    chc.negotiatedCipherSuite = helloRetryRequest.cipherSuite;
    // Check and launch ClientHello extensions.
    SSLExtension[] extTypes = chc.sslConfig.getEnabledExtensions(
        SSLHandshake.HelloRequest.RETRY_REQUEST);
    helloRetryRequest.extensions.consumeOnLoad(chc, extTypes);
    helloRetryRequest.extensions.consumeOnTrade(chc, extTypes);
    SSLHandshake.CLIENT_HELLO.produce(context, helloRetryRequest);
}

Listing 14: Class ServerHello.T13HelloRetryRequestConsumer defines method consume to process incoming (TLS 1.3) HelloRetryRequest messages (further to processing shown in Listing 13). The method updates the active context to include the server’s selected cipher suite (Line 1383) and extensions (Line 1390–1397), and produces a ClientHello message (Line 1459).
where $M'_1$ is the following special, synthetic handshake message, namely,

$$
M'_1 = \begin{cases}
0xFE & /* header type message_hash*/ \\
\| 0x0000 \| \text{Hash.length} & /* (padded) header length */ \\
\| \text{Hash}(M_1) & /* hash of ClientHello message */
\end{cases}
$$

where \text{Hash.length} is the output length in bytes of negotiated hash function \text{Hash}. This special case enables servers to construct transcripts without maintaining state, in particular, they need not store an initial ClientHello message, since it can be stored in extension cookie (§2.2.1).
final class HandshakeHash {
    private TranscriptHash transcriptHash;
    private LinkedList<byte[]> reserves;  // one handshake message per entry

    HandshakeHash() {
        this.transcriptHash = new CacheOnlyHash();
        this.reserves = new LinkedList<>();
    }

    void receive(byte[] input) {
        reserves.add(Arrays.copyOf(input, input.length));
    }

    // For HelloRetryRequest only! Please use this method very carefully!
    void push(byte[] input) {
        reserves.push(Arrays.copyOf(input, input.length));
    }

    void deliver(byte[] input) {
        update();
        transcriptHash.update(input, 0, input.length);
    }

    void update() {
        while (reserves.size() != 0) {
            byte[] holder = reserves.remove();
            transcriptHash.update(holder, 0, holder.length);
        }
    }

    byte[] digest() {
        // Note that the reserve handshake message may be not a part of
        // the expected digest.
        return transcriptHash.digest();
    }

    void finish() {
        this.transcriptHash = new CacheOnlyHash();
        this.reserves = new LinkedList<>();
    }
}

Listing 15: Class HandshakeHash defines field reserves to maintain a list of protocol messages (Line 39), which can be extended (e.g., with incoming messages) using method receive (Lines 85–87), moreover, the class defines field transcriptHash as a message digest algorithm (Line 38, see also Lines 58 & 551–644), whose digest can be updated to include the aforementioned messages using methods deliver (Lines 116–119) and update (Lines 164–170). (The former method is used when the digest should also include an additional message, e.g., an outgoing message, whereas the latter only updates the digest with messages listed by field reserves.) Furthermore, method digest returns the hash over the current digest (Lines 172–176) and method finish resets all fields (Lines 178–182).
chc.handshakeHash.finish(); // reset the handshake hash
// calculate the transcript hash of the 1st ClientHello message
HandshakeOutStream hos = new HandshakeOutStream(null);
try {
    chc.initialClientHelloMsg.write(hos);
} catch (IOException ioe) {
    // unlikely
}
chc.handshakeHash.deliver(hos.toByteArray());
byte[] clientHelloHash = chc.handshakeHash.digest();

// calculate the message_hash
// Transcript–Hash(ClientHello1, HelloRetryRequest, ... Mn) =
// Hash(message_hash | | */ Handshake type */
// | | 00 00 Hash.length | | */ Handshake message length (bytes) */
// | | Hash(ClientHello1) | | */ Hash of ClientHello1 */
// HelloRetryRequest | | ... | | Mn)
int hashLen = chc.negotiatedCipherSuite.hashAlg.hashLength;
byte[] hashedClientHello = new byte[4 + hashLen];
hashedClientHello[0] = SSLHandshake.MESSAGE_HASH.id;
hashedClientHello[1] = (byte)0x00;
hashedClientHello[2] = (byte)0x00;
hashedClientHello[3] = (byte)(hashLen & 0xFF);
System.arraycopy(clientHelloHash, 0,
    hashedClientHello, 4, hashLen);
chc.handshakeHash.finish(); // reset the handshake hash
chc.handshakeHash.deliver(hashedClientHello);
int hrrBodyLen = helloRetryRequest.handshakeRecord.remaining();
byte[] hrrMessage = new byte[4 + hrrBodyLen];
hrrMessage[0] = SSLHandshake.HELLO_RETRY_REQUEST.id;
hrrMessage[1] = (byte)((hrrBodyLen >> 16) & 0xFF);
hrrMessage[2] = (byte)((hrrBodyLen >> 8) & 0xFF);
hrrMessage[3] = (byte)(hrrBodyLen & 0xFF);
ByteBuffer hrrBody = helloRetryRequest.handshakeRecord.duplicate();
hrrBody.get(hrrMessage, 4, hrrBodyLen);
chc.handshakeHash.receive(hrrMessage);

Listing 16: Class ServerHello.T13HelloRetryRequestConsumer (omitted from Listing 14) modifies the transcript hash in the special case of HelloRetryRequest messages: A hash of the ClientHello message is computed (Lines 1401–1415), using variable chc.initialClientHelloMsg that was initialised in Listing 2; a special, synthetic handshake message $M'_1$ is computed, as the concatenation of $0xF$ (Line 1426), $0x0000$ (Lines 1427–1428), Hash.length (Line 1429), and the hashed ClientHello message (Lines 1430–1431); the transcript hash’s digest is reset and the special message is added (Line 1433–1434); a further message is computed as the concatenation of $0x02$ (Line 1438), the HelloRetryRequest message length (Lines 1439–1441), and the HelloRetryRequest (Lines 1443–1444); and that message is added to the transcript hash’s digest too (Line 1446). Thus, the client’s active context includes the expected digest.
Listing 17: Class HelloCookieManager.T13HelloCookieManager processes cookies, in particular, method isCookieValid tests the validity of cookies. That method also updates the transcript hash in the special case of HelloRetryRequest messages: A HelloRetryRequest message is reconstructed and added to the front of the transcript hash's digest (Lines 322–324). Moreover, a special, synthetic handshake message $M'_1$ is computed as the concatenation of 0xFE0000 || Hash.length and the hash (of a ClientHello message) stored in the cookie, and message $M'_1$ is added to the front of the transcript hash's digest (Lines 327–335).
Function Derive-Secret is used (with the empty context) to derive salt for subsequent applications of HKDF-Extract. Indeed, we have

\[
\text{Handshake Secret} = \text{HKDF-Extract}(\text{Derive-Secret}(\text{Early Secret}, \text{"derived"}, \text{""}), K),
\]

where K is 0 for PSK-only key exchange and otherwise the (EC)DHE key, moreover,

\[
\text{Master Secret} = \text{HKDF-Extract}(\text{Derive-Secret}(\text{Handshake Secret}, \text{"derived"}, \text{""}), 0),
\]

noting that Transcript-Hash("") = Hash("") , that is, the hash of the empty string. Traffic secrets are derived from Early Secret, Handshake Secret, and Master Secret, as shown in Figure 2 by adding context. Those secrets are used to derive traffic keys (§2.3.3) to protect the data summarised in the following table:

| Underlying traffic secret                                                       | Protected data                        |
|---------------------------------------------------------------------------------|---------------------------------------|
| client_early_traffic_secret                                                     | 0-RTT                                 |
| [sender]_handshake_traffic_secret                                               | Handshake extensions                  |
| [sender]_application_traffic_secret_N                                          | Application traffic                   |

Table 2: Traffic secrets that underlie traffic keys used to protect data

where [sender] is either client or server, and [sender]_application_traffic_secret_N+1 is defined as follows when N > 0, namely,

\[
\text{[sender]_application_traffic_secret}_N+1 = \text{Derive-Secret}([\text{sender}]_\text{application_traffic_secret}_N, \text{"traffic upd"}, \text{""}),
\]

which is used to update application-traffic secrets.

Function HKDF-Extract is implemented by class HKDF (Listing 18) and function Derive-Secret is implemented by class SSLSecretDerivation (Listing 19 & 20). Application of the former is dependent on the negotiated pre-shared key to derive Early Secret, which is computed by static method ServerHello.setUpPskKD (Listing 21), except for (EC)DHE-only key exchange, which derives Early Secret as HKDF-Extract(0, 0) and is computed by class DHKeyExchange.DHEKAGenerator.DHEKAKeyDerivation or class ECDHKey-Exchange.ECDHEKAKKeyDerivation, which also compute Handshake Secret. (PSK-only key exchange is unsupported and Handshake Secret is only computed with an (EC)DHE key.) Those classes are identical up to constructor names, strings "DiffieHellman" and "ECDH", and whitespace. (Refactoring could eliminate unnecessary code.) So, for brevity, we only present the former class (Listing 22).

2.3.3 Traffic keys

Traffic keys are derived from traffic secrets listed in Table 2 using function HKDF-Expand-Label, defined such that

\[
\text{HKDF-Expand-Label}(\text{Secret}, \text{Label}, \text{Context}, \text{Length}) = \text{HKDF-Expand}(\text{Secret}, \text{HkdfLabel}, \text{Length}),
\]

where HkdfLabel = Length || “tls13_” || Label || Context and function HKDF-Expand is defined by RFC 5869 such that HKDF-Expand(\text{Secret}, \text{ExpLabel}, \text{Length}) outputs the first Length-bytes of
Figure 2: Key derivation process, showing application of functions HKDF-Extract and Derive-Secret to derive working keys. Function HKDF-Extract is shown inputting salt from the top and secrets from the left, and outputs to the bottom, where the output’s name is shown to the right. Moreover, function Derive-Secret is shown inputting secrets from the incoming arrow and the remaining inputs appear inline, and some outputs are named below the function’s application (e.g., Early Secret is input as the secret to generate client_early_traffic_secret) and others serve as salt for subsequent applications of the former (e.g., Early Secret is input as the secret to generate salt for Handshake Secret). Output binder_key is derived by application of function Derive-Secret to “ext binder” | “res binder” which denotes either “ext binder” or “res binder.” The former is used for external PSKs (those provisioned independently of TLS) and the latter is used for resumption PSKs (those provisioned as the resumption master secret of a previous handshake), hence, one type of PSK cannot be substituted for the other.

Source: This figure is an excerpt from RFC 8446.
final class HKDF {
    private final String hmacAlg;
    private final Mac hmacObj;
    private final int hmacLen;

    HKDF(String hashAlg) throws NoSuchAlgorithmException {
        hmacAlg = "Hmac" + hashAlg.replace("-", "");
        hmacObj = JsseJce.getMac(hmacAlg);
        hmacLen = hmacObj.getMacLength();
    }

    SecretKey extract(SecretKey salt, SecretKey inputKey, String keyAlg)
        throws InvalidKeyException {
        if (salt == null) {
            salt = new SecretKeySpec(new byte[hmacLen], "HKDF-Salt");
        }
        hmacObj.init(salt);
        return new SecretKeySpec(hmacObj.doFinal(inputKey.getEncoded()), keyAlg);
    }

    SecretKey extract(byte[] salt, SecretKey inputKey, String keyAlg)
        throws InvalidKeyException {
        if (salt == null) {
            salt = new byte[hmacLen];
        }
        return extract(new SecretKeySpec(salt, "HKDF-Salt"), inputKey, keyAlg);
    }
}

Listing 18: Class HKDF defines method extract to implement function HKDF-Extract (RFC 5869),
over salt values of type SecretKey (Lines 86–95) and byte[] (Lines 114–120), using a HMAC function
derived from the negotiated hash function (Lines 64–65), where JsseJce.getMac(hmacAlg) computes
Mac.getInstance(hmacAlg) or Mac.getInstance(hmacAlg, cryptoProvider), depending on whether
sun.security.ssl.SunJSSE.cryptoProvider is null. Both implementations allow the
salt to be null and will instantiate salt as a zero-filled byte array of the same length as
Hash.length (Lines 88–90 & 116–118). An HMAC is initialised with the salt as a key (Line 91)
and a secret as the message (Line 93), the resulting HMAC is returned as a key of type
javax.crypto.spec.SecretKeySpec.
final class SSLSecretDerivation implements SSLKeyDerivation {
    private final HandshakeContext context;
    private final HashAlg hashAlg;
    private final SecretKey secret;
    private final byte[] transcriptHash; // handshake messages transcript hash

    SSLSecretDerivation(HandshakeContext context, SecretKey secret) {
        this.context = context;
        this.secret = secret;
        this.hashAlg = context.negotiatedCipherSuite.hashAlg;
        context.handshakeHash.update();
        this.transcriptHash = context.handshakeHash.digest();
    }

    public SecretKey deriveKey(String algorithm, AlgorithmParameterSpec params) throws IOException {
        SecretSchedule ks = SecretSchedule.valueOf(algorithm);
        try {
            byte[] expandContext;
            if (ks == SecretSchedule.TlsSaltSecret) {
                if (hashAlg == HashAlg.H_SHA256) {
                    expandContext = sha256EmptyDigest;
                } else if (hashAlg == HashAlg.H_SHA384) {
                    expandContext = sha384EmptyDigest;
                } else {
                    // unlikely, but please update if more hash algorithm
                }
            } else {
                expandContext = transcriptHash;
            }

            byte[] hkdfInfo = createHkdfInfo(ks.label, expandContext, hashAlg.hashLength);

            HKDF hkdf = new HKDF(hashAlg.name);
            return hkdf.expand(secret, hkdfInfo, hashAlg.hashLength, algorithm);
        } catch (GeneralSecurityException gse) {
            throw (SSLHandshakeException) new SSLHandshakeException("Could not generate secret").initCause(gse);
        }
    }
}

Listing 19: Class SSLSecretDerivation implements function Derive-Secret. The class defines a constructor (Lines 69–78) that instantiates fields context, transcriptHash and secret with data including the transcript hash, the hash of the corresponding digest and a secret, respectively. Method deriveKey (Lines 85–114) is instantiated with a string that references a label and returns an HMAC computed by application of method HKDF.expand (Line 109) to inputs including HkdfLabel, which is computed (Lines 105–106) over the negotiated hash function’s output length, the label prepended with “tls13/”, and a hash of either the transcript’s digest (when the resulting output will be used as a secret) or the empty digest (when the resulting output will be used as salt, i.e., when ks == SecretSchedule.TlsSaltSecret), using static method SSLSecretDerivation.createHkdfInfo to handle concatenation.
private enum SecretSchedule {
  // Note that we use enum name as the key/secret name.
  TlsSaltSecret  (*derived*),
  TlsExtBinderKey (*ext_binder*),
  TlsResBinderKey (*res_binder*),
  TlsClientEarlyTrafficSecret (*c_e_traffic*),
  TlsEarlyExporterMasterSecret (*e_exp_master*),
  TlsClientHandshakeTrafficSecret (*c_hs_traffic*),
  TlsServerHandshakeTrafficSecret (*s_hs_traffic*),
  TlsClientAppTrafficSecret (*c_ap_traffic*),
  TlsServerAppTrafficSecret (*s_ap_traffic*),
  TlsExporterMasterSecret (*exp_master*),
  TlsResumptionMasterSecret (*res_master*);

  private final byte[] label;

  private SecretSchedule(String label) {
    this.label = ("tls13_" + label).getBytes();
  }
}

Listing 20: Enum SSLSecretDerivation.SecretSchedule (omitted from Listing 19) maps strings to
labels used by function Derive-Secret, and prepends labels with “tls13.”.

private static void setUpPskKD(HandshakeContext hc,
  SecretKey psk) throws SSLHandshakeException {
  try {
    CipherSuite.HashAlg hashAlg = hc.negotiatedCipherSuite.hashAlg;
    HKDF hkdf = new HKDF(hashAlg.name);
    byte[] zeros = new byte[hashAlg.hashLength];
    SecretKey earlySecret = hkdf.extract(zeros, psk, "TlsEarlySecret");
    hc.handshakeKeyDerivation =
      new SSLSecretDerivation(hc, earlySecret);
  } catch (GeneralSecurityException gse) {
    throw (SSLHandshakeException) new SSLHandshakeException(
      "Could not generate secret").initCause(gse);
  }
}

Listing 21: Static method ServerHello.setUpPskKD (omitted from Listings 8 & 11) derives Early
Secret over a negotiated pre-shared key.
private static final

class DHEKAKeyDerivation implements SSLKeyDerivation {
    private final HandshakeContext context;
    private final PrivateKey localPrivateKey;
    private final PublicKey peerPublicKey;
    public SecretKey deriveKey(String algorithm,
        AlgorithmParameterSpec params) throws IOException {
        return t13DeriveKey(algorithm, params);
    }
    private SecretKey t13DeriveKey(String algorithm,
        AlgorithmParameterSpec params) throws IOException {
        try {
            KeyAgreement ka = JsseJce.getKeyAgreement("DiffieHellman");
            ka.init(localPrivateKey);
            ka.doPhase(peerPublicKey, true);
            SecretKey sharedSecret =
                ka.generateSecret("TlsPremasterSecret");
            HashAlg hashAlg = context.negotiatedCipherSuite.hashAlg;
            SSLKeyDerivation kd = context.handshakeKeyDerivation;
            HKDF hkdf = new HKDF(hashAlg.name);
            if (kd == null) { // No PSK is in use.
                // If PSK is not in use Early Secret will still be
                // HKDF-Extract(0, 0).
                byte[] zeros = new byte[hashAlg.hashLength];
                SecretKeySpec ikm =
                    new SecretKeySpec(zeros, "TlsPreSharedSecret");
                SecretKey earlySecret =
                    hkdf.extract(zeros, ikm, "TlsEarlySecret");
                kd = new SSLSecretDerivation(context, earlySecret);
            }
            // derive salt secret
            SecretKey saltSecret = kd.deriveKey("TlsSaltSecret", null);
            // derive handshake secret
            return hkdf.extract(saltSecret, sharedSecret, algorithm);
        } catch (GeneralSecurityException gse) {
            throw (SSLHandshakeException) new SSLHandshakeException("Could_not_generate_secret").initCause(gse);
        }
    }
}

Listing 22: Class DHKeyExchange.DHEKAKeyGenerator.DHEKAKeyDerivation defines method
t13DeriveKey to derive the negotiated key (Lines 502–506); compute Early Secret, for
(EC)-DHE-only key exchange (Lines 511–520), i.e., when production or consumption of
a ServerHello message did not call method ServerHello setUpPskKD, which instantiates
context.handshakeKeyDerivation; applies Derive-Secret to Early Secret and label “derived”
(Line 523); and uses the resulting output as salt when applying HKDF-Extract to the negotiated
key (Line 526), which produces Handshake Secret.
SecretKey expand(SecretKey pseudoRandKey, byte[] info, int outLen, 
String keyAlg) throws InvalidKeyException {
    byte[] kdfOutput;
    hmacObj.init(pseudoRandKey);
    if (info == null) {
        info = new byte[0];
    }
    int rounds = (outLen + hmacLen - 1) / hmacLen;
    kdfOutput = new byte[rounds * hmacLen];
    int offset = 0;
    int tLength = 0;
    for (int i = 0; i < rounds; i++) {
        try {
            // Add T(i). This will be an empty string on the first
            // iteration since tLength starts at zero. After the first
            // iteration, tLength is changed to the HMAC length for the
            // rest of the loop.
            hmacObj.update(kdfOutput, 
                Math.max(0, offset - hmacLen), tLength);
            hmacObj.update(info); // Add info
            hmacObj.update((byte)(i + 1)); // Add round number
            hmacObj.doFinal(kdfOutput, offset);
            tLength = hmacLen;
            offset += hmacLen; // For next iteration
        } catch (ShortBufferException sbe) {
            // This really shouldn’t happen given that we’ve
            // sized the buffers to their largest possible size up-front,
            // but just in case...
            throw new RuntimeException(sbe);
        }
    }
    return new SecretKeySpec(kdfOutput, 0, outLen, keyAlg);
}

Listing 23: Class HKDF (omitted from Listing 18) defines method expand to implement function 
HKDF-Expand. A buffer kdfOutput of length Hash.length ⌈ Length Hash.length ⌉ is initialised (Lines 139 & 
154–155) and an HMAC is initialised with the input secret as a key (Line 150). The for-loop 
computes \( T_1, T_2, \ldots \) values as HMACs over messages that concatenate the previous round’s output 
(which is the empty string during the first round), label info, and the round number (Lines 167– 
170). Those values are stored in buffer kdfOutput (Line 171), which is returned as a key of type 
javax.crypto.spec.SecretKeySpec after truncating to length outLen (Line 183).

\[ T_1 \parallel \cdots \parallel T_n, \text{ where } n = \lceil \text{Length} \text{Hash.length} \rceil \] and

\[ T_0 = "" \]
\[ T_1 = \text{HMAC} (\text{Secret}, T_0 \parallel \text{ExpLabel} \parallel 0x01) \]
\[ T_2 = \text{HMAC} (\text{Secret}, T_1 \parallel \text{ExpLabel} \parallel 0x02) \]
\[ \vdots \]

Function HKDF-Expand-Label may input Context as the null ASCII character 0x00, denoted \(^{207}\).

Function HKDF-Expand is implemented by class HKDF (Listing 23) and traffic keys are 
derived by class SSLTrafficKeyDerivation (Listing 24).
static final class T13TrafficKeyDerivation implements SSLKeyDerivation {
    private final CipherSuite cs;
    private final SecretKey secret;

    T13TrafficKeyDerivation(
        HandshakeContext context, SecretKey secret) {
        this.secret = secret;
        this.cs = context.negotiatedCipherSuite;
    }

    public SecretKey deriveKey(String algorithm,
        AlgorithmParameterSpec params) throws IOException {
        try {
            HKDF hkdf = new HKDF(cs.hashAlg.name);
            byte[] hkdfInfo =
                createHkdfInfo(ks.label, ks.getKeyLength(cs));
            return hkdf.expand(secret, hkdfInfo,
                ks.getKeyLength(cs),
                ks.getAlgorithm(cs, algorithm));
        } catch (GeneralSecurityException gse) {
            throw (SSLHandshakeException)(
                new SSLHandshakeException(
                    "Could/uni2423not/uni2423generate/uni2423secret").initCause(gse));
        }
    }

    private enum KeySchedule {
        // Note that we use enum name as the key/name.
        TlsKey ("key", false),
        TlsIv ("iv", true),
        TlsUpdateNplus1 ("traffic_upd", false);

        private final byte[] label;
        private final boolean isIv;

        private KeySchedule(String label, boolean isIv) {
            this.label = ("tls13_" + label).getBytes();
            this.isIv = isIv;
        }

        int getKeyLength(CipherSuite cs) {
            if (this == KeySchedule.TlsUpdateNplus1)
                return cs.hashAlg.hashLength;
            return isIv ? cs.bulkCipher.ivSize : cs.bulkCipher.keySize;
        }

        String getAlgorithm(CipherSuite cs, String algorithm) {
            return isIv ? algorithm : cs.bulkCipher.algorithm;
        }
    }

Listing 24: Class SSLTrafficKeyDerivation.T13TrafficKeyDerivation derives traffic keys. Method deriveKey is instantiated with a string that references a label and returns an HMAC computed by application of method HKIDR.expand (Lines 153–155) to inputs including HkdfLabel, which is computed (Lines 151–152) over the negotiated hash function’s output length, the label prepended with "tls13_", and null ASCII character 0x00, using static method SSLTrafficKeyDerivation.T13TrafficKeyDerivation.createHkdfInfo to handle concatenation and to introduce 0x00.
Returning to key derivation, we derive the following traffic keys:

\[
\begin{align*}
\text{[sender]}_\text{write_key} &= \text{HKDF-Expand-Label}([\text{secret}], \text{“key”}, \text{“”}, \text{key\_length}) \\
\text{[sender]}_\text{write_iv} &= \text{HKDF-Expand-Label}([\text{secret}], \text{“iv”}, \text{“”}, \text{iv\_length})
\end{align*}
\]

where \(\text{[sender]}\) is either \textit{client} or \textit{server}, and \([\text{secret}]\) is taken from the secrets listed in Table 2.

Server- and client-side handshake-traffic key derivation is implemented by classes ServerHello.T13ServerHelloProducer and ServerHello.T13ServerHelloConsumer, respectively. The former class defines method produce to write a ServerHello message to an output stream (Listings 8 & 9), and that method derives handshake-traffic keys immediately after writing the ServerHello message; the keys are used to encrypt subsequent outgoing handshake messages (including an EncryptedExtensions message) and to decrypt subsequent incoming handshake messages. Similarly, the latter class defines method consume to read a ServerHello message from an input buffer (Listing 11), and that method derives handshake-traffic keys immediately before reading an EncryptedExtensions message (and prior to reading further extensions, including Certificate and CertificateVerify messages for (EC)DHE-only key exchange, and a Finished message); the keys are used to decrypt subsequent incoming handshake messages, including that EncryptedExtensions message, and to encrypt subsequent outgoing handshake messages. The implementations are identical up to contexts (namely, ServerHandshakeContext and ClientHandshakeContext, that share parent HandshakeContext), labels \(s/\text{uni2423hs}/\text{uni2423 traffic}\) and \(c/\text{uni2423hs}/\text{uni2423 traffic}\) (which are instantiated by enum SSLSecretDerivation.SecretSchedule using strings TlsServerHandshakeTrafficSecret and TlsClientHandshakeTrafficSecret, respectively), treatment of null in tricks to make the compiler happy (cf. \text{return null}; and \text{return}; in catch-branches), \(\alpha\)-renaming of one variable, and whitespace (and some obsolete, commented-out code). (Refactoring could eliminate unnecessary code.) So, for brevity, we only present server-side handshake-traffic key derivation (Listings 25 & 26).

Traffic secrets \textit{client\_handshake\_traffic\_secret} and \textit{server\_handshake\_traffic\_secret} are used to derive handshake-traffic keys that protect handshake extensions (§2.4 & 2.5). After those extensions are processed, application-traffic keys to protect application data can be derived (§2.5.2).

### 2.4 Server parameters: EncryptedExtensions

To request extended functionality, a client may include extensions – beyond those already discussed – in ClientHello messages. Such functionality is not required to establish handshake-traffic keys, hence, those extensions can be encrypted, and a server responds to client requests by including extensions in EncryptedExtensions and Certificate messages. (Appendix A lists all extensions and formally states which extensions can be listed in the extensions field of EncryptedExtensions and Certificate messages, and of other handshake protocol messages.) The former message lists extensions which are not associated with individual certificates, and the latter lists those that are.

An EncryptedExtensions message (which must follow immediately after a ServerHello message) comprises of the following field:

- **extensions**: A list of extensions responding to requests for extended functionalities, i.e., functionalities not required to establish handshake-traffic keys (hence, can be encrypted with such keys), excluding extensions associated with individual certificates.
Listing 25: Class ServerHello.T13ServerHelloProducer (omitted from Listing 9) updates the transcript hash’s digest to include all handshake protocol messages (Line 589), derives an (EC)DHE key (Line 600), and establishes Handshake Secret (Lines 601–602). Variable shc.handshakeKeyExchange is assigned by class KeyShareExtension (PSK-only key exchange is unsupported, hence, ke is not null) as an instance of class SSLKeyExchange parametrised with SSLKeyExchange.T13KeyAgreement (of type SSLKeyAgreement) and ke.createKeyDerivation(shc) returns either ECDHKeyExchange.ecdheKAGenerator.createKeyDerivation(shc) or DHKeyExchange.kaGenerator.createKeyDerivation(shc), i.e., an (EC)DHE key (Line 600). The class also initialises variables kdg (Line 604-605) and kd (Lines 614-615) which will be used to derive traffic secrets and the corresponding traffic keys, respectively. The former is an instance of class SSLTrafficKeyDerivation that overrides method createKeyDerivation such that it returns an instance of class SSLSecretDerivation.T13SecretDerivation.
// update the handshake traffic read keys.
SecretKey readSecret = kd.deriveKey("TlsClientHandshakeTrafficSecret", null);
SSLKeyDerivation readKD =
    kdg.createKeyDerivation(shc, readSecret);
SecretKey readKey = readKD.deriveKey("TlsKey", null);
SecretKey readIvSecret = readKD.deriveKey("TlsIv", null);
IvParameterSpec readIv = new IvParameterSpec(readIvSecret.getEncoded());
SSLReadCipher readCipher;
try {
    readCipher =
        shc.negotiatedCipherSuite.bulkCipher.createReadCipher(
            Authenticator.valueOf(shc.negotiatedProtocol),
            shc.negotiatedProtocol, readKey, readIv,
            shc.sslContext.getSecureRandom());
} catch (GeneralSecurityException gse) {
    // unlikely
}
shc.baseReadSecret = readSecret;
shc.conContext.inputRecord.changeReadCiphers(readCipher);

// update the handshake traffic write secret.
SecretKey writeSecret = kd.deriveKey("TlsServerHandshakeTrafficSecret", null);
SSLKeyDerivation writeKD =
    kdg.createKeyDerivation(shc, writeSecret);
SecretKey writeKey = writeKD.deriveKey("TlsKey", null);
SecretKey writeIvSecret = writeKD.deriveKey("TlsIv", null);
IvParameterSpec writeIv = new IvParameterSpec(writeIvSecret.getEncoded());
SSLWriteCipher writeCipher;
try {
    writeCipher =
        shc.negotiatedCipherSuite.bulkCipher.createWriteCipher(
            Authenticator.valueOf(shc.negotiatedProtocol),
            shc.negotiatedProtocol, writeKey, writeIv,
            shc.sslContext.getSecureRandom());
} catch (GeneralSecurityException gse) {
    // unlikely
}
shc.baseWriteSecret = writeSecret;
shc.conContext.outputRecord.changeWriteCiphers(
    writeCipher, (clientHello.sessionId.length() != 0));

// Update the context for master key derivation.
shc.handshakeKeyDerivation = kd;

Listing 26: Class ServerHello.T13ServerHelloProducer (continued from Listing 25) derives traffic secret client_handshake_traffic_secret (Lines 618–619), constructs an instance of SSLTrafficKeyDerivation.T13TrafficKeyDerivation from that secret (Lines 620–621), and uses that instance to derive the corresponding traffic keys client_write_key (Lines 622–623) and client_write_iv (Lines 624–625), which will be used to decrypt (and read) incoming client traffic (Lines 626–643). Similarly, traffic secret server_handshake_traffic_secret is derived (Lines 646–647), along with traffic keys server_write_key (Lines 650–651) and server_write_iv (Lines 652–653), used to encrypt (and write) outgoing traffic (Lines 654–672).
Each `EncryptedExtensions` message is encrypted using the handshake-traffic key generated from traffic secret `server_handshake_traffic_secret`, as are subsequent handshake messages sent by the server.

`EncryptedExtensions` messages are implemented, produced, and consumed by inner-classes of class `EncryptedExtensions`, namely, inner-classes `EncryptedExtensionsMessage`, `EncryptedExtensionsProducer`, and `EncryptedExtensionsConsumer`, respectively.

### 2.5 Authentication

The handshake protocol concludes with unilateral authentication of the server. (Client authentication is also possible, as discussed in Appendix C). For (EC)DHE-only key exchange, the server must send a `Certificate` message followed by a `CertificateVerify` message (§2.5.1), immediately after an `EncryptedExtensions` message (except when client authentication is requested). Those messages are followed by a `Finished` message (§2.5.2). For PSK-based key exchange, the pre-shared key serves to authenticate the handshake (without certificates), hence, `Certificate` and `CertificateVerify` messages are not sent, and the server only sends a `Finished` message.

#### 2.5.1 Certificate and CertificateVerify

A `Certificate` message contains a certificate (along with its certificate chain) for authentication, and a `CertificateVerify` message contains a signature (using the private key corresponding to the public key in the certificate) over a hash of the handshake protocol’s transcript, thereby, proving possession of the private key used for signing, hence, identifying the server. The former comprises of the following fields:

- `certificate_request_context`: A zero-length identifier. (A `Certificate` message may also be sent in response to a `CertificateRequest` message during post-handshake authentication, as discussed in Appendix C in which case this field echos the identifier used by the `CertificateRequest` message.)

- `certificate_list`: A (non-empty) list of certificates and any associated extensions. Certificates must be DER-encoded X.509v3 certificates, unless an alternative certificate type was negotiated (using extension `server_certificate_type` or `client_certificate_type`). Moreover, the server’s certificate must appear first and every subsequent certificate should certify the previous one (i.e., every subsequent certificate should contain a signature – using the private key corresponding to the certificate’s public key – over the previous certificate’s public key), hence, the list is a certificate chain. That first certificate must be signed using an algorithm amongst those offered by the client (`ClientHello.signature_algorithms`). The remaining certificates must also be, if possible. If not, the chain may rely on algorithms not offered by the client, except for SHA-1, which must not be used, unless offered. (Self-signed certificates or trust anchors may be signed with any algorithm. Furthermore, if raw public keys are negotiated, then the list must contain exactly one certificate.) Any extensions must respond to ones listed in the `CertificateRequest` message.) The entire chain should appear in the first extension listed.

A server’s `Certificate` message is consumed by the client, which aborts with a `decode_error` alert if the `Certificate` message is empty and with a `bad_certificate` alert if a certificate relies on MD5, moreover, it is recommended that a client also aborts with a `bad_certificate` alert if a certificate relies on SHA-1. The client may validate certificates using procedures beyond the scope of TLS. (The TLS 1.3 specification cites RFC 5280 as a reference for validation procedures.)

A `CertificateVerify` message comprises of the following fields:

- `algorithm`: The signing algorithm, which must be amongst those offered by the client (`ClientHello.signature_algorithms`), unless “no valid certificate chain can be produced without unsupported algorithms.”
static final class CertificateEntry {
    final byte[] encoded;  // encoded cert or public key

    private final SSLExtensions extensions;

    CertificateEntry(byte[] encoded, SSLExtensions extensions) {
        this.encoded = encoded;
        this.extensions = extensions;
    }
}

static final class T13CertificateMessage extends HandshakeMessage {
    private final byte[] requestContext;

    T13CertificateMessage(HandshakeContext context, byte[] requestContext, X509Certificate[] certificates) throws SSLException, CertificateException {
        super(context);

        this.requestContext = requestContext.clone();
        this.certEntries = new LinkedList<>();

        for (X509Certificate cert : certificates) {
            byte[] encoded = cert.getEncoded();
            SSLExtensions extensions = new SSLExtensions(this);

            certEntries.add(new CertificateEntry(encoded, extensions));
        }
    }
}

Listing 27: Class CertificateMessage.T13CertificateMessage defines the two fields of a Certificate message (Lines 783–784) and a constructor to instantiate them (Lines 786–798), where the latter field is defined over a list of pairs, comprising a certificate and any associated extensions (Lines 732–777). A further (omitted) constructor is defined to instantiate a Certificate message from an input buffer.

**signature:** A signature computed over the concatenation of: 0x20 repeated 64 times, string “TLS 1.3, server CertificateVerify”, 0x00, and the transcript hash.

A server’s Certificate message is consumed by the client, which aborts with a bad_certificate alert if the signature does not verify.

Certificate and CertificateVerify messages are implemented, produced, and consumed by inner-classes of class CertificateMessage (Listings 27–30) and CertificateVerify (Listings 31–34), respectively.

2.5.2 Finished

The handshake protocol concludes with a Finished message, which provides key confirmation, binds the server’s identity to the exchanged keys (and the client’s identity, if client authentication is used), and, for PSK-based key exchange, authenticates the handshake. A Finished message comprises of the following field:

**verify_data:** An HMAC over the entire handshake.

The HMAC is computed as

$$\text{HMAC}(\text{finished_key}, \text{Transcript-Hash}(\text{Transcript}))$$

where Transcript is a list of handshake protocol messages,

$$\text{finished_key} = \text{HKDF-Expand-Label}(S, “finished”, “”, \text{Hash.length})$$,
static final HandshakeProducer t13HandshakeProducer =
    new T13CertificateProducer();

private static final class T13CertificateProducer implements HandshakeProducer {

    public byte[] produce(ConnectionContext context, HandshakeMessage message) throws IOException {
        HandshakeContext hc = (HandshakeContext)context;
        if (hc.sslConfig.isClientMode) {
            return onProduceCertificate((ClientHandshakeContext)context, message);
        }
        else {
            return onProduceCertificate((ServerHandshakeContext)context, message);
        }
    }

    private byte[] onProduceCertificate(ServerHandshakeContext shc, HandshakeMessage message) throws IOException {
        ClientHelloMessage clientHello = (ClientHelloMessage)message;
        SSLPossession pos = choosePossession(shc, clientHello);
        X509Possession x509Possession = (X509Possession)pos;
        X509Certificate[] localCerts = x509Possession.popCerts;
        // update the context
        shc.handshakePossessions.add(x509Possession);
        shc.handshakeSession.setLocalPrivateKey(x509Possession.popPrivateKey);
        shc.handshakeSession.setLocalCertificates(localCerts);
        T13CertificateMessage cm;
        try {
            cm = new T13CertificateMessage(shc, (new byte[0]), localCerts);
        } catch (SSLException | CertificateException ce) {
            return null; // make the compiler happy
        }
        // Process extensions for each CertificateEntry.
        // Since there can be multiple CertificateEntries within a
        // single CT message, we will pin a specific CertificateEntry
        // into the ServerHandshakeContext so individual extension
        // producers know which X509Certificate it is processing in
        // each call.
        SSLExtension[] enabledCTExts = shc.sslConfig.getEnabledExtensions(
            SSLHandshake.CERTIFICATE,
            Arrays.asList(ProtocolVersion.PROTOCOLS_OF_13));
        for (CertificateEntry certEnt : cm.certEntries) {
            shc.currentCertEntry = certEnt;
            certEnt.extensions.produce(shc, enabledCTExts);
        }
        // Output the handshake message.
        cm.write(shc.handshakeOutput);
        shc.handshakeOutput.flush();
        // The handshake message has been delivered.
        return null;
    }
}

Listing 28: Class CertificateMessage.T13CertificateProducer defines method produce to write
(to an output stream) a Certificate message, originating from a client (Lines 931–932) or
server (Lines 934–935). For the latter, a private key and authenticating certificates are wrapped
inside an instance of class X509Authentication.X509Possession (Lines 943–955), using method
choosePossession (Listing 29); the server’s active context is updated to include that private key
and associated certificates (Lines 964–967); a Certificate message is constructed from the
certificates (Lines 968–975); and the message is written to an output stream (Lines 1002–1003).
private static SSLPossession choosePossession(
    HandshakeContext hc,
    ClientHelloMessage clientHello) throws IOException {
    for (SignatureScheme ss : hc.peerRequestedCertSignSchemes) {
        // Don't select a signature scheme unless we will be able to
        // produce a CertificateVerify message later
        if (SignatureScheme.getPreferableAlgorithm(
            hc.peerRequestedSignatureSchemes,
            ss, hc.negotiatedProtocol) == null) {
            continue;
        }
        SSLAuthentication ka = X509Authentication.valueOf(ss);
        if (ka == null) {
            continue;
        }
        SSLPossession pos = ka.createPossession(hc);
        if (pos == null) {
            continue;
        }
        return pos;
    }
    return null;
}

Listing 29: Class CertificateMessage.T13CertificateProducer (omitted from Listing 28) defines method choosePossession to iterate over the client offered signature algorithms for certificates (defined by extension signature_algorithms_cert, or signature_algorithms if the former is absent), which class CertSignAlgsExtension.CHCertSignatureSchemesUpdate (respectively SignatureAlgorithmsExtension.CHSignatureSchemesUpdate) assigns to variable hc.peerRequestedCertSignSchemes; disregard algorithms not offered for signing CertificateVerify requests (Lines 1033–1044), unsupported algorithms (Lines 1046–1054), or algorithms for which no suitable private key is available (1056–1063); and return a private key for the first suitable algorithm (Line 1065), or null if no such key exists (Line 1071).
static final SSLConsumer t13HandshakeConsumer =
   new T13CertificateConsumer();

private static final class T13CertificateConsumer implements SSLConsumer {
   public void consume(ConnectionContext context, ByteBuffer message)
      throws IOException {
      HandshakeContext hc = (HandshakeContext)context;
      hc.handshakeConsumers.remove(SSLHandshake.CERTIFICATE.id);
      T13CertificateMessage cm = new T13CertificateMessage(hc, message);
      if (hc.sslConfig.isClientMode) {
         onConsumeCertificate((ClientHandshakeContext)context, cm);
      } else {
         onConsumeCertificate((ServerHandshakeContext)context, cm);
      }
   }

   private void onConsumeCertificate(ClientHandshakeContext chc,
       T13CertificateMessage certificateMessage) throws IOException {
      // Each CertificateEntry will have its own set of extensions
      // which must be consumed.
      SSLExtension[] enabledExtensions =
         chc.sslConfig.getEnabledExtensions(SSLHandshake.CERTIFICATE);
      for (CertificateEntry certEnt : certificateMessage.certEntries) {
         certEnt.extensions.consumeOnLoad(chc, enabledExtensions);
      }
      // check server certificate entries
      X509Certificate[] srvCerts =
         checkServerCerts(chc, certificateMessage.certEntries);
      // update
c   chc.handshakeCredentials.add(
      new X509Credentials(srvCerts[0].getPublicKey(), srvCerts));
      chc.handshakeSession.setPeerCertificates(srvCerts);
   }
}

Listing 30: Class CertificateMessage.T13CertificateConsumer defines method consume to instantiate a Certificate message from an input buffer (Line 1145) and consume the message as originating from a server (Line 1151) or client (Lines 1157). For the former, certificates are checked (Lines 1203-1204) and the active context is updated (Lines 1209-1211).
static final class T13CertificateVerifyMessage extends HandshakeMessage {
    private static final byte[] serverSignHead = new byte[] {
        // repeated 0x20 for 64 times
        // "TLS 1.3, server CertificateVerify" + 0x00
    };
    private static final byte[] clientSignHead = new byte[] {
        // repeated 0x20 for 64 times
        // "TLS 1.3, client CertificateVerify" + 0x00
    };
    // the signature algorithm
    private final SignatureScheme signatureScheme;
    // signature bytes
    private final byte[] signature;

    T13CertificateVerifyMessage(HandshakeContext context, X509Possession x509Possession) throws IOException {
        super(context);

        this.signatureScheme = SignatureScheme.getPreferableAlgorithm(
            context.peerRequestedSignatureSchemes,
            x509Possession.popPrivateKey,
            context.negotiatedProtocol);

        byte[] hashValue = context.handshakeHash.digest();
        byte[] contentCovered;
        if (context.sslConfig.isClientMode) {
            contentCovered = Arrays.copyOf(clientSignHead,
                clientSignHead.length + hashValue.length);
            System.arraycopy(hashValue, 0, contentCovered,
                clientSignHead.length, hashValue.length);
        } else {
            contentCovered = Arrays.copyOf(serverSignHead,
                serverSignHead.length + hashValue.length);
            System.arraycopy(hashValue, 0, contentCovered,
                serverSignHead.length, hashValue.length);
        }

        byte[] temporary = null;
        try {
            Signature signer =
                signatureScheme.getSignature(x509Possession.popPrivateKey);
            signer.update(contentCovered);
            temporary = signer.sign();
        } catch (NoSuchAlgorithmException | InvalidAlgorithmParameterException nsae) {
            context.conContext.fatal(Alert.INTERNAL_ERROR,
                "Unsupported_signature_algorithm/" +
                signatureScheme.name +
                ", used in CertificateVerify_handshake_message", nsae);
        } catch (InvalidKeyException | SignatureException ikse) {
            context.conContext.fatal(Alert.HANDSHAKE_FAILURE,
                "Cannot_produce_CertificateVerify_signature", ikse);
        }

        this.signature = temporary;
    }
}

Listing 31: Class CertificateVerify. T13CertificateVerifyMessage defines the two fields of a CertificateVerify message (Lines 858 & 861) and constructors to instantiate them from parameters (Lines 863–910) or an input buffer (Listing 32). The former instantiates the first field with the chosen signature algorithm (Lines 867–870); derives the string over which to compute the signature (Lines 878–890), using constant serverSignHead (Lines 764–823) for messages originating from a server, and constant clientSignHead (Lines 825–854) for messages originating from a client, where bytes used to construct those contents are omitted for brevity; and instantiates the second field as a signature over that string (Lines 892–909).
Listing 32: Class CertificateVerify. T13CertificateVerifyMessage (omitted from Listing 31) defines
a constructor which instantiates a CertificateVerify message from an input buffer, parametrizing
the first field with the chosen signature algorithm (Lines 926–927) and the second with the
signature (Line 957), if the signature verifies (Lines 974–980) with respect to the expected string
(Lines 959–971).
static final HandshakeProducer t13HandshakeProducer =
        new T13CertificateVerifyProducer();

private static final
class T13CertificateVerifyProducer implements HandshakeProducer {
    public byte[] produce(ConnectionContext context,
                           HandshakeMessage message) throws IOException {
        // The producing happens in handshake context only.
        HandshakeContext hc = (HandshakeContext)context;
        X509Possession x509Possession = null;
        for (SSLPossession possession : hc.handshakePossessions) {
            if (possession instanceof X509Possession) {
                x509Possession = (X509Possession) possession;
                break;
            }
        }
        if (hc.sslConfig.isClientMode) {
            return onProduceCertificateVerify((ClientHandshakeContext)context, x509Possession);
        } else {
            return onProduceCertificateVerify((ServerHandshakeContext)context, x509Possession);
        }
    }

    private byte[] onProduceCertificateVerify(ServerHandshakeContext shc, X509Possession x509Possession) throws IOException {
        T13CertificateVerifyMessage cvm =
                new T13CertificateVerifyMessage(shc, x509Possession);
        // Output the handshake message.
        cvm.write(shc.handshakeOutput);
        shc.handshakeOutput.flush();
        // The handshake message has been delivered.
        return null;
    }
}

Listing 33: Class CertificateVerify.T13CertificateVerifyProducer defines method produce to write (to an output stream) a CertificateVerify message, originating from a client (Lines 1067–1068) or server (Lines 1070–1071). For the latter, a CertificateVerify message is constructed (Lines 1077-1078) and written to an output stream (Lines 1085-1086).

static final SSLConsumer t13HandshakeConsumer =
                new T13CertificateVerifyConsumer();

private static final
class T13CertificateVerifyConsumer implements SSLConsumer {
    public void consume(ConnectionContext context,
                         ByteBuffer message) throws IOException {
        HandshakeContext hc = (HandshakeContext)context;
        T13CertificateVerifyMessage cvm =
                new T13CertificateVerifyMessage(hc, message);
    }
}

Listing 34: Class CertificateVerify.T13CertificateVerifyConsumer defines method consume to instantiate a CertificateVerify message from an input buffer (Line 1125–1126), checking validity of the message’s signature as a side effect.
2 HANDSHAKE PROTOCOL

and traffic secret $S$ is `server_handshake_traffic_secret` when the `Finished` message originates from a server to conclude an initial handshake, `client_handshake_traffic_secret` when originating from a client to conclude an initial handshake, and `client_application_traffic_secret_N` when concluding post-handshake authentication.

A `Finished` message is first sent by the server (immediately after a `CertificateVerify` message for (EC)DHE-only key exchange and immediately after an `EncryptedExtensions` message for PSK-based key exchange). That message is consumed by the client, which recomputes the HMAC (using secret `server_handshake_traffic_secret`) and checks that it matches the `Finished` message’s HMAC (`Finished.verify_data`), terminating the connection with a `decrypt_error` alert if the check fails. A client that successfully consumes a server’s `Finished` message responds with its own `Finished` message, which is similarly consumed by the server (albeit using secret `client_handshake_traffic_secret`). (That message is preceded by client generated `Certificate` and `CertificateVerify` messages, if client authentication is used.) Once endpoints have successfully consumed `Finished` messages, (encrypted) application data may be exchanged. Moreover, a server may send (encrypted) application data immediately after sending its `Finished` message (i.e., without consuming a `Finished` message), albeit, since `ClientHello` messages may be replayed, any such data is sent without assurance of the client’s liveness (nor identity).

```
Finished messages are implemented, produced, and consumed by inner-classes of class Finished (Listings 35–42).
```

Traffic secrets `server_application_traffic_secret_0` and `client_application_traffic_secret_0` are used to derive application-traffic keys to protect application data.

2.6 Further features

2.6.1 NewSessionTicket

After receiving a client’s `Finished` message, a server can initiate establishment of a new pre-shared key, which will be derived from the resumption master secret `resumption_master_secret` (Figure 2). Such a pre-shared key may be used to establish subsequent channels. Establishment is initiated with a `NewSessionTicket` message, comprising the following fields:

- `ticket_lifetime`: A 32-bit unsigned integer indicating the lifetime in seconds of the pre-shared key, which must not exceed seven days (604800 seconds).
- `ticket_age_add`: A 32-bit nonce to obscure the lifetime.
- `ticket_nonce`: A nonce for key derivation.
- `ticket`: A key identifier.
- `extensions`: A list of extensions.

The associated pre-shared key is computed as follows:

```
HKDF-Expand-Label(resumption_master_secret, "resumption", ticket_nonce, Hash.length),
```

Since the pre-shared key is computed from nonce `ticket_nonce`, it follows that each `NewSessionTicket` message creates a distinct pre-shared key.

A `NewSessionTicket` is consumed by the client, which derives and stores the pre-shared key along with associated data. That data may be used in extension `pre_shared_key` of subsequent `ClientHello` messages.
private static final class FinishedMessage extends HandshakeMessage {
    private final byte[] verifyData;
    FinishedMessage(HandshakeContext context) throws IOException {
        super(context);
        VerifyDataScheme vds = VerifyDataScheme.valueOf(context.negotiatedProtocol);
        byte[] vd = null;
        try {
            vd = vds.createVerifyData(context, false);
        } catch (IOException ioe) {
            context.conContext.fatal(Alert.INTERNAL_ERROR, "Failed to generate verify data", ioe);
        }
        this.verifyData = vd;
    }
    FinishedMessage(HandshakeContext context, ByteBuffer m) throws IOException {
        super(context);
        int verifyDataLen = 12;
        if (context.negotiatedProtocol == ProtocolVersion.SSL30) {
            verifyDataLen = 36;
        } else if (context.negotiatedProtocol.useTLS13PlusSpec()) {
            verifyDataLen = context.negotiatedCipherSuite.hashAlg.hashLength;
        }
        this.verifyData = new byte[verifyDataLen];
        m.get(verifyData);
        VerifyDataScheme vd = VerifyDataScheme.valueOf(context.negotiatedProtocol);
        byte[] myVerifyData;
        try {
            myVerifyData = vd.createVerifyData(context, true);
        } catch (IOException ioe) {
            return; // make the compiler happy
        }
        if (!MessageDigest.isEqual(myVerifyData, verifyData)) {
            context.conContext.fatal(Alert.INTERNAL_ERROR, "The Finished message cannot be verified.");
        }
    }
}

Listing 35: Class Finished.FinishedMessage defines the one field of a Finished message (Line 70) and two constructors to instantiate it. The first constructor parametrises the field with an HMAC it constructs (Lines 72-87) and the second parses an HMAC from an input buffer (Lines 92-107), recomputes the expected HMAC itself (Lines 109-118), and checks that the HMACs match (Lines 119-122). The HMACs are (indirectly) computed using method T13VerifyDataGenerator.createVerifyData (Listing 36).
Listing 36: Class Finished.T13VerifyDataGenerator defines method createVerifyData to compute HMACs for Finished messages. That method computes variable finishedSecret by indirect application of method HKDF.expand to inputs including secret context.baseReadSecret or context.baseWriteSecret, and HkdfLabel, which is computed over the negotiated hash function’s output length, label “tls13_finished”, and null ASCII character 0x00, using class SSLBasicKeyDerivation to apply method HKDF.expand. (Class SSLBasicKeyDerivation uses method createHkdfInfo to handle concatenation and is reliant on method Record.putBytes8 to introduce 0x00. This differs from a similar application of method HKDF.expand by class SSLTrafficKeyDerivation.T13TrafficKeyDerivation, which introduces 0x00 itself.)
static final HandshakeProducer t13HandshakeProducer =
   new T13FinishedProducer();

private static final
class T13FinishedProducer implements HandshakeProducer {
   public byte[] produce(ConnectionContext context,
      HandshakeMessage message) throws IOException {
      HandshakeContext hc = (HandshakeContext)context;
      if (hc.sslConfig.isClientMode) {
         return onProduceFinished((ClientHandshakeContext)context, message);
      } else {
         return onProduceFinished((ServerHandshakeContext)context, message);
      }
   }

   private byte[] onProduceFinished(ServerHandshakeContext shc,
      HandshakeMessage message) throws IOException {
      // Refresh handshake hash
      shc.handshakeHash.update();
      FinishedMessage fm = new FinishedMessage(shc);
      // Output the handshake message.
      fm.write(shc.handshakeOutput);
      shc.handshakeOutput.flush();
      // Change client/server application traffic secrets.
      SSLKeyDerivation kd = shc.handshakeKeyDerivation;
      SSLTrafficKeyDerivation kdg = SSLTrafficKeyDerivation.valueOf(shc_negotiatedProtocol);
      // derive salt secret
      try {
         SecretKey saltSecret = kd.deriveKey("TlsSaltSecret", null);
         // derive application secrets
         HashAlg hashAlg = shc_negotiatedCipherSuite.hashAlg;
         HKDF hkdf = new HKDF(hashAlg.name);
         byte[] zeros = new byte[hashAlg.hashLength];
         SecretKeySpec sharedSecret =
            new SecretKeySpec(zeros, "TlsZeroSecret");
         SecretKey masterSecret =
            hkdf.extract(saltSecret, sharedSecret, "TlsMasterSecret");
      }
   }

Listing 37: Class Finished.T13FinishedProducer defines method produce to write (to an output stream) a Finished message, originating from a client or a server, and to establish Master Secret when the message originates from such a server. For messages originating from servers, processing proceeds with method onProduceFinished, parametrised by the server’s active context. That method updates the transcript hash’s digest to include all handshake protocol messages (Line 741), instantiates and outputs a Finished message (Lines 743–751), and establishes Master Secret (Lines 782–783). Variable shc.handshakeKeyDerivation (Line 754) is assigned by class ServerHello.T13ServerHelloProducer (Listing 27) as an instance of class SSLSecretDerivation, parametrised by Handshake Secret, hence, the salt necessary to establish Master Secret is correctly derived (Line 774), as-is the necessary Hash.length-length string of zeros (Lines 780–781).
SSLKeyDerivation secretKD =
    new SSLSecretDerivation(shc, masterSecret);

    // update the handshake traffic write keys.
    SecretKey writeSecret = secretKD.deriveKey("TlsServerAppTrafficSecret", null);
    SSLKeyDerivation writeKD =
        kdg.createKeyDerivation(shc, writeSecret);
    SecretKey writeKey = writeKD.deriveKey("TlsKey", null);
    SecretKey writeIvSecret = writeKD.deriveKey("TlsIv", null);
    IvParameterSpec writeIv =
        new IvParameterSpec(writeIvSecret.getEncoded());
    SSLWriteCipher writeCipher =
        shc.negotiatedCipherSuite.bulkCipher.createWriteCipher(  
            Authenticator.valueOf(shc.negotiatedProtocol),
            shc.negotiatedProtocol, writeKey, writeIv,
            shc.sslContext.getSecureRandom());

    shc.baseWriteSecret = writeSecret;
    shc.conContext.outputRecord.changeWriteCiphers(
        writeCipher, false);

    // update the context for the following key derivation
    shc.handshakeKeyDerivation = secretKD;
} catch (GeneralSecurityException gse) {
    return null; // make the compiler happy
}

    // update the context
    shc.handshakeConsumers.put(
        SSLHandshake.FINISHED.id, SSLHandshake.FINISHED);

    // The handshake message has been delivered.
    return null;
}
static final SSLConsumer t13HandshakeConsumer =
    new T13FinishedConsumer();

private static final class T13FinishedConsumer implements SSLConsumer {
    public void consume(ConnectionContext context,
        ByteBuffer message) throws IOException {
        HandshakeContext hc = (HandshakeContext)context;
        if (hc.sslConfig.isClientMode) {
            onConsumeFinished(
                (ClientHandshakeContext)context, message);
        } else {
            onConsumeFinished(
                (ServerHandshakeContext)context, message);
        }
    }

    private void onConsumeFinished(ClientHandshakeContext chc,
        ByteBuffer message) throws IOException {
        // update
        // A change_cipher_spec record received after the peer's Finished
        // message MUST be treated as an unexpected record type.
        chc.conContext.consumers.remove(ContentType.CHANGE_CIPHER_SPEC.id);
        // Change client/server application traffic secrets.
        // Refresh handshake hash
        chc.handshakeHash.update();
        SSLKeyDerivation kd = chc.handshakeKeyDerivation;
        SSLTrafficKeyDerivation kdg =
            SSLTrafficKeyDerivation.valueOf(chc.negotiatedProtocol);
        // derive salt secret
        try {
            SecretKey saltSecret = kd.deriveKey("TlsSaltSecret", null);
            // derive application secrets
            HashAlg hashAlg = chc.negotiatedCipherSuite.hashAlg;
            HKDF hkdf = new HKDF(hashAlg.name);
            byte[] zeros = new byte[hashAlg.hashLength];
            SecretKeySpec sharedSecret =
                new SecretKeySpec(zeros, "TlsZeroSecret");
            SecretKey masterSecret =
                hkdf.extract(saltSecret, sharedSecret, "TlsMasterSecret");
        }
    }
}

Listing 39: Class Finished.T13FinishedConsumer defines method consume to read (from an input buffer) a Finished message, originating from a client or a server, and to establish Master Secret when the message originates from such a server. For messages originating from servers, processing proceeds with method onConsumeFinished, parametrised by the client’s active context. That method instantiates a Finished message from the input buffer (Line 858), updates the transcript hash’s digest to include all handshake protocol messages (Line 883), and establishes Master Secret (Lines 919–920). (Computations are similar to Listing 37 and refactoring could eliminate unnecessary code.)
SSLKeyDerivation secretKD =
   new SSLSecretDerivation(chc, masterSecret);

   // update the handshake traffic read keys.
SecretKey readSecret = secretKD.deriveKey(
   "TlsServerAppTrafficSecret", null);
SSLKeyDerivation writeKD =
   kdg.createKeyDerivation(chc, readSecret);
SecretKey readKey = writeKD.deriveKey(
   "TlsKey", null);
SecretKey readIvSecret = writeKD.deriveKey(
   "TlsIv", null);
IvParameterSpec readIv =
   new IvParameterSpec(readIvSecret.getEncoded());
SSLReadCipher readCipher =
   chc.negotiatedCipherSuite.bulkCipher.createReadCipher(
      Authenticator.valueOf(chc.negotiatedProtocol),
      chc.negotiatedProtocol, readKey, readIv,
      chc.sslContext.getSecureRandom());

   chc.baseReadSecret = readSecret;
   chc.conContext.inputRecord.changeReadCiphers(readCipher);

   // update the context for the following key derivation
   chc.handshakeKeyDerivation = secretKD;
   }
   catch (GeneralSecurityException gse) {
   return;   // make the compiler happy
   }

   // produce
   //
   chc.handshakeProducers.put(SSLHandshake.FINISHED.id,
      SSLHandshake.FINISHED);
SSLHandshake[] probableHandshakeMessages = new SSLHandshake[] {
   // full handshake messages
   SSLHandshake.CERTIFICATE,
   SSLHandshake.CERTIFICATE_VERIFY,
   SSLHandshake.FINISHED
};
   for (SSLHandshake hs : probableHandshakeMessages) {
   HandshakeProducer handshakeProducer =
      chc.handshakeProducers.remove(hs.id);
   if (handshakeProducer != null) {
   handshakeProducer.produce(chc, null);
   }
   }

Listing 40: Class Finished.T13FinishedConsumer defines method consume (continued from Listing 39) to derive traffic secret server_application_traffic_secret_0 (Lines 922–927) and corresponding traffic keys server_write_key (Lines 930–931) and server_write_iv (Lines 932–933), used to decrypt (and read) incoming traffic (Lines 934–943). (Computations are similar to Listing 38 and refactoring could eliminate unnecessary code.) Moreover, the method updates the client’s active context to include a producer for Finished messages (Lines 956–957); constructs an array of producers clients might use during the remainder of the handshake protocol, namely, produces for messages Certificate, CertificateVerify, and Finished, in the order that they might be used (Lines 958–963); and uses those producers to produce messages when the active context includes the producer (Lines 965–971). Since a Finished message producer is included, a Finished message is always produced, using class Finished.T13FinishedProducer (Listing 37 & 41).
private byte[] onProduceFinished(ClientHandshakeContext chc,
HandshakeMessage message) throws IOException {
    // Refresh handshake hash
    chc.handshakeHash.update();
    FinishedMessage fm = new FinishedMessage(chc);
    // Output the handshake message.
    fm.write(chc.handshakeOutput);
    chc.handshakeOutput.flush();
    // Change client/server application traffic secrets.
    SSLKeyDerivation kd = chc.handshakeKeyDerivation;
    SSLTrafficKeyDerivation kdg =
        SSLTrafficKeyDerivation.valueOf(chc.negotiatedProtocol);
    try {
        // update the application traffic read keys.
        SecretKey writeSecret = kd.deriveKey(
            "TlsClientAppTrafficSecret", null);
        SSLKeyDerivation writeKD =
            kdg.createKeyDerivation(chc, writeSecret);
        SecretKey writeKey = writeKD.deriveKey(
            "TlsKey", null);
        SecretKey writeIvSecret = writeKD.deriveKey(
            "TlsIv", null);
        IvParameterSpec writeIv =
            new IvParameterSpec(writeIvSecret.getEncoded());
        SSLWriteCipher writeCipher =
            chc.negotiatedCipherSuite.bulkCipher.createWriteCipher(
                Authenticator.valueOf(chc.negotiatedProtocol),
                chc.negotiatedProtocol, writeKey, writeIv,
                chc.sslContext.getSecureRandom());
        chc.baseWriteSecret = writeSecret;
        chc.conContext.outputRecord.changeWriteCiphers(
            writeCipher, false);
    } catch (GeneralSecurityException gse) {
        return null;  // make the compiler happy
    }
    // The handshake message has been delivered.
    return null;
}

Listing 41: Class Finished.T13FinishedProducer defines method onProduceFinished parametrised by a client’s active context (omitted from Listing 37) to write (to an output stream) a Finished message originating from a client, and to derive traffic secret client_application_traffic_secret_0 (Lines 692–693) and corresponding traffic keys client_write_key (Lines 698–699) and client_write_iv (Lines 700–701), used to encrypt (and write) outgoing traffic (Lines 702–712).
private void onConsumeFinished(ServerHandshakeContext shc,
    ByteBuffer message) throws IOException {
    FinishedMessage fm = new FinishedMessage(shc, message);
    // update
    // Change client/server application traffic secrets.
    SSLKeyDerivation kd = shc.handshakeKeyDerivation;
    SSLTrafficKeyDerivation kdg =
        SSLTrafficKeyDerivation.valueOf(shc.negotiatedProtocol);
    try {
        // update the application traffic read keys.
        SecretKey readSecret = kd.deriveKey(
            "TlsClientAppTrafficSecret", null);
        SSLKeyDerivation readKD =
            kdg.createKeyDerivation(shc, readSecret);
        SecretKey readKey = readKD.deriveKey(
            "TlsKey", null);
        SecretKey readIvSecret = readKD.deriveKey(
            "TlsIv", null);
        IvParameterSpec readIv =
            new IvParameterSpec(readIvSecret.getEncoded());
        SSLReadCipher readCipher =
            shc.negotiatedCipherSuite.bulkCipher.createReadCipher(
                Authenticator.valueOf(shc.negotiatedProtocol),
                shc.negotiatedProtocol, readKey, readIv,
                shc.sslContext.getSecureRandom());
        shc.baseReadSecret = readSecret;
        shc.conContext.inputRecord.changeReadCiphers(readCipher);
    } catch (GeneralSecurityException gse) {
        return; // make the compiler happy
    }
}

Listing 42: Class Finished.T13FinishedConsumer defines method onConsumeFinished parametrised by a server’s active context (omitted from Listing 39) to read (from an input buffer) a Finished message originating from a client, and to derive traffic secret client_application_traffic_secret_0 (Lines 1022–1023) and corresponding traffic keys client_write_key (Lines 1027–1028) and client_write_iv (Lines 1029–1030), used to decrypt (and read) incoming traffic (Lines 1031–1040). (Computations are similar to Listing 41 and refactoring could eliminate unnecessary code.)
**NewSessionTicket** messages are implemented, produced, and consumed by inner-classes of class NewSessionTicket. Those classes define the five fields of a **NewSessionTicket** message and constructors to instantiate them from parameters or an input buffer, moreover, they define methods to write such a message to an output stream and to read such a message from an input buffer.

**Extension pre_shared_key**

The pre-shared key identifiers listed by extension **pre_shared_key** (§2.1) may include identifiers established by **NewSessionTicket** messages, identifiers established out-of-band, or both. Each identifier is coupled with an obfuscated age, which is derived by addition of **NewSessionTicket.ticket_lifetime** and **NewSessionTicket.ticket_age_add** (modulo $2^{32}$) for identifiers established by **NewSessionTicket** messages, and 0 for identifiers established out-of-band. Moreover, extension **pre_shared_key** associates each identifier with a PSK binder, which binds the pre-shared key with the current handshake, and to the session in which the pre-shared key was generated for pre-shared keys established by **NewSessionTicket** messages. The PSK binder is computed as an HMAC over a partial transcript, which excludes binders, namely,

$$\text{HMAC}(\text{binder_key}, \text{Transcript-Hash}((\text{Truncate}(CH))))$$

for transcripts which include only a single **ClientHello** message $CH$, and

$$\text{HMAC}(\text{binder_key}, \text{Transcript-Hash}((CH, HRR, \text{Truncate}(CH'))))$$

for transcripts that include an initial **ClientHello** message $CH$, followed by a **HelloRetryRequest** message $HRR$ and a subsequent **ClientHello** message $CH'$, where function $\text{Truncate}$ removes binders. When consuming a **ClientHello** message that includes extension **pre_shared_key**, a server recomputes the HMAC for their selected pre-shared key and checks that it matches the corresponding binder listed by the extension, aborting if the check fails or the binder is not present.

**Extension pre_shared_key** is implemented, produced, and consumed by inner-classes of class PreSharedKeyExtension. Those classes define fields of extension **pre_shared_key** and constructors to instantiate them from parameters or an input buffer, moreover, they define methods to write such an extension to an output stream and to read such an extension from an input buffer, the latter is reliant on static methods checkBinder, computeBinder and deriveBinderKey to recompute HMACs for pre-shared keys and to check whether they match the corresponding binder listed by the extension.

### 2.6.2 KeyUpdate

After sending a **Finished** message, an endpoint may send a **KeyUpdate** message to notify their peer that they are updating their cryptographic key. The message comprises of the following field:

**request_update**: A bit indicating whether the recipient should respond with their own **KeyUpdate** message and update their own cryptographic key.

After sending a **KeyUpdate** message, the sender must update their application-traffic secret and corresponding application-traffic keys (§2.3.2–2.3.3). A peer that receives a **KeyUpdate** message prior to receiving a **Finished** message aborts with an **unexpected_message** alert, moreover, the peer aborts with an **illegal_parameter** alert if field **request_update** does not contain a bit. Otherwise, the peer updates its receiving keys (§2.3.2–2.3.3). Moreover, when the sender requests that the peer updates their sending keys, the peer must
send a KeyUpdate message of its own (without requesting that the sender update its cryptographic key), prior to sending any further application data.

KeyUpdate messages are implemented, produced, and consumed by inner-classes of class KeyUpdate. Those classes define the one field of a NewSessionTicket message and constructors to instantiate it from parameters or an input buffer, moreover, they define methods to write such a message to an output stream and to read such a message from an input buffer, those methods also update application-traffic secrets and corresponding application-traffic keys, and the latter produces a KeyUpdate message of the receiver when requested by the sender.

2.6.3 Early data

For PSK-based key exchange, clients may send encrypted application data immediately after ClientHello messages (i.e., before receiving a ServerHello message), enabling a zero round-trip time (0-RTT), at the cost of forward secrecy (since application data is solely encrypted by the pre-shared key, which does not afford forward secrecy, as per PSK-only key exchange) and replay protection (since such protection is derived from the server’s nonce, which is constructed after encrypted application data is sent). Such early data requires the ClientHello message to include extension early_data and application data must be encrypted using the client’s first identified pre-shared key. We omit further discussion of early data from this document.

3 Record protocol

Handshake messages are encapsulated into one or more TLSPlaintext records (§3.1), which, for ClientHello, ServerHello and HelloRetryRequest messages, are immediately written to the transport layer, otherwise, TLSPlaintext records are translated to TLSCiphertext records (§3.2), which add protection prior to writing to the transport layer. (Alerts are similarly encapsulated and when appropriate protected. Application data is always encapsulated and protected.)

3.1 TLSPlaintext

Handshake messages are fragmented and each fragment is encapsulated into a TLSPlaintext record, comprising the following fields:

- **type**: Constant 0x22 (handshake). (Other constants are used for records encapsulating data other than handshake messages, e.g., alerts and application data.)

- **legacy_record_version**: Constant 0x0303, except for an initial ClientHello message, which may use constant 0x0301.

- **length**: The byte length of the data fragment, which must not exceed $2^{14}$ bytes.

- **fragment**: The data fragment.

An endpoint that receives a TLSPlaintext record with field length set greater than $2^{14}$ must abort with a record_overflow alert.

3.2 TLSCiphertext

For protection, a TLSPlaintext record is transformed into a TLSCiphertext record, comprising of the following fields:

- **opaque_type**: Constant 0x23.
legacy_record_version: Constant 0x0303.

length: The byte length of the encrypted data including headers, which must not exceed $2^{14} + 256$ bytes.

encrypted_record: The encrypted data.

Encrypted data is computed, using the negotiated AEAD algorithm, as

$$AEAD-Encrypt(write_key, nonce, additional_data, plaintext),$$

where $write_key$ is either $client_write_key$ or $server_write_key$; $nonce$ is derived from a sequence number XORed with $client_write_iv$ or $server_write_iv$, respectively; $additional_data$ is the TLSCiphertext record header, i.e., $additional_data = TLSCiphertext.opaque_type || TLSCiphertext.legacy_record_version || TLSCiphertext.length$; and $plaintext$ comprises of TLSPlainText.fragment prepended with the following fields, namely, $type$ containing $TLSPlainText.type$ and $zeros$ containing an arbitrary-length run of zero-valued bytes, used to pad a TLS record (the resulting plaintext is known as record $TLSInnerPlaintext$).

An endpoint that receives a TLSCiphertext record with field $length$ set greater than $2^{14} + 256$ must abort with a record_overflow alert. Otherwise, the endpoint computes

$$AEAD-Decrypt(write_key, nonce, additional_data, TLSCiphertext.encrypted_record),$$

which outputs a plaintext or terminates with an error. The endpoint aborts with a bad_record_mac alert in the event of such an error.

Per-record nonce. The nonce used by the negotiated AEAD algorithm is derived from a 64-bit sequence number, which is initialised as 0, incremented by one after reading or writing a record, and reset to 0 whenever the key is changed. That sequence number is XORed with $client_write_iv$ or $server_write_iv$ to derive the nonce.

Outgoing records are constructed by class SSLSocketOutputRecord (Listing 43) and parent OutputRecord (Listing 44). (Alternatively, outgoing records are constructed by class SSLEngineOutputRecord, which shares the same parent.) Incoming records are processed by class SSLSocketInputRecord (or SSLEngineInputRecord) and parent InputRecord.

A Extensions

Table 3 formally specifies which extensions can be listed in the extensions field of handshake protocol messages. Endpoints must abort with an illegal_parameter alert if an extension is received in a handshake protocol message for which it is not specified.
final class SSLSocketOutputRecord extends OutputRecord implements SSLRecord {
    private OutputStream deliverStream = null;
    synchronized void encodeHandshake(byte[] source, int offset, int length) throws IOException {
        byte handshakeType = source[0];
        if (handshakeHash.isHashable(handshakeType)) {
            handshakeHash.deliver(source, offset, length);
        }

        int fragLimit = getFragLimit();
        int position = headerSize + writeCipher.getExplicitNonceSize();
        if (count == 0) {
            count = position;
        }

        if ((count - position) < (fragLimit - length)) {
            write(source, offset, length);
            return;
        }

        for (int limit = (offset + length); offset < limit;) {
            int remains = (limit - offset) + (count - position);
            int fragLen = Math.min(fragLimit, remains);
            // use the buf of ByteArrayOutputStream
            write(source, offset, fragLen);
            if (remains < fragLimit) {
                return;
            }

            // Encrypt the fragment and wrap up a record.
            encrypt(writeCipher, ContentType.HANDSHAKE.id, headerSize);
            // deliver this message
            deliverStream.write(buf, 0, count);  // may throw IOException
            deliverStream.flush();  // may throw IOException
            // reset the offset
            offset += fragLen;
            // reset the internal buffer
            count = position;
        }
    }
}

Listing 43: Class SSLSocketOutputRecord writes outgoing data to buffer buf (Lines 159 & 169), using method ByteArrayOutputStream.write, which (if full) is processed by parent OutputRecord (Line 182) and delivered (Lines 185–186). The class is also responsible for adding the encapsulated message to the transcript hash, if appropriate (Lines 147–150).
abstract class OutputRecord
extends ByteArrayOutputStream implements Record, Closeable {
long encrypt(
    SSLWriteCipher encCipher, byte contentType, int headerSize) {
    return t13Encrypt(encCipher, contentType, headerSize);
}

private static final class T13PaddingHolder {
    private static final byte[] zeros = new byte[16];
}

private long t13Encrypt(
    SSLWriteCipher encCipher, byte contentType, int headerSize) {
    if (!encCipher.isNullCipher()) {
        // inner plaintext
        write(contentType);
        write(T13PaddingHolder.zeros, 0, T13PaddingHolder.zeros.length);
    }
    byte[] sequenceNumber = encCipher.authenticator.sequenceNumber();
    int position = headerSize;
    int contentLen = count - position;

    // ensure the capacity
    int requiredPacketSize =
        encCipher.calculatePacketSize(contentLen, headerSize);
    if (requiredPacketSize > buf.length) {
        byte[] newBuf = new byte[requiredPacketSize];
        System.arraycopy(buf, 0, newBuf, 0, count);
        buf = newBuf;
    }

    // use the right TLS.Ciphertext.opaque_type and legacy_record_version
    ProtocolVersion pv = protocolVersion;
    if (!encCipher.isNullCipher()) {
        pv = ProtocolVersion.TLS12;
        contentType = ContentType.APPLICATION_DATA.id;
    } else {
        pv = ProtocolVersion.TLS12;
    }

    ByteBuffer destination = ByteBuffer.wrap(buf, position, contentLen);
    count = headerSize + encCipher.encrypt(contentType, destination);

    // Fill out the header, write it and the message.
    int fragLen = count - headerSize;
    buf[0] = contentType;
    buf[1] = pv.major;
    buf[2] = pv.minor;
    buf[3] = (byte)((fragLen >> 8) & 0xFF);
    buf[4] = (byte)(fragLen & 0xFF);

    return Authenticator.toLong(sequenceNumber);
}

Listing 44: Class OutputRecord prepends ContentType.HANDSHAKE.id (namely, constant 0x22) and padding to buffer buf if outgoing data should be encrypted (Lines 404–408); encrypts the data in that buffer (Lines 432–433), using a null cipher if data should not be encrypted; and adds the header fields for record TLSPlaintext or TLSCiphertext (Lines 438–442), which only differ on the first byte, in particular, the former uses constant 0x22, which is input by child SSLSocketOutputStream, whereas the latter uses constant 0x23 (Line 427).
### B ALERT PROTOCOL

#### Table 3: Extensions and the handshake protocol messages in which they may appear, where such messages are abbreviated as follows: CH (ClientHello), SH (ServerHello), EE (EncryptedExtensions), CT (Certificate), CR (CertificateRequest), NST (NewSessionTicket), and HRR (HelloRetryRequest).

| Extension                                           | RFC | Handshake message |
|-----------------------------------------------------|-----|-------------------|
| application_layer_protocol_negotiation              | 7301| CH, EE            |
| certificateAuthorities                              | 8446| CH, CR            |
| client_certificate_type                             | 7250| CH, EE            |
| cookie                                              | 8446| CH, HRR           |
| earlyData                                           | 8446| CH, EE, NST       |
| heartbeat                                           | 6520| CH, EE            |
| key_share                                           | 8446| CH, SH, HRR       |
| max_fragment_length                                 | 6066| CH, EE            |
| oid_filters                                         | 8446| CR                |
| padding                                             | 7685| CH                |
| post_handshake_auth                                 | 8446| CH                |
| pre_shared_key                                      | 8446| CH, SH            |
| psk_key_exchange_modes                              | 8446| CH                |
| server_certificate_type                             | 7250| CH, EE            |
| server_name                                         | 6066| CH, EE            |
| signature_algorithms                                | 8446| CH, CR            |
| signature_algorithms_cert                           | 8446| CH, CR            |
| signed_certificate_timestamp                         | 6962| CH, CR, CT        |
| status_request                                      | 6066| CH, CR, CT        |
| supported_groups                                    | 7919| CH, EE            |
| supported_versions                                  | 8446| CH, SH, HRR       |
| use_srtp                                            | 5764| CH, EE            |

Extensions are enumerated and instantiated by enum SSLExtension (Listing 45), and class SSLExtensions (Listings 46-47) represents a list of extensions.

### B Alert protocol

TLS defines closure and error alerts, comprising a description field and a legacy severity-level field (which, in TLS 1.3, can be inferred from the alert type). Closure alerts indicate orderly termination of the established channel (in one direction only), and are necessary to avoid truncation attacks. Such closure alerts notify the receiver that the sender will not send any more messages on the channel and any data sent after the alert must be ignored. (The channel must be closed in one direction only to avoid truncating messages in the other direction. This requirement differs from prior versions of TLS, which required the receiver to discard pending messages and immediately send a closure alert of their own, thereby truncating the pending messages.) Error alerts indicate abortive closure and should be sent whenever an error is encountered. Upon transmission or receipt of such an error alert, the established channel must be closed immediately, without sending or receiving any further data. (The listings in this manuscript omit most error alert handling and processing for brevity.) All alerts are encrypted (by the record protocol) after message ServerHello has been successfully consumed.
C Client authentication: CertificateRequest message

For (EC)DHE-only key exchange, a server may request client authentication by sending a CertificateRequest message (immediately after an EncryptedExtensions message) comprising the following fields:

- **certificate_request_context**: A zero-length identifier. (A CertificateRequest message may also be sent to initiate post-handshake authentication, as explained below, in which case a nonce may be used as an identifier.)

- **extensions**: A list of extensions describing authentication properties. The list must contain at least extension signature_algorithms. (Table 3 Appendix A lists other permissible extensions.)

For PSK-based key exchange, a server may only request client authentication if their peer’s ClientHello message included extension post_handshake_auth. Such a request can be made by sending a CertificateRequest message (with a non-zero length identifier) after the handshake protocol completes. A client may decline to authenticate by responding with a Certificate message that does not contain a certificate, followed by Finished message. (The server may continue the handshake without client authentication or abort with a certificate_required alert.) Alternatively, a client may authenticate by responding with Certificate (such that CertificateRequest.certificate_request_context = Certificate.certificate_request_context) and CertificateVerify messages followed by a Finished message. The CertificateVerify message uses string “TLS 1.3, client CertificateVerify”, rather than “TLS 1.3, server CertificateVerify”, to distinguish client- and server-generated CertificateVerify messages and help defend against potential cross-protocol attacks. Moreover, the signature algorithm must be one of those listed in field supported_signature_algorithms of extension signature_algorithms in the CertificateRequest message. (The server may abort the handshake if the client’s certificate chain is unacceptable, e.g., when the chain contains a signature from an unknown or untrusted certificate authority. Alternatively, the server may proceed, considering the client unauthenticated.) Any extensions listed by the Certificate message must respond to ones listed in the CertificateRequest message.
enum SSLExtension implements SSLStringizer {
    private SSLExtension(int id, String name, SSLHandshake handshakeType,
    ProtocolVersion[] supportedProtocols,
    HandshakeProducer producer,
    ExtensionConsumer onLoadConsumer, HandshakeAbsence onLoadAbsence,
    HandshakeConsumer onTradeConsumer, HandshakeAbsence onTradeAbsence,
    SSLStringizer stringize) {
        this.id = id;
        this.handshakeType = handshakeType;
        this.name = name;
        this.supportedProtocols = supportedProtocols;
        this.networkProducer = producer;
        this.onLoadConsumer = onLoadConsumer;
        this.onLoadAbsence = onLoadAbsence;
        this.onTradeConsumer = onTradeConsumer;
        this.onTradeAbsence = onTradeAbsence;
        this.stringizer = stringize;
    }

    public byte[] produce(ConnectionContext context,
        HandshakeMessage message) throws IOException {
        return networkProducer.produce(context, message);
    }

    public void consumeOnLoad(ConnectionContext context,
        HandshakeMessage message, ByteBuffer buffer) throws IOException {
        onLoadConsumer.consume(context, message, buffer);
    }

    public void consumeOnTrade(ConnectionContext context,
        HandshakeMessage message) throws IOException {
        onTradeConsumer.consume(context, message);
    }
}

Listing 45: Enum SSLExtension enumerates and instantiates extensions. Each instantiation defines a hexadecimal value (Line 495) and a name (Line 497). Moreover, they define variable networkProducer of (interface) type HandshakeProducer which is instantiated by a constant ThisNameExtension.messageNetworkProducer, where ThisName corresponds to extension this_name and message is an abbreviation of the message type, e.g., ch abbreviates ClientHello. For instance, constants SupportedVersionsExtension.chNetworkProducer and PreSharedKeyExtension.chNetworkProducer are used for extensions supported_versions and pre_shared_key, respectively, for ClientHello messages. Variable networkProducer is used by method produce to instantiate extensions (Lines 529–537). Variables onLoadConsumer and onTradeConsumer of (interface) type ExtensionConsumer and HandshakeConsumer, respectively, are defined similarly. The former is used by method consumeOnLoad to consume extensions (Lines 539–547) and the latter is used by method consumeOnTrade to update the active context to include extensions (Lines 549–557). Hence, enum SSLExtension is reliant on classes implementing interfaces HandshakeConsumer, HandshakeProducer, and ExtensionConsumer, e.g., inner-classes of class PreSharedKeyExtension.
Listing 46: Class SSLExtensions defines a map of extensions and their associated data (Line 41). That map can be instantiated by method produce (Lines 207–240) or during construction from an input stream (Lines 53-123).
void consumeOnLoad(HandshakeContext context,
SSLExtension[] extensions) throws IOException {
    for (SSLExtension extension : extensions) {
        ByteBuffer m = ByteBuffer.wrap(extMap.get(extension));
        extension.consumeOnLoad(context, handshakeMessage, m);
    }
}

void consumeOnTrade(HandshakeContext context,
SSLExtension[] extensions) throws IOException {
    for (SSLExtension extension : extensions) {
        extension.consumeOnTrade(context, handshakeMessage);
    }
}

void send(HandshakeOutStream hos) throws IOException {
    int extsLen = length();
    if (extsLen == 0) {
        return;
    }
    hos.putInt16(extsLen - 2);
    // extensions must be sent in the order they appear in the enum
    for (SSLExtension ext : SSLExtension.values()) {
        byte[] extData = extMap.get(ext);
        if (extData != null) {
            hos.putInt16(ext.id);
            hos.putInt16(ext.id);
            hos.putBytes16(extData);
        }
    }
}

Listing 47: Class SSLExtensions (continued from Listing 46) defines method consumeOnLoad to consume received extensions (Lines 132–170), using method SSLExtension.consumeOnLoad (Listing 45); consumeOnTrade to update the active context to include extensions (Lines 175–202), using method SSLExtension.consumeOnTrade (Listing 45); and method send to write extensions and associated data to an output stream (Lines 293–307).