Report on TLS 1.3

Aarav Varshney[1]

[1]**Ashoka University**, `aarav.varshney@alumni.ashoka.edu.in`

Abstract

The Internet Engineering Task Force (IETF) formed the Transport Layer Security (TLS) working group to develop a standardized SSL-like protocol. In 1999, the group released the specification for the TLS 1.0 protocol [1], which is a minor variation of SSL 3.0 and is sometimes referred to as SSL version 3.1. In 2006 and 2008, minor updates were made to TLS 1.0, leading to the development of TLS version 1.2. However, due to numerous security vulnerabilities, TLS 1.2 was overhauled in 2017, resulting in the much stronger TLS version 1.3 [2]. TLS has become widely used in software systems worldwide. This report will mostly focus on TLS 1.3.

1. Introduction

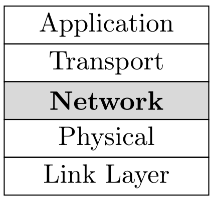


Figure 1. The TCP/IP Reference Model [3]

When two computers want to send data to each other, they may use the **Internet Protocol** (IP) [4] directly. IP fragments data into blocks of data called *packets* (also called datagrams) and transmits them from the source to the destination. The sources and destinations are computers identified by fixed length addresses. However, IP is a low-level protocol that does not interact directly with application data and may cause unreliable data transmission.

In the TCP/IP Reference Model [3], the IP protocol works in the network layer, which is two layers below the application layer (see Figure 1). This layer provides an unreliable, connectionless delivery system (there is no direct connection between two hosts) which does not provide any functionality for error recovery for datagrams that are either duplicated, lost or arrive at the remote host in another order than they were sent. The transport layer, which is the layer above the network layer, is responsible for fixing the unreliable data transmission with transport protocols like **Transmission Control Protocol (TCP)** [1]. TCP is the most commonly used transport protocol on top of IP and includes strategies for packet ordering, retransmission, and maintaining data integrity.

TCP is a connection-oriented protocol, which means that before any data can be sent, a connection must be established between the two hosts. This connection is established by a three-way handshake between the two hosts. The first computer sends a packet with the SYN (a field in the

TCP header) bit set to 1. The second computer responds with a packet with the SYN and ACK (another field in the TCP header) bits set to 1. The first computer then sends a packet with the ACK bit set to 1. Once the connection is established, the two hosts can start sending data to each other.

Now that we have established a connection between two hosts and have the ability to reliably send data across, we can further improve the transmission by using **Transport Layer Security (TLS)** [2] (original [5]). TLS protocol allows two hosts to communicate in a way that is designed to prevent eavesdropping, tampering, or message forgery. While TCP provides reliable data transmission, the data in the packets is unencrypted and can be read by anyone who has access to the network. This is acceptable when the data is not sensitive, but when the data is sensitive, TLS is used to *encrypt* the data. TLS also provides *authentication* of the remote host, which means that the remote host can be trusted to be the host it claims to be.

TLS is primarily used in a client/server setting where the client initiates the connection and the server responds. Setting up a TLS connection requires a handshake between the client and the server. At the end of the handshake, the client and the server have agreed upon a shared secret key which is used to encrypt the data, the encryption schemes and their parameters, and the client is also assured of the server's identity (not without issues [6, 7]).

In this report, we discuss TLS 1.3 [2], which is the latest version of TLS. Section 2 covers the necessary background for understanding TLS protocols; section 3 describes the *handshake protocol* in TLS 1.3. Finally, section 4 discusses the *record protocol* that uses the parameters established by the handshake protocol to protect traffic between the communicating hosts.

2. Background

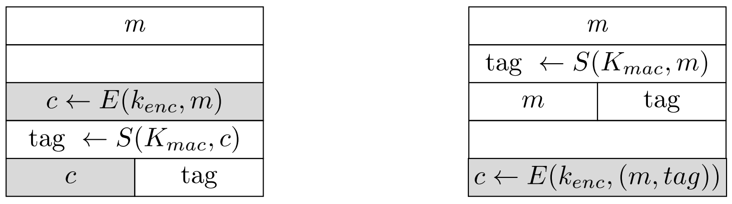
2.1. Authenticated Encryption (AE)

Cryptographic applications commonly require both confidentiality and message authentication. Confidentiality ensures that data is available only to those authorized to obtain it; usually it is realized through encryption. Message authentication is the service that ensures that data has not been altered or forged by unauthorized entities; it can be achieved by using a Message Authentication Code (MAC). This service is also called data integrity. **Authenticated Encryption (AE)** [8] schemes ensure both data secrecy (confidentiality) and data integrity. These schemes can be constructed with either a **generic composition** that combines a CPA-secure cipher with a secure MAC, or to build them directly from a block cipher or a PRF without first constructing either a standalone cipher or MAC [9]. The latter schemes are called **integrated schemes**.

Let (E, D) be a cipher and (S, V) be a MAC. Let k_{enc} be a cipher key and k_{mac} be a MAC key. In a generic composition, these two primitives can be combined using two commonly used options: **Encrypt-then-MAC** and **MAC-then-Encrypt**. In the first option, the plaintext is encrypted using the cipher and the resulting ciphertext is authenticated using the MAC. In the second variant, the plaintext is first authenticated using the MAC and the resulting MAC tag along with the plaintext is encrypted using the cipher (see Figure 2a and Figure 2b). Only the first method is secure for every combination of CPA-secure cipher and secure MAC. The intuition is that the MAC on the ciphertext prevents any tampering with the ciphertext. The second method is known to have attacks in some cases [9].

2.2. Authenticated Encryption with Associated Data (AEAD)

TLS uses a nonce-based **Authenticated Encryption with Associated Data (AEAD)** [10] cipher to encrypt and authenticate packets. AEAD schemes are an extension of AE schemes (discussed above) that adds the ability to check the integrity and authenticity of some Associated Data (AD), also called "additional authenticated data", that is not encrypted. The associated data is required to set context; for example in a networking protocol, authenticated encryption protects the packet body, but the header must be transmitted in the clear so that the network can route the packet to its intended destination [9]. Nonetheless, we want to ensure header integrity so that a malicious adversary can't get away with tampering the header. This header is provided as the


m
$c \leftarrow E(k_{enc}, m)$
tag $\leftarrow S(K_{mac}, c)$
c
tag
m
tag $\leftarrow S(K_{mac}, m)$
m
tag
$c \leftarrow E(k_{enc}, (m, tag))$

(a) Encrypt-then-MAC **(b)** MAC-then-Encrypt

Figure 2. Two different ways to combine a cipher and a MAC.

associated data input to the AEAD encryption scheme. Secondly, the encryption scheme takes an additional input nonce $\mathcal{N}$, which is a random value that is used only once. This nonce ensures that the same plaintext is never encrypted to the same ciphertext. The nonce can be kept with the encrypted message or created just before it's decrypted. The decryption process only needs enough information to reconstruct the nonce. For example, a system could use a nonce consisting of a sequence number in a particular format, in which case it could be inferred from the order of the ciphertexts. If a wrong nonce is used, the decryption process will still detect it as incorrect, so it won't compromise security.

The interface to the AEAD is relatively simple, since it requires only a single key as input and requires only a single identifier to indicate the algorithm in use in a particular case unlike AE where you have to specify the combination of cipher and MAC. The authenticated encryption operation has four inputs, each of which is an octet string:

1. A secret key K, which MUST be generated in a way that is uniformly random or pseudorandom.
2. A nonce $\mathcal{N}$. Each nonce provided to distinct invocations of the Authenticated Encryption operation MUST be distinct, for any particular value of the key, unless each and every nonce is zero- length.
3. A plaintext P, which contains the data to be encrypted and authenticated.
4. The associated data A, which contains the data to be authenticated, but not encrypted.

The output is a ciphertext C, which is at least as long as the plaintext.

2.3. HMAC-based Extract-and-Expand Key Derivation Function (HKDF)

The TLS handshake establishes one or more input secrets which are combined to create the actual working keying material using HKDF which uses HMAC at its core.

Hashed Message Authentication Code (HMAC) [11] is based on a cryptographic hash function $H : \{0,1\}^* \to \{0,1\}^\lambda$ and keyed with some key $K \in \{0,1\}^\lambda$ (larger key material is hashed through H to obtain a λ-bit key). Computing the HMAC value on some message m is then defined as $HMAC(K,m) := H((K \oplus opad) \parallel H((K \oplus ipad) \parallel m))$, where $opad$ and $ipad$ are two λ-bit padding values consisting of repeated bytes 0x5c and 0x36, respectively.

A key derivation function (KDF) is a basic and essential component of cryptographic systems. Its goal is to take some source of initial keying material and derive from it one or more cryptographically strong secret keys [12]. The main difficulty in designing a KDF relates to the form of the initial keying material. When this key material is given as a uniformly random or pseudorandom key K then one can use K to seed a pseudorandom function (PRF) or pseudorandom generator (PRG) to produce additional cryptographic keys. However, when the source keying material is not uniformly random or pseudorandom then the KDF needs to first "extract" from this "imperfect" source a first pseudorandom key from which further keys can be derived using a PRF. Thus, one identifies two logical modules in a KDF: a first module that takes the source keying material and extracts from it a fixed-length pseudorandom key K, and a second module that expands K into several additional pseudorandom cryptographic keys. This two-module paradigm is called the “extract-then-expand” paradigm.

The expansion module is standard in cryptography and can be implemented on the basis of any secure PRF. The extraction functionality, in turn, is well modeled by the notion of randomness extractors [12]. There exists computational extractors, namely randomness extractors where the output is only required to be pseudorandom rather than statistically close to uniform. Computational extractors are well-suited for the cryptographic setting where attackers are computationally bounded and source entropy may only exist in a computational sense. KDFs accept four inputs: a sample from the source of keying material from which the KDF needs to extract cryptographic keys, a parameter defining the number of key bits to be output, an (optional) randomizing salt value as mentioned before, and a fourth “contextual information” field. The latter is an important parameter for the KDF intended to include key-related information that needs to be uniquely and cryptographically bound to the produced key material (e.g., a protocol identifier, identities of principals, timestamps, etc.) [12].

The computation of extract-then-expand KDFs is specified as follows:

$$\begin{aligned} &\mathit{PRK} = \mathit{XTR}\ (\mathit{XTS},\ \mathit{SKM}) \\ &\mathit{KM} = \mathit{PRF}^*(\mathit{PRK},\ \mathit{CTXinfo},\ L) \end{aligned} \tag{1}$$

where XTR is the extractor function, PRF^* is a variable-length output pseudorandom function used in the expansion module, and L is the desired length of the output key material.

HMAC-based key derivation function (HKDF) [12] follows the above mentioned "extract-then-expand" paradigm and uses HMAC as the underlying PRF in both extract and expand modules. HDKF is specified as:

$$\mathrm{HKDF}(\mathit{XTS}, \mathit{SKM}, \mathit{CTXinfo}, L) = K(1) \parallel K(2) \parallel ... \parallel K(t) \tag{2}$$

where the values $K(i)$ are defined as follows:

$$\begin{aligned} &\mathit{PRK} = \mathrm{HMAC}(\mathit{XTS}, \mathit{SKM}) \\ &K(1) = \mathrm{HMAC}(\mathit{PRK}, \mathit{CTXinfo} \parallel 0) \\ &K(i+1) = \mathrm{HMAC}(\mathit{PRK}, K(i) \parallel \mathit{CTXinfo} \parallel i) 1 \leq i \leq t, \end{aligned} \tag{3}$$

where $t = \lceil L/k \rceil$ where k denotes the output (and key) length of the hash function used with HMAC and the value $K(t)$ is truncated to its first $d = L \mod k$ bits; the counter i is of a given fixed size, e.g., a single byte. The source key material is SKM, the extractor salt is XTS (which may be null or constant), the number L of key bits to be produced by KDF, and a “context information” string $CTXinfo$ (which may be null) [12].

2.4. TLS 1.3 Key Derivation

The Key Derivation process in TLS 1.3 makes use of the HKDF-Extract and HKDF-Expand functions as well as the functions defined below [2]:

- **HKDF-Extract**: HKDF-Extract(Salt, IKM) $\rightarrow$ PRK
- **HKDF-Expand**: HKDF-Expand(PRK, Info, L) $\rightarrow$ OKM
- **HKDF-Expand-Label**: HKDF-Expand-Label(Secret, Label, Context, Length) $\rightarrow$ HKDF-Expand(Secret, HkdfLabel, Length)
- **HkdfLabel**: HkdfLabel(Length, Label, Context): (length $=$ Length; label $=$ "tls13 " $+$ Label; context $=$ Context) $\rightarrow$ HkdfLabel
- **Derive-Secret**: Derive-Secret(Secret, Label, Messages) $\rightarrow$ HKDF-Expand-Label(Secret, Label, Transcript-Hash(Messages), Hash.length)

3. TLS Handshake Protocol

For consistency with the notation, we let P play the role of the client and Q play the role of the server. P and Q wish to setup a secure session.

TLS 1.3 supports both one-sided and mutual authentication. In most cases, authentication for the client is optional. In the notation, (E_s, D_s) is a symmetric encryption scheme that provides authenticated encryption, such as AES-128 in GCM mode. Algorithm S refers to a MAC signing algorithm, such as HMAC-SHA256. Algorithms $Sig_P(\cdot)$ and $Sig_Q(\cdot)$ sign the provided data using P's or Q's signing keys. Finally, the hash functions H_1, H_2 are used to derive symmetric keys. They are built from HKDF with hash functions such as SHA-256.

The cipher-suites which determine the symmetric encryption scheme, the hash function in the MAC signing algorithm and HKDF are negotiated during the handshake. The cipher-suites are defined in [2]. They are specified in Table 1. TLS negotiates these algorithms, rather than hard code a specific choice, because some organizations may prefer to use different algorithms based on their policies.

The protocol. In the first step **Client Hello**, P sends a message to Q. The message contains a 32 bytes nonce $\mathcal{N}_c$ generated by a secure random number generator and an "offer". The offer specifies the group, a list of the symmetric cipher options supported by the client, specifically the record protection algorithm (including secret key length) and a hash to be used with HKDF, in descending order of client preference. In fact, the client can provide several groups in his offer using a "supported_groups" extension though constrained to be one of several pre-defined groups, which include both elliptic curves and subgroups of finite fields. A "key_share" extension provides corresponding group elements for each group. Clients which desire the server to authenticate itself via a certificate must send the "signature_algorithms" extension which indicates the signature algorithms the client can accept.

After receiving the first flow, the server Q examines the "offer" sent by the client. It verifies that the group (or groups) preferred by the client coincides with the group (or groups) that the server is able and willing to use. It also selects an encryption scheme (E_s, D_s) and a hash function from the offer that it is willing and able to use, if any. If the server is unable to find a compatible group and encryption/hash schemes, the server may send a special "retry" request to the client.

The server now responds with **Server Hello** containing its own nonce $\mathcal{N}_s$, a "mode" message which indicates the parameter choices (group, encryption/hash scheme) made by the server along with the key share corresponding to the selected group. At this point, the server has enough information to compute the client handshake traffic secret (CHTS) and server handshake traffic secret (SHTS) values, and uses these to derive client and server handshake traffic keys (tk_{chs} and tk_{shs}, respectively)
The server now begins to encrypt all handshake messages under tk_{shs}, and any extensions that

Description	Value
TLS_AES_128_GCM_SHA256	$\{0 \times 13, 0 \times 01\}$
TLS_AES_256_GCM_SHA384	$\{0 \times 13, 0 \times 02\}$
TLS_CHACHA20_POLY1305_SHA256	$\{0 \times 13, 0 \times 03\}$
TLS_AES_128_CCM_SHA256	$\{0 \times 13, 0 \times 04\}$
TLS_AES_128_CCM_8_SHA256	$\{0 \times 13, 0 \times 05\}$

Table 1: Cipher-Suites [2]. Cipher suite names follow the naming convention: CipherSuite TLS_AEAD_HASH = VALUE;

are not required to establish the server handshake traffic key are sent (and encrypted) in the **EncryptedExtensions** (EE) messages [13].

SHTS/CHTS can be derived as follows [2]:

```
early sercret (ES) =  HKDF-Extract(0, 0)
derived early secret (dES) = Derive-Secret(ES, "derived", "")
handshake secret (HS) =  HKDF-Extract(dES, (EC)DHE shared secret)
SHTS = Derive-Secret(HS, "s hs traffic", ClientHello...ServerHello)
CHTS = Derive-Secret(HS, "c hs traffic", ClientHello...ServerHello)
```

The server sends **EncryptedExtensions** message immediately after the ServerHello message for sending extensions that can be protected. This is followed by **Certificate Request** message, which is only sent if the server wishes to authenticate the client. If present, this message specifies the type of certificates the server will accept. Finally, the server sends **Authentication Messages** for authentication, key confirmation (provide explicit proof that an endpoint possesses the private key corresponding to its certificate), and handshake integrity: **Certificate**, **CertificateVerify**, and **Finished**. All handshake messages in this phase are encrypted under tk_{shs} or tk_{chs}. n the full 1-RTT handshake (TODO: Explain 1-RTT and 0-RTT), authentication is based on public key certificates. In pre-shared key handshakes (both PSK and PSK-(EC)DHE), the server and client will authenticate each other by relying on a message authentication code applied to the transcript. (TODO: This part needs clarity)

- ServerCertificate(SV): The server's certificate, and any supporting certificates in the chain.
- ServerCertificateVerify(SCV): A signature over the value Transcript-Hash [1](Handshake Context, Certificate). Here, Handshake Context consists of `ClientHello`, `ServerHello`, `EncryptedExtensions`, and `CertificateRequest` (the conversation so far) using server's signing key.
- ServerFinished(SF): This is the final message in the Authentication Messages. It is essential for providing authentication of the handshake and of the computed keys. The server first derives a server finished key fk_S from SHTS and then computes a MAC tag SF over `Transcript-Hash(Handshake Context,Certificate, CertificateVerify)`. This value is also encrypted under tk_{shs}, sending the output ciphertext to the client. At this point, the server is able to compute the client application traffic secret (CATS), the server application traffic secret (SATS), and the exporter master secret (EMS). It can now compute the client application traffic key tk_{capp} and the server application traffic key tk_{sapp} and it can begin sending encrypted application data to the client.

$$fk_S = \texttt{HKDF-Expand-Label(SHTS, ``finished'', '', Hash.length)} \tag{4}$$

[1]Transcript-Hash is computed by hashing the concatenation of each included handshake message, including the handshake message header carrying the handshake message type and length fields, but not including record layer headers [2].

We can derive the traffic keys as follows:

$$\begin{aligned}
\text{dHS} &= \texttt{Derive-Secret(HS, "derived", "")} \\
\text{MS} &= \texttt{HKDF-Extract(dHS, 0)} \\
\text{CATS} &= \texttt{Derive-Secret(MS, "c ap traffic", ClientHello...Server Finished)} \\
\text{SATS} &= \texttt{Derive-Secret(MS, "s ap traffic", ClientHello...Server Finished)} \\
tk_{capp} &= \texttt{HKDF-Expand-Label(CATS, "key", "", key\_length)} \\
tk_{sapp} &= \texttt{HKDF-Expand-Label(SATS, "key", "", key\_length)}
\end{aligned} \tag{5}$$

Client verification, authentication, key confirmation, and key derivation. The client P, upon receiving these messages, checks that the signature SCV (if in full 1-RTT mode) and the MAC SF verify correctly. If the server Q has requested client authentication, P will begin by sending its digital certificate (carrying its public-key) in the `ClientCertificate` message, after which P will compute its own certificate verify value `ClientCertificateVerify` by signing the session hash, then send it to Q as the `CertificateVerify` message. P finally derives the client finished key fk_C from CHTS and uses fk_C to compute a MAC tag CF over the session hash.

Server verification. The server will verify the final MAC (CF) and optional signature (CCV) messages of the client.

Handshake completion. At this point both parties can compute the resumption master secret (RMS) value that can be used as a pre-shared key for session resumption in the future. Both parties can now derive the client application traffic key (tk_{capp}), and use the record layer for encrypted communication of application data with the resulting keys.

$$RMS = \texttt{Derive-Secret(MS, "res master", ClientHello...Client Finished)} \tag{6}$$

4. TLS Record Protocol

The TLS record protocol [2] protects traffic between peers using parameters established by the handshake protocol. Each data packet is referred to as a record, containing a header and a payload. The header includes record type, protocol version, and payload length, while the payload contains the actual data being sent.

TLS defines a Record Layer to receive uninterpreted data from higher layers in non-empty blocks of arbitrary size [5]. This layer fragments information blocks into `TLSPlaintext` records with data chunks of 2^{14} bytes or less. There are four record types: `ChangeCipherSpec`, `Alert`, `Handshake`, and `ApplicationData`.

The `ChangeCipherSpec` record, now deprecated, is used to change the cipher suite used for encryption. The `Alert` record is used to send error messages. The `Handshake` record is used to send handshake messages, while the `ApplicationData` record is used to send application data.

Delignat-Lavaud et al [14] argue that each sub-protocol of TLS defines its own data stream, and the record layer is responsible for multiplexing all of these streams into one corresponding to network messages after fragmentation, formatting, padding, and optional record-layer encryption (application data is compulsorily encrypted). According to this model, the record layer is the exclusive user for all non-exported keys generated by the handshake.

The record protection functions by translating a `TLSPlaintext` structure into a `TLSCiphertext` structure [2]. `TLSCiphertext` (encrypted record) consists of a plaintext header followed by an encrypted body, which itself contains a type and optional padding.

As per new changes in TLS 1.3, the record layer protection is improved by adopting a single AEAD mode for all ciphersuites, thus deprecating all legacy modes (MAC-only, MAC-pad-encrypt, encrypt-then-MAC [15], compress-then-encrypt) [14]. The new AEAD mode is designed to be provably-secure and modular, supporting algorithms such as AES-GCM, AES-CCM, and ChaCha20-Poly1305 within the same framework. Recall that AEAD algorithms take as input a single key, a nonce, a plaintext, and "additional data" to be included in the authentication check.

The key is either of the traffic key generated in 5, the nonce is derived from the sequence number and the `client write iv` (iv_{capp}) or `server write iv` (iv_{sapp}), and the additional data input is the record header.

The IVs are generated as follows:

$$\begin{aligned} iv_{capp} &= \texttt{HKDF-Expand-Label(CATS, "iv", "", iv\_length)} \\ iv_{sapp} &= \texttt{HKDF-Expand-Label(SATS, "iv", "", iv\_length)} \end{aligned} \tag{7}$$

```
additional_data = TLSCiphertext.opaque_type ||
                  TLSCiphertext.legacy_record_version ||
                  TLSCiphertext.length
```

Here, `opaque_type` is always set to 23 for outward compatibility with middleboxes accustomed to parsing previous versions of TLS. The `legacy_record_version` field is always 0x0303. Finally, the `length` (in bytes) of the ecnrypted record.

A 64-bit sequence number is maintained separately for reading and writing records [2]. The appropriate sequence number is incremented by one after reading or writing each record. Each sequence number is set to zero at the beginning of a connection and whenever the key is changed; the first record transmitted under a particular traffic key must use sequence number 0. The padded sequence number is XORed with either the iv_{capp} or iv_{capp} (we may use handshake traffic ivs and keys in case of handshake messages). This resulting value is used as the per-record nonce for the AEAD algorithm.

References

[1] "Transmission Control Protocol," RFC 793, Sep. 1981. [Online]. Available: https://www.rfc-editor.org/info/rfc793

[2] E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.3," RFC 8446, Aug. 2018. [Online]. Available: https://www.rfc-editor.org/info/rfc8446

[3] A. S. Tanenbaum and D. J. Wetherall, *Computer Networks*, 5th ed. USA: Prentice Hall Press, 2010.

[4] "Internet Protocol," RFC 791, Sep. 1981. [Online]. Available: https://www.rfc-editor.org/info/rfc791

[5] C. Allen and T. Dierks, "The TLS Protocol Version 1.0," RFC 2246, Jan. 1999. [Online]. Available: https://www.rfc-editor.org/info/rfc2246

[6] B. Laurie, A. Langley, and E. Kasper, "Certificate Transparency," RFC 6962, Jun. 2013. [Online]. Available: https://www.rfc-editor.org/info/rfc6962

[7] A. Parsovs, "Practical issues with tls client certificate authentication," 01 2014.

[8] M. Bellare and C. Namprempre, "Authenticated encryption: Relations among notions and analysis of the generic composition paradigm," pp. 531–545, 12 2000.

[9] D. Boneh and V. Shoup, "A graduate course in applied cryptography," 2015. [Online]. Available: https://crypto.stanford.edu/~dabo/cryptobook/draft_0_2.pdf

[10] P. Rogaway, "Authenticated-encryption with associated-data," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, ser. CCS '02. New York, NY, USA: Association for Computing Machinery, 2002, p. 98–107. [Online]. Available: https://doi.org/10.1145/586110.586125

[11] D. H. Krawczyk, M. Bellare, and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication," RFC 2104, Feb. 1997. [Online]. Available: https://www.rfc-editor.org/info/rfc2104

[12] H. Krawczyk, "Cryptographic extraction and key derivation: The hkdf scheme," in *Advances in Cryptology – CRYPTO 2010*, T. Rabin, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 631–648.

[13] B. Dowling, M. Fischlin, F. Günther, and D. Stebila, "A cryptographic analysis of the tls 1.3 handshake protocol candidates," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 1197–1210. [Online]. Available: https://doi.org/10.1145/2810103.2813653

[14] A. Delignat-Lavaud, C. Fournet, M. Kohlweiss, J. Protzenko, A. Rastogi, N. Swamy, S. Zanella-Beguelin, K. Bhargavan, J. Pan, and J. K. Zinzindohoue, "Implementing and proving the tls 1.3 record layer," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 463–482.

[15] P. Gutmann, "Encrypt-then-MAC for Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS)," RFC 7366, Sep. 2014. [Online]. Available: https://www.rfc-editor.org/info/rfc7366