TLS 1.3 for engineers: An exploration of the TLS 1.3 specification and OpenJDK's Java implementation

Ben Smyth

Crypto Stream Ltd. & Ampersand (0x26 Ltd.), UK

September 30, 2020

Contribute

This manuscript is far from perfect: Interesting aspects are omitted, because I didn't have the time, knowledge, or expertise to add them. For instance, the specification hasn't been entirely covered, as is documented (rfc8446-annotated.txt); discussion of security guarantees are notably lacking; and an introduction to the underlying cryptography is absent. (E.g., some details on DHKE, AEAD, etc. would be grand.) Directions for further exploration are missing, hands-on teaching opportunities foregone. For instance, a Davies-style exploration of TLS on-the-wire, with notes on Wireshark and SSLKEYLOGFILE – perhaps as dirty as readers can get, without bursting out soldering irons. (Cf. Davies's *Implementing SSL/TLS: Using Cryptography and PKI.*) Mistakes and issues are no doubt numerous. (Some are flagged by tokens `\ifSpecNotes`, `\ifImplNotes`, and `\ifPresentationNotes`.)

Publication is antiquated; help evolve this manuscript: I encourage *you* to improve this manuscript. Fix a typo. Patch grammar. Revise awkward, overcomplicated, or otherwise poorly-written passages. Contribute an entire section. Help evolve this manuscript:

`https://github.com/BenSmyth/tls-tutorial/`

(Perhaps get in touch prior to writing an entire section! We should probably reach consensus on what to add.) Contributions will be recognised through acknowledgements or co-authorship.

Change history

Date	Description
22 Jan 2019	First draft of §1, §2, & §3
22 Feb 2019	Added first draft of §4, plus minor revisions throughout
27 May 2020	Added first draft of §2.6, plus minor additions and revisions elsewhere
30 Sep 2020	GitHub release, plus minor revisions

Acknowledgements

IETF's TLS mailing list provided useful insights and I am particularly grateful to Eric Rescorla. Adam Petcher corrected misattribution of Java code to Oracle, rather than OpenJDK.

Copyright and licensing

Extracts of Oracle's code 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

Listings

1 Introduction

We are nearing an all-encrypted Internet; 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 OpenJDK'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 externally 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 this instance). Finally, the server sends a *`Finished`* message, containing a Message Authentication Code (MAC) over the protocol's transcript, which provides key confirmation, binds the server's identity to the exchanged keys, and, for PSK-based key exchange, authenticates the handshake. Moreover,

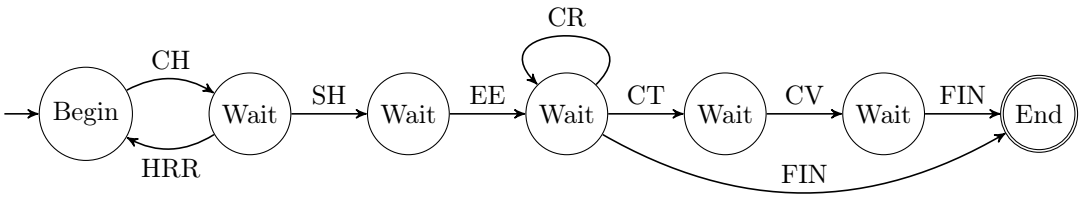

Figure 1: A client initiates the handshake protocol by sending a `ClientHello` (CH) message. After sending that message, the client waits for a `ServerHello` (SH) message followed by an `EncryptedExtensions` (EE) message, or a `HelloRetryRequest` (HRR) message. An `EncryptedExtensions` message might 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.)

the client responds with a `Finished` message of its own. A shared (application traffic) key can then be established to protect communication of application data.

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 OpenJDK'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.

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.)

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

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

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.

`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).[3]

`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:

`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

[3] Support for cipher suite TLS_AES_128_GCM_SHA256 is mandatory (unless an implementation explicitly opts out), and cipher suites TLS_AES_256_GCM_SHA384 and TLS_CHACHA20_POLY1305_SHA256 should also be supported.

```
71  static final class ClientHelloMessage extends HandshakeMessage {
74      final int                   clientVersion;
75      final RandomCookie          clientRandom;
76      final SessionId             sessionId;
79      final List<CipherSuite>     cipherSuites;   // known cipher suites only
80      final byte[]                compressionMethod;
81      final SSLExtensions         extensions;
82
83      private static final byte[] NULL_COMPRESSION = new byte[] {0};
84
85      ClientHelloMessage(HandshakeContext handshakeContext,
86              int clientVersion, SessionId sessionId,
87              List<CipherSuite> cipherSuites, SecureRandom generator) {
88          super(handshakeContext);
90
91          this.clientVersion = clientVersion;
92          this.clientRandom = new RandomCookie(generator);
93          this.sessionId = sessionId;
100         this.cipherSuites = cipherSuites;
102         this.extensions = new SSLExtensions(this);
105         this.compressionMethod = NULL_COMPRESSION;
106     }
160     ClientHelloMessage(HandshakeContext handshakeContext, ByteBuffer m,
161             SSLExtension[] supportedExtensions) throws IOException {
162         super(handshakeContext);
165         this.clientVersion = ((m.get() & 0xFF) << 8) | (m.get() & 0xFF);
166         this.clientRandom = new RandomCookie(m);
167         this.sessionId = new SessionId(Record.getBytes8(m));
185         this.cipherSuiteIds = new int[encodedIds.length >> 1];
186         for (int i = 0, j = 0; i < encodedIds.length; i++, j++) {
187             cipherSuiteIds[j] =
188                 ((encodedIds[i++] & 0xFF) << 8) | (encodedIds[i] & 0xFF);
189         }
190         this.cipherSuites = getCipherSuites(cipherSuiteIds);
192         this.compressionMethod = Record.getBytes8(m);
194         if (m.hasRemaining()) {
195             this.extensions =
196                     new SSLExtensions(this, m, supportedExtensions);
197         } else {
198             this.extensions = new SSLExtensions(this);
199         }
200     }
312     public void send(HandshakeOutStream hos) throws IOException {
313         sendCore(hos);
314         extensions.send(hos);       // In TLS 1.3, use of certain
315                                     // extensions is mandatory.
316     }
318     void sendCore(HandshakeOutStream hos) throws IOException {
319         hos.putInt8((byte) (clientVersion >>> 8));
320         hos.putInt8((byte) clientVersion);
321         hos.write(clientRandom.randomBytes, 0, 32);
322         hos.putBytes8(sessionId.getId());
326         hos.putBytes16(getEncodedCipherSuites());
327         hos.putBytes8(compressionMethod);
328     }
382 }
383
```

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 47, 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 47, Lines 293–307).

```
50  static final SSLProducer kickstartProducer =
51      new ClientHelloKickstartProducer();
387 private static final
388         class ClientHelloKickstartProducer implements SSLProducer {
396     public byte[] produce(ConnectionContext context) throws IOException {
398         ClientHandshakeContext chc = (ClientHandshakeContext)context;
407         SessionId sessionId = SSLSessionImpl.nullSession.getSessionId();
410         List<CipherSuite> cipherSuites = chc.activeCipherSuites;
415         SSLSessionContextImpl ssci = (SSLSessionContextImpl)
416                 chc.sslContext.engineGetClientSessionContext();
417         SSLSessionImpl session = ssci.get(
418                 chc.conContext.transport.getPeerHost(),
419                 chc.conContext.transport.getPeerPort());
445         CipherSuite sessionSuite = null;
446         if (session != null) {
447             sessionSuite = session.getSuite();
542                 cipherSuites = Arrays.asList(sessionSuite);
553             chc.isResumption = true;
554             chc.resumingSession = session;
555         }
615             clientHelloVersion = ProtocolVersion.TLS12;
618
619         ClientHelloMessage chm = new ClientHelloMessage(chc,
620                 clientHelloVersion.id, sessionId, cipherSuites,
621                 chc.sslContext.getSecureRandom());
622
623         // cache the client random number for further using
624         chc.clientHelloRandom = chm.clientRandom;
625         chc.clientHelloVersion = clientHelloVersion.id;
626
627         // Produce extensions for ClientHello handshake message.
628         SSLExtension[] extTypes = chc.sslConfig.getEnabledExtensions(
629                 SSLHandshake.CLIENT_HELLO, chc.activeProtocols);
630         chm.extensions.produce(chc, extTypes);
635
636         // Output the handshake message.
637         chm.write(chc.handshakeOutput);
638         chc.handshakeOutput.flush();
639
640         // Reserve the initial ClientHello message for the follow on
641         // cookie exchange if needed.
642         chc.initialClientHelloMsg = chm;
643
644         // What's the expected response?
645         chc.handshakeConsumers.put(
646                 SSLHandshake.SERVER_HELLO.id, SSLHandshake.SERVER_HELLO);
653
654         // The handshake message has been delivered.
655         return null;
656     }
657 }
```

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 parameterises `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 associated with the pre-shared key, 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).

curves.[4] 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 2.7.1, after `NewSessionTicket` messages – which establish pre-shared keys for subsequent connections – are introduced.) At least one offered cipher suite should define a hash function associated with at least one of the identifiers. 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`).[5] (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.[6] 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

[4] Supported groups include: Finite field groups defined in RFC 7919, namely, ffdhe2048 (0x0100), ffdhe3072 (0x0101), ffdhe4096 (0x0102), ffdhe6144 (0x0103), and ffdhe8192 (0x0104), and elliptic curve groups defined in either FIPS 186-4 or RFC 7748, namely, secp256r1 (0x0017), secp384r1 (0x0018), secp521r1 (0x0019), x25519 (0x001D), and x448 (0x001E). Supporting group secp256r1 is mandatory (unless an implementation explicitly opts out), and group x25519 should also be supported.

[5] Supported signature algorithms include: RSASSA-PKCS1-v1_5 (RFC8017) or RSASSA-PSS (RFC8017) with a corresponding hash function, namely, SHA256, SHA384, or SHA512; ECDSA (American National Standards Institute, 2005) with a corresponding curve & hash function, namely, secp256r1 & SHA256, secp384r1 & SHA384, or secp521r1 & SHA512; and EdDSA (RFC8032). (RSASSA-PKCS1-v1_5 is only supported for `Certificate` messages.) Supporting RSA-based signatures with SHA256 (for certificates) and ECDSA signatures with secp256r1 & SHA256 is mandatory (unless an implementation explicitly opts out). (RSASSA-PSS must also be supported for `CertificateVerify` messages.)

[6] RFC 8446 does not explicitly require servers to check fields `legacy_version` and `legacy_session_id`, it merely requires clients to set those fields correctly. Accordingly, we assume servers *may* perform these checks, rather than mandating them. By comparison, RFC 8446 explicitly requires field `legacy_compression_methods` to be correctly set.

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 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 (that identifier must be associated with a hash function, AEAD algorithm, or both, which are defined by the server-selected cipher suite), 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 RFC 8446, 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 SSLHandshake.SERVER_HELLO – to produce `ServerHello` messages in class ClientHello.T13ClientHelloConsumer (Listing 5).

```
52  static final SSLConsumer handshakeConsumer =
53      new ClientHelloConsumer();
760 private static final class ClientHelloConsumer implements SSLConsumer {
767     public void consume(ConnectionContext context,
768             ByteBuffer message) throws IOException {
770         ServerHandshakeContext shc = (ServerHandshakeContext)context;
781         SSLExtension[] enabledExtensions =
782                 shc.sslConfig.getEnabledExtensions(
783                         SSLHandshake.CLIENT_HELLO);
785         ClientHelloMessage chm =
786                 new ClientHelloMessage(shc, message, enabledExtensions);
791         shc.clientHelloVersion = chm.clientVersion;
792         onClientHello(shc, chm);
793     }
795     private void onClientHello(ServerHandshakeContext context,
796             ClientHelloMessage clientHello) throws IOException {
800         SSLExtension[] extTypes = new SSLExtension[] {
801                 SSLExtension.CH_SUPPORTED_VERSIONS
802             };
803         clientHello.extensions.consumeOnLoad(context, extTypes);
804
805         ProtocolVersion negotiatedProtocol;
806         CHSupportedVersionsSpec svs =
807                 (CHSupportedVersionsSpec)context.handshakeExtensions.get(
808                         SSLExtension.CH_SUPPORTED_VERSIONS);
809         if (svs != null) {
810             negotiatedProtocol =
811                     negotiateProtocol(context, svs.requestedProtocols);
812         } else {
813             negotiatedProtocol =
814                     negotiateProtocol(context, clientHello.clientVersion);
815         }
816         context.negotiatedProtocol = negotiatedProtocol;
830             if (negotiatedProtocol.useTLS13PlusSpec()) {
831                 t13HandshakeConsumer.consume(context, clientHello);
832             } else {
833                 t12HandshakeConsumer.consume(context, clientHello);
834             }
836     }
873     private ProtocolVersion negotiateProtocol(
874             ServerHandshakeContext context,
875             int[] clientSupportedVersions) throws SSLException {
877         // The client supported protocol versions are present in client
878         // preference order.  This implementation chooses to use the server
879         // preference of protocol versions instead.
880         for (ProtocolVersion spv : context.activeProtocols) {
884             for (int cpv : clientSupportedVersions) {
888                 if (spv.id == cpv) {
889                     return spv;
890                 }
891             }
892         }
902     }
903 }
```

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).

```
59   private static final HandshakeConsumer t13HandshakeConsumer =
60           new T13ClientHelloConsumer();
1075 private static final
1076         class T13ClientHelloConsumer implements HandshakeConsumer {
1083     public void consume(ConnectionContext context,
1084             HandshakeMessage message) throws IOException {
1086         ServerHandshakeContext shc = (ServerHandshakeContext)context;
1087         ClientHelloMessage clientHello = (ClientHelloMessage)message;
1097         // Check and launch the "psk_key_exchange_modes" and
1098         // "pre_shared_key" extensions first, which will reset the
1099         // resuming session, no matter the extensions present or not.
1100         shc.isResumption = true;
1101         SSLExtension[] extTypes = new SSLExtension[] {
1102                 SSLExtension.PSK_KEY_EXCHANGE_MODES,
1103                 SSLExtension.CH_PRE_SHARED_KEY
1104             };
1105         clientHello.extensions.consumeOnLoad(shc, extTypes);
1106
1107         // Check and launch ClientHello extensions other than
1108         // "psk_key_exchange_modes", "pre_shared_key", "protocol_version"
1109         // and "key_share" extensions.
1110         //
1111         // These extensions may discard session resumption, or ask for
1112         // hello retry.
1113         extTypes = shc.sslConfig.getExclusiveExtensions(
1114                 SSLHandshake.CLIENT_HELLO,
1115                 Arrays.asList(
1116                         SSLExtension.PSK_KEY_EXCHANGE_MODES,
1117                         SSLExtension.CH_PRE_SHARED_KEY,
1118                         SSLExtension.CH_SUPPORTED_VERSIONS));
1119         clientHello.extensions.consumeOnLoad(shc, extTypes);
1120
1121         if (!shc.handshakeProducers.isEmpty()) {
1122             // Should be HelloRetryRequest producer.
1123             goHelloRetryRequest(shc, clientHello);
1124         } else {
1125             goServerHello(shc, clientHello);
1126         }
1127     }
```

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).

```
1129        private void goHelloRetryRequest(ServerHandshakeContext shc,
1130                ClientHelloMessage clientHello) throws IOException {
1131            HandshakeProducer handshakeProducer =
1132                    shc.handshakeProducers.remove(
1133                            SSLHandshake.HELLO_RETRY_REQUEST.id);
1135                    handshakeProducer.produce(shc, clientHello);
1147        }
1149        private void goServerHello(ServerHandshakeContext shc,
1150                ClientHelloMessage clientHello) throws IOException {
1154            shc.clientHelloRandom = clientHello.clientRandom;
1159            if (!shc.conContext.isNegotiated) {
1160                shc.conContext.protocolVersion = shc.negotiatedProtocol;
1161                shc.conContext.outputRecord.setVersion(shc.negotiatedProtocol);
1162            }
1163
1164            // update the responders
1165            //
1166            // Only ServerHello/HelloRetryRequest producer, which adds
1167            // more responders later.
1168            shc.handshakeProducers.put(SSLHandshake.SERVER_HELLO.id,
1169                SSLHandshake.SERVER_HELLO);
1170
1171            SSLHandshake[] probableHandshakeMessages = new SSLHandshake[] {
1172                SSLHandshake.SERVER_HELLO,
1173
1174                // full handshake messages
1175                SSLHandshake.ENCRYPTED_EXTENSIONS,
1176                SSLHandshake.CERTIFICATE_REQUEST,
1177                SSLHandshake.CERTIFICATE,
1178                SSLHandshake.CERTIFICATE_VERIFY,
1179                SSLHandshake.FINISHED
1180            };
1185            for (SSLHandshake hs : probableHandshakeMessages) {
1186                HandshakeProducer handshakeProducer =
1187                        shc.handshakeProducers.remove(hs.id);
1188                if (handshakeProducer != null) {
1189                    handshakeProducer.produce(shc, clientHello);
1190                }
1191            }
1192        }
1193    }
```

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.

```
85  static final class ServerHelloMessage extends HandshakeMessage {
86      final ProtocolVersion           serverVersion;      // TLS 1.3 legacy
87      final RandomCookie              serverRandom;
88      final SessionId                 sessionId;          // TLS 1.3 legacy
89      final CipherSuite               cipherSuite;
90      final byte                      compressionMethod;  // TLS 1.3 legacy
91      final SSLExtensions             extensions;
92
93      // The HelloRetryRequest producer needs to use the ClientHello message
94      // for cookie generation.  Please don't use this field for other
95      // purpose unless it is really necessary.
96      final ClientHelloMessage        clientHello;
97
98      // Reserved for HelloRetryRequest consumer.  Please don't use this
99      // field for other purpose unless it is really necessary.
100     final ByteBuffer                handshakeRecord;
101
102     ServerHelloMessage(HandshakeContext context,
103             ProtocolVersion serverVersion, SessionId sessionId,
104             CipherSuite cipherSuite, RandomCookie serverRandom,
105             ClientHelloMessage clientHello) {
106         super(context);
107
108         this.serverVersion = serverVersion;
109         this.serverRandom = serverRandom;
110         this.sessionId = sessionId;
111         this.cipherSuite = cipherSuite;
112         this.compressionMethod = 0x00;      // Don't support compression.
113         this.extensions = new SSLExtensions(this);
114
115         // Reserve the ClientHello message for cookie generation.
116         this.clientHello = clientHello;
117
118         // The handshakeRecord field is used for HelloRetryRequest consumer
119         // only.  It's fine to set it to null for generating side of the
120         // ServerHello/HelloRetryRequest message.
121         this.handshakeRecord = null;
122     }
```

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).

```
124         ServerHelloMessage(HandshakeContext context,
125                 ByteBuffer m) throws IOException {
126             super(context);
128             // Reserve for HelloRetryRequest consumer if needed.
129             this.handshakeRecord = m.duplicate();
131             byte major = m.get();
132             byte minor = m.get();
133             this.serverVersion = ProtocolVersion.valueOf(major, minor);
141             this.serverRandom = new RandomCookie(m);
142             this.sessionId = new SessionId(Record.getBytes8(m));
149             int cipherSuiteId = Record.getInt16(m);
150             this.cipherSuite = CipherSuite.valueOf(cipherSuiteId);
151             if (cipherSuite == null || !context.isNegotiable(cipherSuite)) {
152                 context.conContext.fatal(Alert.ILLEGAL_PARAMETER,
153                     "Server selected improper ciphersuite " +
154                     CipherSuite.nameOf(cipherSuiteId));
155             }
157             this.compressionMethod = m.get();
163             SSLExtension[] supportedExtensions;
164             if (serverRandom.isHelloRetryRequest()) {
165                 supportedExtensions = context.sslConfig.getEnabledExtensions(
166                         SSLHandshake.HELLO_RETRY_REQUEST);
167             } else {
168                 supportedExtensions = context.sslConfig.getEnabledExtensions(
169                         SSLHandshake.SERVER_HELLO);
170             }
172             if (m.hasRemaining()) {
173                 this.extensions =
174                     new SSLExtensions(this, m, supportedExtensions);
175             } else {
176                 this.extensions = new SSLExtensions(this);
177             }
179             // The clientHello field is used for HelloRetryRequest producer
180             // only.  It's fine to set it to null for receiving side of
181             // ServerHello/HelloRetryRequest message.
182             this.clientHello = null;          // not used, let it be null;
183         }
205         public void send(HandshakeOutStream hos) throws IOException {
206             hos.putInt8(serverVersion.major);
207             hos.putInt8(serverVersion.minor);
208             hos.write(serverRandom.randomBytes);
209             hos.putBytes8(sessionId.getId());
210             hos.putInt8((cipherSuite.id >> 8) & 0xFF);
211             hos.putInt8(cipherSuite.id & 0xff);
212             hos.putInt8(compressionMethod);
213
214             extensions.send(hos);             // In TLS 1.3, use of certain
215                                               // extensions is mandatory.
216         }
246     }
```

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), checking that the server-selected cipher suite (Lines 149–150) is amongst those offered by the client (Lines 151–155), and method send to write such messages to an output stream, using method SSLExtensions.send to write the extensions field (Listing 47, Lines 293–307).

```
59  static final HandshakeProducer t13HandshakeProducer =
60      new T13ServerHelloProducer();
484 private static final
485         class T13ServerHelloProducer implements HandshakeProducer {
492     public byte[] produce(ConnectionContext context,
493             HandshakeMessage message) throws IOException {
495         ServerHandshakeContext shc = (ServerHandshakeContext)context;
496         ClientHelloMessage clientHello = (ClientHelloMessage)message;
497
498         // If client hasn't specified a session we can resume, start a
499         // new one and choose its cipher suite and compression options,
500         // unless new session creation is disabled for this connection!
501         if (!shc.isResumption || shc.resumingSession == null) {
513             SSLSessionImpl session =
514                     new SSLSessionImpl(shc, CipherSuite.C_NULL);
516             shc.handshakeSession = session;
518             // consider the handshake extension impact
519             SSLExtension[] enabledExtensions =
520                     shc.sslConfig.getEnabledExtensions(
521                         SSLHandshake.CLIENT_HELLO, shc.negotiatedProtocol);
522             clientHello.extensions.consumeOnTrade(shc, enabledExtensions);
523
524             // negotiate the cipher suite.
525             CipherSuite cipherSuite = chooseCipherSuite(shc, clientHello);
531             shc.negotiatedCipherSuite = cipherSuite;
532             shc.handshakeSession.setSuite(cipherSuite);
535         } else {
536             shc.handshakeSession = shc.resumingSession;
538             // consider the handshake extension impact
539             SSLExtension[] enabledExtensions =
540             shc.sslConfig.getEnabledExtensions(
541             SSLHandshake.CLIENT_HELLO, shc.negotiatedProtocol);
542             clientHello.extensions.consumeOnTrade(shc, enabledExtensions);
546             shc.negotiatedCipherSuite = shc.resumingSession.getSuite();
549
550             setUpPskKD(shc,
551                     shc.resumingSession.consumePreSharedKey().get());
557         }
558
559         // update the responders
560         shc.handshakeProducers.put(SSLHandshake.ENCRYPTED_EXTENSIONS.id,
561                 SSLHandshake.ENCRYPTED_EXTENSIONS);
562         shc.handshakeProducers.put(SSLHandshake.FINISHED.id,
563                 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.

```
565             // Generate the ServerHello handshake message.
566             ServerHelloMessage shm = new ServerHelloMessage(shc,
567                     ProtocolVersion.TLS12,       // use legacy version
568                     clientHello.sessionId,       // echo back
569                     shc.negotiatedCipherSuite,
570                     new RandomCookie(shc),
571                     clientHello);
572             shc.serverHelloRandom = shm.serverRandom;
573
574             // Produce extensions for ServerHello handshake message.
575             SSLExtension[] serverHelloExtensions =
576                     shc.sslConfig.getEnabledExtensions(
577                         SSLHandshake.SERVER_HELLO, shc.negotiatedProtocol);
578             shm.extensions.produce(shc, serverHelloExtensions);
582
583             // Output the handshake message.
584             shm.write(shc.handshakeOutput);
585             shc.handshakeOutput.flush();
677             // The handshake message has been delivered.
678             return null;
679         }
680
681         private static CipherSuite chooseCipherSuite(
682                 ServerHandshakeContext shc,
683                 ClientHelloMessage clientHello) throws IOException {
684             List<CipherSuite> preferred;
685             List<CipherSuite> proposed;
686             if (shc.sslConfig.preferLocalCipherSuites) {
687                 preferred = shc.activeCipherSuites;
688                 proposed = clientHello.cipherSuites;
689             } else {
690                 preferred = clientHello.cipherSuites;
691                 proposed = shc.activeCipherSuites;
692             }
694             CipherSuite legacySuite = null;
697             for (CipherSuite cs : preferred) {
698                 if (!HandshakeContext.isNegotiable(
699                         proposed, shc.negotiatedProtocol, cs)) {
700                     continue;
701                 }
713                 return cs;
714             }
725             return null;
726         }
727     }
```

Listing 9: Class ServerHello.T13ServerHelloProducer defines method produce (continued from Listing 8) which instantiates a `ServerHello` message (Lines 566–571), populates the extension field for the server's active context (Lines 575–578), and writes the `ServerHello` message to an output stream (Lines 584–585). The `ServerHello` message parameterises `legacy_version` as constant 0x0303 (Line 567), `legacy_session_id_echo` as `ClientHello.cipher_suites` (Line 568), and `cipher_suite` as the negotiated cipher suite (Line 569), which is the server selected cipher suite for (EC)DHE-only key exchange (Lines 525 & 531, Listing 8), selected using method chooseCipherSuite, or the cipher suite associated with the pre-shared key for PSK-based key exchange (Line 546, Listing 8). The output stream is written to using method ServerHello.ServerHelloMessage.write, defined by parent class SSLHandshake.HandshakeMessage, which in turn uses method ServerHello.ServerHelloMessage.send (Listing 7). (After outputting the message, the server updates the active context to include new keying material in preparation for the server's response, Section 2.3.) Method chooseCipherSuite instantiates lists of preferred and proposed cipher suites as the list of available cipher suites and the list of offered cipher suites, respectively, or vice-versa, depending on the active context (Lines 684–692), and returns the first preferred cipher suite that is amongst those proposed (Lines 694–714) or **null** if no such suite exists (Line 725).

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 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 .hrrRandomBytes) 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, the server-selected cipher suite defines a hash function associated with that identifier, and 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.[7] In addition to mandatory extension `supported_versions`, message `HelloRetryRequest` should include extension `key_share` to indicate the server-selected group.[8] (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

[7] 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 $\frac{1}{2^{128}}$, hence, confusion will not occur in practice.

[8] 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.

```
55  static final SSLConsumer handshakeConsumer =
56      new ServerHelloConsumer();
839 private static final
840         class ServerHelloConsumer implements SSLConsumer {
847     public void consume(ConnectionContext context,
848             ByteBuffer message) throws IOException {
850         ClientHandshakeContext chc = (ClientHandshakeContext)context;
864         ServerHelloMessage shm = new ServerHelloMessage(chc, message);
869         if (shm.serverRandom.isHelloRetryRequest()) {
870             onHelloRetryRequest(chc, shm);
871         } else {
872             onServerHello(chc, shm);
873         }
874     }
928     private void onServerHello(ClientHandshakeContext chc,
929             ServerHelloMessage serverHello) throws IOException {
933         SSLExtension[] extTypes = new SSLExtension[] {
934                 SSLExtension.SH_SUPPORTED_VERSIONS
935             };
936         serverHello.extensions.consumeOnLoad(chc, extTypes);
937
938         ProtocolVersion serverVersion;
939         SHSupportedVersionsSpec svs =
940                 (SHSupportedVersionsSpec)chc.handshakeExtensions.get(
941                         SSLExtension.SH_SUPPORTED_VERSIONS);
943             serverVersion =             // could be null
944                     ProtocolVersion.valueOf(svs.selectedVersion);
948
949         if (!chc.activeProtocols.contains(serverVersion)) {
950             chc.conContext.fatal(Alert.PROTOCOL_VERSION,
951                 "The server selected protocol version " + serverVersion +
952                 " is not accepted by client preferences " +
953                 chc.activeProtocols);
954         }
956         chc.negotiatedProtocol = serverVersion;
984             t13HandshakeConsumer.consume(chc, serverHello);
993     }
994 }
```

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.put(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).

```
69    private static final HandshakeConsumer t13HandshakeConsumer =
70        new T13ServerHelloConsumer();
1172  private static final
1173          class T13ServerHelloConsumer implements HandshakeConsumer {
1180      public void consume(ConnectionContext context,
1181              HandshakeMessage message) throws IOException {
1183          ClientHandshakeContext chc = (ClientHandshakeContext)context;
1184          ServerHelloMessage serverHello = (ServerHelloMessage)message;
1189
1190          chc.negotiatedCipherSuite = serverHello.cipherSuite;
1193          chc.serverHelloRandom = serverHello.serverRandom;
1200          SSLExtension[] extTypes = chc.sslConfig.getEnabledExtensions(
1201                  SSLHandshake.SERVER_HELLO);
1202          serverHello.extensions.consumeOnLoad(chc, extTypes);
1203          if (!chc.isResumption) {
1214              chc.handshakeSession = new SSLSessionImpl(chc,
1215                      chc.negotiatedCipherSuite,
1216                      serverHello.sessionId);
1219          } else {
1221              Optional<SecretKey> psk =
1222                      chc.resumingSession.consumePreSharedKey();
1228              chc.handshakeSession = chc.resumingSession;
1230              setUpPskKD(chc, psk.get());
1231          }
1330          // update the consumers and producers
1339          chc.handshakeConsumers.put(
1340                  SSLHandshake.ENCRYPTED_EXTENSIONS.id,
1341                  SSLHandshake.ENCRYPTED_EXTENSIONS);
1342
1343          // Support cert authentication only, when not PSK.
1344          chc.handshakeConsumers.put(
1345                  SSLHandshake.CERTIFICATE_REQUEST.id,
1346                  SSLHandshake.CERTIFICATE_REQUEST);
1347          chc.handshakeConsumers.put(
1348                  SSLHandshake.CERTIFICATE.id,
1349                  SSLHandshake.CERTIFICATE);
1350          chc.handshakeConsumers.put(
1351                  SSLHandshake.CERTIFICATE_VERIFY.id,
1352                  SSLHandshake.CERTIFICATE_VERIFY);
1353
1354          chc.handshakeConsumers.put(
1355                  SSLHandshake.FINISHED.id,
1356                  SSLHandshake.FINISHED);
1362      }
1363  }
```

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).

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, removing extension `early_data` if present, including a copy of extension `cookie` and associated data if the extension appeared in the `HelloRetryRequest` message, and updating extension `pre_shared_key` by recomputing its obfuscated age and binder values (§2.7.1). Moreover, the client should remove any pre-shared key identifiers that are incompatible with the server-selected cipher suite (i.e., remove identifiers associated with a hash function, AEAD algorithm, or both that are not defined by 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`).

Beyond the above two instances of `ClientHello` messages, a server that receives a `ClientHello` message at any other time must abort with an unexpected_message alert.

> 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. HELLO_RETRY_REQUEST – 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.

```
61  static final HandshakeProducer hrrHandshakeProducer =
62      new T13HelloRetryRequestProducer();
732 private static final
733         class T13HelloRetryRequestProducer implements HandshakeProducer {
740     public byte[] produce(ConnectionContext context,
741             HandshakeMessage message) throws IOException {
742         ServerHandshakeContext shc = (ServerHandshakeContext) context;
743         ClientHelloMessage clientHello = (ClientHelloMessage) message;
745         // negotiate the cipher suite.
746         CipherSuite cipherSuite =
747                 T13ServerHelloProducer.chooseCipherSuite(shc, clientHello);
754         ServerHelloMessage hhrm = new ServerHelloMessage(shc,
755                 ProtocolVersion.TLS12,       // use legacy version
756                 clientHello.sessionId,       //  echo back
757                 cipherSuite,
758                 RandomCookie.hrrRandom,
759                 clientHello
760         );
762         shc.negotiatedCipherSuite = cipherSuite;
766         // Produce extensions for HelloRetryRequest handshake message.
767         SSLExtension[] serverHelloExtensions =
768             shc.sslConfig.getEnabledExtensions(
769                 SSLHandshake.HELLO_RETRY_REQUEST, shc.negotiatedProtocol);
770         hhrm.extensions.produce(shc, serverHelloExtensions);
776         // Output the handshake message.
777         hhrm.write(shc.handshakeOutput);
778         shc.handshakeOutput.flush();
780         // Stateless, shall we clean up the handshake context as well?
781         shc.handshakeHash.finish();     // forgot about the handshake hash
782         shc.handshakeExtensions.clear();
784         // What's the expected response?
785         shc.handshakeConsumers.put(
786                 SSLHandshake.CLIENT_HELLO.id, SSLHandshake.CLIENT_HELLO);
788         // The handshake message has been delivered.
789         return null;
790     }
791 }
```

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).

```
876     private void onHelloRetryRequest(ClientHandshakeContext chc,
877             ServerHelloMessage helloRetryRequest) throws IOException {
881         SSLExtension[] extTypes = new SSLExtension[] {
882                 SSLExtension.HRR_SUPPORTED_VERSIONS
883             };
884         helloRetryRequest.extensions.consumeOnLoad(chc, extTypes);
885
886         ProtocolVersion serverVersion;
887         SHSupportedVersionsSpec svs =
888                 (SHSupportedVersionsSpec)chc.handshakeExtensions.get(
889                         SSLExtension.HRR_SUPPORTED_VERSIONS);
891             serverVersion =             // could be null
892                     ProtocolVersion.valueOf(svs.selectedVersion);
897         if (!chc.activeProtocols.contains(serverVersion)) {
898             chc.conContext.fatal(Alert.PROTOCOL_VERSION,
899                 "The server selected protocol version " + serverVersion +
900                 " is not accepted by client preferences " +
901                 chc.activeProtocols);
902         }
909         chc.negotiatedProtocol = serverVersion;
924             t13HrrHandshakeConsumer.consume(chc, helloRetryRequest);
926     }
```

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).

```
77   private static final HandshakeConsumer t13HrrHandshakeConsumer =
78       new T13HelloRetryRequestConsumer();
1373     public void consume(ConnectionContext context,
1374             HandshakeMessage message) throws IOException {
1376         ClientHandshakeContext chc = (ClientHandshakeContext)context;
1377         ServerHelloMessage helloRetryRequest = (ServerHelloMessage)message;
1383         chc.negotiatedCipherSuite = helloRetryRequest.cipherSuite;
1389         // Check and launch ClientHello extensions.
1390         SSLExtension[] extTypes = chc.sslConfig.getEnabledExtensions(
1391                 SSLHandshake.HELLO_RETRY_REQUEST);
1392         helloRetryRequest.extensions.consumeOnLoad(chc, extTypes);
1397         helloRetryRequest.extensions.consumeOnTrade(chc, extTypes);
1459         SSLHandshake.CLIENT_HELLO.produce(context, helloRetryRequest);
1460     }
1461 }
```

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).

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, as introduced in Section 3), but excluding record-layer headers. The concatenation of messages starts with `ClientHello`, optionally followed by `HelloRetryRequest` and `ClientHello` if present, and proceeded by `ServerHello`. That transcript is used in computing transcript traffic keys (which protect the remaining handshake messages). Thereafter, the concatenation of messages is extended with `EncryptedExtensions` and optionally `CertificateRequest`, `Certificate`, and `CertificateVerify` if sent. A MAC over that transcript is included in a server's `Finished` message, and a signature over the transcript (excluding message `CertificateVerify`) is included in any `CertificateVerify` message. Once extended with that `Finished` message, the transcript is used in computing the application traffic keys (which protect application traffic). Finally, for a client's `Finished` message, the transcript is further extended with their `EndOfEarlyData`, `Certificate`, and `CertificateVerify` messages (as relevant), before computing a MAC, wherein any `CertificateVerify` message includes a signature over that transcript (excluding itself).

To capture a transcript hash (i.e., a hash of a transcript), we introduce function Transcript-Hash such that

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

for handshake protocol messages $M_1, \ldots, M_n$ (sent in that order), where 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.,

$$\mathsf{Transcript\text{-}Hash}(M_1, \ldots, M_n) = \mathsf{Hash}(M'_1 \parallel M_2 \parallel \cdots \parallel M_n),$$

where M'_1 is the following special, synthetic handshake message, namely,

$$\begin{aligned} M'_1 = \ & 0\text{x}FE && \text{/* header type message\_hash*/} \\ & \parallel 0\text{x}0000 \parallel \mathsf{Hash.length} && \text{/* (padded) header length */} \\ & \parallel \mathsf{Hash}(M_1) && \text{/* hash of } \texttt{ClientHello} \text{ message */} \end{aligned}$$

where Hash.length is the output length in bytes of negotiated hash function 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).[9]

> Transcript hashing is implemented by class HandshakeHash (Listing 15). Instances of that class form part of the active client and server contexts (instantiated by classes SSLEngineImpl and SSLSocketImpl), which are updated by classes SSLEngineInputRecord and SSLEngineOutputRecord, respectively classes SSLSocketInputRecord and SSLSocketOutputRecord. Moreover, in the case of a `HelloRetryRequest` message, it is updated by class ServerHello.T13HelloRetryRequestConsumer (Listing 16) and by class HelloCookieManager (Listing 17), when consuming any `cookie` extension associated with a corresponding `ClientHello` message.

2.3.2 Key derivation

The key derivation process combines the negotiated pre-shared key, the (EC)DHE key, or both, with the protocol's transcript. The process uses function HKDF-Extract, which is defined by

[9] `HelloRetryRequest` messages need not be maintained by the server either, since they can be reconstructed from `ClientHello` messages and the special constant value that is used by field `HelloRetryRequest.random`.

```
37  final class HandshakeHash {
38      private TranscriptHash transcriptHash;
39      private LinkedList<byte[]> reserves;     // one handshake message per entry
41
42      HandshakeHash() {
43          this.transcriptHash = new CacheOnlyHash();
44          this.reserves = new LinkedList<>();
46      }
85      void receive(byte[] input) {
86          reserves.add(Arrays.copyOf(input, input.length));
87      }
106     // For HelloRetryRequest only! Please use this method very carefully!
107     void push(byte[] input) {
108         reserves.push(Arrays.copyOf(input, input.length));
109     }
116     void deliver(byte[] input) {
117         update();
118         transcriptHash.update(input, 0, input.length);
119     }
164     void update() {
165         while (reserves.size() != 0) {
166             byte[] holder = reserves.remove();
167             transcriptHash.update(holder, 0, holder.length);
168         }
170     }
172     byte[] digest() {
173         // Note that the reserve handshake message may be not a part of
174         // the expected digest.
175         return transcriptHash.digest();
176     }
178     void finish() {
179         this.transcriptHash = new CacheOnlyHash();
180         this.reserves = new LinkedList<>();
182     }
645 }
```

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).

```
1401            chc.handshakeHash.finish();      // reset the handshake hash
1403            // calculate the transcript hash of the 1st ClientHello message
1404            HandshakeOutStream hos = new HandshakeOutStream(null);
1405            try {
1406                chc.initialClientHelloMsg.write(hos);
1407            } catch (IOException ioe) {
1408                // unlikely
1411            }
1412            chc.handshakeHash.deliver(hos.toByteArray());
1415            byte[] clientHelloHash = chc.handshakeHash.digest();
1416
1417            // calculate the message_hash
1418            //
1419            // Transcript-Hash(ClientHello1, HelloRetryRequest, ... Mn) =
1420            //   Hash(message_hash ||     /* Handshake type */
1421            //      00 00 Hash.length ||  /* Handshake message length (bytes) */
1422            //      Hash(ClientHello1) || /* Hash of ClientHello1 */
1423            //      HelloRetryRequest || ... || Mn)
1424            int hashLen = chc.negotiatedCipherSuite.hashAlg.hashLength;
1425            byte[] hashedClientHello = new byte[4 + hashLen];
1426            hashedClientHello[0] = SSLHandshake.MESSAGE_HASH.id;
1427            hashedClientHello[1] = (byte)0x00;
1428            hashedClientHello[2] = (byte)0x00;
1429            hashedClientHello[3] = (byte)(hashLen & 0xFF);
1430            System.arraycopy(clientHelloHash, 0,
1431                    hashedClientHello, 4, hashLen);
1432
1433            chc.handshakeHash.finish();      // reset the handshake hash
1434            chc.handshakeHash.deliver(hashedClientHello);
1435
1436            int hrrBodyLen = helloRetryRequest.handshakeRecord.remaining();
1437            byte[] hrrMessage = new byte[4 + hrrBodyLen];
1438            hrrMessage[0] = SSLHandshake.HELLO_RETRY_REQUEST.id;
1439            hrrMessage[1] = (byte)((hrrBodyLen >> 16) & 0xFF);
1440            hrrMessage[2] = (byte)((hrrBodyLen >> 8) & 0xFF);
1441            hrrMessage[3] = (byte)(hrrBodyLen & 0xFF);
1442
1443            ByteBuffer hrrBody = helloRetryRequest.handshakeRecord.duplicate();
1444            hrrBody.get(hrrMessage, 4, hrrBodyLen);
1445
1446            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 0xFE (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.

```
200   private static final
201           class T13HelloCookieManager extends HelloCookieManager {
271       boolean isCookieValid(ServerHandshakeContext context,
272           ClientHelloMessage clientHello, byte[] cookie) throws IOException {
278           int csId = ((cookie[0] & 0xFF) << 8) | (cookie[1] & 0xFF);
279           CipherSuite cs = CipherSuite.valueOf(csId);
284           int hashLen = cs.hashAlg.hashLength;
291           byte[] prevClientHelloHash =
292                   Arrays.copyOfRange(cookie, 3 + hashLen, cookie.length);
312           // Use the ClientHello hash in the cookie for transtript
313           // hash calculation for stateless HelloRetryRequest.
314           //
315           // Transcript-Hash(ClientHello1, HelloRetryRequest, ... Mn) =
316           //   Hash(message_hash ||     /* Handshake type */
317           //      00 00 Hash.length ||  /* Handshake message length (bytes) */
318           //      Hash(ClientHello1) || /* Hash of ClientHello1 */
319           //      HelloRetryRequest || ... || Mn)
320
321           // Reproduce HelloRetryRequest handshake message
322           byte[] hrrMessage =
323                   ServerHello.hrrReproducer.produce(context, clientHello);
324           context.handshakeHash.push(hrrMessage);
325
326           // Construct the 1st ClientHello message for transcript hash
327           byte[] hashedClientHello = new byte[4 + hashLen];
328           hashedClientHello[0] = SSLHandshake.MESSAGE_HASH.id;
329           hashedClientHello[1] = (byte)0x00;
330           hashedClientHello[2] = (byte)0x00;
331           hashedClientHello[3] = (byte)(hashLen & 0xFF);
332           System.arraycopy(prevClientHelloHash, 0,
333                   hashedClientHello, 4, hashLen);
334
335           context.handshakeHash.push(hashedClientHello);
338       }
339   }
```

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 $0xFE0000 \parallel \mathsf{Hash.length}$ 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).

RFC 5869 such that

$$\mathsf{HKDF\text{-}Extract}(\mathit{Salt}, \mathit{Secret}) = \begin{cases} \mathsf{HMAC}(0\text{s}, \mathit{Secret}) & \text{if } \mathit{Salt} \text{ is null} \\ \mathsf{HMAC}(\mathit{Salt}, \mathit{Secret}) & \text{otherwise,} \end{cases}$$

where 0s denotes a Hash.length-length string of zeros (and function HMAC is specified by RFC 2104 over keys and messages, hence, the above definition treats *Salt* as a key and *Secret* as a message when applying HMAC). In the context of key derivation, the salt is initially 0s and the secret is initially the pre-shared key or 0s if no such key was negotiated. The function's first output is known as `Early Secret`. It follows that

$$\texttt{Early Secret} = \mathsf{HKDF\text{-}Extract}(0\text{s}, \text{PSK}),$$

where PSK is 0s for (EC)DHE-only key exchange and otherwise the pre-shared key, which provides raw entropy without context. Context is added using function Derive-Secret, defined such that

$$\mathsf{Derive\text{-}Secret}(\mathit{Secret}, \mathit{Label}, \mathit{Transcript}) = \mathsf{HMAC}\left(\mathit{Secret}, \mathit{HkdfLabel} \parallel 0\text{x}01\right),$$

where *Transcript* is a concatenation of the protocol's messages (§2.3.1) and *HkdfLabel* is defined as the following message,[10] namely,

$$\mathit{HkdfLabel} = \mathsf{Hash.length} \parallel \text{“tls13}\_\text{”} \parallel \mathit{Label} \parallel \mathsf{Transcript\text{-}Hash}(\mathit{Transcript}).$$

Function Derive-Secret is used (with the empty context) to derive salt for subsequent applications of HKDF-Extract. Indeed, we have

$$\texttt{Handshake Secret} = \mathsf{HKDF\text{-}Extract}(\mathsf{Derive\text{-}Secret}(\texttt{Early Secret}, \text{“derived”}, \text{“”}), \text{K}),$$

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

$$\texttt{Master Secret} = \mathsf{HKDF\text{-}Extract}(\mathsf{Derive\text{-}Secret}(\texttt{Handshake Secret}, \text{“derived”}, \text{“”}), 0\text{s}),$$

noting that $\mathsf{Transcript\text{-}Hash}(\text{“”}) = \mathsf{Hash}(\text{“”})$, that is, the hash of the empty string (null ASCII character 0x00). 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,

$$\texttt{[sender]\_application\_traffic\_secret\_N+1} = \mathsf{Derive\text{-}Secret}(\texttt{[sender]\_application\_traffic\_secret\_N}, \text{“traffic upd”}, \text{“”}),$$

which is used to update application-traffic secrets.

[10] RFC 8446 defines function Derive-Secret in terms of functions HKDF-Expand-Label and HKDF-Expand, namely, $\mathsf{Derive\text{-}Secret}(\mathit{Secret}, \mathit{Label}, \mathit{Transcript}) = \mathsf{HKDF\text{-}Expand\text{-}Label}(\mathit{Secret}, \mathit{Label}, \mathsf{Transcript\text{-}Hash}(\mathit{Transcript}), \mathsf{Hash.length}) = \mathsf{HKDF\text{-}Expand}(\mathit{Secret}, \mathit{HkdfLabel}, \mathsf{Hash.length}) = \mathsf{HMAC}(\mathit{Secret}, \mathit{HkdfLabel} \parallel 0\text{x}01)$, where $\mathit{HkdfLabel} = \mathsf{Hash.length} \parallel \text{“tls13}\_\text{”} \parallel \mathit{Label} \parallel \mathsf{Transcript\text{-}Hash}(\mathit{Transcript})$. By comparison, we define function Derive-Secret more directly and defer the additional functions to Section 2.3.3, where we consider functions HKDF-Expand-Label and HKDF-Expand more generally (in particular, the former may omit transcript hashes in favour of strings and the latter may consider lengths other than Hash.length).

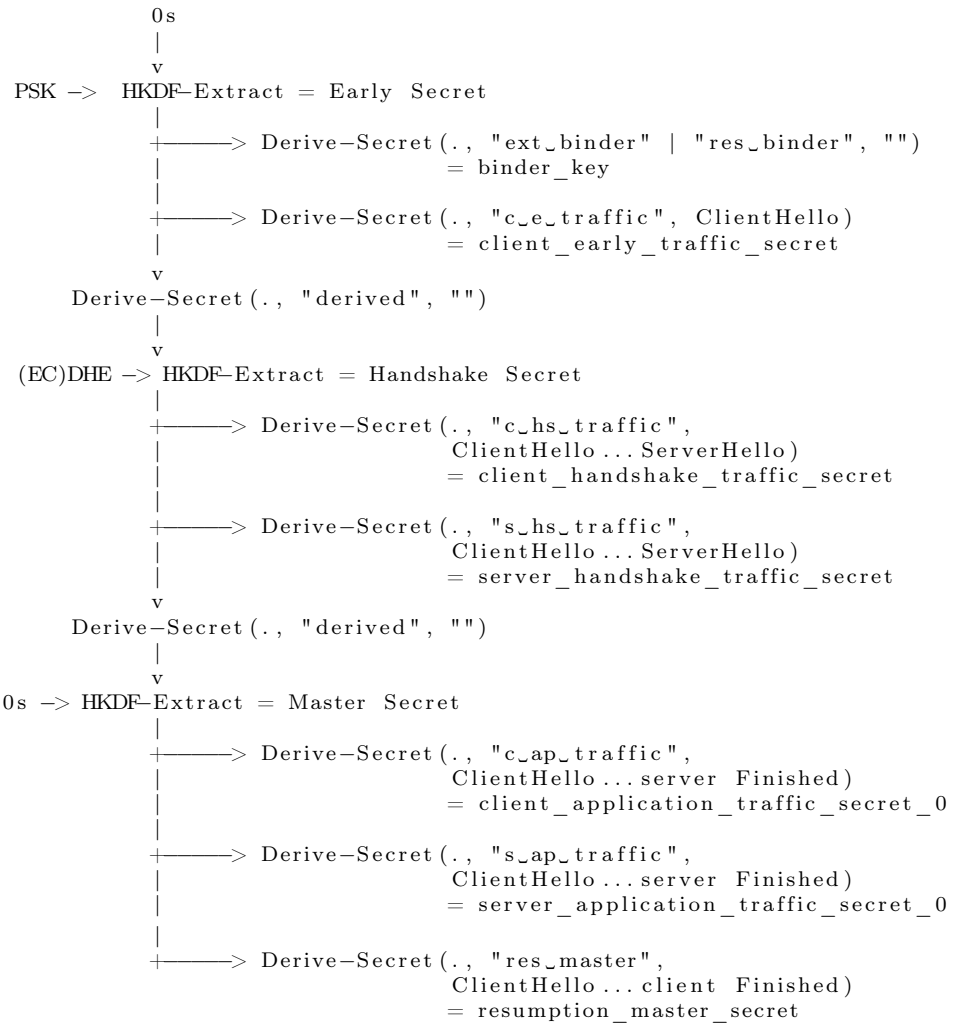

Figure 2: Key derivation process, showing application of functions HKDF-Extract and Derive-Secret to derive working keys. Function HKDF-Extract is shown inputting salt from the top and secrets from the left, and outputs to the bottom, where the output's name is shown to the right. Moreover, function 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 established independently of TLS) and the latter is used for resumption PSKs (those established by `NewSessionTicket` messages, using the resumption master secret of a previous handshake), hence, one type of PSK cannot be substituted for the other.

Source: This figure is excerpted from RFC 8446.

```
46  final class HKDF {
47      private final String hmacAlg;
48      private final Mac hmacObj;
49      private final int hmacLen;
61      HKDF(String hashAlg) throws NoSuchAlgorithmException {
64          hmacAlg = "Hmac" + hashAlg.replace("-", "");
65          hmacObj = JsseJce.getMac(hmacAlg);
66          hmacLen = hmacObj.getMacLength();
67      }
86      SecretKey extract(SecretKey salt, SecretKey inputKey, String keyAlg)
87              throws InvalidKeyException {
88          if (salt == null) {
89              salt = new SecretKeySpec(new byte[hmacLen], "HKDF-Salt");
90          }
91          hmacObj.init(salt);
92
93          return new SecretKeySpec(hmacObj.doFinal(inputKey.getEncoded()),
94                  keyAlg);
95      }
114     SecretKey extract(byte[] salt, SecretKey inputKey, String keyAlg)
115             throws InvalidKeyException {
116         if (salt == null) {
117             salt = new byte[hmacLen];
118         }
119         return extract(new SecretKeySpec(salt, "HKDF-Salt"), inputKey, keyAlg);
120     }
185 }
```

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.

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(0s, 0s) and is computed by class DHKeyExchange.DHEKAGenerator.DHEKAKeyDerivation or class ECDHKeyExchange.ECDHEKAKeyDerivation, 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 has replaced those classes with KAKeyDerivation in JDK-13.) So, for brevity, we only present the former class (Listing 22).

```
36  final class SSLSecretDerivation implements SSLKeyDerivation {
63      private final HandshakeContext context;
65      private final HashAlg hashAlg;
66      private final SecretKey secret;
67      private final byte[] transcriptHash;  // handshake messages transcript hash
69      SSLSecretDerivation(
70              HandshakeContext context, SecretKey secret) {
71          this.context = context;
72          this.secret = secret;
73          this.hashAlg = context.negotiatedCipherSuite.hashAlg;
76          context.handshakeHash.update();
77          this.transcriptHash = context.handshakeHash.digest();
78      }
85      public SecretKey deriveKey(String algorithm,
86              AlgorithmParameterSpec params) throws IOException {
87          SecretSchedule ks = SecretSchedule.valueOf(algorithm);
88          try {
89              byte[] expandContext;
90              if (ks == SecretSchedule.TlsSaltSecret) {
91                  if (hashAlg == HashAlg.H_SHA256) {
92                      expandContext = sha256EmptyDigest;
93                  } else if (hashAlg == HashAlg.H_SHA384) {
94                      expandContext = sha384EmptyDigest;
95                  } else {
96                      // unlikely, but please update if more hash algorithm
101                 }
102             } else {
103                 expandContext = transcriptHash;
104             }
105             byte[] hkdfInfo = createHkdfInfo(ks.label,
106                     expandContext, hashAlg.hashLength);
107
108             HKDF hkdf = new HKDF(hashAlg.name);
109             return hkdf.expand(secret, hkdfInfo, hashAlg.hashLength, algorithm);
110         } catch (GeneralSecurityException gse) {
111             throw (SSLHandshakeException) new SSLHandshakeException(
112                 "Could not generate secret").initCause(gse);
113         }
114     }
```

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.

```
132     private enum SecretSchedule {
133         // Note that we use enum name as the key/secret name.
134         TlsSaltSecret                       ("derived"),
135         TlsExtBinderKey                     ("ext binder"),
136         TlsResBinderKey                     ("res binder"),
137         TlsClientEarlyTrafficSecret         ("c e traffic"),
138         TlsEarlyExporterMasterSecret        ("e exp master"),
139         TlsClientHandshakeTrafficSecret     ("c hs traffic"),
140         TlsServerHandshakeTrafficSecret     ("s hs traffic"),
141         TlsClientAppTrafficSecret           ("c ap traffic"),
142         TlsServerAppTrafficSecret           ("s ap traffic"),
143         TlsExporterMasterSecret             ("exp master"),
144         TlsResumptionMasterSecret           ("res master");
145
146         private final byte[] label;
147
148         private SecretSchedule(String label) {
149             this.label = ("tls13 " + label).getBytes();
150         }
151     }
152 }
```

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

```
1152     private static void setUpPskKD(HandshakeContext hc,
1153             SecretKey psk) throws SSLHandshakeException {
1159         try {
1160             CipherSuite.HashAlg hashAlg = hc.negotiatedCipherSuite.hashAlg;
1161             HKDF hkdf = new HKDF(hashAlg.name);
1162             byte[] zeros = new byte[hashAlg.hashLength];
1163             SecretKey earlySecret = hkdf.extract(zeros, psk, "TlsEarlySecret");
1164             hc.handshakeKeyDerivation =
1165                     new SSLSecretDerivation(hc, earlySecret);
1166         } catch (GeneralSecurityException gse) {
1167             throw (SSLHandshakeException) new SSLHandshakeException(
1168                 "Could not generate secret").initCause(gse);
1169         }
1170     }
```

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

```
449     private static final
450             class DHEKAKeyDerivation implements SSLKeyDerivation {
451         private final HandshakeContext context;
452         private final PrivateKey localPrivateKey;
453         private final PublicKey peerPublicKey;
464         public SecretKey deriveKey(String algorithm,
465                 AlgorithmParameterSpec params) throws IOException {
469             return t13DeriveKey(algorithm, params);
471         }
499         private SecretKey t13DeriveKey(String algorithm,
500                 AlgorithmParameterSpec params) throws IOException {
501             try {
502                 KeyAgreement ka = JsseJce.getKeyAgreement("DiffieHellman");
503                 ka.init(localPrivateKey);
504                 ka.doPhase(peerPublicKey, true);
505                 SecretKey sharedSecret =
506                         ka.generateSecret("TlsPremasterSecret");
507
508                 HashAlg hashAlg = context.negotiatedCipherSuite.hashAlg;
509                 SSLKeyDerivation kd = context.handshakeKeyDerivation;
510                 HKDF hkdf = new HKDF(hashAlg.name);
511                 if (kd == null) {   // No PSK is in use.
512                     // If PSK is not in use Early Secret will still be
513                     // HKDF-Extract(0, 0).
514                     byte[] zeros = new byte[hashAlg.hashLength];
515                     SecretKeySpec ikm =
516                             new SecretKeySpec(zeros, "TlsPreSharedSecret");
517                     SecretKey earlySecret =
518                             hkdf.extract(zeros, ikm, "TlsEarlySecret");
519                     kd = new SSLSecretDerivation(context, earlySecret);
520                 }
521
522                 // derive salt secret
523                 SecretKey saltSecret = kd.deriveKey("TlsSaltSecret", null);
524
525                 // derive handshake secret
526                 return hkdf.extract(saltSecret, sharedSecret, algorithm);
527             } catch (GeneralSecurityException gse) {
528                 throw (SSLHandshakeException) new SSLHandshakeException(
529                     "Could not generate secret").initCause(gse);
530             }
531         }
532     }
```

Listing 22: Class DHKeyExchange.DHEKAGenerator.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`.

2.3.3 Traffic keys

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

$$\begin{aligned}&\mathsf{HKDF\text{-}Expand\text{-}Label}(\mathit{Secret}, \mathit{Label}, \mathit{Context}, \mathit{Length})\\ &\qquad\qquad = \mathsf{HKDF\text{-}Expand}(\mathit{Secret}, \mathit{HkdfLabel}, \mathit{Length}),\end{aligned}$$

where $\mathit{HkdfLabel} = \mathit{Length} \parallel$ "tls13␣" $\parallel \mathit{Label} \parallel \mathit{Context}$ and function HKDF-Expand is defined by RFC 5869 such that $\mathsf{HKDF\text{-}Expand}(\mathit{Secret}, \mathit{ExpLabel}, \mathit{Length})$ outputs the first Length-bytes of $T_1 \parallel \cdots \parallel T_n$, where $n = \lceil \frac{\mathit{Length}}{\mathsf{Hash.length}} \rceil$ and

$$\begin{aligned}T_0 &= \text{“”}\\ T_1 &= \mathsf{HMAC}(\mathit{Secret}, T_0 \parallel \mathit{ExpLabel} \parallel \text{0x01})\\ T_2 &= \mathsf{HMAC}(\mathit{Secret}, T_1 \parallel \mathit{ExpLabel} \parallel \text{0x02})\\ &\vdots\end{aligned}$$

Function HKDF-Expand-Label may input *Context* as the null ASCII character 0x00, denoted “”.

> Function HKDF-Expand is implemented by class HKDF (Listing 23) and traffic keys are derived by class SSLTrafficKeyDerivation (Listing 24).

Returning to key derivation, we derive the following traffic keys:

$$\begin{aligned}\texttt{[sender]\_write\_key} &= \mathsf{HKDF\text{-}Expand\text{-}Label}(\texttt{[secret]}, \text{“}\mathit{key}\text{”}, \text{“”}, \mathit{key\_length})\\ \texttt{[sender]\_write\_iv} &= \mathsf{HKDF\text{-}Expand\text{-}Label}(\texttt{[secret]}, \text{“}\mathit{iv}\text{”}, \text{“”}, \mathit{iv\_length})\end{aligned}$$

where `[sender]` is either `client` or `server`, and `[secret]` is taken from the secrets listed in Table 2.

```
137     SecretKey expand(SecretKey pseudoRandKey, byte[] info, int outLen,
138             String keyAlg) throws InvalidKeyException {
139         byte[] kdfOutput;
150         hmacObj.init(pseudoRandKey);
154         int rounds = (outLen + hmacLen - 1) / hmacLen;
155         kdfOutput = new byte[rounds * hmacLen];
156         int offset = 0;
157         int tLength = 0;
158
159         for (int i = 0; i < rounds ; i++) {
162             try {
163                 // Add T(i).  This will be an empty string on the first
164                 // iteration since tLength starts at zero.  After the first
165                 // iteration, tLength is changed to the HMAC length for the
166                 // rest of the loop.
167                 hmacObj.update(kdfOutput,
168                         Math.max(0, offset - hmacLen), tLength);
169                 hmacObj.update(info);                          // Add info
170                 hmacObj.update((byte)(i + 1));                 // Add round number
171                 hmacObj.doFinal(kdfOutput, offset);
172
173                 tLength = hmacLen;
174                 offset += hmacLen;                          // For next iteration
175             } catch (ShortBufferException sbe) {
176                 // This really shouldn't happen given that we've
177                 // sized the buffers to their largest possible size up-front,
178                 // but just in case...
179                 throw new RuntimeException(sbe);
180             }
181         }
182
183         return new SecretKeySpec(kdfOutput, 0, outLen, keyAlg);
184     }
```

Listing 23: Class HKDF (omitted from Listing 18) defines method expand to implement function HKDF-Expand. A buffer kdfOutput of length $\mathsf{Hash.length} \cdot \lceil \frac{Length}{\mathsf{Hash.length}} \rceil$ is initialised (Lines 139 & 154–155) and an HMAC is initialised with the input secret as a key (Line 150). The for-loop computes $T_1, T_2, ...$ 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).

```
135   static final class T13TrafficKeyDerivation implements SSLKeyDerivation {
136       private final CipherSuite cs;
137       private final SecretKey secret;
139       T13TrafficKeyDerivation(
140               HandshakeContext context, SecretKey secret) {
141           this.secret = secret;
142           this.cs = context.negotiatedCipherSuite;
143       }
146       public SecretKey deriveKey(String algorithm,
147               AlgorithmParameterSpec params) throws IOException {
148           KeySchedule ks = KeySchedule.valueOf(algorithm);
149           try {
150               HKDF hkdf = new HKDF(cs.hashAlg.name);
151               byte[] hkdfInfo =
152                       createHkdfInfo(ks.label, ks.getKeyLength(cs));
153               return hkdf.expand(secret, hkdfInfo,
154                       ks.getKeyLength(cs),
155                       ks.getAlgorithm(cs, algorithm));
156           } catch (GeneralSecurityException gse) {
157               throw (SSLHandshakeException)(new SSLHandshakeException(
158                   "Could not generate secret").initCause(gse));
159           }
160       }
177   }
179   private enum KeySchedule {
180       // Note that we use enum name as the key/ name.
181       TlsKey              ("key", false),
182       TlsIv               ("iv",  true),
183       TlsUpdateNplus1     ("traffic upd", false);
184
185       private final byte[] label;
186       private final boolean isIv;
187
188       private KeySchedule(String label, boolean isIv) {
189           this.label = ("tls13 " + label).getBytes();
190           this.isIv = isIv;
191       }
192
193       int getKeyLength(CipherSuite cs) {
194           if (this == KeySchedule.TlsUpdateNplus1)
195               return cs.hashAlg.hashLength;
196           return isIv ? cs.bulkCipher.ivSize : cs.bulkCipher.keySize;
197       }
198
199       String getAlgorithm(CipherSuite cs, String algorithm) {
200           return isIv ? algorithm : cs.bulkCipher.algorithm;
201       }
202   }
```

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 HKDR.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.

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␣hs␣traffic and c␣hs␣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. **return null**; and **return**; in catch-branches), α-renaming of one variable, and whitespace (and some obsolete, commented-out code). (Refactoring could eliminate unnecessary code.[a]) So, for brevity, we only present server-side handshake-traffic key derivation (Listings 25 & 26).

[a]The OpenJDK team are aware of refactoring opportunities (`https://mail.openjdk.java.net/pipermail/security-dev/2020-May/021928.html`) and are tracking changes (`https://bugs.openjdk.java.net/browse/JDK-8245983`).

Traffic secrets `client_handshake_traffic_secret` and `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.

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.

```
588             // Refresh handshake hash
589             shc.handshakeHash.update();
591             // Change client/server handshake traffic secrets.
592             SSLKeyExchange ke = shc.handshakeKeyExchange;
593             if (ke == null) {
594                 // unlikely
595                 shc.conContext.fatal(Alert.INTERNAL_ERROR,
596                         "Not negotiated key shares");
597                 return null;     // make the compiler happy
598             }
600             SSLKeyDerivation handshakeKD = ke.createKeyDerivation(shc);
601             SecretKey handshakeSecret = handshakeKD.deriveKey(
602                     "TlsHandshakeSecret", null);
603
604             SSLTrafficKeyDerivation kdg =
605                 SSLTrafficKeyDerivation.valueOf(shc.negotiatedProtocol);
613
614             SSLKeyDerivation kd =
615                     new SSLSecretDerivation(shc, handshakeSecret);
```

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 parameterised 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 SSLTrafficKeyDerivation.T13TrafficKeyDerivation.

```
617             // update the handshake traffic read keys.
618             SecretKey readSecret = kd.deriveKey(
619                     "TlsClientHandshakeTrafficSecret", null);
620             SSLKeyDerivation readKD =
621                     kdg.createKeyDerivation(shc, readSecret);
622             SecretKey readKey = readKD.deriveKey(
623                     "TlsKey", null);
624             SecretKey readIvSecret = readKD.deriveKey(
625                     "TlsIv", null);
626             IvParameterSpec readIv =
627                     new IvParameterSpec(readIvSecret.getEncoded());
628             SSLReadCipher readCipher;
629             try {
630                 readCipher =
631                     shc.negotiatedCipherSuite.bulkCipher.createReadCipher(
632                         Authenticator.valueOf(shc.negotiatedProtocol),
633                         shc.negotiatedProtocol, readKey, readIv,
634                         shc.sslContext.getSecureRandom());
635             } catch (GeneralSecurityException gse) {
636                 // unlikely
640             }
641
642             shc.baseReadSecret = readSecret;
643             shc.conContext.inputRecord.changeReadCiphers(readCipher);
644
645             // update the handshake traffic write secret.
646             SecretKey writeSecret = kd.deriveKey(
647                     "TlsServerHandshakeTrafficSecret", null);
648             SSLKeyDerivation writeKD =
649                     kdg.createKeyDerivation(shc, writeSecret);
650             SecretKey writeKey = writeKD.deriveKey(
651                     "TlsKey", null);
652             SecretKey writeIvSecret = writeKD.deriveKey(
653                     "TlsIv", null);
654             IvParameterSpec writeIv =
655                     new IvParameterSpec(writeIvSecret.getEncoded());
656             SSLWriteCipher writeCipher;
657             try {
658                 writeCipher =
659                     shc.negotiatedCipherSuite.bulkCipher.createWriteCipher(
660                         Authenticator.valueOf(shc.negotiatedProtocol),
661                         shc.negotiatedProtocol, writeKey, writeIv,
662                         shc.sslContext.getSecureRandom());
663             } catch (GeneralSecurityException gse) {
664                 // unlikely
668             }
669
670             shc.baseWriteSecret = writeSecret;
671             shc.conContext.outputRecord.changeWriteCiphers(
672                     writeCipher, (clientHello.sessionId.length() != 0));
673
674             // Update the context for master key derivation.
675             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).

`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.[11]

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 (constructed with the private key corresponding to the public key in the certificate) over a hash of the protocol's transcript, thereby, proving possession of the private key used for signing, hence, identifying the server.

A `Certificate` message 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. (Any extensions must respond to ones listed in the `ClientHello` message. Moreover, an extension that applies to the entire chain should appear in the first extension listed.) Certificates must be DER-encoded X.509v3 certificates, unless an alternative certificate type was negotiated (using extension `server_certificate_type`). 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's public key should be compatible with an algorithm amongst those offered, by the client, for `CertificateVerify` messages (i.e., advertised by `ClientHello.signature_algorithms`). Any remaining certificates' public keys should be compatible with an algorithm offered for `Certificate` messages (i.e., those advertised by extension `signature_algorithms_cert` if present and extension `signature_algorithms` otherwise). (When a certificate chain cannot be constructed from compatible algorithms, the chain may rely on algorithms not offered by the client, except for SHA-1, which must not be used, unless offered.) All certificates must (explicitly) permit signature verification (whenever certificates include a Key Usage extension). (Self-signed certificates or trust anchors may be signed with any algorithm, trust anchor certificates may be omitted when they are known to be in the client's possession, and, for raw public keys, the list must contain exactly one certificate.)

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

[11] RFC 8446 does not permit PSK-based key exchange with `Certificate` and `CertificateVerify` messages from the server; (direct) certificate-based server authentication is unsupported for PSK-based key exchange. (The specification notes that future documents may support such authentication.) Certificate-based client authentication is compatible with PSK-based key exchange (Appendix C).

```
732   static final class CertificateEntry {
733       final byte[] encoded;       // encoded cert or public key
734       private final SSLExtensions extensions;
736       CertificateEntry(byte[] encoded, SSLExtensions extensions) {
737           this.encoded = encoded;
738           this.extensions = extensions;
739       }
777   }
782   static final class T13CertificateMessage extends HandshakeMessage {
783       private final byte[] requestContext;
784       private final List<CertificateEntry> certEntries;
786       T13CertificateMessage(HandshakeContext context,
787               byte[] requestContext, X509Certificate[] certificates)
788               throws SSLException, CertificateException  {
789           super(context);
790
791           this.requestContext = requestContext.clone();
792           this.certEntries = new LinkedList<>();
793           for (X509Certificate cert : certificates) {
794               byte[] encoded = cert.getEncoded();
795               SSLExtensions extensions = new SSLExtensions(this);
796               certEntries.add(new CertificateEntry(encoded, extensions));
797           }
798       }
913   }
```

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.

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`: A signing algorithm, which must be amongst those offered by the client (`ClientHello.signature_algorithms`), unless unless a certificate chain cannot be constructed from compatible algorithms.

`signature`: A signature, produced by the aforementioned algorithm, over the concatenation of: 0x20 repeated 64 times, string "TLS 1.3, server CertificateVerify", 0x00, and the transcript hash (§2.3.1).

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:

```
73  static final HandshakeProducer t13HandshakeProducer =
74      new T13CertificateProducer();
918 private static final
919         class T13CertificateProducer implements HandshakeProducer {
926     public byte[] produce(ConnectionContext context,
927             HandshakeMessage message) throws IOException {
929         HandshakeContext hc = (HandshakeContext)context;
930         if (hc.sslConfig.isClientMode) {
931             return onProduceCertificate(
932                     (ClientHandshakeContext)context, message);
933         } else {
934             return onProduceCertificate(
935                     (ServerHandshakeContext)context, message);
936         }
937     }
939     private byte[] onProduceCertificate(ServerHandshakeContext shc,
940             HandshakeMessage message) throws IOException {
941         ClientHelloMessage clientHello = (ClientHelloMessage)message;
943         SSLPossession pos = choosePossession(shc, clientHello);
955         X509Possession x509Possession = (X509Possession)pos;
956         X509Certificate[] localCerts = x509Possession.popCerts;
963         // update the context
964         shc.handshakePossessions.add(x509Possession);
965         shc.handshakeSession.setLocalPrivateKey(
966                 x509Possession.popPrivateKey);
967         shc.handshakeSession.setLocalCertificates(localCerts);
968         T13CertificateMessage cm;
969         try {
970             cm = new T13CertificateMessage(shc, (new byte[0]), localCerts);
971         } catch (SSLException | CertificateException ce) {
974             return null;     // make the complier happy
975         }
983         // Process extensions for each CertificateEntry.
984         // Since there can be multiple CertificateEntries within a
985         // single CT message, we will pin a specific CertificateEntry
986         // into the ServerHandshakeContext so individual extension
987         // producers know which X509Certificate it is processing in
988         // each call.
989         SSLExtension[] enabledCTExts = shc.sslConfig.getEnabledExtensions(
990                 SSLHandshake.CERTIFICATE,
991                 Arrays.asList(ProtocolVersion.PROTOCOLS_OF_13));
992         for (CertificateEntry certEnt : cm.certEntries) {
993             shc.currentCertEntry = certEnt;
994             certEnt.extensions.produce(shc, enabledCTExts);
995         }
1001        // Output the handshake message.
1002        cm.write(shc.handshakeOutput);
1003        shc.handshakeOutput.flush();
1005        // The handshake message has been delivered.
1006        return null;
1007    }
1126 }
```

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).

```
1009        private static SSLPossession choosePossession(
1010                HandshakeContext hc,
1011                ClientHelloMessage clientHello) throws IOException {
1022            for (SignatureScheme ss : hc.peerRequestedCertSignSchemes) {
1031                // Don't select a signature scheme unless we will be able to
1032                // produce a CertificateVerify message later
1033                if (SignatureScheme.getPreferableAlgorithm(
1034                        hc.peerRequestedSignatureSchemes,
1035                        ss, hc.negotiatedProtocol) == null) {
1043                    continue;
1044                }
1046                SSLAuthentication ka = X509Authentication.valueOf(ss);
1047                if (ka == null) {
1053                    continue;
1054                }
1056                SSLPossession pos = ka.createPossession(hc);
1057                if (pos == null) {
1062                    continue;
1063                }
1065                return pos;
1066            }
1071            return null;
1072        }
```

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).

```
71   static final SSLConsumer t13HandshakeConsumer =
72       new T13CertificateConsumer();
1131 private static final class T13CertificateConsumer implements SSLConsumer {
1138     public void consume(ConnectionContext context,
1139             ByteBuffer message) throws IOException {
1141         HandshakeContext hc = (HandshakeContext)context;
1144         hc.handshakeConsumers.remove(SSLHandshake.CERTIFICATE.id);
1145         T13CertificateMessage cm = new T13CertificateMessage(hc, message);
1146         if (hc.sslConfig.isClientMode) {
1151             onConsumeCertificate((ClientHandshakeContext)context, cm);
1152         } else {
1157             onConsumeCertificate((ServerHandshakeContext)context, cm);
1158         }
1159     }
1186     private void onConsumeCertificate(ClientHandshakeContext chc,
1187             T13CertificateMessage certificateMessage )throws IOException {
1194         // Each CertificateEntry will have its own set of extensions
1195         // which must be consumed.
1196         SSLExtension[] enabledExtensions =
1197             chc.sslConfig.getEnabledExtensions(SSLHandshake.CERTIFICATE);
1198         for (CertificateEntry certEnt : certificateMessage.certEntries) {
1199             certEnt.extensions.consumeOnLoad(chc, enabledExtensions);
1200         }
1202         // check server certificate entries
1203         X509Certificate[] srvCerts =
1204                 checkServerCerts(chc, certificateMessage.certEntries);
1207         // update
1209         chc.handshakeCredentials.add(
1210             new X509Credentials(srvCerts[0].getPublicKey(), srvCerts));
1211         chc.handshakeSession.setPeerCertificates(srvCerts);
1212     }
1369 }
```

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).

```
793   static final class T13CertificateVerifyMessage extends HandshakeMessage {
794       private static final byte[] serverSignHead = new byte[] {
795           // repeated 0x20 for 64 times
813           // "TLS 1.3, server CertificateVerify" + 0x00
823       };
825       private static final byte[] clientSignHead = new byte[] {
826           // repeated 0x20 for 64 times
844           // "TLS 1.3, client CertificateVerify" + 0x00
854       };
857       // the signature algorithm
858       private final SignatureScheme signatureScheme;
860       // signature bytes
861       private final byte[] signature;
863       T13CertificateVerifyMessage(HandshakeContext context,
864               X509Possession x509Possession) throws IOException {
865           super(context);
866
867           this.signatureScheme = SignatureScheme.getPreferableAlgorithm(
868                   context.peerRequestedSignatureSchemes,
869                   x509Possession.popPrivateKey,
870                   context.negotiatedProtocol);
878           byte[] hashValue = context.handshakeHash.digest();
879           byte[] contentCovered;
880           if (context.sslConfig.isClientMode) {
881               contentCovered = Arrays.copyOf(clientSignHead,
882                       clientSignHead.length + hashValue.length);
883               System.arraycopy(hashValue, 0, contentCovered,
884                       clientSignHead.length, hashValue.length);
885           } else {
886               contentCovered = Arrays.copyOf(serverSignHead,
887                       serverSignHead.length + hashValue.length);
888               System.arraycopy(hashValue, 0, contentCovered,
889                       serverSignHead.length, hashValue.length);
890           }
891
892           byte[] temproary = null;
893           try {
894               Signature signer =
895                   signatureScheme.getSignature(x509Possession.popPrivateKey);
896               signer.update(contentCovered);
897               temproary = signer.sign();
898           } catch (NoSuchAlgorithmException |
899                   InvalidAlgorithmParameterException nsae) {
900               context.conContext.fatal(Alert.INTERNAL_ERROR,
901                       "Unsupported signature algorithm (" +
902                       signatureScheme.name +
903                       ") used in CertificateVerify handshake message", nsae);
904           } catch (InvalidKeyException | SignatureException ikse) {
905               context.conContext.fatal(Alert.HANDSHAKE_FAILURE,
906                       "Cannot produce CertificateVerify signature", ikse);
907           }
908
909           this.signature = temproary;
910       }
1030  }
```

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).

```
912     T13CertificateVerifyMessage(HandshakeContext context,
913             ByteBuffer m) throws IOException {
914         super(context);
925         // SignatureAndHashAlgorithm algorithm
926         int ssid = Record.getInt16(m);
927         this.signatureScheme = SignatureScheme.valueOf(ssid);
941         // read and verify the signature
942         X509Credentials x509Credentials = null;
943         for (SSLCredentials cd : context.handshakeCredentials) {
944             if (cd instanceof X509Credentials) {
945                 x509Credentials = (X509Credentials)cd;
946                 break;
947             }
948         }
957         this.signature = Record.getBytes16(m);
958
959         byte[] hashValue = context.handshakeHash.digest();
960         byte[] contentCovered;
961         if (context.sslConfig.isClientMode) {
962             contentCovered = Arrays.copyOf(serverSignHead,
963                     serverSignHead.length + hashValue.length);
964             System.arraycopy(hashValue, 0, contentCovered,
965                     serverSignHead.length, hashValue.length);
966         } else {
967             contentCovered = Arrays.copyOf(clientSignHead,
968                     clientSignHead.length + hashValue.length);
969             System.arraycopy(hashValue, 0, contentCovered,
970                     clientSignHead.length, hashValue.length);
971         }
972
973         try {
974             Signature signer =
975                 signatureScheme.getSignature(x509Credentials.popPublicKey);
976             signer.update(contentCovered);
977             if (!signer.verify(signature)) {
978                 context.conContext.fatal(Alert.HANDSHAKE_FAILURE,
979                     "Invalid CertificateVerify signature");
980             }
981         } catch (NoSuchAlgorithmException |
982                 InvalidAlgorithmParameterException nsae) {
983             context.conContext.fatal(Alert.INTERNAL_ERROR,
984                     "Unsupported signature algorithm (" +
985                     signatureScheme.name +
986                     ") used in CertificateVerify handshake message", nsae);
987         } catch (InvalidKeyException | SignatureException ikse) {
988             context.conContext.fatal(Alert.HANDSHAKE_FAILURE,
989                     "Cannot verify CertificateVerify signature", ikse);
990         }
991     }
```

Listing 32: Class CertificateVerify.T13CertificateVerifyMessage (omitted from Listing 31) defines a constructor which instantiates a `CertificateVerify` message from an input buffer, parametrising 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).

```
60  static final HandshakeProducer t13HandshakeProducer =
61      new T13CertificateVerifyProducer();
1035 private static final
1036         class T13CertificateVerifyProducer implements HandshakeProducer {
1043     public byte[] produce(ConnectionContext context,
1044             HandshakeMessage message) throws IOException {
1045         // The producing happens in handshake context only.
1046         HandshakeContext hc = (HandshakeContext)context;
1048         X509Possession x509Possession = null;
1049         for (SSLPossession possession : hc.handshakePossessions) {
1050             if (possession instanceof X509Possession) {
1051                 x509Possession = (X509Possession)possession;
1052                 break;
1053             }
1066         if (hc.sslConfig.isClientMode) {
1067             return onProduceCertificateVerify(
1068                     (ClientHandshakeContext)context, x509Possession);
1069         } else {
1070             return onProduceCertificateVerify(
1071                     (ServerHandshakeContext)context, x509Possession);
1072         }
1073     }
1075     private byte[] onProduceCertificateVerify(ServerHandshakeContext shc,
1076             X509Possession x509Possession) throws IOException {
1077         T13CertificateVerifyMessage cvm =
1078                 new T13CertificateVerifyMessage(shc, x509Possession);
1084         // Output the handshake message.
1085         cvm.write(shc.handshakeOutput);
1086         shc.handshakeOutput.flush();
1088         // The handshake message has been delivered.
1089         return null;
1090     }
1108 }
```

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).

```
58  static final SSLConsumer t13HandshakeConsumer =
59      new T13CertificateVerifyConsumer();
1113 private static final
1114         class T13CertificateVerifyConsumer implements SSLConsumer {
1121     public void consume(ConnectionContext context,
1122             ByteBuffer message) throws IOException {
1124         HandshakeContext hc = (HandshakeContext)context;
1125         T13CertificateVerifyMessage cvm =
1126                 new T13CertificateVerifyMessage(hc, message);
1141     }
1142 }
```

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.

`verify_data`: An HMAC over the entire handshake.

The HMAC is computed as

$$\mathsf{HMAC}(\texttt{finished\_key}, \mathsf{Transcript\text{-}Hash}(\mathit{Transcript}))$$

where *Transcript* is a concatenation of the protocol's messages (§2.3.1),

$$\texttt{finished\_key} = \mathsf{HKDF\text{-}Expand\text{-}Label}(\texttt{[sender]\_handshake\_traffic\_secret}, \text{“}\mathit{finished}\text{”}, \text{“”}, \mathsf{Hash.length}),$$

and `[sender]` is `server` when the `Finished` message originates from a server to conclude a handshake and `client` when originating from a client.

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.

Application data

TLS protects application-layer communication independently of specific applications. Independence is readily apparent from the specification: There is no mention of interaction between applications and TLS. Designers and implementors must decide for themselves how to use TLS within their applications. For instance, when to initiate a handshake and how to validate certificates.

2.6 Early data

For PSK-based key exchange, clients may exceptionally start sending encrypted application data immediately after `ClientHello` messages (before receiving a `ServerHello` message),[12] 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).[13] Such early data requires the `ClientHello` message to include extensions `early_data` and `pre_shared_key`, and application data must be

[12] RFC 8446 only permits clients to send early data when the pre-shared key is associated with data permitting them to do so. For resumption PSKs, permission is granted by inclusion of extension `early_data` in `NewSessionTicket` messages (§2.7.1).

[13] RFC 8446 discusses anti-replay defences and notes that single-use PSKs enjoy forward secrecy.

```
69  private static final class FinishedMessage extends HandshakeMessage {
70      private final byte[] verifyData;
72      FinishedMessage(HandshakeContext context) throws IOException {
73          super(context);
75          VerifyDataScheme vds =
76                  VerifyDataScheme.valueOf(context.negotiatedProtocol);
78          byte[] vd = null;
79          try {
80              vd = vds.createVerifyData(context, false);
81          } catch (IOException ioe) {
82              context.conContext.fatal(Alert.ILLEGAL_PARAMETER,
83                      "Failed to generate verify_data", ioe);
84          }
86          this.verifyData = vd;
87      }
89      FinishedMessage(HandshakeContext context,
90              ByteBuffer m) throws IOException {
91          super(context);
92          int verifyDataLen = 12;
93          if (context.negotiatedProtocol == ProtocolVersion.SSL30) {
94              verifyDataLen = 36;
95          } else if (context.negotiatedProtocol.useTLS13PlusSpec()) {
96              verifyDataLen =
97                      context.negotiatedCipherSuite.hashAlg.hashLength;
98          }
106         this.verifyData = new byte[verifyDataLen];
107         m.get(verifyData);
109         VerifyDataScheme vd =
110                 VerifyDataScheme.valueOf(context.negotiatedProtocol);
111         byte[] myVerifyData;
112         try {
113             myVerifyData = vd.createVerifyData(context, true);
114         } catch (IOException ioe) {
117             return;      // make the compiler happy
118         }
119         if (!MessageDigest.isEqual(myVerifyData, verifyData)) {
120             context.conContext.fatal(Alert.ILLEGAL_PARAMETER,
121                     "The Finished message cannot be verified.");
122         }
123     }
156 }
```

Listing 35: Class Finished.FinishedMessage defines the one field of a `Finished` message (Line 70) and two constructors to instantiate it. The first constructor parameterises 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).

```
326   private static final
327           class T13VerifyDataGenerator implements VerifyDataGenerator {
328       private static final byte[] hkdfLabel = "tls13 finished".getBytes();
329       private static final byte[] hkdfContext = new byte[0];
332       public byte[] createVerifyData(HandshakeContext context,
333               boolean isValidation) throws IOException {
335           HashAlg hashAlg =
336                   context.negotiatedCipherSuite.hashAlg;
337           SecretKey secret = isValidation ?
338                   context.baseReadSecret : context.baseWriteSecret;
339           SSLBasicKeyDerivation kdf = new SSLBasicKeyDerivation(
340                   secret, hashAlg.name,
341                   hkdfLabel, hkdfContext, hashAlg.hashLength);
342           AlgorithmParameterSpec keySpec =
343                   new SecretSizeSpec(hashAlg.hashLength);
344           SecretKey finishedSecret =
345                   kdf.deriveKey("TlsFinishedSecret", keySpec);
346
347           String hmacAlg =
348               "Hmac" + hashAlg.name.replace("-", "");
349           try {
350               Mac hmac = JsseJce.getMac(hmacAlg);
351               hmac.init(finishedSecret);
352               return hmac.doFinal(context.handshakeHash.digest());
353           } catch (NoSuchAlgorithmException |InvalidKeyException ex) {
354               throw new ProviderException(
355                       "Failed to generate verify_data", ex);
356           }
357       }
358   }
```

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.)

```
63      static final HandshakeProducer t13HandshakeProducer =
64          new T13FinishedProducer();
630     private static final
631             class T13FinishedProducer implements HandshakeProducer {
638         public byte[] produce(ConnectionContext context,
639                 HandshakeMessage message) throws IOException {
641             HandshakeContext hc = (HandshakeContext)context;
642             if (hc.sslConfig.isClientMode) {
643                 return onProduceFinished(
644                         (ClientHandshakeContext)context, message);
645             } else {
646                 return onProduceFinished(
647                         (ServerHandshakeContext)context, message);
648             }
649         }
738         private byte[] onProduceFinished(ServerHandshakeContext shc,
739                 HandshakeMessage message) throws IOException {
740             // Refresh handshake hash
741             shc.handshakeHash.update();
743             FinishedMessage fm = new FinishedMessage(shc);
749             // Output the handshake message.
750             fm.write(shc.handshakeOutput);
751             shc.handshakeOutput.flush();
753             // Change client/server application traffic secrets.
754             SSLKeyDerivation kd = shc.handshakeKeyDerivation;
762             SSLTrafficKeyDerivation kdg =
763                     SSLTrafficKeyDerivation.valueOf(shc.negotiatedProtocol);
772             // derive salt secret
773             try {
774                 SecretKey saltSecret = kd.deriveKey("TlsSaltSecret", null);
775
776                 // derive application secrets
777                 HashAlg hashAlg = shc.negotiatedCipherSuite.hashAlg;
778                 HKDF hkdf = new HKDF(hashAlg.name);
779                 byte[] zeros = new byte[hashAlg.hashLength];
780                 SecretKeySpec sharedSecret =
781                         new SecretKeySpec(zeros, "TlsZeroSecret");
782                 SecretKey masterSecret =
783                     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, parameterised 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, parameterised 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).

```
785                 SSLKeyDerivation secretKD =
786                         new SSLSecretDerivation(shc, masterSecret);
788                 // update the handshake traffic write keys.
789                 SecretKey writeSecret = secretKD.deriveKey(
790                         "TlsServerAppTrafficSecret", null);
791                 SSLKeyDerivation writeKD =
792                         kdg.createKeyDerivation(shc, writeSecret);
793                 SecretKey writeKey = writeKD.deriveKey(
794                         "TlsKey", null);
795                 SecretKey writeIvSecret = writeKD.deriveKey(
796                         "TlsIv", null);
797                 IvParameterSpec writeIv =
798                         new IvParameterSpec(writeIvSecret.getEncoded());
799                 SSLWriteCipher writeCipher =
800                         shc.negotiatedCipherSuite.bulkCipher.createWriteCipher(
801                                 Authenticator.valueOf(shc.negotiatedProtocol),
802                                 shc.negotiatedProtocol, writeKey, writeIv,
803                                 shc.sslContext.getSecureRandom());
804
805                 shc.baseWriteSecret = writeSecret;
806                 shc.conContext.outputRecord.changeWriteCiphers(
807                         writeCipher, false);
808
809                 // update the context for the following key derivation
810                 shc.handshakeKeyDerivation = secretKD;
811             } catch (GeneralSecurityException gse) {
814                 return null;    // make the compiler happy
815             }
824             // update the context
825             shc.handshakeConsumers.put(
826                     SSLHandshake.FINISHED.id, SSLHandshake.FINISHED);
827
828             // The handshake message has been delivered.
829             return null;
830         }
831     }
```

Listing 38: Class Finished.T13FinishedProducer defines method onProduceFinished (continued from Listing 37) to derive traffic secret `server_application_traffic_secret_0` from an instance of class SSLSecretDerivation, parameterised by `Master Secret` (Lines 785–790); constructs an instance of SSLTrafficKeyDerivation.T13TrafficKeyDerivation from that secret (Lines 791–792); and uses that instance to derive the corresponding traffic keys `server_write_key` (Lines 793–794) and `server_write_iv` (Lines 795–796), used to encrypt (and write) outgoing traffic (Lines 797–807). Moreover, the method prepares the server's active context for the client's response (Lines 825–826).

```
61      static final SSLConsumer t13HandshakeConsumer =
62          new T13FinishedConsumer();
836     private static final class T13FinishedConsumer implements SSLConsumer {
843         public void consume(ConnectionContext context,
844                 ByteBuffer message) throws IOException {
846             HandshakeContext hc = (HandshakeContext)context;
847             if (hc.sslConfig.isClientMode) {
848                 onConsumeFinished(
849                         (ClientHandshakeContext)context, message);
850             } else {
851                 onConsumeFinished(
852                         (ServerHandshakeContext)context, message);
853             }
854         }
856         private void onConsumeFinished(ClientHandshakeContext chc,
857                 ByteBuffer message) throws IOException {
858             FinishedMessage fm = new FinishedMessage(chc, message);
874             //
875             // update
876             //
877             // A change_cipher_spec record received after the peer's Finished
878             // message MUST be treated as an unexpected record type.
879             chc.conContext.consumers.remove(ContentType.CHANGE_CIPHER_SPEC.id);
880
881             // Change client/server application traffic secrets.
882             // Refresh handshake hash
883             chc.handshakeHash.update();
884             SSLKeyDerivation kd = chc.handshakeKeyDerivation;
892             SSLTrafficKeyDerivation kdg =
893                     SSLTrafficKeyDerivation.valueOf(chc.negotiatedProtocol);
909             // derive salt secret
910             try {
911                 SecretKey saltSecret = kd.deriveKey("TlsSaltSecret", null);
912
913                 // derive application secrets
914                 HashAlg hashAlg = chc.negotiatedCipherSuite.hashAlg;
915                 HKDF hkdf = new HKDF(hashAlg.name);
916                 byte[] zeros = new byte[hashAlg.hashLength];
917                 SecretKeySpec sharedSecret =
918                         new SecretKeySpec(zeros, "TlsZeroSecret");
919                 SecretKey masterSecret =
920                     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, parameterised 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.)

```
922                 SSLKeyDerivation secretKD =
923                         new SSLSecretDerivation(chc, masterSecret);
924
925                 // update the handshake traffic read keys.
926                 SecretKey readSecret = secretKD.deriveKey(
927                         "TlsServerAppTrafficSecret", null);
928                 SSLKeyDerivation writeKD =
929                         kdg.createKeyDerivation(chc, readSecret);
930                 SecretKey readKey = writeKD.deriveKey(
931                         "TlsKey", null);
932                 SecretKey readIvSecret = writeKD.deriveKey(
933                         "TlsIv", null);
934                 IvParameterSpec readIv =
935                         new IvParameterSpec(readIvSecret.getEncoded());
936                 SSLReadCipher readCipher =
937                         chc.negotiatedCipherSuite.bulkCipher.createReadCipher(
938                                 Authenticator.valueOf(chc.negotiatedProtocol),
939                                 chc.negotiatedProtocol, readKey, readIv,
940                                 chc.sslContext.getSecureRandom());
941
942                 chc.baseReadSecret = readSecret;
943                 chc.conContext.inputRecord.changeReadCiphers(readCipher);
944
945                 // update the context for the following key derivation
946                 chc.handshakeKeyDerivation = secretKD;
947             } catch (GeneralSecurityException gse) {
950                 return;     // make the compiler happy
951             }
953             //
954             // produce
955             //
956             chc.handshakeProducers.put(SSLHandshake.FINISHED.id,
957                         SSLHandshake.FINISHED);
958             SSLHandshake[] probableHandshakeMessages = new SSLHandshake[] {
959                 // full handshake messages
960                 SSLHandshake.CERTIFICATE,
961                 SSLHandshake.CERTIFICATE_VERIFY,
962                 SSLHandshake.FINISHED
963             };
965             for (SSLHandshake hs : probableHandshakeMessages) {
966                 HandshakeProducer handshakeProducer =
967                         chc.handshakeProducers.remove(hs.id);
968                 if (handshakeProducer != null) {
969                     handshakeProducer.produce(chc, null);
970                 }
971             }
972         }
```

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).

```
651         private byte[] onProduceFinished(ClientHandshakeContext chc,
652                 HandshakeMessage message) throws IOException {
653             // Refresh handshake hash
654             chc.handshakeHash.update();
656             FinishedMessage fm = new FinishedMessage(chc);
662             // Output the handshake message.
663             fm.write(chc.handshakeOutput);
664             chc.handshakeOutput.flush();
672             // Change client/server application traffic secrets.
673             SSLKeyDerivation kd = chc.handshakeKeyDerivation;
681             SSLTrafficKeyDerivation kdg =
682                     SSLTrafficKeyDerivation.valueOf(chc.negotiatedProtocol);
691             try {
692                 // update the application traffic read keys.
693                 SecretKey writeSecret = kd.deriveKey(
694                         "TlsClientAppTrafficSecret", null);
695
696                 SSLKeyDerivation writeKD =
697                         kdg.createKeyDerivation(chc, writeSecret);
698                 SecretKey writeKey = writeKD.deriveKey(
699                         "TlsKey", null);
700                 SecretKey writeIvSecret = writeKD.deriveKey(
701                         "TlsIv", null);
702                 IvParameterSpec writeIv =
703                         new IvParameterSpec(writeIvSecret.getEncoded());
704                 SSLWriteCipher writeCipher =
705                         chc.negotiatedCipherSuite.bulkCipher.createWriteCipher(
706                                 Authenticator.valueOf(chc.negotiatedProtocol),
707                                 chc.negotiatedProtocol, writeKey, writeIv,
708                                 chc.sslContext.getSecureRandom());
709
710                 chc.baseWriteSecret = writeSecret;
711                 chc.conContext.outputRecord.changeWriteCiphers(
712                         writeCipher, false);
714             } catch (GeneralSecurityException gse) {
717                 return null;     // make the compiler happy
718             }
734             // The handshake message has been delivered.
735             return null;
736         }
```

Listing 41: Class Finished.T13FinishedProducer defines method onProduceFinished parameterised 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).

```
974         private void onConsumeFinished(ServerHandshakeContext shc,
975                 ByteBuffer message) throws IOException {
976             FinishedMessage fm = new FinishedMessage(shc, message);
991             //
992             // update
993             //
994             // Change client/server application traffic secrets.
995             SSLKeyDerivation kd = shc.handshakeKeyDerivation;
1003            SSLTrafficKeyDerivation kdg =
1004                    SSLTrafficKeyDerivation.valueOf(shc.negotiatedProtocol);
1020            try {
1021                // update the application traffic read keys.
1022                SecretKey readSecret = kd.deriveKey(
1023                        "TlsClientAppTrafficSecret", null);
1024
1025                SSLKeyDerivation readKD =
1026                        kdg.createKeyDerivation(shc, readSecret);
1027                SecretKey readKey = readKD.deriveKey(
1028                        "TlsKey", null);
1029                SecretKey readIvSecret = readKD.deriveKey(
1030                        "TlsIv", null);
1031                IvParameterSpec readIv =
1032                        new IvParameterSpec(readIvSecret.getEncoded());
1033                SSLReadCipher readCipher =
1034                        shc.negotiatedCipherSuite.bulkCipher.createReadCipher(
1035                                Authenticator.valueOf(shc.negotiatedProtocol),
1036                                shc.negotiatedProtocol, readKey, readIv,
1037                                shc.sslContext.getSecureRandom());
1038
1039                shc.baseReadSecret = readSecret;
1040                shc.conContext.inputRecord.changeReadCiphers(readCipher);
1049            } catch (GeneralSecurityException gse) {
1052                return;     // make the compiler happy
1053            }
1075        }
```

Listing 42: Class Finished.T13FinishedConsumer defines method onConsumeFinished parameterised 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.)

encrypted using the client's first identified pre-shared key. (Extension `early_data` is not associated with data, encrypted application data is sent separately.)

To consume early data, a server must select the client's first pre-shared key identifier and an offered cipher suite associated with that identifier. The server must check the identifier is associated with the server-selected protocol version and (if extension `application_layer_protocol_negotiation` is present) application protocol. (These checks are a superset of those for PSK-based key exchange without early data.) Additionally, for resumption PSKs, the server must check that the PSK is not beyond its lifetime. If checks succeed (and the server is willing to consume early data), then the server will include a corresponding `early_data` extension in their `EncryptedExtensions` message. (When consuming that extension, the client must check that the server selected the client's first pre-shared key identifier, aborting with an illegal_parameter alert, if the check fails.) Otherwise, no such extension will be sent and no early data will be consumed (extension `early_data` is ignored), and the server will proceed in one of the following two ways (which must also be followed by servers not supporting early data): Respond with a `ServerHello` message, excluding extension `early_data`, or respond with a `HelloRetryRequest` message, forcing the client to send a second `ClientHello` message without extension `early_data`. In both cases, the server must skip past early data. For the former, given that all messages will be encrypted, the server must decrypt messages with the handshake traffic key, discard messages when decryption fails, and treat the first successfully decrypted message as the client's next handshake message, thereafter proceeding as if no early data were sent. For the latter, the second `ClientHello` message will be unencrypted and the server can discard all encrypted messages (identified by record type application_data (0x23), rather than type handshake (0x22), as introduced in Section 3), before proceeding as if no early data were sent when the second `ClientHello` message is identified. (When the pre-shared key is associated with a maximum amount of early data, the server should abort with an unexpected_message alert if the maximum is exceeded when skipping past early data.)

> Early data is not supported by JDK 11 (`https://openjdk.java.net/jeps/332`), nor subsequent versions: When extension `early_data` is included in message `ClientHello`, that extension will be processed (Line 1119, Listing 4) and runtime exception UnsupportedOperationException will be thrown (omitted from Listing 46).

Data associated with pre-shared keys

An external PSK (established independently of TLS) must minimally be associated with a hash function and an identity. (The hash function may default to SHA-256, if no function is explicitly associated.) Such a PSK grants freedom over AEAD algorithms, whilst fixing the hash function. By comparison, a resumption PSK (established by `NewSessionTicket` messages) is associated with values negotiated in the connection that provisioned the PSK, which fixes a cipher suite, hence, an AEAD algorithm and a hash function.[14] (It follows that a connection established using a resumption PSK will inherit security from the connection in which the resumption PSK was established.) Resumption PSKs are compatible with early data by default (assuming suitable provisioning with extension `application_layer_protocol_negotiation`, if relevant), whereas (minimally associated) external PSKs are not. They must be associated with a cipher suite (rather than just a hash function), a protocol version, and (if relevant) an application protocol, for compatibility with early data.

2.6.1 `EndOfEarlyData`

A client can transmit early data until they receive a server's `Finished` message. After which, the client transmits an `EndOfEarlyData` message (encrypted using a key derived from secret

[14] Although resumption PSKs are associated with cipher suites, they need not be used with defined AEAD algorithms, except for compatibility with early data.

client_early_traffic_secret), if the server's EncryptedExtensions message included extension early_data. Otherwise, early data has not and will not be consumed by the server, and no EndOfEarlyData message is sent. The EndOfEarlyData message indicates that all early data has been transmitted and subsequent handshake messages will be encrypted with the client's handshake-traffic key. (Servers must not send EndOfEarlyData messages and clients receiving such messages must abort with an unexpected_message alert.)

2.7 Further features

2.7.1 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, currently limited to extension early_data, which indicates that early data is permitted and defines a maximum amount of early data.

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 (including the negotiated hash function). That data may be stored by client and used in extension pre_shared_key of subsequent ClientHello messages. Data must not be used longer than seven days or beyond its lifetime (specified by ticket_lifetime), whichever is shorter, and endpoints may retire data early.

The sole extension currently defined for NewSessionTicket is "early_data", indicating that the ticket may be used to send 0-RTT data (Section 4.2.10). It contains the following value:

max_early_data_size: The maximum amount of 0-RTT data that the client is allowed to send when using this ticket, in bytes. Only Application Data payload (i.e., plaintext but not padding or the inner content type byte) is counted. A server receiving more than max_early_data_size bytes of 0-RTT data SHOULD terminate the connection with an "unexpected_message" alert. Note that servers that reject early data due to lack of cryptographic material will be unable to differentiate padding from content, so clients SHOULD NOT depend on being able to send large quantities of padding in early data records.

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 externally, 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 externally. 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,

$$\mathsf{HMAC}(\texttt{binder\_key}, \mathsf{Transcript\text{-}Hash}(\mathsf{Truncate}(CH)))$$

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

$$\mathsf{HMAC}(\texttt{binder\_key}, \mathsf{Transcript\text{-}Hash}(CH, HRR, \mathsf{Truncate}(CH')))$$

for transcripts that include an initial `ClientHello` message CH, followed by a `HelloRetryRequest` message HRR and a subsequent `ClientHello` message CH', where function 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.

2.7.2 `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.

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 following field (namely, `fragment`), which must not exceed 2^{14} bytes.

`fragment`: A handshake message 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 following field (namely, `encrypted_record`), which must not exceed $2^{14} + 256$ bytes.

`encrypted_record`: 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` appended with type `TLSPlaintext.type` and field `zeros`, which contains an arbitrary-length run of zero-valued bytes and is 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

$$\text{AEAD-Decrypt}(\textit{write\_key}, \textit{nonce}, \textit{additional\_data}, \texttt{TLSCiphertext.encrypted\_record}),$$

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 produced by class SSLSocketOutputRecord (Listing 43) and parent OutputRecord (Listing 44), using enum SSLCipher (Listing 45) to protect outgoing records. (Alternatively, outgoing records are constructed by class SSLEngineOutputRecord, which shares the same parent.) Incoming records are consumed by class SSLSocketInputRecord (or SSLEngineInputRecord) and parent InputRecord, which uses enum SSLCipher for record protection.

4 Java Secure Socket Extension (JSSE)

Java programmers need not concern themselves with the intricacies of TLS: They can use the Java Secure Socket Extension (JSSE), which provides an abstract, high-level API to establish a TLS channel. Doing otherwise is outright dangerous! TLS 1.3 was developed over four years by a team of almost one hundred security experts from more than forty institutions, including tech behemoths Amazon, Apple, Google, IBM, and Microsoft. Their work involved iterating over the subtle details to ensure that security objectives were achieved. JSSE provides OpenJDK's implementation of TLS; using it simplifies development and reduces risk. We present toy applications that demonstrate the use of JSSE (Section 4.1) and then delve into JSSE itself (Section 4.2).

4.1 Examples for code monkeys: Toy client and server

JSSE trivialises the development of toy applications. For instance, the following code snippet establishes a TLS socket:[15]

```
//Open TCP connection to <<host>> : <<port>>, using default SSL socket factory
final String host = "example.com";
final int port = 443;
SSLSocket socket = (SSLSocket) SSLSocketFactory.getDefault().createSocket(host, port);

//TLS handshake
socket.startHandshake();
```

The established TLS socket protects communication, for example, the following HTTP request and response is protected:

[15] Prepending the snippet with System.setProperty("javax.net.debug", "ssl,handshake,verbose") prints additional information, which can be useful.

```
38 final class SSLSocketOutputRecord extends OutputRecord implements SSLRecord {
39     private OutputStream deliverStream = null;
94     synchronized void encodeHandshake(byte[] source,
95             int offset, int length) throws IOException {
147         byte handshakeType = source[0];
148         if (handshakeHash.isHashable(handshakeType)) {
149             handshakeHash.deliver(source, offset, length);
150         }
151
152         int fragLimit = getFragLimit();
153         int position = headerSize + writeCipher.getExplicitNonceSize();
154         if (count == 0) {
155             count = position;
156         }
157
158         if ((count - position) < (fragLimit - length)) {
159             write(source, offset, length);
160             return;
161         }
162
163         for (int limit = (offset + length); offset < limit;) {
165             int remains = (limit - offset) + (count - position);
166             int fragLen = Math.min(fragLimit, remains);
168             // use the buf of ByteArrayOutputStream
169             write(source, offset, fragLen);
170             if (remains < fragLimit) {
171                 return;
172             }
181             // Encrypt the fragment and wrap up a record.
182             encrypt(writeCipher, ContentType.HANDSHAKE.id, headerSize);
184             // deliver this message
185             deliverStream.write(buf, 0, count);     // may throw IOException
186             deliverStream.flush();                  // may throw IOException
193             // reset the offset
194             offset += fragLen;
196             // reset the internal buffer
197             count = position;
198         }
199     }
```

Listing 43: Class SSLSocketOutputRecord defines method encodeHandshake to fragment outgoing handshake messages and write fragments to (its parent's) 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).

```
42  abstract class OutputRecord
43          extends ByteArrayOutputStream implements Record, Closeable {
389     long encrypt(
390             SSLWriteCipher encCipher, byte contentType, int headerSize) {
392             return t13Encrypt(encCipher, contentType, headerSize);
396     }
398     private static final class T13PaddingHolder {
399         private static final byte[] zeros = new byte[16];
400     }
402     private long t13Encrypt(
403             SSLWriteCipher encCipher, byte contentType, int headerSize) {
404         if (!encCipher.isNullCipher()) {
405             // inner plaintext
406             write(contentType);
407             write(T13PaddingHolder.zeros, 0, T13PaddingHolder.zeros.length);
408         }
409
410         byte[] sequenceNumber = encCipher.authenticator.sequenceNumber();
411         int position = headerSize;
412         int contentLen = count - position;
413
414         // ensure the capacity
415         int requiredPacketSize =
416                 encCipher.calculatePacketSize(contentLen, headerSize);
417         if (requiredPacketSize > buf.length) {
418             byte[] newBuf = new byte[requiredPacketSize];
419             System.arraycopy(buf, 0, newBuf, 0, count);
420             buf = newBuf;
421         }
422
423         // use the right TLSCiphertext.opaque_type and legacy_record_version
424         ProtocolVersion pv = protocolVersion;
425         if (!encCipher.isNullCipher()) {
426             pv = ProtocolVersion.TLS12;
427             contentType = ContentType.APPLICATION_DATA.id;
428         } else {
429             pv = ProtocolVersion.TLS12;
430         }
431
432         ByteBuffer destination = ByteBuffer.wrap(buf, position, contentLen);
433         count = headerSize + encCipher.encrypt(contentType, destination);
434
435         // Fill out the header, write it and the message.
436         int fragLen = count - headerSize;
437
438         buf[0] = contentType;
439         buf[1] = pv.major;
440         buf[2] = pv.minor;
441         buf[3] = (byte)((fragLen >> 8) & 0xFF);
442         buf[4] = (byte)(fragLen & 0xFF);
443
444         return Authenticator.toLong(sequenceNumber);
445     }
```

Listing 44: Class OutputRecord defines method t13Encrypt which appends constant 0x22 (defined by variable ContentType.HANDSHAKE.id) and padding (defined by constant zeros) to buffer buf if outgoing data should be encrypted (Lines 404–408), i.e., when producing record `TLSCiphertext`, as opposed to `TLSPlaintext`; encrypts the data in that buffer (Lines 432–433), using a null cipher (SSLCipher.NullReadCipherGenerator) if data should not be encrypted and a cipher in Galois/-Counter Mode (SSLCipher.T13GcmWriteCipherGenerator) otherwise; 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 SSLSocketOutputRecord, in the context of Listing 43), whereas the latter uses constant 0x23 (Line 427).

```
1945     private static final
1946             class T13GcmWriteCipherGenerator implements WriteCipherGenerator {
1956         private static final class GcmWriteCipher extends SSLWriteCipher {
1957             private final Cipher cipher;
1958             private final int tagSize;
1959             private final Key key;
1960             private final byte[] iv;
1961             private final SecureRandom random;
1962
1963             GcmWriteCipher(Authenticator authenticator,
1964                     ProtocolVersion protocolVersion,
1965                     SSLCipher sslCipher, String algorithm,
1966                     Key key, AlgorithmParameterSpec params,
1967                     SecureRandom random) throws GeneralSecurityException {
1968                 super(authenticator, protocolVersion);
1969                 this.cipher = JsseJce.getCipher(algorithm);
1970                 this.tagSize = sslCipher.tagSize;
1971                 this.key = key;
1972                 this.iv = ((IvParameterSpec)params).getIV();
1973                 this.random = random;
1987             }
1990             public int encrypt(byte contentType,
1991                     ByteBuffer bb) {
1992                 byte[] sn = authenticator.sequenceNumber();
1993                 byte[] nonce = iv.clone();
1994                 int offset = nonce.length - sn.length;
1995                 for (int i = 0; i < sn.length; i++) {
1996                     nonce[offset + i] ^= sn[i];
1997                 }
1998
1999                 // initialize the AEAD cipher for the unique IV
2000                 GCMParameterSpec spec =
2001                         new GCMParameterSpec(tagSize * 8, nonce);
2002                 try {
2003                     cipher.init(Cipher.ENCRYPT_MODE, key, spec, random);
2004                 } catch (InvalidKeyException |
2005                         InvalidAlgorithmParameterException ikae) {
2006                     // unlikely to happen
2009                 }
2011                 // Update the additional authentication data, using the
2012                 // implicit sequence number of the authenticator.
2013                 int outputSize = cipher.getOutputSize(bb.remaining());
2014                 byte[] aad = authenticator.acquireAuthenticationBytes(
2015                                         contentType, outputSize, sn);
2016                 cipher.updateAAD(aad);
2017
2018                 int len, pos = bb.position();
2035                 try {
2036                     len = cipher.doFinal(dup, bb);
2037                 } catch (IllegalBlockSizeException |
2038                         BadPaddingException | ShortBufferException ibse) {
2039                     // unlikely to happen
2043                 }
2054                 return len;
2055             }
2082         }
2083     }
```

Listing 45: Class SSLCipher.T13GcmWriteCipherGenerator defines method encrypt which XORs the sequence number and initialisation vector (Lines 1992–1997); initialises a cipher (Line 2003), using algorithm parameters that define the bit length of the authentication tag and the initialisation vector (Lines 2000-2001); and encrypts the input data (Line 2036), appending the authentication tag (Lines 2013–2016), which increments the sequence number as a side effect. (Method Authenticator.TLS13Authenticator.acquireAuthenticationBytes increments the sequence number using method Authenticator.increaseSequenceNumber.)

```
//Request index page from <<host>>
OutputStreamWriter out = new OutputStreamWriter(socket.getOutputStream());
out.write("GET / HTTP/1.1\r\n");
out.write("Host: " + host + "\r\n");  //mandatory for HTTP/1.1 requests
out.write("\r\n");  //header concludes with blank line
out.flush();

//Print response
InputStreamReader in = new InputStreamReader(socket.getInputStream());
int c;
while ((c = in.read()) != -1)
  System.out.print((char)c);

out.close();
in.close();
socket.close();
```

JSSE uses a "provider"-based architecture, whereby services (e.g., SSLSocket and SSLSocketFactory) and implementations (e.g., SSLSocketImpl and SSLSocketFactoryImpl) are defined independently, and are (typically) instantiated by factory methods (e.g., SSLSocketFactory.getDefault()). Hence, programmers need not concern themselves with the inner-workings of implementations, such as those provided by *SunJSSE* (we will nonetheless take a brief look in Section 4.2). Let us now consider a toy server application, to compliment our (above) toy client.

Our client uses the default client-side context, whereas our server cannot, because server-side authentication is mandatory and a certification must be initialised, hence, we start by initialising a suitable context:

```
//Init. default SSL context parametrised with key store args[0], using pwd args[1]
SSLContext sslContext = SSLContext.getInstance("TLSv1.3");

KeyManagerFactory kmf = KeyManagerFactory.getInstance(
  KeyManagerFactory.getDefaultAlgorithm());
kmf.init(KeyStore.getInstance(new File(args[0]), args[1].toCharArray()),
  args[1].toCharArray());

sslContext.init(kmf.getKeyManagers(), null, null);
```

That context can be used to establish a TLS socket:

```
//Init. incoming TCP connection on <<host>> : <<port>>, using default SSL socket factory
final InetAddress host = InetAddress.getLocalHost();
final int port = 8443;
final int backlog = 50;
SSLServerSocket socket = (SSLServerSocket) sslContext.getServerSocketFactory()
  .createServerSocket(port, backlog, host);

//Wait for a connection
Socket session = socket.accept();
```

Communication over the TLS socket is protected, for example, any incoming character is protected as is any subsequent response:

```
//Read an incoming character
InputStreamReader in = new InputStreamReader(session.getInputStream());
if (in.read() == -1)
  System.exit(1);

//Response (regardless of input)
OutputStreamWriter out = new OutputStreamWriter(session.getOutputStream());
out.write("HELLO WORLD\r\n");
out.flush();
```

Our client and server can communicate by assigning InetAddress.getLocalHost() to variable host and 8443 to variable port,[16] rather than example.com and 443, respectively. The key store necessary for this example can be constructed using keytool as follows:

[16]Unix systems protect ports under 1024, hence, we use port 8443, rather than port 443.

```
bas $ keytool -genkey -keyalg RSA -keystore store
Enter keystore password:
Re-enter new password:
What is your first and last name?
  [Unknown]:  127.0.1.1
What is the name of your organizational unit?
  [Unknown]:
What is the name of your organization?
  [Unknown]:
What is the name of your City or Locality?
  [Unknown]:
What is the name of your State or Province?
  [Unknown]:
What is the two-letter country code for this unit?
  [Unknown]:
Is CN=127.0.1.1, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown correct?
  [no]:  yes

bas $
```

where InetAddress.getLocalHost() is 127.0.1.1 and filename store can be replaced with alternatives. Since the above key store is self-signed, it must be added to the Java virtual machine's trust store, which can be achieved by prepending client code with the following: System.setProperty("javax.net.ssl.trustStore", "store"); System.setProperty("javax.net.ssl.trustStorePassword", <<pwd>>).

4.2 SunJSSE provider for architects, researchers, and the curious

Our toy client uses statement SSLSocketFactory.getDefault().createSocket(host, port) to instantiate an instance of SSLSocketFactoryImpl parameterised with an initial context, and uses that context along with variables host and port to instantiate and return an instance of SSLSocketImpl. Method SSLSocketImpl.startHandshake() proceeds as follows:

```
377   public void startHandshake() throws IOException {
395               conContext.kickstart();
396
397               // All initial handshaking goes through this operation until we
398               // have a valid SSL connection.
401               if (!conContext.isNegotiated) {
402                   readHandshakeRecord();
403               }
411   }
```

Line 395 indirectly calls method ClientHello.kickstartProducer.produce (§2.1) and processes responses (Line 401–403) as follows:

```
1060  private int readHandshakeRecord() throws IOException {
1061      while (!conContext.isInboundClosed()) {
1063              Plaintext plainText = decode(null);
1064              if ((plainText.contentType == ContentType.HANDSHAKE.id) &&
1065                      conContext.isNegotiated) {
1066                  return 0;
1067              }
1077      }
1079      return -1;
1080  }
1148  private Plaintext decode(ByteBuffer destination) throws IOException {
1149      Plaintext plainText;
1152              plainText = SSLTransport.decode(conContext,
```

```
1153                        null, 0, 0, null, 0, 0);
1170        return plainText;
1171    }
```

Hence, until a connection is negotiated (Lines 1064–1067), responses are processed by method SSLTransport.decode (Line 1152–1153) as follows:

```
101   static Plaintext decode(TransportContext context,
102       ByteBuffer[] srcs, int srcsOffset, int srcsLength,
103       ByteBuffer[] dsts, int dstsOffset, int dstsLength) throws IOException {
104
105       Plaintext[] plaintexts = null;
107           plaintexts =
108                   context.inputRecord.decode(srcs, srcsOffset, srcsLength);
145       Plaintext finalPlaintext = Plaintext.PLAINTEXT_NULL;
146       for (Plaintext plainText : plaintexts) {
147           // plainText should never be null for TLS protocols
148           if (plainText == Plaintext.PLAINTEXT_NULL) {
149               // Only happens for DTLS protocols.
162           } else if (plainText != null &&
163                   plainText.contentType != ContentType.APPLICATION_DATA.id) {
164               context.dispatch(plainText);
165           }
200           finalPlaintext = plainText;
201       }
202
203       return finalPlaintext;
204   }
```

Line 108 parses a handshake (or an alert) record header and calls method SSLSocketInputRecord .decodeInputRecord, providing the header as input. That method parses and decodes the complete record, which is then processed by method TransportContext.dispatch (Line 164). Method SSLSocketInputRecord.decodeInputRecord proceeds as follows:

```
204   private Plaintext[] decodeInputRecord(
205           byte[] header) throws IOException, BadPaddingException {
206       byte contentType = header[0];                    // pos: 0
207       byte majorVersion = header[1];                   // pos: 1
208       byte minorVersion = header[2];                   // pos: 2
209       int contentLen = ((header[3] & 0xFF) << 8) +
210                        (header[4] & 0xFF);             // pos: 3, 4
230       // Read a complete record.
232       ByteBuffer destination = ByteBuffer.allocate(headerSize + contentLen);
233       int dstPos = destination.position();
234       destination.put(temporary, 0, headerSize);
235       while (contentLen > 0) {
236           int howmuch = Math.min(temporary.length, contentLen);
237           int really = read(is, temporary, 0, howmuch);
238           if (really < 0) {
239               throw new EOFException("SSL peer shut down incorrectly");
240           }
241
242           destination.put(temporary, 0, howmuch);
243           contentLen -= howmuch;
244       }
245       destination.flip();
246       destination.position(dstPos + headerSize);
257       // Decrypt the fragment
259       ByteBuffer fragment;
261           Plaintext plaintext =
262                   readCipher.decrypt(contentType, destination, null);
263           fragment = plaintext.fragment;
264           contentType = plaintext.contentType;
280       // parse handshake messages
282       if (contentType == ContentType.HANDSHAKE.id) {
```

```
283             ByteBuffer handshakeFrag = fragment;
284             if ((handshakeBuffer != null) &&
285                     (handshakeBuffer.remaining() != 0)) {
286                 ByteBuffer bb = ByteBuffer.wrap(new byte[
287                         handshakeBuffer.remaining() + fragment.remaining()]);
288                 bb.put(handshakeBuffer);
289                 bb.put(fragment);
290                 handshakeFrag = bb.rewind();
291                 handshakeBuffer = null;
292             }
293 
294             ArrayList<Plaintext> plaintexts = new ArrayList<>(5);
295             while (handshakeFrag.hasRemaining()) {
296                 int remaining = handshakeFrag.remaining();
297                 if (remaining < handshakeHeaderSize) {
298                     handshakeBuffer = ByteBuffer.wrap(new byte[remaining]);
299                     handshakeBuffer.put(handshakeFrag);
300                     handshakeBuffer.rewind();
301                     break;
302                 }
303 
304                 handshakeFrag.mark();
305                 // skip the first byte: handshake type
306                 byte handshakeType = handshakeFrag.get();
307                 int handshakeBodyLen = Record.getInt24(handshakeFrag);
308                 handshakeFrag.reset();
309                 int handshakeMessageLen =
310                         handshakeHeaderSize + handshakeBodyLen;
311                 if (remaining < handshakeMessageLen) {
312                     handshakeBuffer = ByteBuffer.wrap(new byte[remaining]);
313                     handshakeBuffer.put(handshakeFrag);
314                     handshakeBuffer.rewind();
315                     break;
316                 } if (remaining == handshakeMessageLen) {
317                     if (handshakeHash.isHashable(handshakeType)) {
318                         handshakeHash.receive(handshakeFrag);
319                     }
320 
321                     plaintexts.add(
322                         new Plaintext(contentType,
323                             majorVersion, minorVersion, -1, -1L, handshakeFrag)
324                     );
325                     break;
326                 } else {
327                     int fragPos = handshakeFrag.position();
328                     int fragLim = handshakeFrag.limit();
329                     int nextPos = fragPos + handshakeMessageLen;
330                     handshakeFrag.limit(nextPos);
331 
332                     if (handshakeHash.isHashable(handshakeType)) {
333                         handshakeHash.receive(handshakeFrag);
334                     }
335 
336                     plaintexts.add(
337                         new Plaintext(contentType, majorVersion, minorVersion,
338                             -1, -1L, handshakeFrag.slice())
339                     );
340 
341                     handshakeFrag.position(nextPos);
342                     handshakeFrag.limit(fragLim);
343                 }
344             }
346             return plaintexts.toArray(new Plaintext[0]);
347         }
348 
349         return new Plaintext[] {
350                 new Plaintext(contentType,
```

```
351                     majorVersion, minorVersion, -1, -1L, fragment)
352             };
353     }
```

Finally, method TransportContext.dispatch proceeds as follows:

```
143     // Dispatch plaintext to a specific consumer.
144     void dispatch(Plaintext plaintext) throws IOException {
149         ContentType ct = ContentType.valueOf(plaintext.contentType);
156         switch (ct) {
157             case HANDSHAKE:
158                 byte type = HandshakeContext.getHandshakeType(this,
159                         plaintext);
160                 if (handshakeContext == null) {
161                     if (type == SSLHandshake.KEY_UPDATE.id ||
162                             type == SSLHandshake.NEW_SESSION_TICKET.id) {
163                         if (isNegotiated &&
164                                 protocolVersion.useTLS13PlusSpec()) {
165                             handshakeContext = new PostHandshakeContext(this);
166                         } else {
167                             fatal(Alert.UNEXPECTED_MESSAGE,
168                                     "Unexpected post-handshake message: " +
169                                     SSLHandshake.nameOf(type));
170                         }
171                     } else {
172                         handshakeContext = sslConfig.isClientMode ?
173                                 new ClientHandshakeContext(sslContext, this) :
174                                 new ServerHandshakeContext(sslContext, this);
175                         outputRecord.initHandshaker();
176                     }
177                 }
178                 handshakeContext.dispatch(type, plaintext);
179                 break;
191         }
192     }
```

Line 178 indirectly calls HandshakeContext.dispatch on variables handshakeType and plaintext. fragment, which calls the relevant consumer and updates the handshake hash.

A Extensions

Extensions listed by an endpoint are generally followed by a corresponding extension from their peer. Corresponding extensions must not be sent without solicitation, and endpoints must abort with an unsupported_extension alert upon receipt of such unsolicited extensions. For instance, a `ClientHello` message listing extension `supported_groups` is followed by a `ServerHello` message listing the same extension, whereas a `ServerHello` message must not list that extension in response to a `ClientHello` message that does not and a client should abort in such cases.

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. Support for the following extensions is mandatory (unless an implementation explicitly opts out): `cookie`, `key_share`, `server_name`, `signature_algorithms`, `signature_algorithms_cert`, `supported_groups`, and `supported_versions`. A client requesting a non-mandatory extension may abort if the extension is not supported by the server. A server may require `ClientHello` messages to include extension `server_name` and should abort with an missing_extension alert if the extension is missing.

Extension	RFC	Handshake message
application_layer_protocol_negotiation	7301	CH, EE
certificate_authorities	8446	CH, CR
client_certificate_type	7250	CH, EE
cookie	8446	CH, HRR
early_data	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

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`).

When designing new extensions, the following considerations should be taken into account: First, a server that does not support a client-requested extension should indicate that the extension is unsupported by inclusion of a suitable extension in their response, rather than aborting. By comparison, a server should abort when a client-supplied extension is erroneous. Secondly, prior to authentication, active attackers can remove and inject messages, hence, they can modify handshake messages. Since an HMAC is computed over the entire handshake, such modifications can typically be detected and endpoints can abort. However, to quote RFC 8446, "extreme care is needed when the extension changes the meaning of messages sent in the handshake phase." Thus, extensions should be designed to prevent an active adversary from unduly influencing parameter negotiation, i.e., endpoints should negotiate their preferred parameters, even in the presence of an adversary. In addition, any interactions with early data must be defined.

Extensions are enumerated and instantiated by enum SSLExtension (Listing 46), and class SSLExtensions (Listings 47–48) 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

```
38  enum SSLExtension implements SSLStringizer {
489     private SSLExtension(int id, String name, SSLHandshake handshakeType,
490             ProtocolVersion[] supportedProtocols,
491             HandshakeProducer producer,
492             ExtensionConsumer onLoadConsumer, HandshakeAbsence onLoadAbsence,
493             HandshakeConsumer onTradeConsumer, HandshakeAbsence onTradeAbsence,
494             SSLStringizer stringize) {
495         this.id = id;
496         this.handshakeType = handshakeType;
497         this.name = name;
498         this.supportedProtocols = supportedProtocols;
499         this.networkProducer = producer;
500         this.onLoadConsumer = onLoadConsumer;
501         this.onLoadAbsence = onLoadAbsence;
502         this.onTradeConsumer = onTradeConsumer;
503         this.onTradeAbsence = onTradeAbsence;
504         this.stringizer = stringize;
505     }
529     public byte[] produce(ConnectionContext context,
530             HandshakeMessage message) throws IOException {
532             return networkProducer.produce(context, message);
537     }
539     public void consumeOnLoad(ConnectionContext context,
540             HandshakeMessage message, ByteBuffer buffer) throws IOException {
542             onLoadConsumer.consume(context, message, buffer);
547     }
549     public void consumeOnTrade(ConnectionContext context,
550             HandshakeMessage message) throws IOException {
552             onTradeConsumer.consume(context, message);
557     }
685 }
```

Listing 46: 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.

```
39  final class SSLExtensions {
40      private final HandshakeMessage handshakeMessage;
41      private Map<SSLExtension, byte[]> extMap = new LinkedHashMap<>();
47
48      SSLExtensions(HandshakeMessage handshakeMessage) {
49          this.handshakeMessage = handshakeMessage;
51      }
53      SSLExtensions(HandshakeMessage hm,
54              ByteBuffer m, SSLExtension[] extensions) throws IOException {
55          this.handshakeMessage = hm;
56
57          int len = Record.getInt16(m);
58          encodedLength = len + 2;          // 2: the length of the extensions.
59          while (len > 0) {
60              int extId = Record.getInt16(m);
61              int extLen = Record.getInt16(m);
77              boolean isSupported = false;
78              for (SSLExtension extension : extensions) {
79                  if ((extension.id != extId) ||
80                          (extension.onLoadConsumer == null)) {
81                      continue;
82                  }
91                  byte[] extData = new byte[extLen];
92                  m.get(extData);
93                  extMap.put(extension, extData);
98                  isSupported = true;
99                  break;
100             }
102             if (!isSupported) {
103                 if (logMap != null) {
104                     // cache the extension for debug logging
114                 } else {
115                     // ignore the extension
116                     int pos = m.position() + extLen;
117                     m.position(pos);
118                 }
119             }
121             len -= extLen + 4;
122         }
123     }
207     void produce(HandshakeContext context,
208             SSLExtension[] extensions) throws IOException {
209         for (SSLExtension extension : extensions) {
228             byte[] encoded = extension.produce(context, handshakeMessage);
229             if (encoded != null) {
230                 extMap.put(extension, encoded);
233             } else if (SSLLogger.isOn && SSLLogger.isOn("ssl,handshake")) {
234                 // The extension is not available in the context.
238             }
239         }
240     }
```

Listing 47: 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).

```
132     void consumeOnLoad(HandshakeContext context,
133             SSLExtension[] extensions) throws IOException {
134         for (SSLExtension extension : extensions) {
163             ByteBuffer m = ByteBuffer.wrap(extMap.get(extension));
164             extension.consumeOnLoad(context, handshakeMessage, m);
169         }
170     }
175     void consumeOnTrade(HandshakeContext context,
176             SSLExtension[] extensions) throws IOException {
177         for (SSLExtension extension : extensions) {
197             extension.consumeOnTrade(context, handshakeMessage);
201         }
202     }
293     void send(HandshakeOutStream hos) throws IOException {
294         int extsLen = length();
295         if (extsLen == 0) {
296             return;
297         }
298         hos.putInt16(extsLen - 2);
299         // extensions must be sent in the order they appear in the enum
300         for (SSLExtension ext : SSLExtension.values()) {
301             byte[] extData = extMap.get(ext);
302             if (extData != null) {
303                 hos.putInt16(ext.id);
304                 hos.putBytes16(extData);
305             }
306         }
307     }
362 }
```

Listing 48: Class SSLExtensions (continued from Listing 47) defines method consumeOnLoad to consume received extensions (Lines 132–170), using method SSLExtension.consumeOnLoad (Listing 46); consumeOnTrade to update the active context to include extensions (Lines 175–202), using method SSLExtension.consumeOnTrade (Listing 46); and method send to write extensions and associated data to an output stream (Lines 293–307).

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`

A server may request client authentication by sending a `CertificateRequest` 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.)

A client may decline to authenticate by responding with a `Certificate` message that does not contain a certificate, followed by a `Finished` message. (The server may continue without client authentication or abort with a certificate_required alert.) Alternatively, a client may authenticate by responding with `Certificate` and `CertificateVerify` messages (such that `CertificateRequest.certificate_request_context` = `Certificate.certificate_request_context`), followed by a `Finished` message. The `CertificateVerify` message includes a signature over string "TLS 1.3, client CertificateVerify", rather than "TLS 1.3, server CertificateVerify", to distinguish client- and server-generated `CertificateVerify` messages, and to help defend against potential cross-protocol attacks. 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 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.

For (EC)DHE-only key exchange, client authentication is possible during a handshake: a server includes a `CertificateRequest` message immediately after their `EncryptedExtensions` message (and before `Certificate`, `CertificateVerify`, and `Finished` messages), and a client responds with `Certificate`, (optionally) `CertificateVerify`, and `Finished` messages. For PSK-based key exchange, a server must 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 responds with `Certificate`, (optionally) `CertificateVerify`, and `Finished` messages, computing the HMAC with

$$\texttt{finished\_key} = \mathsf{HKDF\text{-}Expand\text{-}Label}(\texttt{client\_application\_traffic\_secret\_N}, \text{“}finished\text{”}, \text{“”}, \mathsf{Hash.length})$$

(Post-handshake authentication is only concerned with updating the client's application-traffic key, for the purposes of blinding the client's identity to that key. Hence, secret `finished_key` is not concerned with traffic secret `server_application_traffic_secret_N`. Beyond traffic keys, a key established by a `NewSessionTicket` message, sent after post-handshake authentication, will also be bound to the client's identity.) A client receiving an unsolicited post-handshake authentication

request (i.e., message `ClientHello` did not include extension `post_handshake_auth`) must abort with an unexpected_message alert.