                Ephemeral Diffie-Hellman Over COSE (EDHOC)
                      draft-ietf-lake-edhoc-06

Abstract

   This document specifies Ephemeral Diffie-Hellman Over COSE (EDHOC), a
   very compact and lightweight authenticated Diffie-Hellman key
   exchange with ephemeral keys.  EDHOC provides mutual authentication,
   perfect forward secrecy, and identity protection.  EDHOC is intended
   for usage in constrained scenarios and a main use case is to
   establish an OSCORE security context.  By reusing COSE for
   cryptography, CBOR for encoding, and CoAP for transport, the
   additional code size can be kept very low.

Status of This Memo

   This Internet-Draft is submitted in full conformance with the
   provisions of BCP 78 and BCP 79.

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

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

   This Internet-Draft will expire on 23 October 2021.

Table of Contents

1.  Introduction

1.1.  Motivation

   Many Internet of Things (IoT) deployments require technologies which
   are highly performant in constrained environments [RFC7228].  IoT
   devices may be constrained in various ways, including memory,
   storage, processing capacity, and power.  The connectivity for these
   settings may also exhibit constraints such as unreliable and lossy
   channels, highly restricted bandwidth, and dynamic topology.  The
   IETF has acknowledged this problem by standardizing a range of
   lightweight protocols and enablers designed for the IoT, including
   the Constrained Application Protocol (CoAP, [RFC7252]), Concise
   Binary Object Representation (CBOR, [RFC8949]), and Static Context
   Header Compression (SCHC, [RFC8724]).

   The need for special protocols targeting constrained IoT deployments
   extends also to the security domain [I-D.ietf-lake-reqs].  Important
   characteristics in constrained environments are the number of round
   trips and protocol message sizes, which if kept low can contribute to
   good performance by enabling transport over a small number of radio
   frames, reducing latency due to fragmentation or duty cycles, etc.
   Another important criteria is code size, which may be prohibitive for
   certain deployments due to device capabilities or network load during
   firmware update.  Some IoT deployments also need to support a variety
   of underlying transport technologies, potentially even with a single
   connection.

   Some security solutions for such settings exist already.  CBOR Object
   Signing and Encryption (COSE, [I-D.ietf-cose-rfc8152bis-struct])
   specifies basic application-layer security services efficiently
   encoded in CBOR.  Another example is Object Security for Constrained
   RESTful Environments (OSCORE, [RFC8613]) which is a lightweight
   communication security extension to CoAP using CBOR and COSE.  In
   order to establish good quality cryptographic keys for security
   protocols such as COSE and OSCORE, the two endpoints may run an
   authenticated Diffie-Hellman key exchange protocol, from which shared
   secret key material can be derived.  Such a key exchange protocol
   should also be lightweight; to prevent bad performance in case of
   repeated use, e.g., due to device rebooting or frequent rekeying for
   security reasons; or to avoid latencies in a network formation
   setting with many devices authenticating at the same time.

This document specifies Ephemeral Diffie-Hellman Over COSE (EDHOC), a lightweight authenticated key exchange protocol providing good security properties including perfect forward secrecy, identity protection, and cipher suite negotiation.  Authentication can be based on raw public keys (RPK) or public key certificates, and requires the application to provide input on how to verify that endpoints are trusted.  This specification focuses on referencing instead of transporting credentials to reduce message overhead.

EDHOC makes use of known protocol constructions, such as SIGMA [SIGMA] and Extract-and-Expand [RFC5869].  COSE also provides crypto agility and enables the use of future algorithms targeting IoT.

## 1.2.  Use of EDHOC

EDHOC is designed for highly constrained settings making it especially suitable for low-power wide area networks [RFC8376] such as Cellular IoT, 6TiSCH, and LoRaWAN.  A main objective for EDHOC is to be a lightweight authenticated key exchange for OSCORE, i.e. to provide authentication and session key establishment for IoT use cases such as those built on CoAP [RFC7252].  CoAP is a specialized web transfer protocol for use with constrained nodes and networks, providing a request/response interaction model between application endpoints.  As such, EDHOC is targeting a large variety of use cases involving 'things' with embedded microcontrollers, sensors, and actuators.

A typical setting is when one of the endpoints is constrained or in a constrained network, and the other endpoint is a node on the Internet (such as a mobile phone) or at the edge of the constrained network (such as a gateway).  Thing-to-thing interactions over constrained networks are also relevant since both endpoints would then benefit from the lightweight properties of the protocol.  EDHOC could e.g. be run when a device connects for the first time, or to establish fresh keys which are not revealed by a later compromise of the long-term keys.  Further security properties are described in Section 8.1.

EDHOC enables the reuse of the same lightweight primitives as OSCORE: CBOR for encoding, COSE for cryptography, and CoAP for transport.  By reusing existing libraries the additional code size can be kept very low.  Note that, while CBOR and COSE primitives are built into the protocol messages, EDHOC is not bound to a particular transport.  However, it is recommended to transfer EDHOC messages in CoAP payloads as is detailed in Section 7.2.

1.3.  Message Size Examples

   Compared to the DTLS 1.3 handshake [I-D.ietf-tls-dtls13] with ECDHE
   and connection ID, the number of bytes in EDHOC + CoAP can be less
   than 1/6 when RPK authentication is used, see
   [I-D.ietf-lwig-security-protocol-comparison].  Figure 1 shows two
   examples of message sizes for EDHOC with different kinds of
   authentication keys and different COSE header parameters for
   identification: static Diffie-Hellman keys identified by 'kid'
   [I-D.ietf-cose-rfc8152bis-struct], and X.509 signature certificates
   identified by a hash value using 'x5t' [I-D.ietf-cose-x509].

```
               ================================
                                 kid      x5t
               --------------------------------
               message_1          37       37
               message_2          46      117
               message_3          20       91
               --------------------------------
               Total             103      245
               ================================
```

                 Figure 1: Example of message sizes in bytes.

1.4.  Document Structure

   The remainder of the document is organized as follows: Section 2
   outlines EDHOC authenticated with digital signatures, Section 3
   describes the protocol elements of EDHOC, including message flow, and
   formatting of the ephemeral public keys, Section 4 describes the key
   derivation, Section 5 specifies EDHOC with authentication based on
   signature keys or static Diffie-Hellman keys, Section 6 specifies the
   EDHOC error message, and Section 7 describes how EDHOC can be
   transferred in CoAP and used to establish an OSCORE security context.

1.5.  Terminology and Requirements Language

   The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT",
   "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and
   "OPTIONAL" in this document are to be interpreted as described in BCP
   14 [RFC2119] [RFC8174] when, and only when, they appear in all
   capitals, as shown here.

   Readers are expected to be familiar with the terms and concepts
   described in CBOR [RFC8949], CBOR Sequences [RFC8742], COSE
   structures and process [I-D.ietf-cose-rfc8152bis-struct], COSE
   algorithms [I-D.ietf-cose-rfc8152bis-algs], and CDDL [RFC8610].  The
   Concise Data Definition Language (CDDL) is used to express CBOR data

structures [RFC8949].  Examples of CBOR and CDDL are provided in
Appendix A.1.  When referring to CBOR, this specification always
refer to Deterministically Encoded CBOR as specified in Sections
4.2.1 and 4.2.2 of [RFC8949].

The single output from authenticated encryption (including the
authentication tag) is called 'ciphertext', following [RFC5116].

## 2.  EDHOC Outline

EDHOC specifies different authentication methods of the Diffie-
Hellman key exchange: digital signatures and static Diffie-Hellman
keys.  This section outlines the digital signature based method.
Further details of protocol elements and other authentication methods
are provided in the remainder of this document.

SIGMA (SIGn-and-MAc) is a family of theoretical protocols with a
large number of variants [SIGMA].  Like IKEv2 [RFC7296] and (D)TLS
1.3 [RFC8446], EDHOC authenticated with digital signatures is built
on a variant of the SIGMA protocol which provides identity protection
of the initiator (SIGMA-I), and like IKEv2 [RFC7296], EDHOC
implements the SIGMA-I variant as MAC-then-Sign.  The SIGMA-I
protocol using an authenticated encryption algorithm is shown in
Figure 2.

```
   Initiator                                             Responder
     |                              G_X                       |
     +------------------------------------------------------->|
     |                                                        |
     |   G_Y, AEAD( K_2; ID_CRED_R, Sig(R; CRED_R, G_X, G_Y) )|
     |<-------------------------------------------------------+
     |                                                        |
     |        AEAD( K_3; ID_CRED_I, Sig(I; CRED_I, G_Y, G_X) )|
     +------------------------------------------------------->|
     |                                                        |
```
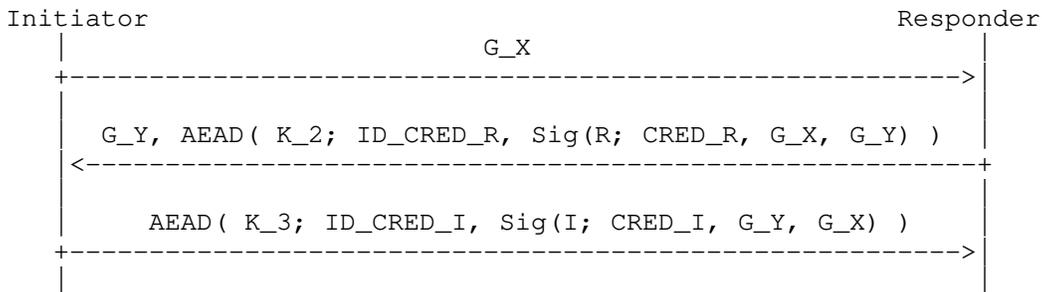
Figure 2: Authenticated encryption variant of the SIGMA-I protocol.

The parties exchanging messages are called Initiator (I) and
Responder (R).  They exchange ephemeral public keys, compute a shared
secret, and derive symmetric application keys used to protect
application data.

*  G_X and G_Y are the ECDH ephemeral public keys of I and R,
   respectively.

*  CRED_I and CRED_R are the credentials containing the public
   authentication keys of I and R, respectively.

* ID_CRED_I and ID_CRED_R are credential identifiers enabling the
  recipient party to retrieve the credential of I and R,
  respectively.

* Sig(I; . ) and Sig(R; . ) denote signatures made with the private
  authentication key of I and R, respectively.

* AEAD(K; . ) denotes authenticated encryption with additional data
  using a key K derived from the shared secret.

In order to create a "full-fledged" protocol some additional protocol
elements are needed.  EDHOC adds:

* Explicit connection identifiers C_I, C_R chosen by I and R,
  respectively, enabling the recipient to find the protocol state.

* Transcript hashes (hashes of message data) TH_2, TH_3, TH_4 used
  for key derivation and as additional authenticated data.

* Computationally independent keys derived from the ECDH shared
  secret and used for authenticated encryption of different
  messages.

* An optional fourth message giving explicit key confirmation to I
  in deployments where no protected application data is sent from R
  to I.

* A key material exporter and a key update function enabling
  frequent forward secrecy.

* Verification of a common preferred cipher suite:

  - The Initiator lists supported cipher suites in order of
    preference

  - The Responder verifies that the selected cipher suite is the
    first supported cipher suite (or else rejects and states
    supported cipher suites).

* Method types and error handling.

* Transport of opaque auxiliary data.

EDHOC is designed to encrypt and integrity protect as much
information as possible, and all symmetric keys are derived using as
much previous information as possible.  EDHOC is furthermore designed
to be as compact and lightweight as possible, in terms of message
sizes, processing, and the ability to reuse already existing CBOR,
COSE, and CoAP libraries.

To simplify for implementors, the use of CBOR and COSE in EDHOC is
summarized in Appendix A and test vectors including CBOR diagnostic
notation are given in Appendix B.

## 3.  Protocol Elements

### 3.1.  General

An EDHOC message flow consists of three mandatory messages
(message_1, message_2, message_3) between Initiator and Responder, an
optional fourth message (message_4), plus an EDHOC error message.
EDHOC messages are CBOR Sequences [RFC8742], see Figure 3.  The
protocol elements in the figure are introduced in the following
sections.  Message formatting and processing is specified in
Section 5 and Section 6.  An implementation may support only
Initiator or only Responder.

Application data is protected using the agreed application algorithms
(AEAD, hash) in the selected cipher suite (see Section 3.4) and the
application can make use of the established connection identifiers
C_1, C_I, and C_R (see Section 3.2.4).  EDHOC may be used with the
media type application/edhoc defined in Section 9.

The Initiator can derive symmetric application keys after creating
EDHOC message_3, see Section 4.1.  Application protected data can
therefore be sent in parallel or together with EDHOC message_3.

```
Initiator                                                      Responder
|              C_1, METHOD_CORR, SUITES_I, G_X, C_I, AD_1            |
+------------------------------------------------------------------>|
|                             message_1                             |
|                                                                   |
|       C_I, G_Y, C_R, Enc(ID_CRED_R, Signature_or_MAC_2, AD_2)     |
|<------------------------------------------------------------------+
|                             message_2                             |
|                                                                   |
|        C_R, AEAD(K_3ae; ID_CRED_I, Signature_or_MAC_3, AD_3)      |
+------------------------------------------------------------------>|
|                             message_3                             |
```
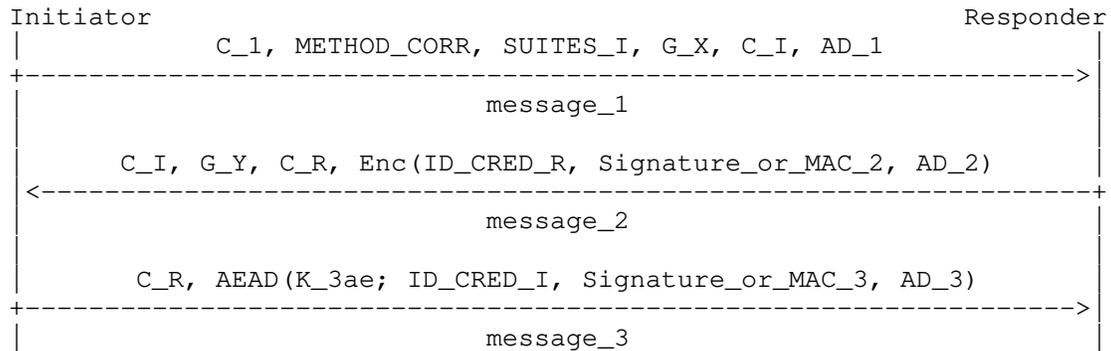
Figure 3: EDHOC Message Flow

## 3.2.  Method and Correlation

The data item METHOD_CORR in message_1 (see Section 5.3.1), is an
integer specifying the method and the correlation properties of the
transport, which are described in this section.

### 3.2.1.  Method

EDHOC supports authentication with signature or static Diffie-Hellman
keys, as defined in the four authentication methods: 0, 1, 2, and 3,
see Figure 4.  (Method 0 corresponds to the case outlined in
Section 2 where both Initiator and Responder authenticate with
signature keys.)

An implementation may support only a single method.  The Initiator
and the Responder need to have agreed on a single method to be used
for EDHOC, see Section 3.7.

```
+-------+------------------+------------------+------------------+
| Value | Initiator        | Responder        | Reference        |
+-------+------------------+------------------+------------------+
|     0 | Signature Key    | Signature Key    | [[this document]] |
|     1 | Signature Key    | Static DH Key    | [[this document]] |
|     2 | Static DH Key    | Signature Key    | [[this document]] |
|     3 | Static DH Key    | Static DH Key    | [[this document]] |
+-------+------------------+------------------+------------------+
```

Figure 4: Method Types

### 3.2.2.  Connection Identifiers

EDHOC includes optional connection identifiers (C_1, C_I, C_R).  The
connection identifiers C_1, C_I, and C_R do not have any
cryptographic purpose in EDHOC.  They contain information
facilitating retrieval of the protocol state and may therefore be
very short.  C_1 is always set to "null", while C_I and C_R are
chosen by I and R, respectively.  One byte connection identifiers are
realistic in many scenarios as most constrained devices only have a
few connections.  In cases where a node only has one connection, the
identifiers may even be the empty byte string.

The connection identifier MAY be used with an application protocol
(e.g.  OSCORE) for which EDHOC establishes keys, in which case the
connection identifiers SHALL adhere to the requirements for that
protocol.  Each party choses a connection identifier it desires the
other party to use in outgoing messages.  (For OSCORE this results in
the endpoint selecting its Recipient ID, see Section 3.1 of
[RFC8613]).

3.2.3.  Transport

   Cryptographically, EDHOC does not put requirements on the lower
   layers.  EDHOC is not bound to a particular transport layer, and can
   be used in environments without IP.  The transport is responsible to
   handle message loss, reordering, message duplication, fragmentation,
   and denial of service protection, where necessary.

   The Initiator and the Responder need to have agreed on a transport to
   be used for EDHOC, see Section 3.7.  It is recommended to transport
   EDHOC in CoAP payloads, see Section 7.

3.2.4.  Message Correlation

   If the whole transport path provides a mechanism for correlating
   messages received with messages previously sent, then some of the
   connection identifiers may be omitted.  There are four cases:

   *  corr = 0, the transport does not provide a correlation mechanism.

   *  corr = 1, the transport provides a correlation mechanism that
      enables the Responder to correlate message_2 and message_1 as well
      as message_4 and message_3.

   *  corr = 2, the transport provides a correlation mechanism that
      enables the Initiator to correlate message_3 and message_2.

   *  corr = 3, the transport provides a correlation mechanism that
      enables both parties to correlate all three messages.

   For example, if the key exchange is transported over CoAP, the CoAP
   Token can be used to correlate messages, see Section 7.2.

3.3.  Authentication Parameters

### 3.3.1.  Authentication Keys

The authentication key MUST be a signature key or static Diffie-Hellman key.  The Initiator and the Responder MAY use different types of authentication keys, e.g. one uses a signature key and the other uses a static Diffie-Hellman key.  When using a signature key, the authentication is provided by a signature.  When using a static Diffie-Hellman key the authentication is provided by a Message Authentication Code (MAC) computed from an ephemeral-static ECDH shared secret which enables significant reductions in message sizes. The MAC is implemented with an AEAD algorithm.  When using static Diffie-Hellman keys the Initiator's and Responder's private authentication keys are called I and R, respectively, and the public authentication keys are called G_I and G_R, respectively.

*  Only the Responder SHALL have access to the Responder's private
   authentication key.

*  Only the Initiator SHALL have access to the Initiator's private
   authentication key.

### 3.3.2.  Identities

EDHOC assumes the existence of mechanisms (certification authority, trusted third party, manual distribution, etc.) for specifying and distributing authentication keys and identities.  Policies are set based on the identity of the other party, and parties typically only allow connections from a specific identity or a small restricted set of identities.  For example, in the case of a device connecting to a network, the network may only allow connections from devices which authenticate with certificates having a particular range of serial numbers in the subject field and signed by a particular CA.  On the other side, the device may only be allowed to connect to a network which authenticates with a particular public key (information of which may be provisioned, e.g., out of band or in the Auxiliary Data, see Section 3.6).

The EDHOC implementation must be able to receive and enforce information from the application about what is the intended endpoint, and in particular whether it is a specific identity or a set of identities.

*  When a Public Key Infrastructure (PKI) is used, the trust anchor
   is a Certification Authority (CA) certificate, and the identity is
   the subject whose unique name (e.g. a domain name, NAI, or EUI) is
   included in the endpoint's certificate.  Before running EDHOC each
   party needs at least one CA public key certificate, or just the
   public key, and a specific identity or set of identities it is

allowed to communicate with.  Only validated public-key
certificates with an allowed subject name, as specified by the
application, are to be accepted.  EDHOC provides proof that the
other party possesses the private authentication key corresponding
to the public authentication key in its certificate.  The
certification path provides proof that the subject of the
certificate owns the public key in the certificate.

*  When public keys are used but not with a PKI (RPK, self-signed
   certificate), the trust anchor is the public authentication key of
   the other party.  In this case, the identity is typically directly
   associated to the public authentication key of the other party.
   For example, the name of the subject may be a canonical
   representation of the public key.  Alternatively, if identities
   can be expressed in the form of unique subject names assigned to
   public keys, then a binding to identity can be achieved by
   including both public key and associated subject name in the
   protocol message computation: CRED_I or CRED_R may be a self-
   signed certificate or COSE_Key containing the public
   authentication key and the subject name, see Section 3.3.3.
   Before running EDHOC, each endpoint needs a specific public
   authentication key/unique associated subject name, or a set of
   public authentication keys/unique associated subject names, which
   it is allowed to communicate with.  EDHOC provides proof that the
   other party possesses the private authentication key corresponding
   to the public authentication key.

3.3.3.  Authentication Credentials

   The authentication credentials, CRED_I and CRED_R, contain the public
   authentication key of the Initiator and the Responder, respectively.
   The Initiator and the Responder MAY use different types of
   credentials, e.g. one uses an RPK and the other uses a public key
   certificate.

   The credentials CRED_I and CRED_R are signed or MAC:ed (depending on
   method) by the Initiator and the Responder, respectively, see
   Section 5.5 and Section 5.4.

   When the credential is a certificate, CRED_x is an end-entity
   certificate (i.e. not the certificate chain) encoded as a CBOR bstr.
   In X.509 certificates, signature keys typically have key usage
   "digitalSignature" and Diffie-Hellman keys typically have key usage
   "keyAgreement".

   To prevent misbinding attacks in systems where an attacker can
   register public keys without proving knowledge of the private key,
   SIGMA [SIGMA] enforces a MAC to be calculated over the "Identity",

which in case of a X.509 certificate would be the 'subject' and
'subjectAltName' fields.  EDHOC follows SIGMA by calculating a MAC
over the whole certificate.  While the SIGMA paper only focuses on
the identity, the same principle is true for any information such as
policies connected to the public key.

When the credential is a COSE_Key, CRED_x is a CBOR map only
containing specific fields from the COSE_Key identifying the public
key, and optionally the "Identity".  CRED_x needs to be defined such
that it is identical when generated by Initiator or Responder.  The
parameters SHALL be encoded in bytewise lexicographic order of their
deterministic encodings as specified in Section 4.2.1 of [RFC8949].

If the parties have agreed on an identity besides the public key, the
identity is included in the CBOR map with the label "subject name",
otherwise the subject name is the empty text string.  The public key
parameters depend on key type.

*  For COSE_Keys of type OKP the CBOR map SHALL, except for subject
   name, only include the parameters 1 (kty), -1 (crv), and -2
   (x-coordinate).

*  For COSE_Keys of type EC2 the CBOR map SHALL, except for subject
   name, only include the parameters 1 (kty), -1 (crv), -2
   (x-coordinate), and -3 (y-coordinate).

An example of CRED_x when the RPK contains an X25519 static Diffie-
Hellman key and the parties have agreed on an EUI-64 identity is
shown below:

```
CRED_x = {
  1:  1,
 -1:  4,
 -2:  h'b1a3e89460e88d3a8d54211dc95f0b90
       3ff205eb71912d6db8f4af980d2db83a',
 "subject name" : "42-50-31-FF-EF-37-32-39"
}
```

3.3.4.  Identification of Credentials

ID_CRED_I and ID_CRED_R are identifiers of the public authentication
keys of the Initiator and the Responder, respectively.  ID_CRED_I and
ID_CRED_R do not have any cryptographic purpose in EDHOC.

*  ID_CRED_R is intended to facilitate for the Initiator to retrieve
   the Responder's public authentication key.

   *  ID_CRED_I is intended to facilitate for the Responder to retrieve
      the Initiator's public authentication key.

   The identifiers ID_CRED_I and ID_CRED_R are COSE header_maps, i.e.
   CBOR maps containing Common COSE Header Parameters, see Section 3.1
   of [I-D.ietf-cose-rfc8152bis-struct]).  In the following we give some
   examples of COSE header_maps.

   Raw public keys are most optimally stored as COSE_Key objects and
   identified with a 'kid' parameter:

   *  ID_CRED_x = { 4 : kid_x }, where kid_x : bstr, for x = I or R.

   Public key certificates can be identified in different ways.  Header
   parameters for identifying C509 certificates and X.509 certificates
   are defined in [I-D.mattsson-cose-cbor-cert-compress] and
   [I-D.ietf-cose-x509], for example:

   *  by a hash value with the 'c5t' or 'x5t' parameters;

      -  ID_CRED_x = { 34 : COSE_CertHash }, for x = I or R,

      -  ID_CRED_x = { TDB3 : COSE_CertHash }, for x = I or R,

   *  by a URI with the 'c5u' or 'x5u' parameters;

      -  ID_CRED_x = { 35 : uri }, for x = I or R,

      -  ID_CRED_x = { TBD4 : uri }, for x = I or R,

   *  ID_CRED_x MAY contain the actual credential used for
      authentication, CRED_x.  For example, a certificate chain can be
      transported in ID_CRED_x with COSE header parameter c5c or
      x5chain, defined in [I-D.mattsson-cose-cbor-cert-compress] and
      [I-D.ietf-cose-x509].

   It is RECOMMENDED that ID_CRED_x uniquely identify the public
   authentication key as the recipient may otherwise have to try several
   keys.  ID_CRED_I and ID_CRED_R are transported in the 'ciphertext',
   see Section 5.5 and Section 5.4.

   When ID_CRED_x does not contain the actual credential it may be very
   short.  One byte credential identifiers are realistic in many
   scenarios as most constrained devices only have a few keys.  In cases
   where a node only has one key, the identifier may even be the empty
   byte string.

3.4.  Cipher Suites

   An EDHOC cipher suite consists of an ordered set of COSE code points
   from the "COSE Algorithms" and "COSE Elliptic Curves" registries:

   *  EDHOC AEAD algorithm

   *  EDHOC hash algorithm

   *  EDHOC ECDH curve

   *  EDHOC signature algorithm

   *  EDHOC signature algorithm curve

   *  Application AEAD algorithm

   *  Application hash algorithm

   Each cipher suite is identified with a pre-defined int label.

   EDHOC can be used with all algorithms and curves defined for COSE.
   Implementation can either use one of the pre-defined cipher suites
   (Section 9.1) or use any combination of COSE algorithms to define
   their own private cipher suite.  Private cipher suites can be
   identified with any of the four values -24, -23, -22, -21.

   The following cipher suites are for constrained IoT where message
   overhead is a very important factor:

      0.  ( 10, -16, 4, -8, 6, 10, -16 )
          (AES-CCM-16-64-128, SHA-256, X25519, EdDSA, Ed25519,
           AES-CCM-16-64-128, SHA-256)

      1.  ( 30, -16, 4, -8, 6, 10, -16 )
          (AES-CCM-16-128-128, SHA-256, X25519, EdDSA, Ed25519,
           AES-CCM-16-64-128, SHA-256)

      2.  ( 10, -16, 1, -7, 1, 10, -16 )
          (AES-CCM-16-64-128, SHA-256, P-256, ES256, P-256,
           AES-CCM-16-64-128, SHA-256)

      3.  ( 30, -16, 1, -7, 1, 10, -16 )
          (AES-CCM-16-128-128, SHA-256, P-256, ES256, P-256,
           AES-CCM-16-64-128, SHA-256)

The following cipher suite is for general non-constrained
applications.  It uses very high performance algorithms that also are
widely supported:

    4. ( 1, -16, 4, -7, 1, 1, -16 )
       (A128GCM, SHA-256, X25519, ES256, P-256,
        A128GCM, SHA-256)

The following cipher suite is for high security application such as
government use and financial applications.  It is compatible with the
CNSA suite [CNSA].

    5. ( 3, -43, 2, -35, 2, 3, -43 )
       (A256GCM, SHA-384, P-384, ES384, P-384,
        A256GCM, SHA-384)

The different methods use the same cipher suites, but some algorithms
are not used in some methods.  The EDHOC signature algorithm and the
EDHOC signature algorithm curve are not used in methods without
signature authentication.

The Initiator needs to have a list of cipher suites it supports in
order of preference.  The Responder needs to have a list of cipher
suites it supports.  SUITES_I is a CBOR array containing cipher
suites that the Initiator supports.  SUITES_I is formatted and
processed as detailed in Section 5.3.1 to secure the cipher suite
negotiation.  Examples of cipher suite negotiation are given in
Section 6.3.2.

3.5.  Ephemeral Public Keys

The ECDH ephemeral public keys are formatted as a COSE_Key of type
EC2 or OKP according to Sections 7.1 and 7.2 of
[I-D.ietf-cose-rfc8152bis-algs], but only the 'x' parameter is
included G_X and G_Y.  For Elliptic Curve Keys of type EC2, compact
representation as per [RFC6090] MAY be used also in the COSE_Key.  If
the COSE implementation requires an 'y' parameter, any of the
possible values of the y-coordinate can be used, see Appendix C of
[RFC6090].  COSE always use compact output for Elliptic Curve Keys of
type EC2.

3.6.  Auxiliary Data

   In order to reduce round trips and number of messages, and in some
   cases also streamline processing, certain security applications may
   be integrated into EDHOC by transporting auxiliary data together with
   the messages.  One example is the transport of third-party
   authorization information protected outside of EDHOC
   [I-D.selander-ace-ake-authz].  Another example is the embedding of a
   certificate enrolment request or a newly issued certificate.

   EDHOC allows opaque auxiliary data (AD) to be sent in the EDHOC
   messages.  Unprotected Auxiliary Data (AD_1, AD_2) may be sent in
   message_1 and message_2, respectively.  Protected Auxiliary Data
   (AD_3) may be sent in message_3.

   Since data carried in AD_1 and AD_2 may not be protected, and the
   content of AD_3 is available to both the Initiator and the Responder,
   special considerations need to be made such that the availability of
   the data a) does not violate security and privacy requirements of the
   service which uses this data, and b) does not violate the security
   properties of EDHOC.

3.7.  Applicability Statement

   EDHOC requires certain parameters to be agreed upon between Initiator
   and Responder.  Some parameters can be agreed through the protocol
   execution (specifically cipher suite negotiation, see Section 3.4)
   but other parameters may need to be known out-of-band (e.g., which
   authentication method is used, see Section 3.2.1).

   The purpose of the applicability statement is describe the intended
   use of EDHOC to allow for the relevant processing and verifications
   to be made, including things like:

   1.  How the endpoint detects that an EDHOC message is received.  This
       includes how EDHOC messages are transported, for example in the
       payload of a CoAP message with a certain Uri-Path or Content-
       Format; see Section 7.2.

   2.  Method and correlation of underlying transport messages
       (METHOD_CORR; see Section 3.2.1 and Section 3.2.4).  This gives
       information about the optional connection identifier fields.

   3.  How message_1 is identified, in particular if the optional
       initial C_1 = "null" of message_1 is present; see Section 5.3.1

   4.  Authentication credentials (CRED_I, CRED_R; see Section 3.3.3).

5. Type used to identify authentication credentials (ID_CRED_I,
   ID_CRED_R; see Section 3.3.4).

6. Use and type of Auxiliary Data (AD_1, AD_2, AD_3; see
   Section 3.6).

7. Identifier used as identity of endpoint; see Section 3.3.2.

8. If message_4 shall be sent/expected, and if not, how to ensure a
   protected application message is sent from the Responder to the
   Initiator; see Section 7.1.

The applicability statement may also contain information about
supported cipher suites.  The procedure for selecting and verifying
cipher suite is still performed as specified by the protocol, but it
may become simplified by this knowledge.

An example of an applicability statement is shown in Appendix C.

For some parameters, like METHOD_CORR, ID_CRED_x, type of AD_x, the
receiver is able to verify compliance with applicability statement,
and if it needs to fail because of incompliance, to infer the reason
why the protocol failed.

For other parameters, like CRED_x in the case that it is not
transported, it may not be possible to verify that incompliance with
applicability statement was the reason for failure: Integrity
verification in message_2 or message_3 may fail not only because of
wrong authentication credential.  For example, in case the Initiator
uses public key certificate by reference (i.e. not transported within
the protocol) then both endpoints need to use an identical data
structure as CRED_I or else the integrity verification will fail.

Note that it is not necessary for the endpoints to specify a single
transport for the EDHOC messages.  For example, a mix of CoAP and
HTTP may be used along the path, and this may still allow correlation
between messages.

The applicability statement may be dependent on the identity of the
other endpoint, but this applies only to the later phases of the
protocol when identities are known.  (Initiator does not know
identity of Responder before having verified message_2, and Responder
does not know identity of Initiator before having verified
message_3.)

Other conditions may be part of the applicability statement, such as
target application or use (if there is more than one application/use)
to the extent that EDHOC can distinguish between them.  In case

multiple applicability statements are used, the receiver needs to be able to determine which is applicable for a given protocol instance, for example based on URI or Auxiliary Data type.

4.  Key Derivation

EDHOC uses Extract-and-Expand [RFC5869] with the EDHOC hash algorithm in the selected cipher suite to derive keys used in EDHOC and in the application.  Extract is used to derive fixed-length uniformly pseudorandom keys (PRK) from ECDH shared secrets.  Expand is used to derive additional output keying material (OKM) from the PRKs.  The PRKs are derived using Extract.

    PRK = Extract( salt, IKM )

If the EDHOC hash algorithm is SHA-2, then Extract( salt, IKM ) = HKDF-Extract( salt, IKM ) [RFC5869].  If the EDHOC hash algorithm is SHAKE128, then Extract( salt, IKM ) = KMAC128( salt, IKM, 256, "" ). If the EDHOC hash algorithm is SHAKE256, then Extract( salt, IKM ) = KMAC256( salt, IKM, 512, "" ).

PRK_2e is used to derive a keystream to encrypt message_2.  PRK_3e2m is used to derive keys and IVs to produce a MAC in message_2 and to encrypt message_3.  PRK_4x3m is used to derive keys and IVs to produce a MAC in message_3 and to derive application specific data.

PRK_2e is derived with the following input:

*  The salt SHALL be the empty byte string.  Note that [RFC5869] specifies that if the salt is not provided, it is set to a string of zeros (see Section 2.2 of [RFC5869]).  For implementation purposes, not providing the salt is the same as setting the salt to the empty byte string.

*  The input keying material (IKM) SHALL be the ECDH shared secret G_XY (calculated from G_X and Y or G_Y and X) as defined in Section 6.3.1 of [I-D.ietf-cose-rfc8152bis-algs].

Example: Assuming the use of SHA-256 the extract phase of HKDF produces PRK_2e as follows:

    PRK_2e = HMAC-SHA-256( salt, G_XY )

where salt = 0x (the empty byte string).

The pseudorandom keys PRK_3e2m and PRK_4x3m are defined as follow:

   *   If the Responder authenticates with a static Diffie-Hellman key,
       then PRK_3e2m = Extract( PRK_2e, G_RX ), where G_RX is the ECDH
       shared secret calculated from G_R and X, or G_X and R, else
       PRK_3e2m = PRK_2e.

   *   If the Initiator authenticates with a static Diffie-Hellman key,
       then PRK_4x3m = Extract( PRK_3e2m, G_IY ), where G_IY is the ECDH
       shared secret calculated from G_I and Y, or G_Y and I, else
       PRK_4x3m = PRK_3e2m.

   Example: Assuming the use of curve25519, the ECDH shared secrets
   G_XY, G_RX, and G_IY are the outputs of the X25519 function
   [RFC7748]:

       G_XY = X25519( Y, G_X ) = X25519( X, G_Y )

   The keys and IVs used in EDHOC are derived from PRK using Expand
   [RFC5869] where the EDHOC-KDF is instantiated with the EDHOC AEAD
   algorithm in the selected cipher suite.

       OKM = EDHOC-KDF( PRK, transcript_hash, label, length )
           = Expand( PRK, info, length )

   where info is the CBOR encoding of

   info = [
      edhoc_aead_id : int / tstr,
      transcript_hash : bstr,
      label : tstr,
      length : uint
   ]

   where

   *   edhoc_aead_id is an int or tstr containing the algorithm
       identifier of the EDHOC AEAD algorithm in the selected cipher
       suite encoded as defined in [I-D.ietf-cose-rfc8152bis-algs].  Note
       that a single fixed edhoc_aead_id is used in all invocations of
       EDHOC-KDF, including the derivation of KEYSTREAM_2 and invocations
       of the EDHOC-Exporter.

   *   transcript_hash is a bstr set to one of the transcript hashes
       TH_2, TH_3, or TH_4 as defined in Sections 5.4.1, 5.5.1, and 4.1.

   *   label is a tstr set to the name of the derived key or IV, i.e.
       "K_2m", "IV_2m", "KEYSTREAM_2", "K_3m", "IV_3m", "K_3ae", or
       "IV_3ae".

* length is the length of output keying material (OKM) in bytes

If the EDHOC hash algorithm is SHA-2, then Expand( PRK, info, length ) = HKDF-Expand( PRK, info, length ) [RFC5869].  If the EDHOC hash algorithm is SHAKE128, then Expand( PRK, info, length ) = KMAC128( PRK, info, L, "" ).  If the EDHOC hash algorithm is SHAKE256, then Expand( PRK, info, length ) = KMAC256( PRK, info, L, "" ).

KEYSTREAM_2 are derived using the transcript hash TH_2 and the pseudorandom key PRK_2e.  K_2m and IV_2m are derived using the transcript hash TH_2 and the pseudorandom key PRK_3e2m.  K_3ae and IV_3ae are derived using the transcript hash TH_3 and the pseudorandom key PRK_3e2m.  K_3m and IV_3m are derived using the transcript hash TH_3 and the pseudorandom key PRK_4x3m.  IVs are only used if the EDHOC AEAD algorithm uses IVs.

## 4.1.  EDHOC-Exporter Interface

Application keys and other application specific data can be derived using the EDHOC-Exporter interface defined as:

    EDHOC-Exporter(label, length)
      = EDHOC-KDF(PRK_4x3m, TH_4, label, length)

where label is a tstr defined by the application and length is a uint defined by the application.  The label SHALL be different for each different exporter value.  The transcript hash TH_4 is a CBOR encoded bstr and the input to the hash function is a CBOR Sequence.

    TH_4 = H( TH_3, CIPHERTEXT_3 )

where H() is the hash function in the selected cipher suite.  Example use of the EDHOC-Exporter is given in Sections 7.2.1.

To provide forward secrecy in an even more efficient way than re-running EDHOC, EDHOC provides the function EDHOC-KeyUpdate.  When EDHOC-KeyUpdate is called the old PRK_4x3m is deleted and the new PRk_4x3m is calculated as a "hash" of the old key using the Extract function as illustrated by the following pseudocode:

    EDHOC-KeyUpdate( nonce ):
        PRK_4x3m = Extract( nonce, PRK_4x3m )

## 5.  Message Formatting and Processing

This section specifies formatting of the messages and processing steps.  Error messages are specified in Section 6.

An EDHOC message is encoded as a sequence of CBOR data (CBOR
Sequence, [RFC8742]).  Additional optimizations are made to reduce
message overhead.

While EDHOC uses the COSE_Key, COSE_Sign1, and COSE_Encrypt0
structures, only a subset of the parameters is included in the EDHOC
messages.  The unprotected COSE header in COSE_Sign1, and
COSE_Encrypt0 (not included in the EDHOC message) MAY contain
parameters (e.g. 'alg').

5.1.  Encoding of bstr_identifier

   Byte strings are encoded in CBOR as two or more bytes, whereas
   integers in the interval -24 to 23 are encoded in CBOR as one byte.

   bstr_identifier is a special encoding of byte strings, used
   throughout the protocol to enable the encoding of the shortest byte
   strings as integers that only require one byte of CBOR encoding.

   The bstr_identifier encoding is defined as follows: Byte strings in
   the interval h'00' to h'2f' are encoded as the corresponding integer
   minus 24, which are all represented by one byte CBOR ints.  Other
   byte strings are encoded as CBOR byte strings.

   For example, the byte string h'59e9' encoded as a bstr_identifier is
   equal to h'59e9', while the byte string h'2a' is encoded as the
   integer 18.

   The CDDL definition of the bstr_identifier is given below:

   bstr_identifier = bstr / int

   Note that, despite what could be interpreted by the CDDL definition
   only, bstr_identifier once decoded are always byte strings.

5.2.  Message Processing Outline

   This section outlines the message processing of EDHOC.

   For each protocol instance, the endpoints are assumed to keep an
   associated protocol state containing connection identifiers, keys,
   etc. used for subsequent processing of protocol related data.  The
   protocol state is assumed to be associated to an applicability
   statement (Section 3.7) which provides the context for how messages
   are transported, identified and processed.

EDHOC messages SHALL be processed according to the current protocol
state.  The following steps are expected to be performed at reception
of an EDHOC message:

1.  Detect that an EDHOC message has been received, for example by
    means of port number, URI, or media type (Section 3.7).

2.  Retrieve the protocol state, e.g. using the received connection
    identifier (Section 3.2.2) or with the help of message
    correlation provided by the transport protocol (Section 3.2.4).
    If there is no protocol state, in the case of message_1, a new
    protocol state is created.  An initial C_1 = "null" byte in
    message_1 (Section 5.3.1) can be used to distinguish message_1
    from other messages.  The Responder endpoint needs to make use of
    available Denial-of-Service mitigation (Section 8.5).

3.  If the message received is an error message then process
    according to Section 6, else process as the expected next message
    according to the protocol state.

If the processing fails, then the protocol is discontinued, an error
message sent, and the protocol state erased.  Further details are
provided in the following subsections.

Different instances of the same message MUST NOT be processed in one
protocol instance.  Note that processing will fail if the same
message appears a second time for EDHOC processing because the state
of the protocol has moved on and now expects something else.  This
assumes that message duplication due to re-transmissions is handled
by the transport protocol, see Section 3.2.3.  The case when the
transport does not support message deduplication is addressed in
Appendix D.

5.3.  EDHOC Message 1

5.3.1.  Formatting of Message 1

message_1 SHALL be a CBOR Sequence (see Appendix A.1) as defined
below

```
message_1 = (
  ? C_1 : null,
  METHOD_CORR : int,
  SUITES_I : [ selected : suite, supported : 2* suite ] / suite,
  G_X : bstr,
  C_I : bstr_identifier,
  ? AD_1 : bstr,
)

suite = int
```

where:

* C_1 - an initial CBOR simple value "null" (= 0xf6) MAY be used to distinguish message_1 from other messages.

* METHOD_CORR = 4 * method + corr, where method = 0, 1, 2, or 3 (see Figure 4) and the correlation parameter corr is chosen based on the transport and determines which connection identifiers that are omitted (see Section 3.2.4).

* SUITES_I - cipher suites which the Initiator supports in order of (decreasing) preference.  The list of supported cipher suites can be truncated at the end, as is detailed in the processing steps below and Section 6.3.  One of the supported cipher suites is selected.  The selected suite is the first suite in the SUITES_I CBOR array.  If a single supported cipher suite is conveyed then that cipher suite is selected and SUITES_I is encoded as an int instead of an array.

* G_X - the ephemeral public key of the Initiator

* C_I - variable length connection identifier, encoded as a bstr_identifier (see Section 5.1).

* AD_1 - bstr containing unprotected opaque auxiliary data

5.3.2.  Initiator Processing of Message 1

   The Initiator SHALL compose message_1 as follows:

*  The supported cipher suites and the order of preference MUST NOT
   be changed based on previous error messages.  However, the list
   SUITES_I sent to the Responder MAY be truncated such that cipher
   suites which are the least preferred are omitted.  The amount of
   truncation MAY be changed between sessions, e.g. based on previous
   error messages (see next bullet), but all cipher suites which are
   more preferred than the least preferred cipher suite in the list
   MUST be included in the list.

*  The Initiator MUST select its most preferred cipher suite,
   conditioned on what it can assume to be supported by the
   Responder.  If the Initiator previously received from the
   Responder an error message with error code 1 (see Section 6.3)
   indicating cipher suites supported by the Responder which also are
   supported by the Initiator, then the Initiator SHOULD select the
   most preferred cipher suite of those (note that error messages are
   not authenticated and may be forged).

*  Generate an ephemeral ECDH key pair as specified in Section 5 of
   [SP-800-56A] using the curve in the selected cipher suite and
   format it as a COSE_Key.  Let G_X be the 'x' parameter of the
   COSE_Key.

*  Choose a connection identifier C_I and store it for the length of
   the protocol.

*  Encode message_1 as a sequence of CBOR encoded data items as
   specified in Section 5.3.1

5.3.3.  Responder Processing of Message 1

   The Responder SHALL process message_1 as follows:

*  Decode message_1 (see Appendix A.1).

*  Verify that the selected cipher suite is supported and that no
   prior cipher suite in SUITES_I is supported.

*  Pass AD_1 to the security application.

   If any verification step fails, the Responder MUST send an EDHOC
   error message back, formatted as defined in Section 6, and the
   protocol MUST be discontinued.

5.4.  EDHOC Message 2

5.4.1.  Formatting of Message 2

   message_2 and data_2 SHALL be CBOR Sequences (see Appendix A.1) as
   defined below

   message_2 = (
     data_2,
     CIPHERTEXT_2 : bstr,
   )

   data_2 = (
     ? C_I : bstr_identifier,
     G_Y : bstr,
     C_R : bstr_identifier,
   )

   where:

   *  G_Y – the ephemeral public key of the Responder

   *  C_R – variable length connection identifier, encoded as a
      bstr_identifier (see Section 5.1).

5.4.2.  Responder Processing of Message 2

   The Responder SHALL compose message_2 as follows:

   *  If corr (METHOD_CORR mod 4) equals 1 or 3, C_I is omitted,
      otherwise C_I is not omitted.

   *  Generate an ephemeral ECDH key pair as specified in Section 5 of
      [SP-800-56A] using the curve in the selected cipher suite and
      format it as a COSE_Key.  Let G_Y be the 'x' parameter of the
      COSE_Key.

   *  Choose a connection identifier C_R and store it for the length of
      the protocol.

   *  Compute the transcript hash TH_2 = H(message_1, data_2) where H()
      is the hash function in the selected cipher suite.  The transcript
      hash TH_2 is a CBOR encoded bstr and the input to the hash
      function is a CBOR Sequence.

   *  Compute an inner COSE_Encrypt0 as defined in Section 5.3 of
      [I-D.ietf-cose-rfc8152bis-struct], with the EDHOC AEAD algorithm
      in the selected cipher suite, K_2m, IV_2m, and the following
      parameters:

- protected = << ID_CRED_R >>

    o  ID_CRED_R - identifier to facilitate retrieval of CRED_R,
       see Section 3.3.4

- external_aad = << TH_2, CRED_R, ? AD_2 >>

    o  CRED_R - bstr containing the credential of the Responder,
       see Section 3.3.4.

    o  AD_2 = bstr containing opaque unprotected auxiliary data

- plaintext = h''

COSE constructs the input to the AEAD [RFC5116] as follows:

- Key K = EDHOC-KDF( PRK_3e2m, TH_2, "K_2m", length )

- Nonce N = EDHOC-KDF( PRK_3e2m, TH_2, "IV_2m", length )

- Plaintext P = 0x (the empty string)

- Associated data A =

    [ "Encrypt0", << ID_CRED_R >>, << TH_2, CRED_R, ? AD_2 >> ]

MAC_2 is the 'ciphertext' of the inner COSE_Encrypt0.

*  If the Responder authenticates with a static Diffie-Hellman key
   (method equals 1 or 3), then Signature_or_MAC_2 is MAC_2.  If the
   Responder authenticates with a signature key (method equals 0 or
   2), then Signature_or_MAC_2 is the 'signature' of a COSE_Sign1
   object as defined in Section 4.4 of
   [I-D.ietf-cose-rfc8152bis-struct] using the signature algorithm in
   the selected cipher suite, the private authentication key of the
   Responder, and the following parameters:

- protected = << ID_CRED_R >>

- external_aad = << TH_2, CRED_R, ? AD_2 >>

- payload = MAC_2

COSE constructs the input to the Signature Algorithm as:

- The key is the private authentication key of the Responder.

- The message M to be signed =

```
      [ "Signature1", << ID_CRED_R >>, << TH_2, CRED_R, ? AD_2 >>,
      MAC_2 ]
```

* CIPHERTEXT_2 is encrypted by using the Expand function as a binary additive stream cipher.

  - plaintext = ( ID_CRED_R / bstr_identifier, Signature_or_MAC_2, ? AD_2 )

    o Note that if ID_CRED_R contains a single 'kid' parameter, i.e., ID_CRED_R = { 4 : kid_R }, only the byte string kid_R is conveyed in the plaintext encoded as a bstr_identifier, see Section 3.3.4 and Section 5.1.

  - CIPHERTEXT_2 = plaintext XOR KEYSTREAM_2

* Encode message_2 as a sequence of CBOR encoded data items as specified in Section 5.4.1.

## 5.4.3.  Initiator Processing of Message 2

The Initiator SHALL process message_2 as follows:

* Decode message_2 (see Appendix A.1).

* Retrieve the protocol state using the connection identifier C_I and/or other external information such as the CoAP Token and the 5-tuple.

* Decrypt CIPHERTEXT_2, see Section 5.4.2.

* Verify that the identity of the Responder is an allowed identity for this connection, see Section 3.3.

* Verify Signature_or_MAC_2 using the algorithm in the selected cipher suite.  The verification process depends on the method, see Section 5.4.2.

* Pass AD_2 to the security application.

If any verification step fails, the Initiator MUST send an EDHOC error message back, formatted as defined in Section 6, and the protocol MUST be discontinued.

## 5.5.  EDHOC Message 3

5.5.1.  Formatting of Message 3

   message_3 and data_3 SHALL be CBOR Sequences (see Appendix A.1) as
   defined below

   message_3 = (
     data_3,
     CIPHERTEXT_3 : bstr,
   )

   data_3 = (
     ? C_R : bstr_identifier,
   )

5.5.2.  Initiator Processing of Message 3

   The Initiator SHALL compose message_3 as follows:

   *  If corr (METHOD_CORR mod 4) equals 2 or 3, C_R is omitted,
      otherwise C_R is not omitted.

   *  Compute the transcript hash TH_3 = H(TH_2 , CIPHERTEXT_2, data_3)
      where H() is the hash function in the selected cipher suite.  The
      transcript hash TH_3 is a CBOR encoded bstr and the input to the
      hash function is a CBOR Sequence.

   *  Compute an inner COSE_Encrypt0 as defined in Section 5.3 of
      [I-D.ietf-cose-rfc8152bis-struct], with the EDHOC AEAD algorithm
      in the selected cipher suite, K_3m, IV_3m, and the following
      parameters:

      -  protected = << ID_CRED_I >>

         o  ID_CRED_I - identifier to facilitate retrieval of CRED_I,
            see Section 3.3.4

      -  external_aad = << TH_3, CRED_I, ? AD_3 >>

         o  CRED_I - bstr containing the credential of the Initiator,
            see Section 3.3.4.

         o  AD_3 = bstr containing opaque protected auxiliary data

      -  plaintext = h''

      COSE constructs the input to the AEAD [RFC5116] as follows:

      -  Key K = EDHOC-KDF( PRK_4x3m, TH_3, "K_3m", length )

      -  Nonce N = EDHOC-KDF( PRK_4x3m, TH_3, "IV_3m", length )

      -  Plaintext P = 0x (the empty string)

      -  Associated data A =

         [ "Encrypt0", << ID_CRED_I >>, << TH_3, CRED_I, ? AD_3 >> ]

      MAC_3 is the 'ciphertext' of the inner COSE_Encrypt0.

   *  If the Initiator authenticates with a static Diffie-Hellman key
      (method equals 2 or 3), then Signature_or_MAC_3 is MAC_3.  If the
      Initiator authenticates with a signature key (method equals 0 or
      1), then Signature_or_MAC_3 is the 'signature' of a COSE_Sign1
      object as defined in Section 4.4 of
      [I-D.ietf-cose-rfc8152bis-struct] using the signature algorithm in
      the selected cipher suite, the private authentication key of the
      Initiator, and the following parameters:

      -  protected = << ID_CRED_I >>

      -  external_aad = << TH_3, CRED_I, ? AD_3 >>

      -  payload = MAC_3

      COSE constructs the input to the Signature Algorithm as:

      -  The key is the private authentication key of the Initiator.

      -  The message M to be signed =

         [ "Signature1", << ID_CRED_I >>, << TH_3, CRED_I, ? AD_3 >>,
         MAC_3 ]

   *  Compute an outer COSE_Encrypt0 as defined in Section 5.3 of
      [I-D.ietf-cose-rfc8152bis-struct], with the EDHOC AEAD algorithm
      in the selected cipher suite, K_3ae, IV_3ae, and the following
      parameters.  The protected header SHALL be empty.

      -  external_aad = TH_3

      -  plaintext = ( ID_CRED_I / bstr_identifier, Signature_or_MAC_3,
         ? AD_3 )

         o  Note that if ID_CRED_I contains a single 'kid' parameter,
            i.e., ID_CRED_I = { 4 : kid_I }, only the byte string kid_I
            is conveyed in the plaintext encoded as a bstr_identifier,
            see Section 3.3.4 and Section 5.1.

COSE constructs the input to the AEAD [RFC5116] as follows:

- Key K = EDHOC-KDF( PRK_3e2m, TH_3, "K_3ae", length )

- Nonce N = EDHOC-KDF( PRK_3e2m, TH_3, "IV_3ae", length )

- Plaintext P = ( ID_CRED_I / bstr_identifier,
  Signature_or_MAC_3, ? AD_3 )

- Associated data A = [ "Encrypt0", h'', TH_3 ]

CIPHERTEXT_3 is the 'ciphertext' of the outer COSE_Encrypt0.

* Encode message_3 as a sequence of CBOR encoded data items as
  specified in Section 5.5.1.

Pass the connection identifiers (C_I, C_R) and the application
algorithms in the selected cipher suite to the application.  The
application can now derive application keys using the EDHOC-Exporter
interface.

After sending message_3, the Initiator is assured that no other party
than the Responder can compute the key PRK_4x3m (implicit key
authentication).  The Initiator does however not know that the
Responder has actually computed the key PRK_4x3m.  While the
Initiator can securely send protected application data, the Initiator
SHOULD NOT permanently store the keying material PRK_4x3m and TH_4
until the Initiator is assured that the Responder has actually
computed the key PRK_4x3m (explicit key confirmation).  Explicit key
confirmation is e.g. assured when the Initiator has verified an
OSCORE message or message_4 from the Responder.

5.5.3.  Responder Processing of Message 3

The Responder SHALL process message_3 as follows:

* Decode message_3 (see Appendix A.1).

* Retrieve the protocol state using the connection identifier C_R
  and/or other external information such as the CoAP Token and the
  5-tuple.

* Decrypt and verify the outer COSE_Encrypt0 as defined in
  Section 5.3 of [I-D.ietf-cose-rfc8152bis-struct], with the EDHOC
  AEAD algorithm in the selected cipher suite, K_3ae, and IV_3ae.

* Verify that the identity of the Initiator is an allowed identity
  for this connection, see Section 3.3.

 *  Verify Signature_or_MAC_3 using the algorithm in the selected
    cipher suite.  The verification process depends on the method, see
    Section 5.5.2.

 *  Pass AD_3, the connection identifiers (C_I, C_R), and the
    application algorithms in the selected cipher suite to the
    security application.  The application can now derive application
    keys using the EDHOC-Exporter interface.

If any verification step fails, the Responder MUST send an EDHOC
error message back, formatted as defined in Section 6, and the
protocol MUST be discontinued.

After verifying message_3, the Responder is assured that the
Initiator has calculated the key PRK_4x3m (explicit key confirmation)
and that no other party than the Responder can compute the key.  The
Responder can securely send protected application data and store the
keying material PRK_4x3m and TH_4.

6.  Error Handling

This section defines the format for error messages.

An EDHOC error message can be sent by either endpoint as a reply to
any non-error EDHOC message.  How errors at the EDHOC layer are
transported depends on lower layers, which need to enable error
messages to be sent and processed as intended.

All error messages in EDHOC are fatal.  After sending an error
message, the sender MUST discontinue the protocol.  The receiver
SHOULD treat an error message as an indication that the other party
likely has discontinued the protocol.  But as the error message is
not authenticated, a received error messages might also have been
sent by an attacker and the receiver MAY therefore try to continue
the protocol.

error SHALL be a CBOR Sequence (see Appendix A.1) as defined below

```
error = (
  ? C_x : bstr_identifier,
  ERR_CODE : int,
  ERR_INFO : any
)
```

Figure 5: EDHOC Error Message

where:

   *  C_x – (optional) variable length connection identifier, encoded as
      a bstr_identifier (see Section 5.1).  If error is sent by the
      Responder and corr (METHOD_CORR mod 4) equals 0 or 2 then C_x is
      set to C_I, else if error is sent by the Initiator and corr
      (METHOD_CORR mod 4) equals 0 or 1 then C_x is set to C_R, else C_x
      is omitted.

   *  ERR_CODE – error code encoded as an integer.

   *  ERR_INFO – error information.  Content and encoding depend on
      error code.

   The remainder of this section specifies the currently defined error
   codes, see Figure 6.  Error codes 1, 0 and -1 MUST be supported.
   Additional error codes and corresponding error information may be
   specified.

   +----------+---------------+-------------------------------------+
   | ERR_CODE | ERR_INFO Type | Description                         |
   +==========+===============+=====================================+
   |       -1 | TBD           | Success                             |
   +----------+---------------+-------------------------------------+
   |        0 | tstr          | Unspecified                         |
   +----------+---------------+-------------------------------------+
   |        1 | SUITES_R      | Wrong selected cipher suite         |
   +----------+---------------+-------------------------------------+

                Figure 6: Error Codes and Error Information

6.1.  Success

   TBD

6.2.  Unspecified

   Error code 0 is used for unspecified errors and contain a diagnostic
   message.

   For error messages with ERR_CODE == 0, ERR_INFO MUST be a text string
   containing a human-readable diagnostic message written in English.
   The diagnostic text message is mainly intended for software engineers
   that during debugging need to interpret it in the context of the
   EDHOC specification.  The diagnostic message SHOULD be provided to
   the calling application where it SHOULD be logged.

6.3.  Wrong Selected Cipher Suite

   Error code 1 MUST only be used in a response to message_1 in case the
   cipher suite selected by the Initiator is not supported by the
   Responder, or if the Responder supports a cipher suite more preferred
   by the Initiator than the selected cipher suite, see Section 5.3.3.

   ERR_INFO is of type SUITES_R:

   SUITES_R : [ supported : 2* suite ] / suite

   If the Responder does not support the selected cipher suite, then
   SUITES_R MUST include one or more supported cipher suites.  If the
   Responder does not support the selected cipher suite, but supports
   another cipher suite in SUITES_I, then SUITES_R MUST include the
   first supported cipher suite in SUITES_I.

6.3.1.  Cipher Suite Negotiation

   After receiving SUITES_R, the Initiator can determine which cipher
   suite to select for the next EDHOC run with the Responder.

   If the Initiator intends to contact the Responder in the future, the
   Initiator SHOULD remember which selected cipher suite to use until
   the next message_1 has been sent, otherwise the Initiator and
   Responder will likely run into an infinite loop.  After a successful
   run of EDHOC, the Initiator MAY remember the selected cipher suite to
   use in future EDHOC runs.  Note that if the Initiator or Responder is
   updated with new cipher suite policies, any cached information may be
   outdated.

6.3.2.  Examples

   Assume that the Initiator supports the five cipher suites 5, 6, 7, 8,
   and 9 in decreasing order of preference.  Figures 7 and 8 show
   examples of how the Initiator can truncate SUITES_I and how SUITES_R
   is used by Responders to give the Initiator information about the
   cipher suites that the Responder supports.

   In the first example (Figure 7), the Responder supports cipher suite
   6 but not the initially selected cipher suite 5.

```
Initiator                                                     Responder
|           METHOD_CORR, SUITES_I = 5, G_X, C_I, AD_1                 |
+-------------------------------------------------------------------->|
|                            message_1                               |
|                                                                    |
|               C_I, DIAG_MSG, SUITES_R = 6                          |
|<-------------------------------------------------------------------+
|                             error                                  |
|                                                                    |
|        METHOD_CORR, SUITES_I = [6, 5, 6], G_X, C_I, AD_1           |
+-------------------------------------------------------------------->|
|                            message_1                               |
```
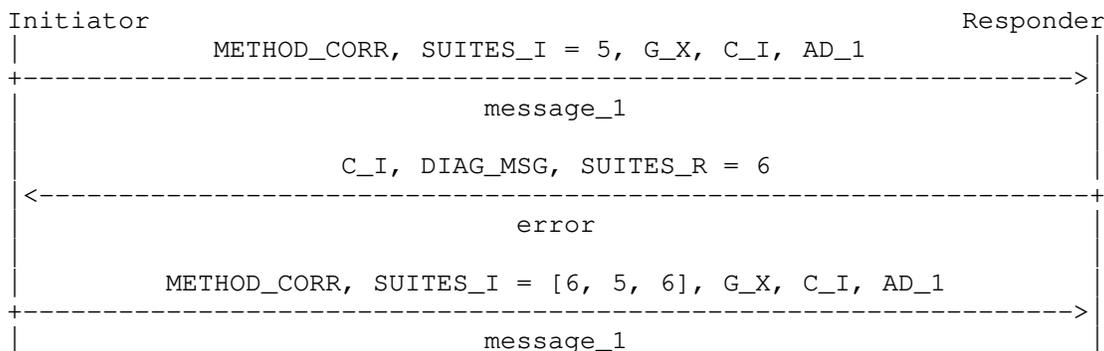
       Figure 7: Example of Responder supporting suite 6 but not suite 5.

In the second example (Figure 8), the Responder supports cipher
suites 8 and 9 but not the more preferred (by the Initiator) cipher
suites 5, 6 or 7.  To illustrate the negotiation mechanics we let the
Initiator first make a guess that the Responder supports suite 6 but
not suite 5.  Since the Responder supports neither 5 nor 6, it
responds with an error and SUITES_R, after which the Initiator
selects its most preferred supported suite.  The order of cipher
suites in SUITES_R does not matter.  (If the Responder had supported
suite 5, it would include it in SUITES_R of the response, and it
would in that case have become the selected suite in the second
message_1.)

```
Initiator                                                     Responder
|          METHOD_CORR, SUITES_I = [6, 5, 6], G_X, C_I, AD_1         |
+-------------------------------------------------------------------->|
|                            message_1                               |
|                                                                    |
|               C_I, DIAG_MSG, SUITES_R = [9, 8]                     |
|<-------------------------------------------------------------------+
|                             error                                  |
|                                                                    |
|      METHOD_CORR, SUITES_I = [8, 5, 6, 7, 8], G_X, C_I, AD_1       |
+-------------------------------------------------------------------->|
|                            message_1                               |
```
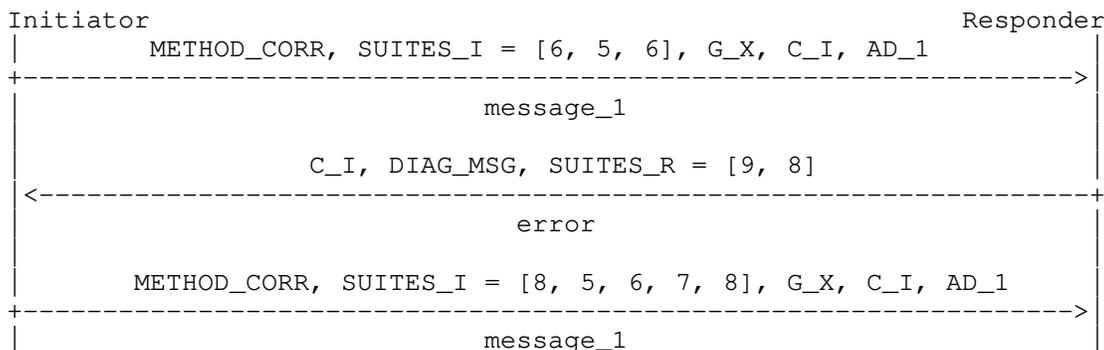
        Figure 8: Example of Responder supporting suites 8 and 9 but not
                               5, 6 or 7.

Note that the Initiator's list of supported cipher suites and order of preference is fixed (see Section 5.3.1 and Section 5.3.2). Furthermore, the Responder shall only accept message_1 if the selected cipher suite is the first cipher suite in SUITES_I that the Responder supports (see Section 5.3.3). Following this procedure ensures that the selected cipher suite is the most preferred (by the Initiator) cipher suite supported by both parties.

If the selected cipher suite is not the first cipher suite which the Responder supports in SUITES_I received in message_1, then Responder MUST discontinue the protocol, see Section 5.3.3. If SUITES_I in message_1 is manipulated then the integrity verification of message_2 containing the transcript hash TH_2 = H( message_1, data_2 ) will fail and the Initiator will discontinue the protocol.

7.  Transferring EDHOC and Deriving an OSCORE Context

7.1.  EDHOC Message 4

This section specifies message_4 which is OPTIONAL to support. Key confirmation is normally provided by sending an application message from the Responder to the Initiator protected with a key derived with the EDHOC-Exporter, e.g., using OSCORE (see Section 7.2.1). In deployments where no protected application message is sent from the Responder to the Initiator, the Responder MUST send message_4. Two examples of such deployments:

1.  When EDHOC is only used for authentication and no application data is sent.

2.  When application data is only sent from the Initiator to the Responder.

Further considerations are provided in Section 3.7.

7.1.1.  Formatting of Message 4

message_4 and data_4 SHALL be CBOR Sequences (see Appendix A.1) as defined below

```
message_4 = (
  data_4,
  MAC_4 : bstr,
)

data_4 = (
  ? C_I : bstr_identifier,
)
```

7.1.2.  Responder Processing of Message 4

   The Responder SHALL compose message_4 as follows:

   *  If corr (METHOD_CORR mod 4) equals 1 or 3, C_I is omitted,
      otherwise C_I is not omitted.

   *  Compute an inner COSE_Encrypt0 as defined in Section 5.3 of
      [I-D.ietf-cose-rfc8152bis-struct], with the EDHOC AEAD algorithm
      in the selected cipher suite, and the following parameters:

      -  protected = h''

      -  external_aad = << TH_4 >>

      -  plaintext = h''

      COSE constructs the input to the AEAD [RFC5116] as follows:

      -  Key K = EDHOC-Exporter( "EDHOC_message_4_Key", length )

      -  Nonce N = EDHOC-Exporter( "EDHOC_message_4_Nonce", length )

      -  Plaintext P = 0x (the empty string)

      -  Associated data A =

         [ "Encrypt0", h'', << TH_4 >> ]

      MAC_4 is the 'ciphertext' of the COSE_Encrypt0.

   *  Encode message_4 as a sequence of CBOR encoded data items as
      specified in Section 7.1.1.

7.1.3.  Initiator Processing of Message 4

   The Initiator SHALL process message_4 as follows:

   *  Decode message_4 (see Appendix A.1).

   *  Retrieve the protocol state using the connection identifier C_I
      and/or other external information such as the CoAP Token and the
      5-tuple.

   *  Verify MAC_4 as defined in Section 5.3 of
      [I-D.ietf-cose-rfc8152bis-struct], with the EDHOC AEAD algorithm
      in the selected cipher suite, and the parameters defined in
      Section 7.1.2.

If any verification step fails the Initiator MUST send an EDHOC error message back, formatted as defined in Section 6, and the protocol MUST be discontinued.

7.2.  Transferring EDHOC in CoAP

It is recommended to transport EDHOC as an exchange of CoAP [RFC7252] messages.  CoAP is a reliable transport that can preserve packet ordering and handle message duplication.  CoAP can also perform fragmentation and protect against denial of service attacks.  It is recommended to carry the EDHOC messages in Confirmable messages, especially if fragmentation is used.

By default, the CoAP client is the Initiator and the CoAP server is the Responder, but the roles SHOULD be chosen to protect the most sensitive identity, see Section 8.  By default, EDHOC is transferred in POST requests and 2.04 (Changed) responses to the Uri-Path: "/.well-known/edhoc", but an application may define its own path that can be discovered e.g. using resource directory [I-D.ietf-core-resource-directory].

By default, the message flow is as follows: EDHOC message_1 is sent in the payload of a POST request from the client to the server's resource for EDHOC.  EDHOC message_2 or the EDHOC error message is sent from the server to the client in the payload of a 2.04 (Changed) response.  EDHOC message_3 or the EDHOC error message is sent from the client to the server's resource in the payload of a POST request. If needed, an EDHOC error message is sent from the server to the client in the payload of a 2.04 (Changed) response.  Alternatively, if EDHOC message_4 is used, it is sent from the server to the client in the payload of a 2.04 (Changed) response analogously to message_2.

An example of a successful EDHOC exchange using CoAP is shown in Figure 9.  In this case the CoAP Token enables the Initiator to correlate message_1 and message_2 so the correlation parameter corr = 1.

```
          Client     Server
             |         |
            +--------->| Header: POST (Code=0.02)
             |   POST  | Uri-Path: "/.well-known/edhoc"
             |         | Content-Format: application/edhoc
             |         | Payload: EDHOC message_1
             |         |
             |<--------+ Header: 2.04 Changed
             |   2.04  | Content-Format: application/edhoc
             |         | Payload: EDHOC message_2
             |         |
            +--------->| Header: POST (Code=0.02)
             |   POST  | Uri-Path: "/.well-known/edhoc"
             |         | Content-Format: application/edhoc
             |         | Payload: EDHOC message_3
             |         |
             |<--------+ Header: 2.04 Changed
             |   2.04  |
             |         |
```
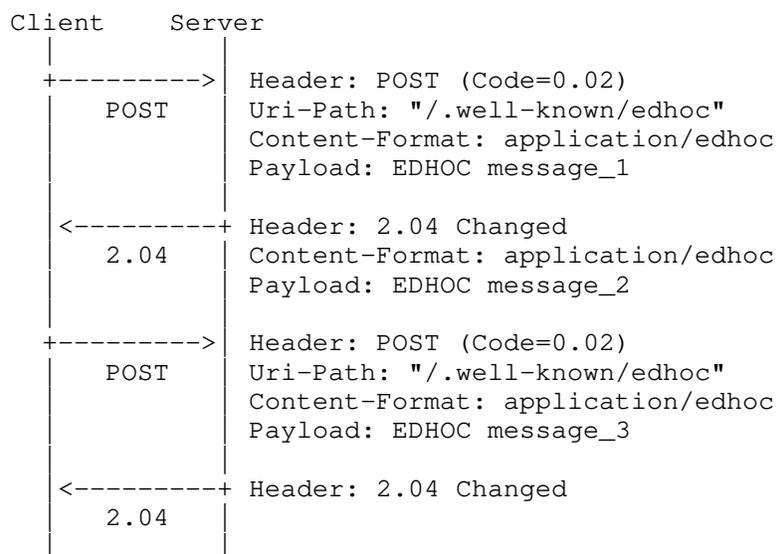
Figure 9: Transferring EDHOC in CoAP when the Initiator is CoAP
Client

   The exchange in Figure 9 protects the client identity against active
   attackers and the server identity against passive attackers.  An
   alternative exchange that protects the server identity against active
   attackers and the client identity against passive attackers is shown
   in Figure 10.  In this case the CoAP Token enables the Responder to
   correlate message_2 and message_3 so the correlation parameter corr =
   2.  If EDHOC message_4 is used, it is transported with CoAP in the
   payload of a POST request with a 2.04 (Changed) response.

```
              Client     Server
                |          |
                +--------->| Header: POST (Code=0.02)
                |   POST   | Uri-Path: "/.well-known/edhoc"
                |          |
                |<--------+ Header: 2.04 Changed
                |   2.04   | Content-Format: application/edhoc
                |          | Payload: EDHOC message_1
                |          |
                +--------->| Header: POST (Code=0.02)
                |   POST   | Uri-Path: "/.well-known/edhoc"
                |          | Content-Format: application/edhoc
                |          | Payload: EDHOC message_2
                |          |
                |<--------+ Header: 2.04 Changed
                |   2.04   | Content-Format: application/edhoc
                |          | Payload: EDHOC message_3
                |          |
```
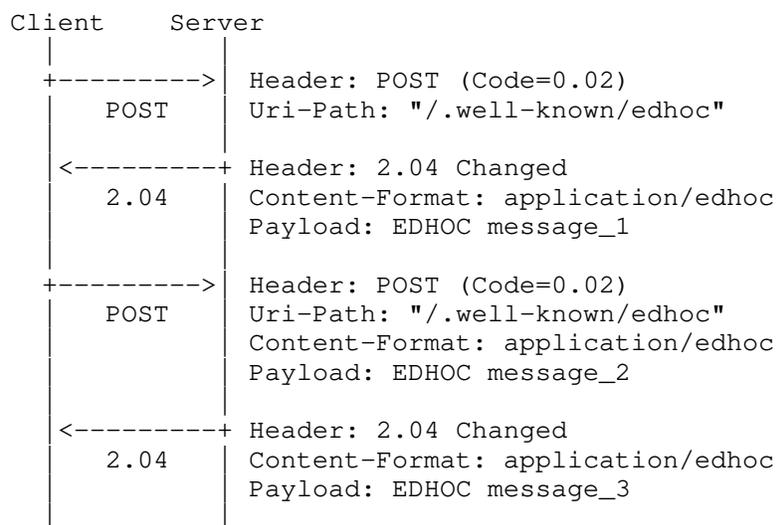
Figure 10: Transferring EDHOC in CoAP when the Initiator is CoAP
Server

To protect against denial-of-service attacks, the CoAP server MAY
respond to the first POST request with a 4.01 (Unauthorized)
containing an Echo option [I-D.ietf-core-echo-request-tag].  This
forces the initiator to demonstrate its reachability at its apparent
network address.  If message fragmentation is needed, the EDHOC
messages may be fragmented using the CoAP Block-Wise Transfer
mechanism [RFC7959].

7.2.1.  Deriving an OSCORE Context from EDHOC

When EDHOC is used to derive parameters for OSCORE [RFC8613], the
parties make sure that the EDHOC connection identifiers are unique,
i.e. C_R MUST NOT be equal to C_I.  The CoAP client and server MUST
be able to retrieve the OSCORE protocol state using its chosen
connection identifier and optionally other information such as the
5-tuple.  In case that the CoAP client is the Initiator and the CoAP
server is the Responder:

   *  The client's OSCORE Sender ID is C_R and the server's OSCORE
      Sender ID is C_I, as defined in this document

   *  The AEAD Algorithm and the hash algorithm are the application AEAD
      and hash algorithms in the selected cipher suite.

   * The Master Secret and Master Salt are derived as follows.  By
     default key_length is the key length (in bytes) of the application
     AEAD Algorithm and salt_length is 8 bytes.  The Initiator and
     Responder MAY agree out-of-band on a longer key_length than the
     default and a different salt_length.

   Master Secret = EDHOC-Exporter( "OSCORE Master Secret", key_length )
   Master Salt   = EDHOC-Exporter( "OSCORE Master Salt", salt_length )

## 7.2.2.  Error Messages with CoAP Transport

   EDHOC does not restrict how error messages are transported with CoAP,
   as long as the appropriate error message can to be transported in
   response to a message that failed (see Section 6).  In case of
   combining EDHOC and OSCORE as specified in
   [I-D.ietf-core-oscore-edhoc], an error message following a combined
   EDHOC message_3/OSCORE request MUST be sent with a CoAP error code
   and SHALL contain the ERR_INFO as payload (see Section 6).

## 8.  Security Considerations

## 8.1.  Security Properties

   EDHOC inherits its security properties from the theoretical SIGMA-I
   protocol [SIGMA].  Using the terminology from [SIGMA], EDHOC provides
   perfect forward secrecy, mutual authentication with aliveness,
   consistency, and peer awareness.  As described in [SIGMA], peer
   awareness is provided to the Responder, but not to the Initiator.

   EDHOC protects the credential identifier of the Initiator against
   active attacks and the credential identifier of the Responder against
   passive attacks.  The roles should be assigned to protect the most
   sensitive identity/identifier, typically that which is not possible
   to infer from routing information in the lower layers.

   Compared to [SIGMA], EDHOC adds an explicit method type and expands
   the message authentication coverage to additional elements such as
   algorithms, auxiliary data, and previous messages.  This protects
   against an attacker replaying messages or injecting messages from
   another session.

   EDHOC also adds negotiation of connection identifiers and downgrade
   protected negotiation of cryptographic parameters, i.e. an attacker
   cannot affect the negotiated parameters.  A single session of EDHOC
   does not include negotiation of cipher suites, but it enables the
   Responder to verify that the selected cipher suite is the most
   preferred cipher suite by the Initiator which is supported by both
   the Initiator and the Responder.

As required by [RFC7258], IETF protocols need to mitigate pervasive monitoring when possible.  One way to mitigate pervasive monitoring is to use a key exchange that provides perfect forward secrecy.  EDHOC therefore only supports methods with perfect forward secrecy.  To limit the effect of breaches, it is important to limit the use of symmetrical group keys for bootstrapping.  EDHOC therefore strives to make the additional cost of using raw public keys and self-signed certificates as small as possible.  Raw public keys and self-signed certificates are not a replacement for a public key infrastructure, but SHOULD be used instead of symmetrical group keys for bootstrapping.

Compromise of the long-term keys (private signature or static DH keys) does not compromise the security of completed EDHOC exchanges.  Compromising the private authentication keys of one party lets an active attacker impersonate that compromised party in EDHOC exchanges with other parties, but does not let the attacker impersonate other parties in EDHOC exchanges with the compromised party.  Compromise of the long-term keys does not enable a passive attacker to compromise future session keys.  Compromise of the HDKF input parameters (ECDH shared secret) leads to compromise of all session keys derived from that compromised shared secret.  Compromise of one session key does not compromise other session keys.  Compromise of PRK_4x3m leads to compromise of all exported keying material derived after the last invocation of the EDHOC-KeyUpdate function.

EDHOC provides a minimum of 64-bit security against online brute force attacks and a minimum of 128-bit security against offline brute force attacks.  This is in line with IPsec, TLS, and COSE.  To break 64-bit security against online brute force an attacker would on average have to send 4.3 billion messages per second for 68 years, which is infeasible in constrained IoT radio technologies.

After sending message_3, the Initiator is assured that no other party than the Responder can compute the key PRK_4x3m (implicit key authentication).  The Initiator does however not know that the Responder has actually computed the key PRK_4x3m.  While the Initiator can securely send protected application data, the Initiator SHOULD NOT permanently store the keying material PRK_4x3m and TH_4 until the Initiator is assured that the Responder has actually computed the key PRK_4x3m (explicit key confirmation).  Explicit key confirmation is e.g. assured when the Initiator has verified an OSCORE message or message_4 from the Responder.  After verifying message_3, the Responder is assured that the Initiator has calculated the key PRK_4x3m (explicit key confirmation) and that no other party than the Responder can compute the key.  The Responder can securely send protected application data and store the keying material PRK_4x3m and TH_4.

Key compromise impersonation (KCI): In EDHOC authenticated with
signature keys, EDHOC provides KCI protection against an attacker
having access to the long term key or the ephemeral secret key.  With
static Diffie-Hellman key authentication, KCI protection would be
provided against an attacker having access to the long-term Diffie-
Hellman key, but not to an attacker having access to the ephemeral
secret key.  Note that the term KCI has typically been used for
compromise of long-term keys, and that an attacker with access to the
ephemeral secret key can only attack that specific protocol run.

Repudiation: In EDHOC authenticated with signature keys, the
Initiator could theoretically prove that the Responder performed a
run of the protocol by presenting the private ephemeral key, and vice
versa.  Note that storing the private ephemeral keys violates the
protocol requirements.  With static Diffie-Hellman key
authentication, both parties can always deny having participated in
the protocol.

Two earlier versions of EDHOC have been formally analyzed [Norrman20]
[Bruni18] and the specification has been updated based on the
analysis.

8.2.  Cryptographic Considerations

The security of the SIGMA protocol requires the MAC to be bound to
the identity of the signer.  Hence the message authenticating
functionality of the authenticated encryption in EDHOC is critical:
authenticated encryption MUST NOT be replaced by plain encryption
only, even if authentication is provided at another level or through
a different mechanism.  EDHOC implements SIGMA-I using a MAC-then-
Sign approach.

To reduce message overhead EDHOC does not use explicit nonces and
instead rely on the ephemeral public keys to provide randomness to
each session.  A good amount of randomness is important for the key
generation, to provide liveness, and to protect against interleaving
attacks.  For this reason, the ephemeral keys MUST NOT be reused, and
both parties SHALL generate fresh random ephemeral key pairs.

As discussed the [SIGMA], the encryption of message_2 does only need to protect against passive attacker as active attackers can always get the Responders identity by sending their own message_1.  EDHOC uses the Expand function (typically HKDF-Expand) as a binary additive stream cipher.  HKDF-Expand provides better confidentiality than AES-CTR but is not often used as it is slow on long messages, and most applications require both IND-CCA confidentiality as well as integrity protection.  For the encryption of message_2, any speed difference is negligible, IND-CCA does not increase security, and integrity is provided by the inner MAC (and signature depending on method).

The data rates in many IoT deployments are very limited.  Given that the application keys are protected as well as the long-term authentication keys they can often be used for years or even decades before the cryptographic limits are reached.  If the application keys established through EDHOC need to be renewed, the communicating parties can derive application keys with other labels or run EDHOC again.

8.3.  Cipher Suites and Cryptographic Algorithms

For many constrained IoT devices it is problematic to support more than one cipher suite.  Existing devices can be expected to support either ECDSA or EdDSA.  To enable as much interoperability as we can reasonably achieve, less constrained devices SHOULD implement both cipher suite 0 (AES-CCM-16-64-128, SHA-256, X25519, EdDSA, Ed25519, AES-CCM-16-64-128, SHA-256) and cipher suite 2 (AES-CCM-16-64-128, SHA-256, P-256, ES256, P-256, AES-CCM-16-64-128, SHA-256). Constrained endpoints SHOULD implement cipher suite 0 or cipher suite 2.  Implementations only need to implement the algorithms needed for their supported methods.

When using private cipher suite or registering new cipher suites, the choice of key length used in the different algorithms needs to be harmonized, so that a sufficient security level is maintained for certificates, EDHOC, and the protection of application data.  The Initiator and the Responder should enforce a minimum security level.

The hash algorithms SHA-1 and SHA-256/64 (256-bit Hash truncated to 64-bits) SHALL NOT be supported for use in EDHOC except for certificate identification with x5u and c5u.  Note that secp256k1 is only defined for use with ECDSA and not for ECDH.

8.4.  Unprotected Data

   The Initiator and the Responder must make sure that unprotected data
   and metadata do not reveal any sensitive information.  This also
   applies for encrypted data sent to an unauthenticated party.  In
   particular, it applies to AD_1, ID_CRED_R, AD_2, and ERR_MSG.  Using
   the same AD_1 in several EDHOC sessions allows passive eavesdroppers
   to correlate the different sessions.  Another consideration is that
   the list of supported cipher suites may potentially be used to
   identify the application.

   The Initiator and the Responder must also make sure that
   unauthenticated data does not trigger any harmful actions.  In
   particular, this applies to AD_1 and ERR_MSG.

8.5.  Denial-of-Service

   EDHOC itself does not provide countermeasures against Denial-of-
   Service attacks.  By sending a number of new or replayed message_1 an
   attacker may cause the Responder to allocate state, perform
   cryptographic operations, and amplify messages.  To mitigate such
   attacks, an implementation SHOULD rely on lower layer mechanisms such
   as the Echo option in CoAP [I-D.ietf-core-echo-request-tag] that
   forces the initiator to demonstrate reachability at its apparent
   network address.

8.6.  Implementation Considerations

   The availability of a secure random number generator is essential for
   the security of EDHOC.  If no true random number generator is
   available, a truly random seed MUST be provided from an external
   source and used with a cryptographically secure pseudorandom number
   generator.  As each pseudorandom number must only be used once, an
   implementation need to get a new truly random seed after reboot, or
   continuously store state in nonvolatile memory, see ([RFC8613],
   Appendix B.1.1) for issues and solution approaches for writing to
   nonvolatile memory.  Intentionally or unintentionally weak or
   predictable pseudorandom number generators can be abused or exploited
   for malicious purposes.  [RFC8937] describes a way for security
   protocol implementations to augment their (pseudo)random number
   generators using a long-term private keys and a deterministic
   signature function.  This improves randomness from broken or
   otherwise subverted random number generators.  The same idea can be
   used with other secrets and functions such as a Diffie-Hellman
   function or a symmetric secret and a PRF like HMAC or KMAC.  It is
   RECOMMENDED to not trust a single source of randomness and to not put
   unaugmented random numbers on the wire.

If ECDSA is supported, "deterministic ECDSA" as specified in
[RFC6979] MAY be used.  Pure deterministic elliptic-curve signatures
such as deterministic ECDSA and EdDSA have gained popularity over
randomized ECDSA as their security do not depend on a source of high-
quality randomness.  Recent research has however found that
implementations of these signature algorithms may be vulnerable to
certain side-channel and fault injection attacks due to their
determinism.  See e.g.  Section 1 of
[I-D.mattsson-cfrg-det-sigs-with-noise] for a list of attack papers.
As suggested in Section 6.1.2 of [I-D.ietf-cose-rfc8152bis-algs] this
can be addressed by combining randomness and determinism.

The referenced processing instructions in [SP-800-56A] must be
complied with, including deleting the intermediate computed values
along with any ephemeral ECDH secrets after the key derivation is
completed.  The ECDH shared secrets, keys, and IVs MUST be secret.
Implementations should provide countermeasures to side-channel
attacks such as timing attacks.  Depending on the selected curve, the
parties should perform various validations of each other's public
keys, see e.g.  Section 5 of [SP-800-56A].

The Initiator and the Responder are responsible for verifying the
integrity of certificates.  The selection of trusted CAs should be
done very carefully and certificate revocation should be supported.
The private authentication keys MUST be kept secret.

The Initiator and the Responder are allowed to select the connection
identifiers C_I and C_R, respectively, for the other party to use in
the ongoing EDHOC protocol as well as in a subsequent application
protocol (e.g.  OSCORE [RFC8613]).  The choice of connection
identifier is not security critical in EDHOC but intended to simplify
the retrieval of the right security context in combination with using
short identifiers.  If the wrong connection identifier of the other
party is used in a protocol message it will result in the receiving
party not being able to retrieve a security context (which will
terminate the protocol) or retrieve the wrong security context (which
also terminates the protocol as the message cannot be verified).

The Responder MUST finish the verification step of message_3 before
passing AD_3 to the application.

If two nodes unintentionally initiate two simultaneous EDHOC message
exchanges with each other even if they only want to complete a single
EDHOC message exchange, they MAY terminate the exchange with the
lexicographically smallest G_X.  If the two G_X values are equal, the
received message_1 MUST be discarded to mitigate reflection attacks.
Note that in the case of two simultaneous EDHOC exchanges where the
nodes only complete one and where the nodes have different preferred
cipher suites, an attacker can affect which of the two nodes'
preferred cipher suites will be used by blocking the other exchange.

If supported by the device, it is RECOMMENDED that at least the long-
term private keys is stored in a Trusted Execution Environment (TEE)
and that sensitive operations using these keys are performed inside
the TEE.  To achieve even higher security it is RECOMMENDED that
additional operations such as ephemeral key generation, all
computations of shared secrets, and storage of the PRK keys can be
done inside the TEE.  The TEE can also be used to protect the EDHOC
and application protocol (e.g.  OSCORE) implementation using some
form of "secure boot", memory protection etc.  The use of a TEE
enforces that code within that environment cannot be tampered with,
and that any data used by such code cannot be read or tampered with
by code outside that environment.

9.  IANA Considerations

9.1.  EDHOC Cipher Suites Registry

IANA has created a new registry titled "EDHOC Cipher Suites" under
the new heading "EDHOC".  The registration procedure is "Expert
Review".  The columns of the registry are Value, Array, Description,
and Reference, where Value is an integer and the other columns are
text strings.  The initial contents of the registry are:

Value: -24
Algorithms: N/A
Desc: Reserved for Private Use
Reference: [[this document]]

Value: -23
Algorithms: N/A
Desc: Reserved for Private Use
Reference: [[this document]]

Value: -22
Algorithms: N/A
Desc: Reserved for Private Use
Reference: [[this document]]

```
   Value: -21
   Algorithms: N/A
   Desc: Reserved for Private Use
   Reference: [[this document]]

   Value: 0
   Array: 10, 5, 4, -8, 6, 10, 5
   Desc: AES-CCM-16-64-128, SHA-256, X25519, EdDSA, Ed25519,
         AES-CCM-16-64-128, SHA-256
   Reference: [[this document]]

   Value: 1
   Array: 30, 5, 4, -8, 6, 10, 5
   Desc: AES-CCM-16-128-128, SHA-256, X25519, EdDSA, Ed25519,
         AES-CCM-16-64-128, SHA-256
   Reference: [[this document]]

   Value: 2
   Array: 10, 5, 1, -7, 1, 10, 5
   Desc: AES-CCM-16-64-128, SHA-256, P-256, ES256, P-256,
         AES-CCM-16-64-128, SHA-256
   Reference: [[this document]]

   Value: 3
   Array: 30, 5, 1, -7, 1, 10, 5
   Desc: AES-CCM-16-128-128, SHA-256, P-256, ES256, P-256,
         AES-CCM-16-64-128, SHA-256
   Reference: [[this document]]

   Value: 4
   Array: 1, -16, 4, -7, 1, 1, -16
   Desc: A128GCM, SHA-256, X25519, ES256, P-256,
         A128GCM, SHA-256
   Reference: [[this document]]

   Value: 5
   Array: 3, -43, 2, -35, 2, 3, -43
   Desc: A256GCM, SHA-384, P-384, ES384, P-384,
         A256GCM, SHA-384
   Reference: [[this document]]
```

9.2.  EDHOC Method Type Registry

   IANA has created a new registry entitled "EDHOC Method Type" under
   the new heading "EDHOC".  The registration procedure is "Expert
   Review".  The columns of the registry are Value, Description, and
   Reference, where Value is an integer and the other columns are text
   strings.  The initial contents of the registry is shown in Figure 4.

9.3.  EDHOC Error Codes Registry

   IANA has created a new registry entitled "EDHOC Error Codes" under
   the new heading "EDHOC".  The registration procedure is
   "Specification Required".  The columns of the registry are ERR_CODE,
   ERR_INFO Type and Description, where ERR_CODE is an integer, ERR_INFO
   is a CDDL defined type, and Description is a text string.  The
   initial contents of the registry is shown in Figure 6.

9.4.  The Well-Known URI Registry

   IANA has added the well-known URI 'edhoc' to the Well-Known URIs
   registry.

   *  URI suffix: edhoc

   *  Change controller: IETF

   *  Specification document(s): [[this document]]

   *  Related information: None

9.5.  Media Types Registry

   IANA has added the media type 'application/edhoc' to the Media Types
   registry.

   *  Type name: application

   *  Subtype name: edhoc

   *  Required parameters: N/A

   *  Optional parameters: N/A

   *  Encoding considerations: binary

   *  Security considerations: See Section 7 of this document.

   *  Interoperability considerations: N/A

   *  Published specification: [[this document]] (this document)

   *  Applications that use this media type: To be identified

   *  Fragment identifier considerations: N/A

   *  Additional information:

          - Magic number(s): N/A

          - File extension(s): N/A

          - Macintosh file type code(s): N/A

     * Person & email address to contact for further information: See
       "Authors' Addresses" section.

     * Intended usage: COMMON

     * Restrictions on usage: N/A

     * Author: See "Authors' Addresses" section.

     * Change Controller: IESG

9.6.  CoAP Content-Formats Registry

     IANA has added the media type 'application/edhoc' to the CoAP
     Content-Formats registry.

     * Media Type: application/edhoc

     * Encoding:

     * ID: TBD42

     * Reference: [[this document]]

9.7.  Expert Review Instructions

     The IANA Registries established in this document is defined as
     "Expert Review".  This section gives some general guidelines for what
     the experts should be looking for, but they are being designated as
     experts for a reason so they should be given substantial latitude.

     Expert reviewers should take into consideration the following points:

     * Clarity and correctness of registrations.  Experts are expected to
       check the clarity of purpose and use of the requested entries.
       Expert needs to make sure the values of algorithms are taken from
       the right registry, when that's required.  Expert should consider
       requesting an opinion on the correctness of registered parameters
       from relevant IETF working groups.  Encodings that do not meet
       these objective of clarity and completeness should not be
       registered.

   *  Experts should take into account the expected usage of fields when
      approving point assignment.  The length of the encoded value
      should be weighed against how many code points of that length are
      left, the size of device it will be used on, and the number of
      code points left that encode to that size.

   *  Specifications are recommended.  When specifications are not
      provided, the description provided needs to have sufficient
      information to verify the points above.

10.  References

10.1.  Normative References

   [RFC2119]  Bradner, S., "Key words for use in RFCs to Indicate
              Requirement Levels", BCP 14, RFC 2119,
              DOI 10.17487/RFC2119, March 1997,
              <https://www.rfc-editor.org/info/rfc2119>.

   [RFC5116]  McGrew, D., "An Interface and Algorithms for Authenticated
              Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008,
              <https://www.rfc-editor.org/info/rfc5116>.

   [RFC5869]  Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand
              Key Derivation Function (HKDF)", RFC 5869,
              DOI 10.17487/RFC5869, May 2010,
              <https://www.rfc-editor.org/info/rfc5869>.

   [RFC6090]  McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic
              Curve Cryptography Algorithms", RFC 6090,
              DOI 10.17487/RFC6090, February 2011,
              <https://www.rfc-editor.org/info/rfc6090>.

   [RFC6979]  Pornin, T., "Deterministic Usage of the Digital Signature
              Algorithm (DSA) and Elliptic Curve Digital Signature
              Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August
              2013, <https://www.rfc-editor.org/info/rfc6979>.

   [RFC7252]  Shelby, Z., Hartke, K., and C. Bormann, "The Constrained
              Application Protocol (CoAP)", RFC 7252,
              DOI 10.17487/RFC7252, June 2014,
              <https://www.rfc-editor.org/info/rfc7252>.

   [RFC7748]  Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves
              for Security", RFC 7748, DOI 10.17487/RFC7748, January
              2016, <https://www.rfc-editor.org/info/rfc7748>.

   [RFC8949]  Bormann, C. and P. Hoffman, "Concise Binary Object
              Representation (CBOR)", STD 94, RFC 8949,
              DOI 10.17487/RFC8949, December 2020,
              <https://www.rfc-editor.org/info/rfc8949>.

   [RFC7959]  Bormann, C. and Z. Shelby, Ed., "Block-Wise Transfers in
              the Constrained Application Protocol (CoAP)", RFC 7959,
              DOI 10.17487/RFC7959, August 2016,
              <https://www.rfc-editor.org/info/rfc7959>.

   [RFC8174]  Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC
              2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174,
              May 2017, <https://www.rfc-editor.org/info/rfc8174>.

   [RFC8376]  Farrell, S., Ed., "Low-Power Wide Area Network (LPWAN)
              Overview", RFC 8376, DOI 10.17487/RFC8376, May 2018,
              <https://www.rfc-editor.org/info/rfc8376>.

   [RFC8610]  Birkholz, H., Vigano, C., and C. Bormann, "Concise Data
              Definition Language (CDDL): A Notational Convention to
              Express Concise Binary Object Representation (CBOR) and
              JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610,
              June 2019, <https://www.rfc-editor.org/info/rfc8610>.

   [RFC8613]  Selander, G., Mattsson, J., Palombini, F., and L. Seitz,
              "Object Security for Constrained RESTful Environments
              (OSCORE)", RFC 8613, DOI 10.17487/RFC8613, July 2019,
              <https://www.rfc-editor.org/info/rfc8613>.

   [RFC8724]  Minaburo, A., Toutain, L., Gomez, C., Barthel, D., and JC.
              Zúñiga, "SCHC: Generic Framework for Static Context Header
              Compression and Fragmentation", RFC 8724,
              DOI 10.17487/RFC8724, April 2020,
              <https://www.rfc-editor.org/info/rfc8724>.

   [RFC8742]  Bormann, C., "Concise Binary Object Representation (CBOR)
              Sequences", RFC 8742, DOI 10.17487/RFC8742, February 2020,
              <https://www.rfc-editor.org/info/rfc8742>.

   [I-D.ietf-cose-rfc8152bis-struct]
              Schaad, J., "CBOR Object Signing and Encryption (COSE):
              Structures and Process", Work in Progress, Internet-Draft,
              draft-ietf-cose-rfc8152bis-struct-14, 24 September 2020,
              <http://www.ietf.org/internet-drafts/draft-ietf-cose-
              rfc8152bis-struct-14.txt>.

[I-D.ietf-cose-rfc8152bis-algs]
          Schaad, J., "CBOR Object Signing and Encryption (COSE):
          Initial Algorithms", Work in Progress, Internet-Draft,
          draft-ietf-cose-rfc8152bis-algs-12, 24 September 2020,
          <http://www.ietf.org/internet-drafts/draft-ietf-cose-
          rfc8152bis-algs-12.txt>.

[I-D.ietf-cose-x509]
          Schaad, J., "CBOR Object Signing and Encryption (COSE):
          Header parameters for carrying and referencing X.509
          certificates", Work in Progress, Internet-Draft, draft-
          ietf-cose-x509-08, 14 December 2020, <http://www.ietf.org/
          internet-drafts/draft-ietf-cose-x509-08.txt>.

[I-D.ietf-core-echo-request-tag]
          Amsuess, C., Mattsson, J., and G. Selander, "CoAP: Echo,
          Request-Tag, and Token Processing", Work in Progress,
          Internet-Draft, draft-ietf-core-echo-request-tag-11, 2
          November 2020, <http://www.ietf.org/internet-drafts/draft-
          ietf-core-echo-request-tag-11.txt>.

[I-D.ietf-lake-reqs]
          Vucinic, M., Selander, G., Mattsson, J., and D. Garcia-
          Carillo, "Requirements for a Lightweight AKE for OSCORE",
          Work in Progress, Internet-Draft, draft-ietf-lake-reqs-04,
          8 June 2020, <http://www.ietf.org/internet-drafts/draft-
          ietf-lake-reqs-04.txt>.

10.2.  Informative References

[RFC7228]  Bormann, C., Ersue, M., and A. Keranen, "Terminology for
          Constrained-Node Networks", RFC 7228,
          DOI 10.17487/RFC7228, May 2014,
          <https://www.rfc-editor.org/info/rfc7228>.

[RFC7258]  Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an
          Attack", BCP 188, RFC 7258, DOI 10.17487/RFC7258, May
          2014, <https://www.rfc-editor.org/info/rfc7258>.

[RFC7296]  Kaufman, C., Hoffman, P., Nir, Y., Eronen, P., and T.
          Kivinen, "Internet Key Exchange Protocol Version 2
          (IKEv2)", STD 79, RFC 7296, DOI 10.17487/RFC7296, October
          2014, <https://www.rfc-editor.org/info/rfc7296>.

[RFC8446]  Rescorla, E., "The Transport Layer Security (TLS) Protocol
          Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018,
          <https://www.rfc-editor.org/info/rfc8446>.

   [RFC8937]  Cremers, C., Garratt, L., Smyshlyaev, S., Sullivan, N.,
              and C. Wood, "Randomness Improvements for Security
              Protocols", RFC 8937, DOI 10.17487/RFC8937, October 2020,
              <https://www.rfc-editor.org/info/rfc8937>.

   [I-D.ietf-core-resource-directory]
              Amsuess, C., Shelby, Z., Koster, M., Bormann, C., and P.
              Stok, "CoRE Resource Directory", Work in Progress,
              Internet-Draft, draft-ietf-core-resource-directory-26, 2
              November 2020, <http://www.ietf.org/internet-drafts/draft-
              ietf-core-resource-directory-26.txt>.

   [I-D.ietf-lwig-security-protocol-comparison]
              Mattsson, J., Palombini, F., and M. Vucinic, "Comparison
              of CoAP Security Protocols", Work in Progress, Internet-
              Draft, draft-ietf-lwig-security-protocol-comparison-05, 2
              November 2020, <http://www.ietf.org/internet-drafts/draft-
              ietf-lwig-security-protocol-comparison-05.txt>.

   [I-D.ietf-tls-dtls13]
              Rescorla, E., Tschofenig, H., and N. Modadugu, "The
              Datagram Transport Layer Security (DTLS) Protocol Version
              1.3", Work in Progress, Internet-Draft, draft-ietf-tls-
              dtls13-40, 20 January 2021, <http://www.ietf.org/internet-
              drafts/draft-ietf-tls-dtls13-40.txt>.

   [I-D.selander-ace-ake-authz]
              Selander, G., Mattsson, J., Vucinic, M., Richardson, M.,
              and A. Schellenbaum, "Lightweight Authorization for
              Authenticated Key Exchange.", Work in Progress, Internet-
              Draft, draft-selander-ace-ake-authz-02, 2 November 2020,
              <http://www.ietf.org/internet-drafts/draft-selander-ace-
              ake-authz-02.txt>.

   [I-D.ietf-core-oscore-edhoc]
              Palombini, F., Tiloca, M., Hoeglund, R., Hristozov, S.,
              and G. Selander, "Combining EDHOC and OSCORE", Work in
              Progress, Internet-Draft, draft-ietf-core-oscore-edhoc-00,
              1 April 2021, <https://www.ietf.org/internet-drafts/draft-
              ietf-core-oscore-edhoc-00.txt>.

   [I-D.mattsson-cose-cbor-cert-compress]
              Raza, S., Hoglund, J., Selander, G., Mattsson, J., and M.
              Furuhed, "CBOR Encoding of X.509 Certificates (CBOR
              Certificates)", Work in Progress, Internet-Draft, draft-
              mattsson-cose-cbor-cert-compress-06, 19 January 2021,
              <http://www.ietf.org/internet-drafts/draft-mattsson-cose-
              cbor-cert-compress-06.txt>.

[I-D.mattsson-cfrg-det-sigs-with-noise]
          Mattsson, J., Thormarker, E., and S. Ruohomaa,
          "Deterministic ECDSA and EdDSA Signatures with Additional
          Randomness", Work in Progress, Internet-Draft, draft-
          mattsson-cfrg-det-sigs-with-noise-02, 11 March 2020,
          <http://www.ietf.org/internet-drafts/draft-mattsson-cfrg-
          det-sigs-with-noise-02.txt>.

[SP-800-56A]
          Barker, E., Chen, L., Roginsky, A., Vassilev, A., and R.
          Davis, "Recommendation for Pair-Wise Key-Establishment
          Schemes Using Discrete Logarithm Cryptography",
          NIST Special Publication 800-56A Revision 3, April 2018,
          <https://doi.org/10.6028/NIST.SP.800-56Ar3>.

[SIGMA]   Krawczyk, H., "SIGMA - The 'SIGn-and-MAc' Approach to
          Authenticated Diffie-Hellman and Its Use in the IKE-
          Protocols (Long version)", June 2003,
          <http://webee.technion.ac.il/~hugo/sigma-pdf.pdf>.

[CNSA]    (Placeholder), ., "Commercial National Security Algorithm
          Suite", August 2015,
          <https://apps.nsa.gov/iaarchive/programs/iad-initiatives/
          cnsa-suite.cfm>.

[Norrman20]
          Norrman, K., Sundararajan, V., and A. Bruni, "Formal
          Analysis of EDHOC Key Establishment for Constrained IoT
          Devices", September 2020,
          <https://arxiv.org/abs/2007.11427>.

[Bruni18] Bruni, A., Sahl Jørgensen, T., Grønbech Petersen, T., and
          C. Schürmann, "Formal Verification of Ephemeral Diffie-
          Hellman Over COSE (EDHOC)", November 2018,
          <https://www.springerprofessional.de/en/formal-
          verification-of-ephemeral-diffie-hellman-over-cose-
          edhoc/16284348>.

[CborMe]  Bormann, C., "CBOR Playground", May 2018,
          <http://cbor.me/>.

Appendix A.  Use of CBOR, CDDL and COSE in EDHOC

   This Appendix is intended to simplify for implementors not familiar
   with CBOR [RFC8949], CDDL [RFC8610], COSE
   [I-D.ietf-cose-rfc8152bis-struct], and HKDF [RFC5869].

A.1.  CBOR and CDDL

   The Concise Binary Object Representation (CBOR) [RFC8949] is a data
   format designed for small code size and small message size.  CBOR
   builds on the JSON data model but extends it by e.g. encoding binary
   data directly without base64 conversion.  In addition to the binary
   CBOR encoding, CBOR also has a diagnostic notation that is readable
   and editable by humans.  The Concise Data Definition Language (CDDL)
   [RFC8610] provides a way to express structures for protocol messages
   and APIs that use CBOR.  [RFC8610] also extends the diagnostic
   notation.

   CBOR data items are encoded to or decoded from byte strings using a
   type-length-value encoding scheme, where the three highest order bits
   of the initial byte contain information about the major type.  CBOR
   supports several different types of data items, in addition to
   integers (int, uint), simple values (e.g. null), byte strings (bstr),
   and text strings (tstr), CBOR also supports arrays [] of data items,
   maps {} of pairs of data items, and sequences [RFC8742] of data
   items.  Some examples are given below.  For a complete specification
   and more examples, see [RFC8949] and [RFC8610].  We recommend
   implementors to get used to CBOR by using the CBOR playground
   [CborMe].

   | Diagnostic        | Encoded         | Type             |
   |-------------------|-----------------|------------------|
   | 1                 | 0x01            | unsigned integer |
   | 24                | 0x1818          | unsigned integer |
   | -24               | 0x37            | negative integer |
   | -25               | 0x3818          | negative integer |
   | null              | 0xf6            | simple value     |
   | h'12cd'           | 0x4212cd        | byte string      |
   | '12cd'            | 0x4431326364    | byte string      |
   | "12cd"            | 0x6431326364    | text string      |
   | { 4 : h'cd' }     | 0xa10441cd      | map              |
   | << 1, 2, null >>  | 0x430102f6      | byte string      |
   | [ 1, 2, null ]    | 0x830102f6      | array            |
   | ( 1, 2, null )    | 0x0102f6        | sequence         |
   | 1, 2, null        | 0x0102f6        | sequence         |

A.2.  CDDL Definitions

   This sections compiles the CDDL definitions for ease of reference.

```
bstr_identifier = bstr / int

suite = int

SUITES_R : [ supported : 2* suite ] / suite

message_1 = (
  ? C_1 : null,
  METHOD_CORR : int,
  SUITES_I : [ selected : suite, supported : 2* suite ] / suite,
  G_X : bstr,
  C_I : bstr_identifier,
  ? AD_1 : bstr,
)

message_2 = (
  data_2,
  CIPHERTEXT_2 : bstr,
)

data_2 = (
  ? C_I : bstr_identifier,
  G_Y : bstr,
  C_R : bstr_identifier,
)

message_3 = (
  data_3,
  CIPHERTEXT_3 : bstr,
)

data_3 = (
  ? C_R : bstr_identifier,
)

message_4 = (
  data_4,
  MAC_4 : bstr,
)

data_4 = (
  ? C_I : bstr_identifier,
)

error = (
  ? C_x : bstr_identifier,
  ERR_CODE : int,
  ERR_INFO : any
```

```
   )

   info = [
      edhoc_aead_id : int / tstr,
      transcript_hash : bstr,
      label : tstr,
      length : uint
   ]
```

A.3.  COSE

   CBOR Object Signing and Encryption (COSE)
   [I-D.ietf-cose-rfc8152bis-struct] describes how to create and process
   signatures, message authentication codes, and encryption using CBOR.
   COSE builds on JOSE, but is adapted to allow more efficient
   processing in constrained devices.  EDHOC makes use of COSE_Key,
   COSE_Encrypt0, and COSE_Sign1 objects.

Appendix B.  Test Vectors

   This appendix provides detailed test vectors compatible with versions
   -05 and -06 of this specification, to ease implementation and ensure
   interoperability.  In addition to hexadecimal, all CBOR data items
   and sequences are given in CBOR diagnostic notation.  The test
   vectors use the default mapping to CoAP where the Initiator acts as
   CoAP client (this means that corr = 1).

   A more extensive test vector suite covering more combinations of
   authentication method used between Initiator and Responder and
   related code to generate them can be found at https://github.com/
   lake-wg/edhoc/tree/master/test-vectors-05.

   NOTE 1.  In the previous and current test vectors the same name is
   used for certain byte strings and their CBOR bstr encodings.  For
   example the transcript hash TH_2 is used to denote both the output of
   the hash function and the input into the key derivation function,
   whereas the latter is a CBOR bstr encoding of the former.  Some
   attempts are made to clarify that in this Appendix (e.g. using "CBOR
   encoded"/"CBOR unencoded").

   NOTE 2.  If not clear from the context, remember that CBOR sequences
   and CBOR arrays assume CBOR encoded data items as elements.

B.1.  Test Vectors for EDHOC Authenticated with Signature Keys (x5t)

   EDHOC with signature authentication and X.509 certificates is used.
   In this test vector, the hash value 'x5t' is used to identify the
   certificate.  The optional C_1 in message_1 is omitted.  No auxiliary
   data is sent in the message exchange.

   method (Signature Authentication)
   0

   CoAP is used as transport and the Initiator acts as CoAP client:

   corr (the Initiator can correlate message_1 and message_2)
   1

   From there, METHOD_CORR has the following value:

   METHOD_CORR (4 * method + corr) (int)
   1

   The Initiator indicates only one cipher suite in the (potentially
   truncated) list of cipher suites.

   Supported Cipher Suites (1 byte)
   00

   The Initiator selected the indicated cipher suite.

   Selected Cipher Suite (int)
   0

   Cipher suite 0 is supported by both the Initiator and the Responder,
   see Section 3.4.

B.1.1.  Message_1

   The Initiator generates its ephemeral key pair.

 X (Initiator's ephemeral private key) (32 bytes)
 8f 78 1a 09 53 72 f8 5b 6d 9f 61 09 ae 42 26 11 73 4d 7d bf a0 06 9a 2d
 f2 93 5b b2 e0 53 bf 35

 G_X (Initiator's ephemeral public key, CBOR unencoded) (32 bytes)
 89 8f f7 9a 02 06 7a 16 ea 1e cc b9 0f a5 22 46 f5 aa 4d d6 ec 07 6b ba
 02 59 d9 04 b7 ec 8b 0c

   The Initiator chooses a connection identifier C_I:

Connection identifier chosen by Initiator (1 byte)
09

Note that since C_I is a byte string in the interval h'00' to h'2f',
it is encoded as the corresponding integer subtracted by 24 (see
bstr_identifier in Section 5.1).  Thus 0x09 = 09, 9 - 24 = -15, and
-15 in CBOR encoding is equal to 0x2e.

C_I (1 byte)
2e

Since no auxiliary data is sent:

AD_1 (0 bytes)

The list of supported cipher suites needs to contain the selected
cipher suite.  The initiator truncates the list of supported cipher
suites to one cipher suite only.  In this case there is only one
supported cipher suite indicated, 00.

Because one single selected cipher suite is conveyed, it is encoded
as an int instead of an array:

SUITES_I (int)
0

message_1 is constructed as the CBOR Sequence of the data items above
encoded as CBOR.  In CBOR diagnostic notation:

message_1 =
 (
   1,
   0,
   h'898FF79A02067A16EA1ECCB90FA52246F5AA4DD6EC076BBA0259D904B7EC8B0C',
   -15
 )

Which as a CBOR encoded data item is:

message_1 (CBOR Sequence) (37 bytes)
01 00 58 20 89 8f f7 9a 02 06 7a 16 ea 1e cc b9 0f a5 22 46 f5 aa 4d d6
ec 07 6b ba 02 59 d9 04 b7 ec 8b 0c 2e

B.1.2.  Message_2

Since METHOD_CORR mod 4 equals 1, C_I is omitted from data_2.

The Responder generates the following ephemeral key pair.

 Y (Responder's ephemeral private key) (32 bytes)
 fd 8c d8 77 c9 ea 38 6e 6a f3 4f f7 e6 06 c4 b6 4c a8 31 c8 ba 33 13 4f
 d4 cd 71 67 ca ba ec da

 G_Y (Responder's ephemeral public key, CBOR unencoded) (32 bytes)
 71 a3 d5 99 c2 1d a1 89 02 a1 ae a8 10 b2 b6 38 2c cd 8d 5f 9b f0 19 52
 81 75 4c 5e bc af 30 1e

   From G_X and Y or from G_Y and X the ECDH shared secret is computed:

 G_XY (ECDH shared secret) (32 bytes)
 2b b7 fa 6e 13 5b c3 35 d0 22 d6 34 cb fb 14 b3 f5 82 f3 e2 e3 af b2 b3
 15 04 91 49 5c 61 78 2b

   The key and nonce for calculating the 'ciphertext' are calculated as
   follows, as specified in Section 4.

   HKDF SHA-256 is the HKDF used (as defined by cipher suite 0).

   PRK_2e = HMAC-SHA-256(salt, G_XY)

   Salt is the empty byte string.

   salt (0 bytes)

   From there, PRK_2e is computed:

 PRK_2e (32 bytes)
 ec 62 92 a0 67 f1 37 fc 7f 59 62 9d 22 6f bf c4 e0 68 89 49 f6 62 a9 7f
 d8 2f be b7 99 71 39 4a

   The Responder's sign/verify key pair is the following:

 SK_R (Responder's private authentication key) (32 bytes)
 df 69 27 4d 71 32 96 e2 46 30 63 65 37 2b 46 83 ce d5 38 1b fc ad cd 44
 0a 24 c3 91 d2 fe db 94

 PK_R (Responder's public authentication key) (32 bytes)
 db d9 dc 8c d0 3f b7 c3 91 35 11 46 2b b2 38 16 47 7c 6b d8 d6 6e f5 a1
 a0 70 ac 85 4e d7 3f d2

   Since neither the Initiator nor the Responder authenticates with a
   static Diffie-Hellman key, PRK_3e2m = PRK_2e

 PRK_3e2m (32 bytes)
 ec 62 92 a0 67 f1 37 fc 7f 59 62 9d 22 6f bf c4 e0 68 89 49 f6 62 a9 7f
 d8 2f be b7 99 71 39 4a

The Responder chooses a connection identifier C_R.

Connection identifier chosen by Responder (1 byte)
00

Note that since C_R is a byte string in the interval h'00' to h'2f',
it is encoded as the corresponding integer subtracted by 24 (see
bstr_identifier in Section 5.1).  Thus 0x00 = 0, 0 - 24 = -24, and
-24 in CBOR encoding is equal to 0x37.

C_R (1 byte)
37

Data_2 is constructed as the CBOR Sequence of G_Y and C_R, encoded as
CBOR byte strings.  The CBOR diagnostic notation is:

```
data_2 =
(
  h'71a3d599c21da18902a1aea810b2b6382ccd8d5f9bf0195281754c5ebcaf301e',
  -24
)
```

Which as a CBOR encoded data item is:

```
data_2 (CBOR Sequence) (35 bytes)
58 20 71 a3 d5 99 c2 1d a1 89 02 a1 ae a8 10 b2 b6 38 2c cd 8d 5f 9b f0
19 52 81 75 4c 5e bc af 30 1e 37
```

From data_2 and message_1, compute the input to the transcript hash
TH_2 = H( message_1, data_2 ), as a CBOR Sequence of these 2 data
items.

```
Input to calculate TH_2 (CBOR Sequence) (72 bytes)
01 00 58 20 89 8f f7 9a 02 06 7a 16 ea 1e cc b9 0f a5 22 46 f5 aa 4d d6
ec 07 6b ba 02 59 d9 04 b7 ec 8b 0c 2e 58 20 71 a3 d5 99 c2 1d a1 89 02
a1 ae a8 10 b2 b6 38 2c cd 8d 5f 9b f0 19 52 81 75 4c 5e bc af 30 1e 37
```

And from there, compute the transcript hash TH_2 = SHA-256(
message_1, data_2 )

```
TH_2 (CBOR unencoded) (32 bytes)
86 4e 32 b3 6a 7b 5f 21 f1 9e 99 f0 c6 6d 91 1e 0a ce 99 72 d3 76 d2 c2
c1 53 c1 7f 8e 96 29 ff
```

The Responder's subject name is the empty string:

Responder's subject name (text string)
""

In this version of the test vectors CRED_R is not a DER encoded X.509
certificate, but a string of random bytes.

CRED_R (CBOR unencoded) (100 bytes)
c7 88 37 00 16 b8 96 5b db 20 74 bf f8 2e 5a 20 e0 9b ec 21 f8 40 6e 86
44 2b 87 ec 3f f2 45 b7 0a 47 62 4d c9 cd c6 82 4b 2a 4c 52 e9 5e c9 d6
b0 53 4b 71 c2 b4 9e 4b f9 03 15 00 ce e6 86 99 79 c2 97 bb 5a 8b 38 1e
98 db 71 41 08 41 5e 5c 50 db 78 97 4c 27 15 79 b0 16 33 a3 ef 62 71 be
5c 22 5e b2

CRED_R is defined to be the CBOR bstr containing the credential of
the Responder.

CRED_R (102 bytes)
58 64 c7 88 37 00 16 b8 96 5b db 20 74 bf f8 2e 5a 20 e0 9b ec 21 f8 40
6e 86 44 2b 87 ec 3f f2 45 b7 0a 47 62 4d c9 cd c6 82 4b 2a 4c 52 e9 5e
c9 d6 b0 53 4b 71 c2 b4 9e 4b f9 03 15 00 ce e6 86 99 79 c2 97 bb 5a 8b
38 1e 98 db 71 41 08 41 5e 5c 50 db 78 97 4c 27 15 79 b0 16 33 a3 ef 62
71 be 5c 22 5e b2

And because certificates are identified by a hash value with the
'x5t' parameter, ID_CRED_R is the following:

ID_CRED_R = { 34 : COSE_CertHash }. In this example, the hash
algorithm used is SHA-2 256-bit with hash truncated to 64-bits (value
-15).  The hash value is calculated over the CBOR unencoded CRED_R.
The CBOR diagnostic notation is:

ID_CRED_R =
{
  34: [-15, h'6844078A53F312F5']
}

which when encoded as a CBOR map becomes:

ID_CRED_R (14 bytes)
a1 18 22 82 2e 48 68 44 07 8a 53 f3 12 f5

Since no auxiliary data is sent:

AD_2  (0 bytes)

The plaintext is defined as the empty string:

P_2m (0 bytes)

The Enc_structure is defined as follows: [ "Encrypt0",
<< ID_CRED_R >>, << TH_2, CRED_R >> ], indicating that ID_CRED_R is
encoded as a CBOR byte string, and that the concatenation of the CBOR
byte strings TH_2 and CRED_R is wrapped as a CBOR bstr.  The CBOR
diagnostic notation is the following:

```
A_2m =
[
  "Encrypt0",
  h'A11822822E486844078A53F312F5',
  h'5820864E32B36A7B5F21F19E99F0C66D911E0ACE9972D376D2C2C153C17F8E9629FF
  5864C788370016B8965BDB2074BFF82E5A20E09BEC21F8406E86442B87EC3FF245B70A
  47624DC9CDC6824B2A4C52E95EC9D6B0534B71C2B49E4BF9031500CEE6869979C297BB
  5A8B381E98DB714108415E5C50DB78974C271579B01633A3EF6271BE5C225EB2'
  ]
```

Which encodes to the following byte string to be used as Additional
Authenticated Data:

```
A_2m (CBOR-encoded) (163 bytes)
83 68 45 6e 63 72 79 70 74 30 4e a1 18 22 82 2e 48 68 44 07 8a 53 f3 12
f5 58 88 58 20 86 4e 32 b3 6a 7b 5f 21 f1 9e 99 f0 c6 6d 91 1e 0a ce 99
72 d3 76 d2 c2 c1 53 c1 7f 8e 96 29 ff 58 64 c7 88 37 00 16 b8 96 5b db
20 74 bf f8 2e 5a 20 e0 9b ec 21 f8 40 6e 86 44 2b 87 ec 3f f2 45 b7 0a
47 62 4d c9 cd c6 82 4b 2a 4c 52 e9 5e c9 d6 b0 53 4b 71 c2 b4 9e 4b f9
03 15 00 ce e6 86 99 79 c2 97 bb 5a 8b 38 1e 98 db 71 41 08 41 5e 5c 50
db 78 97 4c 27 15 79 b0 16 33 a3 ef 62 71 be 5c 22 5e b2
```

info for K_2m is defined as follows in CBOR diagnostic notation:

```
info for K_2m =
[
  10,
  h'864E32B36A7B5F21F19E99F0C66D911E0ACE9972D376D2C2C153C17F8E9629FF',
  "K_2m",
  16
]
```

Which as a CBOR encoded data item is:

```
info for K_2m (CBOR-encoded) (42 bytes)
84 0a 58 20 86 4e 32 b3 6a 7b 5f 21 f1 9e 99 f0 c6 6d 91 1e 0a ce 99 72
d3 76 d2 c2 c1 53 c1 7f 8e 96 29 ff 64 4b 5f 32 6d 10
```

From these parameters, K_2m is computed.  Key K_2m is the output of
HKDF-Expand(PRK_3e2m, info, L), where L is the length of K_2m, so 16
bytes.

   K_2m (16 bytes)
   80 cc a7 49 ab 58 f5 69 ca 35 da ee 05 be d1 94

   info for IV_2m is defined as follows, in CBOR diagnostic notation (10
   is the COSE algorithm no. of the AEAD algorithm in the selected
   cipher suite 0):

 info for IV_2m =
 [
   10,
   h'864E32B36A7B5F21F19E99F0C66D911E0ACE9972D376D2C2C153C17F8E9629FF',
   "IV_2m",
   13
 ]

   Which as a CBOR encoded data item is:

 info for IV_2m (CBOR-encoded) (43 bytes)
 84 0a 58 20 86 4e 32 b3 6a 7b 5f 21 f1 9e 99 f0 c6 6d 91 1e 0a ce 99 72
 d3 76 d2 c2 c1 53 c1 7f 8e 96 29 ff 65 49 56 5f 32 6d 0d

   From these parameters, IV_2m is computed.  IV_2m is the output of
   HKDF-Expand(PRK_3e2m, info, L), where L is the length of IV_2m, so 13
   bytes.

   IV_2m (13 bytes)
   c8 1e 1a 95 cc 93 b3 36 69 6e d5 02 55

   Finally, COSE_Encrypt0 is computed from the parameters above.

   *  protected header = CBOR-encoded ID_CRED_R

   *  external_aad = A_2m

   *  empty plaintext = P_2m

   MAC_2 (CBOR unencoded) (8 bytes)
   fa bb a4 7e 56 71 a1 82

   To compute the Signature_or_MAC_2, the key is the private
   authentication key of the Responder and the message M_2 to be signed
   = [ "Signature1", << ID_CRED_R >>, << TH_2, CRED_R, ? AD_2 >>, MAC_2
   ].  ID_CRED_R is encoded as a CBOR byte string, the concatenation of
   the CBOR byte strings TH_2 and CRED_R is wrapped as a CBOR bstr, and
   MAC_2 is encoded as a CBOR bstr.

```
M_2 =
[
  "Signature1",
  h'A11822822E486844078A53F312F5',
  h'5820864E32B36A7B5F21F19E99F0C66D911E0ACE9972D376D2C2C153C17F8E9629F
  F5864C788370016B8965BDB2074BFF82E5A20E09BEC21F8406E86442B87EC3FF245B7
  0A47624DC9CDC6824B2A4C52E95EC9D6B0534B71C2B49E4BF9031500CEE6869979C29
  7BB5A8B381E98DB714108415E5C50DB78974C271579B01633A3EF6271BE5C225EB2',
  h'FABBA47E5671A182'
]
```

   Which as a CBOR encoded data item is:

```
M_2 (174 bytes)
84 6a 53 69 67 6e 61 74 75 72 65 31 4e a1 18 22 82 2e 48 68 44 07 8a 53
f3 12 f5 58 88 58 20 86 4e 32 b3 6a 7b 5f 21 f1 9e 99 f0 c6 6d 91 1e 0a
ce 99 72 d3 76 d2 c2 c1 53 c1 7f 8e 96 29 ff 58 64 c7 88 37 00 16 b8 96
5b db 20 74 bf f8 2e 5a 20 e0 9b ec 21 f8 40 6e 86 44 2b 87 ec 3f f2 45
b7 0a 47 62 4d c9 cd c6 82 4b 2a 4c 52 e9 5e c9 d6 b0 53 4b 71 c2 b4 9e
4b f9 03 15 00 ce e6 86 99 79 c2 97 bb 5a 8b 38 1e 98 db 71 41 08 41 5e
5c 50 db 78 97 4c 27 15 79 b0 16 33 a3 ef 62 71 be 5c 22 5e b2 48 fa bb
a4 7e 56 71 a1 82
```

   Since the method = 0, Signature_or_MAC_2 is a signature.  The
   algorithm with selected cipher suite 0 is Ed25519 and the output is
   64 bytes.

```
Signature_or_MAC_2 (CBOR unencoded) (64 bytes)
1f 17 00 6a 98 48 c9 77 cb bd ca a7 57 b6 fd 46 82 c8 17 39 e1 5c 99 37
c2 1c f5 e9 a0 e6 03 9f 54 fd 2a 6c 3a 11 18 f2 b9 d8 eb cd 48 23 48 b9
9c 3e d7 ed 1b d9 80 6c 93 c8 90 68 e8 36 b4 0f
```

   CIPHERTEXT_2 is the ciphertext resulting from XOR between plaintext
   and KEYSTREAM_2 which is derived from TH_2 and the pseudorandom key
   PRK_2e.

   *  plaintext = CBOR Sequence of the items ID_CRED_R and
      Signature_or_MAC_2 encoded as CBOR byte strings, in this order
      (AD_2 is empty).

   The plaintext is the following:

```
P_2e (CBOR Sequence) (80 bytes)
a1 18 22 82 2e 48 68 44 07 8a 53 f3 12 f5 58 40 1f 17 00 6a 98 48 c9 77
cb bd ca a7 57 b6 fd 46 82 c8 17 39 e1 5c 99 37 c2 1c f5 e9 a0 e6 03 9f
54 fd 2a 6c 3a 11 18 f2 b9 d8 eb cd 48 23 48 b9 9c 3e d7 ed 1b d9 80 6c
93 c8 90 68 e8 36 b4 0f
```

```
  KEYSTREAM_2 = HKDF-Expand( PRK_2e, info, length ), where length is
  the length of the plaintext, so 80.
```

```
info for KEYSTREAM_2 =
[
  10,
  h'864E32B36A7B5F21F19E99F0C66D911E0ACE9972D376D2C2C153C17F8E9629FF',
  "KEYSTREAM_2",
  80
]
```

Which as a CBOR encoded data item is:

```
info for KEYSTREAM_2 (CBOR-encoded) (50 bytes)
84 0a 58 20 86 4e 32 b3 6a 7b 5f 21 f1 9e 99 f0 c6 6d 91 1e 0a ce 99 72
d3 76 d2 c2 c1 53 c1 7f 8e 96 29 ff 6b 4b 45 59 53 54 52 45 41 4d 5f 32
18 50
```

From there, KEYSTREAM_2 is computed:

```
KEYSTREAM_2 (80 bytes)
ae ea 8e af 50 cf c6 70 09 da e8 2d 8d 85 b0 e7 60 91 bf 0f 07 0b 79 53
6c 83 23 dc 3d 9d 61 13 10 35 94 63 f4 4b 12 4b ea b3 a1 9d 09 93 82 d7
30 80 17 f4 92 62 06 e4 f5 44 9b 9f c9 24 bc b6 bd 78 ec 45 0a 66 83 fb
8a 2f 5f 92 4f c4 40 4f
```

Using the parameters above, the ciphertext CIPHERTEXT_2 can be
computed:

```
CIPHERTEXT_2 (CBOR unencoded) (80 bytes)
0f f2 ac 2d 7e 87 ae 34 0e 50 bb de 9f 70 e8 a7 7f 86 bf 65 9f 43 b0 24
a7 3e e9 7b 6a 2b 9c 55 92 fd 83 5a 15 17 8b 7c 28 af 54 74 a9 75 81 48
64 7d 3d 98 a8 73 1e 16 4c 9c 70 52 81 07 f4 0f 21 46 3b a8 11 bf 03 97
19 e7 cf fa a7 f2 f4 40
```

message_2 is the CBOR Sequence of data_2 and CIPHERTEXT_2, in this
order:

```
message_2 =
(
  data_2,
  h'0FF2AC2D7E87AE340E50BBDE9F70E8A77F86BF659F43B024A73EE97B6A2B9C5592FD
  835A15178B7C28AF5474A9758148647D3D98A8731E164C9C70528107F40F21463BA811
  BF039719E7CFFAA7F2F440'
)
```

Which as a CBOR encoded data item is:

```
message_2 (CBOR Sequence) (117 bytes)
58 20 71 a3 d5 99 c2 1d a1 89 02 a1 ae a8 10 b2 b6 38 2c cd 8d 5f 9b f0
19 52 81 75 4c 5e bc af 30 1e 37 58 50 0f f2 ac 2d 7e 87 ae 34 0e 50 bb
de 9f 70 e8 a7 7f 86 bf 65 9f 43 b0 24 a7 3e e9 7b 6a 2b 9c 55 92 fd 83
5a 15 17 8b 7c 28 af 54 74 a9 75 81 48 64 7d 3d 98 a8 73 1e 16 4c 9c 70
52 81 07 f4 0f 21 46 3b a8 11 bf 03 97 19 e7 cf fa a7 f2 f4 40
```

B.1.3.  Message_3

   Since corr equals 1, C_R is not omitted from data_3.

   The Initiator's sign/verify key pair is the following:

```
SK_I (Initiator's private authentication key) (32 bytes)
2f fc e7 a0 b2 b8 25 d3 97 d0 cb 54 f7 46 e3 da 3f 27 59 6e e0 6b 53 71
48 1d c0 e0 12 bc 34 d7
```

```
PK_I (Responder's public authentication key) (32 bytes)
38 e5 d5 45 63 c2 b6 a4 ba 26 f3 01 5f 61 bb 70 6e 5c 2e fd b5 56 d2 e1
69 0b 97 fc 3c 6d e1 49
```

   HKDF SHA-256 is the HKDF used (as defined by cipher suite 0).

   PRK_4x3m = HMAC-SHA-256 (PRK_3e2m, G_IY)

```
PRK_4x3m (32 bytes)
ec 62 92 a0 67 f1 37 fc 7f 59 62 9d 22 6f bf c4 e0 68 89 49 f6 62 a9 7f
d8 2f be b7 99 71 39 4a
```

   data 3 is equal to C_R.

```
data_3 (CBOR Sequence) (1 byte)
37
```

   From data_3, CIPHERTEXT_2, and TH_2, compute the input to the
   transcript hash TH_3 = H(TH_2 , CIPHERTEXT_2, data_3), as a CBOR
   Sequence of these 3 data items.

```
Input to calculate TH_3 (CBOR Sequence) (117 bytes)
58 20 86 4e 32 b3 6a 7b 5f 21 f1 9e 99 f0 c6 6d 91 1e 0a ce 99 72 d3 76
d2 c2 c1 53 c1 7f 8e 96 29 ff 58 50 0f f2 ac 2d 7e 87 ae 34 0e 50 bb de
9f 70 e8 a7 7f 86 bf 65 9f 43 b0 24 a7 3e e9 7b 6a 2b 9c 55 92 fd 83 5a
15 17 8b 7c 28 af 54 74 a9 75 81 48 64 7d 3d 98 a8 73 1e 16 4c 9c 70 52
81 07 f4 0f 21 46 3b a8 11 bf 03 97 19 e7 cf fa a7 f2 f4 40 37
```

   And from there, compute the transcript hash TH_3 = SHA-256(TH_2 ,
   CIPHERTEXT_2, data_3)

TH_3 (CBOR unencoded) (32 bytes)
f2 4d 18 ca fc e3 74 d4 e3 73 63 29 c1 52 ab 3a ea 9c 7c 0f 65 0c 30 70
b6 f5 1e 68 e2 ae bb 60

   The Initiator's subject name is the empty string:

   Initiator's subject name (text string)
   ""

   In this version of the test vectors CRED_I is not a DER encoded X.509
   certificate, but a string of random bytes.

CRED_I (CBOR unencoded) (101 bytes)
54 13 20 4c 3e bc 34 28 a6 cf 57 e2 4c 9d ef 59 65 17 70 44 9b ce 7e c6
56 1e 52 43 3a a5 5e 71 f1 fa 34 b2 2a 9c a4 a1 e1 29 24 ea e1 d1 76 60
88 09 84 49 cb 84 8f fc 79 5f 88 af c4 9c be 8a fd d1 ba 00 9f 21 67 5e
8f 6c 77 a4 a2 c3 01 95 60 1f 6f 0a 08 52 97 8b d4 3d 28 20 7d 44 48 65
02 ff 7b dd a6

   CRED_I is defined to be the CBOR bstr containing the credential of
   the Initiator.

CRED_I (103 bytes)
58 65 54 13 20 4c 3e bc 34 28 a6 cf 57 e2 4c 9d ef 59 65 17 70 44 9b ce
7e c6 56 1e 52 43 3a a5 5e 71 f1 fa 34 b2 2a 9c a4 a1 e1 29 24 ea e1 d1
76 60 88 09 84 49 cb 84 8f fc 79 5f 88 af c4 9c be 8a fd d1 ba 00 9f 21
67 5e 8f 6c 77 a4 a2 c3 01 95 60 1f 6f 0a 08 52 97 8b d4 3d 28 20 7d 44
48 65 02 ff 7b dd a6

   And because certificates are identified by a hash value with the
   'x5t' parameter, ID_CRED_I is the following:

   ID_CRED_I = { 34 : COSE_CertHash }. In this example, the hash
   algorithm used is SHA-2 256-bit with hash truncated to 64-bits (value
   -15).  The hash value is calculated over the CBOR unencoded CRED_I.

   ID_CRED_I =
   {
     34: [-15, h'705D5845F36FC6A6']
   }

   which when encoded as a CBOR map becomes:

   ID_CRED_I (14 bytes)
   a1 18 22 82 2e 48 70 5d 58 45 f3 6f c6 a6

   Since no auxiliary data is exchanged:

   AD_3 (0 bytes)

   The plaintext of the COSE_Encrypt is the empty string:

   P_3m (0 bytes)

   The associated data is the following: [ "Encrypt0", << ID_CRED_I >>,
   << TH_3, CRED_I, ? AD_3 >> ].

 A_3m (CBOR-encoded) (164 bytes)
 83 68 45 6e 63 72 79 70 74 30 4e a1 18 22 82 2e 48 70 5d 58 45 f3 6f c6
 a6 58 89 58 20 f2 4d 18 ca fc e3 74 d4 e3 73 63 29 c1 52 ab 3a ea 9c 7c
 0f 65 0c 30 70 b6 f5 1e 68 e2 ae bb 60 58 65 54 13 20 4c 3e bc 34 28 a6
 cf 57 e2 4c 9d ef 59 65 17 70 44 9b ce 7e c6 56 1e 52 43 3a a5 5e 71 f1
 fa 34 b2 2a 9c a4 a1 e1 29 24 ea e1 d1 76 60 88 09 84 49 cb 84 8f fc 79
 5f 88 af c4 9c be 8a fd d1 ba 00 9f 21 67 5e 8f 6c 77 a4 a2 c3 01 95 60
 1f 6f 0a 08 52 97 8b d4 3d 28 20 7d 44 48 65 02 ff 7b dd a6

   Info for K_3m is computed as follows:

 info for K_3m =
 [
   10,
   h'F24D18CAFCE374D4E3736329C152AB3AEA9C7C0F650C3070B6F51E68E2AEBB60',
   "K_3m",
   16
 ]

   Which as a CBOR encoded data item is:

 info for K_3m (CBOR-encoded) (42 bytes)
 84 0a 58 20 f2 4d 18 ca fc e3 74 d4 e3 73 63 29 c1 52 ab 3a ea 9c 7c 0f
 65 0c 30 70 b6 f5 1e 68 e2 ae bb 60 64 4b 5f 33 6d 10

   From these parameters, K_3m is computed.  Key K_3m is the output of
   HKDF-Expand(PRK_4x3m, info, L), where L is the length of K_2m, so 16
   bytes.

   K_3m (16 bytes)
   83 a9 c3 88 02 91 2e 7f 8f 0d 2b 84 14 d1 e5 2c

   Nonce IV_3m is the output of HKDF-Expand(PRK_4x3m, info, L), where L
   = 13 bytes.

   Info for IV_3m is defined as follows:

```
  info for IV_3m =
  [
    10,
    h'F24D18CAFCE374D4E3736329C152AB3AEA9C7C0F650C3070B6F51E68E2AEBB60',
    "IV_3m",
    13
  ]
```

   Which as a CBOR encoded data item is:

```
 info for IV_3m (CBOR-encoded) (43 bytes)
 84 0a 58 20 f2 4d 18 ca fc e3 74 d4 e3 73 63 29 c1 52 ab 3a ea 9c 7c 0f
 65 0c 30 70 b6 f5 1e 68 e2 ae bb 60 65 49 56 5f 33 6d 0d
```

   From these parameters, IV_3m is computed:

```
   IV_3m (13 bytes)
   9c 83 9c 0e e8 36 42 50 5a 8e 1c 9f b2
```

   MAC_3 is the 'ciphertext' of the COSE_Encrypt0:

```
   MAC_3 (CBOR unencoded) (8 bytes)
   2f a1 e3 9e ae 7d 5f 8d
```

   Since the method = 0, Signature_or_MAC_3 is a signature.  The
   algorithm with selected cipher suite 0 is Ed25519.

   *  The message M_3 to be signed = [ "Signature1", << ID_CRED_I >>,
      << TH_3, CRED_I >>, MAC_3 ], i.e. ID_CRED_I encoded as CBOR bstr,
      the concatenation of the CBOR byte strings TH_3 and CRED_I wrapped
      as a CBOR bstr, and MAC_3 encoded as a CBOR bstr.

   *  The signing key is the private authentication key of the
      Initiator.

```
 M_3 =
 [
   "Signature1",
   h'A11822822E48705D5845F36FC6A6',
   h'5820F24D18CAFCE374D4E3736329C152AB3AEA9C7C0F650C3070B6F51E68E2AEBB6
   058655413204C3EBC3428A6CF57E24C9DEF59651770449BCE7EC6561E52433AA55E71
   F1FA34B22A9CA4A1E12924EAE1D1766088098449CB848FFC795F88AFC49CBE8AFDD1B
   A009F21675E8F6C77A4A2C30195601F6F0A0852978BD43D28207D44486502FF7BDD
   A6',
   h'2FA1E39EAE7D5F8D']
```

   Which as a CBOR encoded data item is:

M_3 (175 bytes)
```
84 6a 53 69 67 6e 61 74 75 72 65 31 4e a1 18 22 82 2e 48 70 5d 58 45 f3
6f c6 a6 58 89 58 20 f2 4d 18 ca fc e3 74 d4 e3 73 63 29 c1 52 ab 3a ea
9c 7c 0f 65 0c 30 70 b6 f5 1e 68 e2 ae bb 60 58 65 54 13 20 4c 3e bc 34
28 a6 cf 57 e2 4c 9d ef 59 65 17 70 44 9b ce 7e c6 56 1e 52 43 3a a5 5e
71 f1 fa 34 b2 2a 9c a4 a1 e1 29 24 ea e1 d1 76 60 88 09 84 49 cb 84 8f
fc 79 5f 88 af c4 9c be 8a fd d1 ba 00 9f 21 67 5e 8f 6c 77 a4 a2 c3 01
95 60 1f 6f 0a 08 52 97 8b d4 3d 28 20 7d 44 48 65 02 ff 7b dd a6 48 2f
a1 e3 9e ae 7d 5f 8d
```

   From there, the 64 byte signature can be computed:

Signature_or_MAC_3 (CBOR unencoded) (64 bytes)
```
ab 9f 7b bd eb c4 eb f8 a3 d3 04 17 9b cc a3 9d 9c 8a 76 73 65 76 fb 3c
32 d2 fa c7 e2 59 34 e5 33 dc c7 02 2e 4d 68 61 c8 f5 fe cb e9 2d 17 4e
b2 be af 0a 59 a4 15 84 37 2f 43 2e 6b f4 7b 04
```

   Finally, the outer COSE_Encrypt0 is computed.

   The plaintext is the CBOR Sequence of the items ID_CRED_I and the
   CBOR encoded Signature_or_MAC_3, in this order (AD_3 is empty).

P_3ae (CBOR Sequence) (80 bytes)
```
a1 18 22 82 2e 48 70 5d 58 45 f3 6f c6 a6 58 40 ab 9f 7b bd eb c4 eb f8
a3 d3 04 17 9b cc a3 9d 9c 8a 76 73 65 76 fb 3c 32 d2 fa c7 e2 59 34 e5
33 dc c7 02 2e 4d 68 61 c8 f5 fe cb e9 2d 17 4e b2 be af 0a 59 a4 15 84
37 2f 43 2e 6b f4 7b 04
```

   The Associated data A is the following: Associated data A = [
   "Encrypt0", h'', TH_3 ]

A_3ae (CBOR-encoded) (45 bytes)
```
83 68 45 6e 63 72 79 70 74 30 40 58 20 f2 4d 18 ca fc e3 74 d4 e3 73 63
29 c1 52 ab 3a ea 9c 7c 0f 65 0c 30 70 b6 f5 1e 68 e2 ae bb 60
```

   Key K_3ae is the output of HKDF-Expand(PRK_3e2m, info, L).

   info is defined as follows:

```
 info for K_3ae =
 [
   10,
   h'F24D18CAFCE374D4E3736329C152AB3AEA9C7C0F650C3070B6F51E68E2AEBB60',
   "K_3ae",
   16
 ]
```

   Which as a CBOR encoded data item is:

info for K_3ae (CBOR-encoded) (43 bytes)
84 0a 58 20 f2 4d 18 ca fc e3 74 d4 e3 73 63 29 c1 52 ab 3a ea 9c 7c 0f
65 0c 30 70 b6 f5 1e 68 e2 ae bb 60 65 4b 5f 33 61 65 10

  L is the length of K_3ae, so 16 bytes.

  From these parameters, K_3ae is computed:

  K_3ae (16 bytes)
  b8 79 9f e3 d1 50 4f d8 eb 22 c4 72 62 cd bb 05

  Nonce IV_3ae is the output of HKDF-Expand(PRK_3e2m, info, L).

  info is defined as follows:

 info for IV_3ae =
 [
   10,
   h'F24D18CAFCE374D4E3736329C152AB3AEA9C7C0F650C3070B6F51E68E2AEBB60',
   "IV_3ae",
   13
 ]

  Which as a CBOR encoded data item is:

 info for IV_3ae (CBOR-encoded) (44 bytes)
 84 0a 58 20 f2 4d 18 ca fc e3 74 d4 e3 73 63 29 c1 52 ab 3a ea 9c 7c 0f
 65 0c 30 70 b6 f5 1e 68 e2 ae bb 60 66 49 56 5f 33 61 65 0d

  L is the length of IV_3ae, so 13 bytes.

  From these parameters, IV_3ae is computed:

  IV_3ae (13 bytes)
  74 c7 de 41 b8 4a 5b b7 19 0a 85 98 dc

  Using the parameters above, the 'ciphertext' CIPHERTEXT_3 can be
  computed:

 CIPHERTEXT_3 (CBOR unencoded) (88 bytes)
 f5 f6 de bd 82 14 05 1c d5 83 c8 40 96 c4 80 1d eb f3 5b 15 36 3d d1 6e
 bd 85 30 df dc fb 34 fc d2 eb 6c ad 1d ac 66 a4 79 fb 38 de aa f1 d3 0a
 7e 68 17 a2 2a b0 4f 3d 5b 1e 97 2a 0d 13 ea 86 c6 6b 60 51 4c 96 57 ea
 89 c5 7b 04 01 ed c5 aa 8b bc ab 81 3c c5 d6 e7

  From the parameter above, message_3 is computed, as the CBOR Sequence
  of the following CBOR encoded data items: (C_R, CIPHERTEXT_3).

```
message_3 =
(
  -24,
  h'F5F6DEBD8214051CD583C84096C4801DEBF35B15363DD16EBD8530DFDCFB34FCD2EB
  6CAD1DAC66A479FB38DEAAF1D30A7E6817A22AB04F3D5B1E972A0D13EA86C66B60514C
  9657EA89C57B0401EDC5AA8BBCAB813CC5D6E7'
)
```

Which encodes to the following byte string:

```
message_3 (CBOR Sequence) (91 bytes)
37 58 58 f5 f6 de bd 82 14 05 1c d5 83 c8 40 96 c4 80 1d eb f3 5b 15 36
3d d1 6e bd 85 30 df dc fb 34 fc d2 eb 6c ad 1d ac 66 a4 79 fb 38 de aa
f1 d3 0a 7e 68 17 a2 2a b0 4f 3d 5b 1e 97 2a 0d 13 ea 86 c6 6b 60 51 4c
96 57 ea 89 c5 7b 04 01 ed c5 aa 8b bc ab 81 3c c5 d6 e7
```

B.1.4.  OSCORE Security Context Derivation

From here, the Initiator and the Responder can derive an OSCORE
Security Context, using the EDHOC-Exporter interface.

From TH_3 and CIPHERTEXT_3, compute the input to the transcript hash
TH_4 = H( TH_3, CIPHERTEXT_3 ), as a CBOR Sequence of these 2 data
items.

```
Input to calculate TH_4 (CBOR Sequence) (124 bytes)
58 20 f2 4d 18 ca fc e3 74 d4 e3 73 63 29 c1 52 ab 3a ea 9c 7c 0f 65 0c
30 70 b6 f5 1e 68 e2 ae bb 60 58 58 f5 f6 de bd 82 14 05 1c d5 83 c8 40
96 c4 80 1d eb f3 5b 15 36 3d d1 6e bd 85 30 df dc fb 34 fc d2 eb 6c ad
1d ac 66 a4 79 fb 38 de aa f1 d3 0a 7e 68 17 a2 2a b0 4f 3d 5b 1e 97 2a
0d 13 ea 86 c6 6b 60 51 4c 96 57 ea 89 c5 7b 04 01 ed c5 aa 8b bc ab 81
3c c5 d6 e7
```

And from there, compute the transcript hash TH_4 = SHA-256(TH_3 ,
CIPHERTEXT_4)

```
TH_4 (CBOR unencoded) (32 bytes)
3b 69 a6 7f ec 7e 73 6c c1 a9 52 6c da 00 02 d4 09 f5 b9 ea 0a 2b e9 60
51 a6 e3 0d 93 05 fd 51
```

The Master Secret and Master Salt are derived as follows:

Master Secret = EDHOC-Exporter( "OSCORE Master Secret", 16 ) = EDHOC-
KDF(PRK_4x3m, TH_4, "OSCORE Master Secret", 16) = HKDF-Expand(
PRK_4x3m, info_ms, 16 )

   Master Salt = EDHOC-Exporter( "OSCORE Master Salt", 8 ) = EDHOC-
   KDF(PRK_4x3m, TH_4, "OSCORE Master Salt", 8) = HKDF-Expand( PRK_4x3m,
   info_salt, 8 )

   info_ms for OSCORE Master Secret is defined as follows:

```
 info_ms = [
   10,
   h'3B69A67FEC7E736CC1A9526CDA0002D409F5B9EA0A2BE96051A6E30D9305FD51',
   "OSCORE Master Secret",
   16
 ]
```

   Which as a CBOR encoded data item is:

```
 info_ms for OSCORE Master Secret (CBOR-encoded) (58 bytes)
 84 0a 58 20 3b 69 a6 7f ec 7e 73 6c c1 a9 52 6c da 00 02 d4 09 f5 b9 ea
 0a 2b e9 60 51 a6 e3 0d 93 05 fd 51 74 4f 53 43 4f 52 45 20 4d 61 73 74
 65 72 20 53 65 63 72 65 74 10
```

   info_salt for OSCORE Master Salt is defined as follows:

```
 info_salt = [
   10,
   h'3B69A67FEC7E736CC1A9526CDA0002D409F5B9EA0A2BE96051A6E30D9305FD51',
   "OSCORE Master Salt",
   8
 ]
```

   Which as a CBOR encoded data item is:

```
 info for OSCORE Master Salt (CBOR-encoded) (56 Bytes)
 84 0a 58 20 3b 69 a6 7f ec 7e 73 6c c1 a9 52 6c da 00 02 d4 09 f5 b9 ea
 0a 2b e9 60 51 a6 e3 0d 93 05 fd 51 72 4f 53 43 4f 52 45 20 4d 61 73 74
 65 72 20 53 61 6c 74 08
```

   From these parameters, OSCORE Master Secret and OSCORE Master Salt
   are computed:

   OSCORE Master Secret (16 bytes)
   96 aa 88 ce 86 5e ba 1f fa f3 89 64 13 2c c4 42

   OSCORE Master Salt (8 bytes)
   5e c3 ee 41 7c fb ba e9

   The client's OSCORE Sender ID is C_R and the server's OSCORE Sender
   ID is C_I.

Client's OSCORE Sender ID (1 byte)
00

Server's OSCORE Sender ID (1 byte)
09

The AEAD Algorithm and the hash algorithm are the application AEAD
and hash algorithms in the selected cipher suite.

OSCORE AEAD Algorithm (int)
10

OSCORE Hash Algorithm (int)
-16

B.2.  Test Vectors for EDHOC Authenticated with Static Diffie-Hellman
      Keys

EDHOC with static Diffie-Hellman keys and raw public keys is used.
In this test vector, a key identifier is used to identify the raw
public key.  The optional C_1 in message_1 is omitted.  No auxiliary
data is sent in the message exchange.

method (Static DH Based Authentication)
3

CoAP is used as transport and the Initiator acts as CoAP client:

corr (the Initiator can correlate message_1 and message_2)
1

From there, METHOD_CORR has the following value:

METHOD_CORR (4 * method + corr) (int)
13

The Initiator indicates only one cipher suite in the (potentially
truncated) list of cipher suites.

Supported Cipher Suites (1 byte)
00

The Initiator selected the indicated cipher suite.

Selected Cipher Suite (int)
0

Cipher suite 0 is supported by both the Initiator and the Responder, see Section 3.4.

B.2.1.  Message_1

The Initiator generates its ephemeral key pair.

X (Initiator's ephemeral private key) (32 bytes)
ae 11 a0 db 86 3c 02 27 e5 39 92 fe b8 f5 92 4c 50 d0 a7 ba 6e ea b4 ad
1f f2 45 72 f4 f5 7c fa

G_X (Initiator's ephemeral public key, CBOR unencoded) (32 bytes)
8d 3e f5 6d 1b 75 0a 43 51 d6 8a c2 50 a0 e8 83 79 0e fc 80 a5 38 a4 44
ee 9e 2b 57 e2 44 1a 7c

The Initiator chooses a connection identifier C_I:

Connection identifier chosen by Initiator (1 byte)
16

Note that since C_I is a byte string in the interval h'00' to h'2f', it is encoded as the corresponding integer – 24 (see bstr_identifier in Section 5.1), i.e. 0x16 = 22, 22 – 24 = –2, and –2 in CBOR encoding is equal to 0x21.

C_I (1 byte)
21

Since no auxiliary data is sent:

AD_1 (0 bytes)

Since the list of supported cipher suites needs to contain the selected cipher suite, the initiator truncates the list of supported cipher suites to one cipher suite only, 00.

Because one single selected cipher suite is conveyed, it is encoded as an int instead of an array:

SUITES_I (int)
0

message_1 is constructed as the CBOR Sequence of the data items above encoded as CBOR.  In CBOR diagnostic notation:

```
message_1 =
 (
   13,
   0,
   h'8D3EF56D1B750A4351D68AC250A0E883790EFC80A538A444EE9E2B57E2441A7C',
   -2
 )
```

   Which as a CBOR encoded data item is:

```
message_1 (CBOR Sequence) (37 bytes)
0d 00 58 20 8d 3e f5 6d 1b 75 0a 43 51 d6 8a c2 50 a0 e8 83 79 0e fc 80
a5 38 a4 44 ee 9e 2b 57 e2 44 1a 7c 21
```

B.2.2.  Message_2

   Since METHOD_CORR mod 4 equals 1, C_I is omitted from data_2.

   The Responder generates the following ephemeral key pair.

```
Y (Responder's ephemeral private key) (32 bytes)
c6 46 cd dc 58 12 6e 18 10 5f 01 ce 35 05 6e 5e bc 35 f4 d4 cc 51 07 49
a3 a5 e0 69 c1 16 16 9a
```

```
G_Y (Responder's ephemeral public key, CBOR unencoded) (32 bytes)
52 fb a0 bd c8 d9 53 dd 86 ce 1a b2 fd 7c 05 a4 65 8c 7c 30 af db fc 33
01 04 70 69 45 1b af 35
```

   From G_X and Y or from G_Y and X the ECDH shared secret is computed:

```
G_XY (ECDH shared secret) (32 bytes)
de fc 2f 35 69 10 9b 3d 1f a4 a7 3d c5 e2 fe b9 e1 15 0d 90 c2 5e e2 f0
66 c2 d8 85 f4 f8 ac 4e
```

   The key and nonce for calculating the 'ciphertext' are calculated as
   follows, as specified in Section 4.

   HKDF SHA-256 is the HKDF used (as defined by cipher suite 0).

   PRK_2e = HMAC-SHA-256(salt, G_XY)

   Salt is the empty byte string.

   salt (0 bytes)

   From there, PRK_2e is computed:

PRK_2e (32 bytes)
93 9f cb 05 6d 2e 41 4f 1b ec 61 04 61 99 c2 c7 63 d2 7f 0c 3d 15 fa 16
71 fa 13 4e 0d c5 a0 4d

   The Responder's static Diffie-Hellman key pair is the following:

R (Responder's private authentication key) (32 bytes)
bb 50 1a ac 67 b9 a9 5f 97 e0 ed ed 6b 82 a6 62 93 4f bb fc 7a d1 b7 4c
1f ca d6 6a 07 94 22 d0

G_R (Responder's public authentication key) (32 bytes)
a3 ff 26 35 95 be b3 77 d1 a0 ce 1d 04 da d2 d4 09 66 ac 6b cb 62 20 51
b8 46 59 18 4d 5d 9a 32

   Since the Responder authenticates with a static Diffie-Hellman key,
   PRK_3e2m = HKDF-Extract( PRK_2e, G_RX ), where G_RX is the ECDH
   shared secret calculated from G_R and X, or G_X and R.

   From the Responder's authentication key and the Initiator's ephemeral
   key (see Appendix B.2.1), the ECDH shared secret G_RX is calculated.

G_RX (ECDH shared secret) (32 bytes)
21 c7 ef f4 fb 69 fa 4b 67 97 d0 58 84 31 5d 84 11 a3 fd a5 4f 6d ad a6
1d 4f cd 85 e7 90 66 68

PRK_3e2m (32 bytes)
75 07 7c 69 1e 35 01 2d 48 bc 24 c8 4f 2b ab 89 f5 2f ac 03 fe dd 81 3e
43 8c 93 b1 0b 39 93 07

   The Responder chooses a connection identifier C_R.

   Connection identifier chosen by Responder (1 byte)
   00

   Note that since C_R is a byte string in the interval h'00' to h'2f',
   it is encoded as the corresponding integer - 24 (see bstr_identifier
   in Section 5.1), i.e. 0x00 = 0, 0 - 24 = -24, and -24 in CBOR
   encoding is equal to 0x37.

   C_R (1 byte)
   37

   Data_2 is constructed as the CBOR Sequence of G_Y and C_R.

```
 data_2 =
 (
   h'52FBA0BDC8D953DD86CE1AB2FD7C05A4658C7C30AFDBFC3301047069451BAF35',
   -24
 )
```

   Which as a CBOR encoded data item is:

```
data_2 (CBOR Sequence) (35 bytes)
58 20 52 fb a0 bd c8 d9 53 dd 86 ce 1a b2 fd 7c 05 a4 65 8c 7c 30 af db
fc 33 01 04 70 69 45 1b af 35 37
```

   From data_2 and message_1, compute the input to the transcript hash
   TH_2 = H( message_1, data_2 ), as a CBOR Sequence of these 2 data
   items.

```
Input to calculate TH_2 (CBOR Sequence) (72 bytes)
0d 00 58 20 8d 3e f5 6d 1b 75 0a 43 51 d6 8a c2 50 a0 e8 83 79 0e fc 80
a5 38 a4 44 ee 9e 2b 57 e2 44 1a 7c 21 58 20 52 fb a0 bd c8 d9 53 dd 86
ce 1a b2 fd 7c 05 a4 65 8c 7c 30 af db fc 33 01 04 70 69 45 1b af 35 37
```

   And from there, compute the transcript hash TH_2 = SHA-256(
   message_1, data_2 )

```
TH_2 (CBOR unencoded) (32 bytes)
de cf d6 4a 36 67 64 0a 02 33 b0 4a a8 aa 91 f6 89 56 b8 a5 36 d0 cf 8c
73 a6 e8 a7 c3 62 1e 26
```

   The Responder's subject name is the empty string:

```
   Responder's subject name (text string)
   ""
```

   ID_CRED_R is the following:

```
   ID_CRED_R =
   {
     4: h'05'
   }
```

```
   ID_CRED_R (4 bytes)
   a1 04 41 05
```

   CRED_R is the following COSE_Key:

```
{
  1: 1,
  -1: 4,
  -2: h'A3FF263595BEB377D1A0CE1D04DAD2D40966AC6BCB622051B84659184D5D9A32,
  "subject name": ""
}
```

   Which encodes to the following byte string:

```
 CRED_R (54 bytes)
 a4 01 01 20 04 21 58 20 a3 ff 26 35 95 be b3 77 d1 a0 ce 1d 04 da d2 d4
 09 66 ac 6b cb 62 20 51 b8 46 59 18 4d 5d 9a 32 6c 73 75 62 6a 65 63 74
 20 6e 61 6d 65 60
```

   Since no auxiliary data is sent:

   AD_2  (0 bytes)

   The plaintext is defined as the empty string:

   P_2m (0 bytes)

   The Enc_structure is defined as follows: [ "Encrypt0",
   << ID_CRED_R >>, << TH_2, CRED_R >> ], so ID_CRED_R is encoded as a
   CBOR bstr, and the concatenation of the CBOR byte strings TH_2 and
   CRED_R is wrapped in a CBOR bstr.

```
 A_2m =
 [
   "Encrypt0",
   h'A1044105',
   h'5820DECFD64A3667640A0233B04AA8AA91F68956B8A536D0CF8C73A6E8A7C3621E2
   6A401012004215820A3FF263595BEB377D1A0CE1D04DAD2D40966AC6BCB622051B846
   59184D5D9A326C7375626A656374206E616D6560'
 ]
```

   Which encodes to the following byte string to be used as Additional
   Authenticated Data:

```
 A_2m (CBOR-encoded) (105 bytes)
 83 68 45 6e 63 72 79 70 74 30 44 a1 04 41 05 58 58 58 20 de cf d6 4a 36
 67 64 0a 02 33 b0 4a a8 aa 91 f6 89 56 b8 a5 36 d0 cf 8c 73 a6 e8 a7 c3
 62 1e 26 a4 01 01 20 04 21 58 20 a3 ff 26 35 95 be b3 77 d1 a0 ce 1d 04
 da d2 d4 09 66 ac 6b cb 62 20 51 b8 46 59 18 4d 5d 9a 32 6c 73 75 62 6a
 65 63 74 20 6e 61 6d 65 60
```

   info for K_2m is defined as follows:

```
  info for K_2m =
  [
    10,
    h'DECFD64A3667640A0233B04AA8AA91F68956B8A536D0CF8C73A6E8A7C3621E26',
    "K_2m",
    16
  ]
```

   Which as a CBOR encoded data item is:

```
 info for K_2m (CBOR-encoded) (42 bytes)
 84 0a 58 20 de cf d6 4a 36 67 64 0a 02 33 b0 4a a8 aa 91 f6 89 56 b8 a5
 36 d0 cf 8c 73 a6 e8 a7 c3 62 1e 26 64 4b 5f 32 6d 10
```

   From these parameters, K_2m is computed.  Key K_2m is the output of
   HKDF-Expand(PRK_3e2m, info, L), where L is the length of K_2m, so 16
   bytes.

```
   K_2m (16 bytes)
   4e cd ef ba d8 06 81 8b 62 51 b9 d7 86 78 bc 76
```

   info for IV_2m is defined as follows:

```
  info for IV_2m =
  [
    10,
    h'A51C76463E8AE58FD3B8DC5EDE1E27143CC92D223EACD9E5FF6E3FAC876658A5',
    "IV_2m",
    13
  ]
```

   Which as a CBOR encoded data item is:

```
 info for IV_2m (CBOR-encoded) (43 bytes)
 84 0a 58 20 de cf d6 4a 36 67 64 0a 02 33 b0 4a a8 aa 91 f6 89 56 b8 a5
 36 d0 cf 8c 73 a6 e8 a7 c3 62 1e 26 65 49 56 5f 32 6d 0d
```

   From these parameters, IV_2m is computed.  IV_2m is the output of
   HKDF-Expand(PRK_3e2m, info, L), where L is the length of IV_2m, so 13
   bytes.

```
   IV_2m (13 bytes)
   e9 b8 e4 b1 bd 02 f4 9a 82 0d d3 53 4f
```

   Finally, COSE_Encrypt0 is computed from the parameters above.

   *  protected header = CBOR-encoded ID_CRED_R

   *  external_aad = A_2m

   *  empty plaintext = P_2m

   MAC_2 is the 'ciphertext' of the COSE_Encrypt0 with empty plaintext.
   In case of cipher suite 0 the AEAD is AES-CCM truncated to 8 bytes:

   MAC_2 (CBOR unencoded) (8 bytes)
   42 e7 99 78 43 1e 6b 8f

   Since method = 2, Signature_or_MAC_2 is MAC_2:

   Signature_or_MAC_2 (CBOR unencoded) (8 bytes)
   42 e7 99 78 43 1e 6b 8f

   CIPHERTEXT_2 is the ciphertext resulting from XOR between plaintext
   and KEYSTREAM_2 which is derived from TH_2 and the pseudorandom key
   PRK_2e.

   The plaintext is the CBOR Sequence of the items ID_CRED_R and the
   CBOR encoded Signature_or_MAC_2, in this order (AD_2 is empty).

   Note that since ID_CRED_R contains a single 'kid' parameter, i.e.,
   ID_CRED_R = { 4 : kid_R }, only the byte string kid_R is conveyed in
   the plaintext encoded as a bstr_identifier. kid_R is encoded as the
   corresponding integer - 24 (see bstr_identifier in Section 5.1), i.e.
   0x05 = 5, 5 - 24 = -19, and -19 in CBOR encoding is equal to 0x32.

   The plaintext is the following:

   P_2e (CBOR Sequence) (10 bytes)
   32 48 42 e7 99 78 43 1e 6b 8f

   KEYSTREAM_2 = HKDF-Expand( PRK_2e, info, length ), where length is
   the length of the plaintext, so 10.

  info for KEYSTREAM_2 =
  [
    10,
    h'DECFD64A3667640A0233B04AA8AA91F68956B8A536D0CF8C73A6E8A7C3621E26',
    "KEYSTREAM_2",
    10
  ]

   Which as a CBOR encoded data item is:

```
 info for KEYSTREAM_2 (CBOR-encoded) (49 bytes)
 84 0a 58 20 de cf d6 4a 36 67 64 0a 02 33 b0 4a a8 aa 91 f6 89 56 b8 a5
 36 d0 cf 8c 73 a6 e8 a7 c3 62 1e 26 6b 4b 45 59 53 54 52 45 41 4d 5f 32
 0a
```

   From there, KEYSTREAM_2 is computed:

   KEYSTREAM_2 (10 bytes)
   91 b9 ff ba 9b f5 5a d1 57 16

   Using the parameters above, the ciphertext CIPHERTEXT_2 can be
   computed:

   CIPHERTEXT_2 (CBOR unencoded) (10 bytes)
   a3 f1 bd 5d 02 8d 19 cf 3c 99

   message_2 is the CBOR Sequence of data_2 and CIPHERTEXT_2, in this
   order:

   message_2 =
   (
    data_2,
    h'A3F1BD5D028D19CF3C99'
   )

   Which as a CBOR encoded data item is:

```
 message_2 (CBOR Sequence) (46 bytes)
 58 20 52 fb a0 bd c8 d9 53 dd 86 ce 1a b2 fd 7c 05 a4 65 8c 7c 30 af db
 fc 33 01 04 70 69 45 1b af 35 37 4a a3 f1 bd 5d 02 8d 19 cf 3c 99
```

B.2.3.  Message_3

   Since corr equals 1, C_R is not omitted from data_3.

   The Initiator's static Diffie-Hellman key pair is the following:

```
 I (Initiator's private authentication key) (32 bytes)
 2b be a6 55 c2 33 71 c3 29 cf bd 3b 1f 02 c6 c0 62 03 38 37 b8 b5 90 99
 a4 43 6f 66 60 81 b0 8e
```

```
 G_I (Initiator's public authentication key, CBOR unencoded) (32 bytes)
 2c 44 0c c1 21 f8 d7 f2 4c 3b 0e 41 ae da fe 9c aa 4f 4e 7a bb 83 5e c3
 0f 1d e8 8a db 96 ff 71
```

   HKDF SHA-256 is the HKDF used (as defined by cipher suite 0).

   From the Initiator's authentication key and the Responder's ephemeral
   key (see Appendix B.2.2), the ECDH shared secret G_IY is calculated.

 G_IY (ECDH shared secret) (32 bytes)
 cb ff 8c d3 4a 81 df ec 4c b6 5d 9a 57 2e bd 09 64 45 0c 78 56 3d a4 98
 1d 80 d3 6c 8b 1a 75 2a

   PRK_4x3m = HMAC-SHA-256 (PRK_3e2m, G_IY).

 PRK_4x3m (32 bytes)
 02 56 2f 1f 01 78 5c 0a a5 f5 94 64 0c 49 cb f6 9f 72 2e 9e 6c 57 83 7d
 8e 15 79 ec 45 fe 64 7a

   data 3 is equal to C_R.

   data_3 (CBOR Sequence) (1 byte)
   37

   From data_3, CIPHERTEXT_2, and TH_2, compute the input to the
   transcript hash TH_3 = H(TH_2 , CIPHERTEXT_2, data_3), as a CBOR
   Sequence of these 3 data items.

 Input to calculate TH_3 (CBOR Sequence) (46 bytes)
 58 20 de cf d6 4a 36 67 64 0a 02 33 b0 4a a8 aa 91 f6 89 56 b8 a5 36 d0
 cf 8c 73 a6 e8 a7 c3 62 1e 26 4a a3 f1 bd 5d 02 8d 19 cf 3c 99 37

   And from there, compute the transcript hash TH_3 = SHA-256(TH_2 ,
   CIPHERTEXT_2, data_3)

 TH_3 (CBOR unencoded) (32 bytes)
 b6 cd 80 4f c4 b9 d7 ca c5 02 ab d7 7c da 74 e4 1c b0 11 82 d7 cb 8b 84
 db 03 ff a5 83 a3 5f cb

   The initiator's subject name is the empty string:

   Initiator's subject name (text string)
   ""

   And its credential is:

   ID_CRED_I =
   {
     4: h'23'
   }

   ID_CRED_I (4 bytes)
   a1 04 41 23

   CRED_I is the following COSE_Key:

```
{
  1: 1,
  -1: 4,
   -2: h'2C440CC121F8D7F24C3B0E41AEDAFE9CAA4F4E7ABB835EC30F1DE88ADB96FF71',
   "subject name": ""
 }
```

   Which encodes to the following byte string:

```
 CRED_I (54 bytes)
 a4 01 01 20 04 21 58 20 2c 44 0c c1 21 f8 d7 f2 4c 3b 0e 41 ae da fe 9c
 aa 4f 4e 7a bb 83 5e c3 0f 1d e8 8a db 96 ff 71 6c 73 75 62 6a 65 63 74
 20 6e 61 6d 65 60
```

   Since no auxiliary data is exchanged:

   AD_3 (0 bytes)

   The plaintext of the COSE_Encrypt is the empty string:

   P_3m (0 bytes)

   The associated data is the following: [ "Encrypt0", << ID_CRED_I >>,
   << TH_3, CRED_I, ? AD_3 >> ].

```
 A_3m (CBOR-encoded) (105 bytes)
 83 68 45 6e 63 72 79 70 74 30 44 a1 04 41 23 58 58 58 20 b6 cd 80 4f c4
 b9 d7 ca c5 02 ab d7 7c da 74 e4 1c b0 11 82 d7 cb 8b 84 db 03 ff a5 83
 a3 5f cb a4 01 01 20 04 21 58 20 2c 44 0c c1 21 f8 d7 f2 4c 3b 0e 41 ae
 da fe 9c aa 4f 4e 7a bb 83 5e c3 0f 1d e8 8a db 96 ff 71 6c 73 75 62 6a
 65 63 74 20 6e 61 6d 65 60
```

   Info for K_3m is computed as follows:

```
 info for K_3m =
 [
    10,
    h'B6CD804FC4B9D7CAC502ABD77CDA74E41CB01182D7CB8B84DB03FFA583A35FCB',
    "K_3m",
    16
 ]
```

   Which as a CBOR encoded data item is:

```
info for K_3m (CBOR-encoded) (42 bytes)
84 0a 58 20 b6 cd 80 4f c4 b9 d7 ca c5 02 ab d7 7c da 74 e4 1c b0 11 82
d7 cb 8b 84 db 03 ff a5 83 a3 5f cb 64 4b 5f 33 6d 10
```

From these parameters, K_3m is computed.  Key K_3m is the output of HKDF-Expand(PRK_4x3m, info, L), where L is the length of K_2m, so 16 bytes.

```
K_3m (16 bytes)
02 c7 e7 93 89 9d 90 d1 28 28 10 26 96 94 c9 58
```

Nonce IV_3m is the output of HKDF-Expand(PRK_4x3m, info, L), where L = 13 bytes.

Info for IV_3m is defined as follows:

```
info for IV_3m =
[
  10,
  h'B6CD804FC4B9D7CAC502ABD77CDA74E41CB01182D7CB8B84DB03FFA583A35FCB',
  "IV_3m",
  13
]
```

Which as a CBOR encoded data item is:

```
info for IV_3m (CBOR-encoded) (43 bytes)
84 0a 58 20 b6 cd 80 4f c4 b9 d7 ca c5 02 ab d7 7c da 74 e4 1c b0 11 82
d7 cb 8b 84 db 03 ff a5 83 a3 5f cb 65 49 56 5f 33 6d 0d
```

From these parameters, IV_3m is computed:

```
IV_3m (13 bytes)
0d a7 cc 3a 6f 9a b2 48 52 ce 8b 37 a6
```

MAC_3 is the 'ciphertext' of the COSE_Encrypt0 with empty plaintext. In case of cipher suite 0 the AEAD is AES-CCM truncated to 8 bytes:

```
MAC_3 (CBOR unencoded) (8 bytes)
ee 59 8e a6 61 17 dc c3
```

Since method = 3, Signature_or_MAC_3 is MAC_3:

```
Signature_or_MAC_3 (CBOR unencoded) (8 bytes)
ee 59 8e a6 61 17 dc c3
```

Finally, the outer COSE_Encrypt0 is computed.

The plaintext is the CBOR Sequence of the items ID_CRED_I and the
CBOR encoded Signature_or_MAC_3, in this order (AD_3 is empty).

Note that since ID_CRED_I contains a single 'kid' parameter, i.e.,
ID_CRED_I = { 4 : kid_I }, only the byte string kid_I is conveyed in
the plaintext encoded as a bstr_identifier. kid_I is encoded as the
corresponding integer - 24 (see bstr_identifier in Section 5.1), i.e.
0x23 = 35, 35 - 24 = 11, and 11 in CBOR encoding is equal to 0x0b.

P_3ae (CBOR Sequence) (10 bytes)
0b 48 ee 59 8e a6 61 17 dc c3

The Associated data A is the following: Associated data A = [
"Encrypt0", h'', TH_3 ]

A_3ae (CBOR-encoded) (45 bytes)
83 68 45 6e 63 72 79 70 74 30 40 58 20 b6 cd 80 4f c4 b9 d7 ca c5 02 ab
d7 7c da 74 e4 1c b0 11 82 d7 cb 8b 84 db 03 ff a5 83 a3 5f cb

Key K_3ae is the output of HKDF-Expand(PRK_3e2m, info, L).

info is defined as follows:

info for K_3ae =
[
  10,
  h'B6CD804FC4B9D7CAC502ABD77CDA74E41CB01182D7CB8B84DB03FFA583A35FCB',
  "K_3ae",
  16
]

Which as a CBOR encoded data item is:

info for K_3ae (CBOR-encoded) (43 bytes)
84 0a 58 20 b6 cd 80 4f c4 b9 d7 ca c5 02 ab d7 7c da 74 e4 1c b0 11 82
d7 cb 8b 84 db 03 ff a5 83 a3 5f cb 65 4b 5f 33 61 65 10

L is the length of K_3ae, so 16 bytes.

From these parameters, K_3ae is computed:

K_3ae (16 bytes)
6b a4 c8 83 1d e3 ae 23 e9 8e f7 35 08 d0 95 86

Nonce IV_3ae is the output of HKDF-Expand(PRK_3e2m, info, L).

info is defined as follows:

```
  info for IV_3ae =
  [
    10,
    h'97D8AD42334833EB25B960A5EB0704505F89671A0168AA1115FAF92D9E67EF04',
    "IV_3ae",
    13
  ]
```

   Which as a CBOR encoded data item is:

```
 info for IV_3ae (CBOR-encoded) (44 bytes)
 84 0a 58 20 b6 cd 80 4f c4 b9 d7 ca c5 02 ab d7 7c da 74 e4 1c b0 11 82
 d7 cb 8b 84 db 03 ff a5 83 a3 5f cb 66 49 56 5f 33 61 65 0d
```

   L is the length of IV_3ae, so 13 bytes.

   From these parameters, IV_3ae is computed:

```
   IV_3ae (13 bytes)
   6c 6d 0f e1 1e 9a 1a f3 7b 87 84 55 10
```

   Using the parameters above, the 'ciphertext' CIPHERTEXT_3 can be
   computed:

```
   CIPHERTEXT_3 (CBOR unencoded) (18 bytes)
   d5 53 5f 31 47 e8 5f 1c fa cd 9e 78 ab f9 e0 a8 1b bf
```

   From the parameter above, message_3 is computed, as the CBOR Sequence
   of the following items: (C_R, CIPHERTEXT_3).

```
   message_3 =
   (
     -24,
     h'D5535F3147E85F1CFACD9E78ABF9E0A81BBF'
   )
```

   Which encodes to the following byte string:

```
   message_3 (CBOR Sequence) (20 bytes)
   37 52 d5 53 5f 31 47 e8 5f 1c fa cd 9e 78 ab f9 e0 a8 1b bf
```

B.2.4.  OSCORE Security Context Derivation

   From here, the Initiator and the Responder can derive an OSCORE
   Security Context, using the EDHOC-Exporter interface.

From TH_3 and CIPHERTEXT_3, compute the input to the transcript hash TH_4 = H( TH_3, CIPHERTEXT_3 ), as a CBOR Sequence of these 2 data items.

```
Input to calculate TH_4 (CBOR Sequence) (53 bytes)
58 20 b6 cd 80 4f c4 b9 d7 ca c5 02 ab d7 7c da 74 e4 1c b0 11 82 d7 cb
8b 84 db 03 ff a5 83 a3 5f cb 52 d5 53 5f 31 47 e8 5f 1c fa cd 9e 78 ab
f9 e0 a8 1b bf
```

And from there, compute the transcript hash TH_4 = SHA-256(TH_3 , CIPHERTEXT_4)

```
TH_4 (CBOR unencoded) (32 bytes)
7c cf de dc 2c 10 ca 03 56 e9 57 b9 f6 a5 92 e0 fa 74 db 2a b5 4f 59 24
40 96 f9 a2 ac 56 d2 07
```

The Master Secret and Master Salt are derived as follows:

Master Secret = EDHOC-Exporter( "OSCORE Master Secret", 16 ) = EDHOC-KDF(PRK_4x3m, TH_4, "OSCORE Master Secret", 16) = HKDF-Expand( PRK_4x3m, info_ms, 16 )

Master Salt = EDHOC-Exporter( "OSCORE Master Salt", 8 ) = EDHOC-KDF(PRK_4x3m, TH_4, "OSCORE Master Salt", 8) = HKDF-Expand( PRK_4x3m, info_salt, 8 )

info_ms for OSCORE Master Secret is defined as follows:

```
info_ms = [
  10,
  h'7CCFDEDC2C10CA0356E957B9F6A592E0FA74DB2AB54F59244096F9A2AC56D207',
  "OSCORE Master Secret",
  16
]
```

Which as a CBOR encoded data item is:

```
info_ms for OSCORE Master Secret (CBOR-encoded) (58 bytes)
84 0a 58 20 7c cf de dc 2c 10 ca 03 56 e9 57 b9 f6 a5 92 e0 fa 74 db 2a
b5 4f 59 24 40 96 f9 a2 ac 56 d2 07 74 4f 53 43 4f 52 45 20 4d 61 73 74
65 72 20 53 65 63 72 65 74 10
```

info_salt for OSCORE Master Salt is defined as follows:

```
  info_salt = [
    10,
    h'7CCFDEDC2C10CA0356E957B9F6A592E0FA74DB2AB54F59244096F9A2AC56D207',
    "OSCORE Master Salt",
    8
  ]
```

   Which as a CBOR encoded data item is:

```
 info for OSCORE Master Salt (CBOR-encoded) (56 Bytes)
 84 0a 58 20 7c cf de dc 2c 10 ca 03 56 e9 57 b9 f6 a5 92 e0 fa 74 db 2a
 b5 4f 59 24 40 96 f9 a2 ac 56 d2 07 72 4f 53 43 4f 52 45 20 4d 61 73 74
 65 72 20 53 61 6c 74 08
```

   From these parameters, OSCORE Master Secret and OSCORE Master Salt
   are computed:

   OSCORE Master Secret (16 bytes)
   c3 4a 50 6d 0e bf bd 17 03 04 86 13 5f 9c b3 50

   OSCORE Master Salt (8 bytes)
   c2 24 34 9d 9b 34 ca 8c

   The client's OSCORE Sender ID is C_R and the server's OSCORE Sender
   ID is C_I.

   Client's OSCORE Sender ID (1 byte)
   00

   Server's OSCORE Sender ID (1 byte)
   16

   The AEAD Algorithm and the hash algorithm are the application AEAD
   and hash algorithms in the selected cipher suite.

   OSCORE AEAD Algorithm (int)
   10

   OSCORE Hash Algorithm (int)
   -16

Appendix C.  Applicability Template

   This appendix contains an example of an applicability statement, see
   Section 3.7.

   For use of EDHOC in the XX protocol, the following assumptions are
   made on the parameters:

   * METHOD_CORR = 5

     - method = 1 (I uses signature key, R uses static DH key.)

     - corr = 1 (CoAP Token or other transport data enables
       correlation between message_1 and message_2.)

   * EDHOC requests are expected by the server at /app1-edh, no
     Content-Format needed.

   * C_1 = "null" is present to identify message_1

   * CRED_I is an 802.1AR IDevID encoded as a C509 Certificate of type
     0 [I-D.mattsson-cose-cbor-cert-compress].

     - R acquires CRED_I out-of-band, indicated in AD_1

     - ID_CRED_I = {4: h''} is a kid with value empty byte string

   * CRED_R is a COSE_Key of type OKP as specified in Section 3.3.4.

     - The CBOR map has parameters 1 (kty), -1 (crv), and -2
       (x-coordinate).

   * ID_CRED_R = CRED_R

   * AD_1 contains Auxiliary Data of type A (TBD)

   * AD_2 contains Auxiliary Data of type B (TBD)

   * No use of message_4: the application sends protected messages from
     R to I.

   * Auxiliary Data is processed as specified in
     [I-D.selander-ace-ake-authz].

Appendix D.   EDHOC Message Deduplication

   EDHOC by default assumes that message duplication is handled by the
   transport, in this section exemplified with CoAP.

   Deduplication of CoAP messages is described in Section 4.5 of
   [RFC7252].  This handles the case when the same Confirmable (CON)
   message is received multiple times due to missing acknowledgement on
   CoAP messaging layer.  The recommended processing in [RFC7252] is
   that the duplicate message is acknowledged (ACK), but the received
   message is only processed once by the CoAP stack.

Message deduplication is resource demanding and therefore not
supported in all CoAP implementations.  Since EDHOC is targeting
constrained environments, it is desirable that EDHOC can optionally
support transport layers which does not handle message duplication.
Special care is needed to avoid issues with duplicate messages, see
Section 5.2.

The guiding principle here is similar to the deduplication processing
on CoAP messaging layer: a received duplicate EDHOC message SHALL NOT
result in a response consisting of another instance of the next EDHOC
message.  The result MAY be that a duplicate EDHOC response is sent,
provided it is still relevant with respect the current protocol
state.  In any case, the received message MUST NOT be processed more
than once by the same EDHOC instance.  This is called "EDHOC message
deduplication".

An EDHOC implementation MAY store the previously sent EDHOC message
to be able to resend it.  An EDHOC implementation MAY keep the
protocol state to be able to recreate the previously sent EDHOC
message and resend it.  The previous message or protocol state MUST
NOT be kept longer than what is required for retransmission, for
example, in the case of CoAP transport, no longer than the
EXCHANGE_LIFETIME (see Section 4.8.2 of [RFC7252]).

Note that the requirements in Section 5.2 still apply because
duplicate messages are not processed by the EDHOC state machine:

*  EDHOC messages SHALL be processed according to the current
   protocol state.

*  Different instances of the same message MUST NOT be processed in
   one protocol instance.

Appendix E.  Change Log

   Main changes:

*  From -05 to -06:

   -  New section 5.2 "Message Processing Outline"

   -  Optional inital byte C_1 = null in message_1

   -  New format of error messages, table of error codes, IANA
      registry

   -  Change of recommendation transport of error in CoAP

      - Merge of content in 3.7 and appendix C into new section 3.7
        "Applicability Statement"

      - Requiring use of deterministic CBOR

      - New section on message deduplication

      - New appendix containin all CDDL definitions

      - New appendix with change log

      - Removed section "Other Documents Referncing EDHOC"

      - Clarifications based on review comments

   *  From -04 to -05:

      - EDHOC-Rekey-FS -> EDHOC-KeyUpdate

      - Clarification of cipher suite negotiation

      - Updated security considerations

      - Updated test vectors

      - Updated applicability statement template

   *  From -03 to -04:

      - Restructure of section 1

      - Added references to C509 Certificates

      - Change in CIPHERTEXT_2 -> plaintext XOR KEYSTREAM_2 (test
        vector not updated)

      - "K_2e", "IV_2e" -> KEYSTREAM_2

      - Specified optional message 4

      - EDHOC-Exporter-FS -> EDHOC-Rekey-FS

      - Less constrained devices SHOULD implement both suite 0 and 2

      - Clarification of error message

      - Added exporter interface test vector

   *  From -02 to -03:

      -  Rearrangements of section 3 and beginning of section 4

      -  Key derivation new section 4

      -  Cipher suites 4 and 5 added

      -  EDHOC-EXPORTER-FS - generate a new PRK_4x3m from an old one

      -  Change in CIPHERTEXT_2 -> COSE_Encrypt0 without tag (no change
         to test vector)

      -  Clarification of error message

      -  New appendix C applicability statement

   *  From -01 to -02:

      -  New section 1.2 Use of EDHOC

      -  Clarification of identities

      -  New section 4.3 clarifying bstr_identifier

      -  Updated security considerations

      -  Updated text on cipher suite negotiation and key confirmation

      -  Test vector for static DH

   *  From -00 to -01:

      -  Removed PSK method

      -  Removed references to certificate by value

Acknowledgments

      indebted to Jim Schaad for his continuous reviewing and
      implementation of different versions of the draft.

Authors' Addresses

   Göran Selander
   Ericsson AB

   Email: goran.selander@ericsson.com


   John Preuß Mattsson
   Ericsson AB

   Email: john.mattsson@ericsson.com


   Francesca Palombini
   Ericsson AB

   Email: francesca.palombini@ericsson.com