ï»¿

Internet Engineering Task Force (IETF)
Request for Comments: 9052
STD: 96
Obsoletes: 8152
Category: Standards Track
ISSN: 2070-1721

J. Schaad
August Cellars

August 2022

CBOR Object Signing and Encryption (COSE): Structures and Process

Abstract

Concise Binary Object Representation (CBOR) is a data format designed for small code size and small message size. There is a need to be able to define basic security services for this data format. This document defines the CBOR Object Signing and Encryption (COSE) protocol. This specification describes how to create and process signatures, message authentication codes, and encryption using CBOR for serialization. This specification additionally describes how to represent cryptographic keys using CBOR.

This document, along with RFC 9053, obsoletes RFC 8152.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 7841.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9052.

Table of Contents

1. Introduction

There has been an increased focus on small, constrained devices that make up the Internet of Things (IoT). One of the standards that has come out of this process is "Concise Binary Object Representation (CBOR)" [STD94]. CBOR extended the data model of JavaScript Object Notation (JSON) [STD90] by allowing for binary data, among other changes. CBOR has been adopted by several of the IETF working groups dealing with the IoT world as their method of encoding data structures. CBOR was designed specifically to be small in terms of both messages transported and implementation size and to have a schema-free decoder. A need exists to provide message security services for IoT, and using CBOR as the message-encoding format makes sense.

The JOSE Working Group produced a set of documents [RFC7515] [RFC7516] [RFC7517] [RFC7518] that specified how to process encryption, signatures, and Message Authentication Code (MAC) operations and how to encode keys using JSON. This document defines the CBOR Object Signing and Encryption (COSE) standard, which does the same thing for the CBOR encoding format. This document is combined with [RFC9053], which provides an initial set of algorithms. While there is a strong attempt to keep the flavor of the original JSON Object Signing and Encryption (JOSE) documents, two considerations are taken into account:

* CBOR has capabilities that are not present in JSON and are appropriate to use. One example of this is the fact that CBOR has a method of encoding binary data directly without first converting it into a base64-encoded text string.

* COSE is not a direct copy of the JOSE specification. In the process of creating COSE, decisions that were made for JOSE were re-examined. In many cases, different results were decided on, as the criteria were not always the same.

This document contains:

* The description of the structure for the CBOR objects that are transmitted over the wire. Two objects each are defined for encryption, signing, and message authentication. One object is defined for transporting keys and one for transporting groups of keys.

* The procedures used to build the inputs to the cryptographic functions required for each of the structures.

* A set of attributes that apply to the different security objects.

This document does not contain the rules and procedures for using specific cryptographic algorithms. Details on specific algorithms can be found in [RFC9053] and [RFC8230]. Details for additional algorithms are expected to be defined in future documents.

COSE was initially designed as part of a solution to provide security to Constrained RESTful Environments (CoRE), and this is done using [RFC8613] and [CORE-GROUPCOMM]. However, COSE is not restricted to just these cases and can be used in any place where one would consider either JOSE or Cryptographic Message Syntax (CMS) [RFC5652] for the purpose of providing security services. COSE, like JOSE and CMS, is only for use in store-and-forward or offline protocols. The use of COSE in online protocols needing encryption requires that an online key establishment process be done before sending objects back and forth. Any application that uses COSE for security services first needs to determine what security services are required and then select the appropriate COSE structures and cryptographic algorithms based on those needs. Section 10 provides additional information on what applications need to specify when using COSE.

One feature that is present in CMS that is not present in this standard is a digest structure. This omission is deliberate. It is better for the structure to be defined in each protocol as different

protocols will want to include a different set of fields as part of the structure. While an algorithm identifier and the digest value are going to be common to all applications, the two values may not always be adjacent, as the algorithm could be defined once with multiple values. Applications may additionally want to define additional data fields as part of the structure. One such application-specific element would be to include a URI or other pointer to where the data that is being hashed can be obtained. [RFC9054] contains one such possible structure and defines a set of digest algorithms.

During the process of advancing COSE to Internet Standard, it was noticed that the description of the security properties of countersignatures was incorrect for the COSE_Sign1 structure. Since the security properties that were described -- those of a true countersignature -- were those that the working group desired, the decision was made to remove all of the countersignature text from this document and create a new document [COSE-COUNTERSIGN] to both deprecate the old countersignature algorithm and header parameters and define a new algorithm and header parameters with the desired security properties.

1.1. Requirements Terminology

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

1.2. Changes from RFC 8152

* Split the original document into this document and [RFC9053].

* Added some text describing why there is no digest structure defined by COSE.

* Made text clarifications and changes in terminology.

* Removed all of the details relating to countersignatures and placed them in [COSE-COUNTERSIGN].

1.3. Design Changes from JOSE

* A single overall message structure has been defined so that encrypted, signed, and MACed messages can easily be identified and still have a consistent view.

* Signed messages distinguish between the protected and unprotected header parameters that relate to the content and those that relate to the signature.

* MACed messages are separated from signed messages.

* MACed messages have the ability to use the same set of recipient algorithms as enveloped messages for obtaining the MAC authentication key.

* Binary encodings are used, rather than base64url encodings, to encode binary data.

* The authentication tag for encryption algorithms has been combined with the ciphertext.

* The set of cryptographic algorithms has been expanded in some directions and trimmed in others.

1.4. CDDL Grammar for CBOR Data Structures

When COSE was originally written, the Concise Data Definition Language (CDDL) [RFC8610] had not yet been published in an RFC, so it

could not be used as the data description language to normatively describe the CBOR data structures employed by COSE. For that reason, the CBOR data objects defined here are described in prose. Additional (non-normative) descriptions of the COSE data objects are provided in a subset of CDDL, described below.

This document was developed by first working on the grammar and then developing the prose to go with it. An artifact of this is that the prose was written using the primitive-type strings defined by Concise Data Definition Language (CDDL) [RFC8610]. In this specification, the following primitive types are used:

any: A nonspecific value that permits all CBOR values to be placed here.

bool: A boolean value (true: major type 7, value 21; false: major type 7, value 20).

bstr: Byte string (major type 2).

int: An unsigned integer or a negative integer.

nil: A null value (major type 7, value 22).

nint: A negative integer (major type 1).

tstr: A UTF-8 text string (major type 3).

uint: An unsigned integer (major type 0).

Three syntaxes from CDDL appear in this document as shorthand. These are:

FOO / BAR: Indicates that either FOO or BAR can appear here.

[+ FOO]: Indicates that the type FOO appears one or more times in an array.

* FOO: Indicates that the type FOO appears zero or more times.

Two of the constraints defined by CDDL are also used in this document. These are:

type1 .cbor type2: Indicates that the contents of type1, usually bstr, contains a value of type2.

type1 .size integer: Indicates that the contents of type1 is integer bytes long.

As well as the prose description, a grammar for the CBOR data structures is presented in the subset of CDDL described previously. The CDDL grammar is informational; the prose description is normative.

The collected CDDL can be extracted from the XML version of this document via the XPath expression below. (Depending on the XPath evaluator one is using, it may be necessary to deal with > as an entity.)

```
//sourcecode[@type='cddl']/text()
```

CDDL expects the initial nonterminal symbol to be the first symbol in the file. For this reason, the first fragment of CDDL is presented here.

```
start = COSE_Messages / COSE_Key / COSE_KeySet / Internal_Types

; This is defined to make the tool quieter:
Internal_Types = Sig_structure / Enc_structure / MAC_structure
```

The nonterminal Internal_Types is defined for dealing with the

automated validation tools used during the writing of this document. It references those nonterminals that are used for security computations but are not emitted for transport.

1.5. CBOR-Related Terminology

In JSON, maps are called objects and only have one kind of map key: a text string. In COSE, we use text strings, negative integers, and unsigned integers as map keys. The integers are used for compactness of encoding and easy comparison. The inclusion of text strings allows for an additional range of short encoded values to be used as well. Since the word "key" is mainly used in its other meaning, as a cryptographic key, we use the term "label" for this usage as a map key.

In a CBOR map defined by this specification, the presence a label that is neither a text string nor an integer is an error. Applications can either fail processing or process messages by ignoring incorrect labels; however, they MUST NOT create messages with incorrect labels.

A CDDL grammar fragment defines the nonterminal "label", as in the previous paragraph, and "values", which permits any value to be used.

```
label = int / tstr
values = any
```

1.6. Document Terminology

In this document, we use the following terminology:

Byte: A synonym for octet.

Constrained Application Protocol (CoAP): A specialized web transfer protocol for use in constrained systems. It is defined in [RFC7252].

Authenticated Encryption (AE) algorithms [RFC5116]: Encryption algorithms that provide an authentication check of the contents along with the encryption service. An example of an AE algorithm used in COSE is AES Key Wrap [RFC3394]. These algorithms are used for key encryption, but Authenticated Encryption with Associated Data (AEAD) algorithms would be preferred.

AEAD algorithms [RFC5116]: Encryption algorithms that provide the same authentication service of the content as AE algorithms do, and also allow associated data that is not part of the encrypted body to be included in the authentication service. An example of an AEAD algorithm used in COSE is AES-GCM [RFC5116]. These algorithms are used for content encryption and can be used for key encryption as well.

"Context" is used throughout the document to represent information that is not part of the COSE message. Information that is part of the context can come from several different sources, including protocol interactions, associated key structures, and program configuration. The context to use can be implicit, identified using the "kid context" header parameter defined in [RFC8613], or identified by a protocol-specific identifier. Context should generally be included in the cryptographic construction; for more details, see Section 4.3.

The term "byte string" is used for sequences of bytes, while the term "text string" is used for sequences of characters.

2. Basic COSE Structure

The COSE object structure is designed so that there can be a large amount of common code when parsing and processing the different types of security messages. All of the message structures are built on the CBOR array type. The first three elements of the array always

contain the same information:

1. The protected header parameters, encoded and wrapped in a bstr.

2. The unprotected header parameters as a map.

3. The content of the message. The content is either the plaintext or the ciphertext, as appropriate. The content may be detached (i.e., transported separately from the COSE structure), but the location is still used. The content is wrapped in a bstr when present and is a nil value when detached.

Elements after this point are dependent on the specific message type.

COSE messages are built using the concept of layers to separate different types of cryptographic concepts. As an example of how this works, consider the COSE_Encrypt message (Section 5.1). This message type is broken into two layers: the content layer and the recipient layer. The content layer contains the encrypted plaintext and information about the encrypted message. The recipient layer contains the encrypted content encryption key (CEK) and information about how it is encrypted, for each recipient. A single-layer version of the encryption message COSE_Encrypt0 (Section 5.2) is provided for cases where the CEK is preshared.

Identification of which type of message has been presented is done by the following methods:

1. The specific message type is known from the context. This may be defined by a marker in the containing structure or by restrictions specified by the application protocol.

2. The message type is identified by a CBOR tag. Messages with a CBOR tag are known in this specification as tagged messages, while those without the CBOR tag are known as untagged messages. This document defines a CBOR tag for each of the message structures. These tags can be found in Table 1.

3. When a COSE object is carried in a media type of "application/cose", the optional parameter "cose-type" can be used to identify the embedded object. The parameter is OPTIONAL if the tagged version of the structure is used. The parameter is REQUIRED if the untagged version of the structure is used. The value to use with the parameter for each of the structures can be found in Table 1.

4. When a COSE object is carried as a CoAP payload, the CoAP Content-Format Option can be used to identify the message content. The CoAP Content-Format values can be found in Table 2. The CBOR tag for the message structure is not required, as each security message is uniquely identified.

CBOR Tag	cose-type	Data Item	Semantics
98	cose-sign	COSE_Sign	COSE Signed Data Object
18	cose-sign1	COSE_Sign1	COSE Single Signer Data Object
96	cose-encrypt	COSE_Encrypt	COSE Encrypted Data Object
16	cose-encrypt0	COSE_Encrypt0	COSE Single Recipient Encrypted Data Object
97	cose-mac	COSE_Mac	COSE MACed Data Object
17	cose-mac0	COSE_Mac0	COSE Mac w/o

			Recipients Object

Table 1: COSE Message Identification

Media Type	Encoding	ID	Reference
application/cose; cose-type="cose-sign"		98	RFC 9052
application/cose; cose-type="cose-sign1"		18	RFC 9052
application/cose; cose-type="cose-encrypt"		96	RFC 9052
application/cose; cose-type="cose-encrypt0"		16	RFC 9052
application/cose; cose-type="cose-mac"		97	RFC 9052
application/cose; cose-type="cose-mac0"		17	RFC 9052
application/cose-key		101	RFC 9052
application/cose-key-set		102	RFC 9052

Table 2: CoAP Content-Formats for COSE

The following CDDL fragment identifies all of the top messages defined in this document. Separate nonterminals are defined for the tagged and untagged versions of the messages.

```
COSE_Messages = COSE_Untagged_Message / COSE_Tagged_Message

COSE_Untagged_Message = COSE_Sign / COSE_Sign1 /
    COSE_Encrypt / COSE_Encrypt0 /
    COSE_Mac / COSE_Mac0

COSE_Tagged_Message = COSE_Sign_Tagged / COSE_Sign1_Tagged /
    COSE_Encrypt_Tagged / COSE_Encrypt0_Tagged /
    COSE_Mac_Tagged / COSE_Mac0_Tagged
```

3. Header Parameters

The structure of COSE has been designed to have two buckets of information that are not considered to be part of the payload itself, but are used for holding information about content, algorithms, keys, or evaluation hints for the processing of the layer. These two buckets are available for use in all of the structures except for keys. While these buckets are present, they may not always be usable in all instances. For example, while the protected bucket is defined as part of the recipient structure, some of the algorithms used for recipient structures do not provide for authenticated data. If this is the case, the protected bucket is left empty.

Both buckets are implemented as CBOR maps. The map key is a "label" (Section 1.5). The value portion is dependent on the definition for the label. Both maps use the same set of label/value pairs. The integer and text-string values for labels have been divided into several sections, including a standard range, a private use range, and a range that is dependent on the algorithm selected. The defined labels can be found in the "COSE Header Parameters" IANA registry (Section 11.1).

The two buckets are:

protected: Contains parameters about the current layer that are cryptographically protected. This bucket MUST be empty if it is not going to be included in a cryptographic computation. This bucket is encoded in the message as a binary object. This value is obtained by CBOR encoding the protected map and wrapping it in a bstr object. Senders SHOULD encode a zero-length map as a zero-length byte string rather than as a zero-length map (encoded as h'a0'). The zero-length byte string encoding is preferred, because it is both shorter and the version used in the serialization structures for cryptographic computation. Recipients MUST accept both a zero-length byte string and a zero-length map encoded in a byte string.

Wrapping the encoding with a byte string allows the protected map to be transported with a greater chance that it will not be altered accidentally in transit. (Badly behaved intermediates could decode and re-encode, but this will result in a failure to verify unless the re-encoded byte string is identical to the decoded byte string.) This avoids the problem of all parties needing to be able to do a common canonical encoding of the map for input to cryptographic operations.

unprotected: Contains parameters about the current layer that are not cryptographically protected.

Only header parameters that deal with the current layer are to be placed at that layer. As an example of this, the header parameter "content type" describes the content of the message being carried in the message. As such, this header parameter is placed only in the content layer and is not placed in the recipient or signature layers. In principle, one should be able to process any given layer without reference to any other layer. With the exception of the COSE_Sign structure, the only data that needs to cross layers is the cryptographic key.

The buckets are present in all of the security objects defined in this document. The fields, in order, are the "protected" bucket (as a CBOR "bstr" type) and then the "unprotected" bucket (as a CBOR "map" type). The presence of both buckets is required. The header parameters that go into the buckets come from the IANA "COSE Header Parameters" registry (Section 11.1). Some header parameters are defined in the next section.

Labels in each of the maps MUST be unique. When processing messages, if a label appears multiple times, the message MUST be rejected as malformed. Applications SHOULD verify that the same label does not occur in both the protected and unprotected header parameters. If the message is not rejected as malformed, attributes MUST be obtained from the protected bucket, and only if an attribute is not found in the protected bucket can that attribute be obtained from the unprotected bucket.

The following CDDL fragment represents the two header-parameter buckets. A group "Headers" is defined in CDDL that represents the two buckets in which attributes are placed. This group is used to provide these two fields consistently in all locations. A type is also defined that represents the map of common header parameters.

```
Headers = (
    protected : empty_or_serialized_map,
    unprotected : header_map
)

header_map = {
    Generic_Headers,
    * label => values
}

empty_or_serialized_map = bstr .cbor header_map / bstr .size 0
```

3.1. Common COSE Header Parameters

This section defines a set of common header parameters. A summary of these header parameters can be found in Table 3. This table should be consulted to determine the value of the label and the type of the value.

The set of header parameters defined in this section is as follows:

alg: This header parameter is used to indicate the algorithm used for the security processing. This header parameter MUST be authenticated where the ability to do so exists. This support is provided by AEAD algorithms or construction (e.g., COSE_Sign and COSE_Mac0). This authentication can be done either by placing the header parameter in the protected-header-parameters bucket or as part of the externally supplied data (Section 4.3). The value is taken from the "COSE Algorithms" registry (see [COSE.Algorithms]).

crit: This header parameter is used to indicate which protected header parameters an application that is processing a message is required to understand. Header parameters defined in this document do not need to be included, as they should be understood by all implementations. Additionally, the header parameter "counter signature" (label 7) defined by [RFC8152] must be understood by new implementations, to remain compatible with senders that adhere to that document and assume all implementations will understand it. When present, the "crit" header parameter MUST be placed in the protected-header-parameters bucket. The array MUST have at least one value in it.

Not all header-parameter labels need to be included in the "crit" header parameter. The rules for deciding which header parameters are placed in the array are:

* Integer labels in the range of 0 to 7 SHOULD be omitted.

* Integer labels in the range -1 to -128 can be omitted. Algorithms can assign labels in this range where the ability to process the content of the label is considered to be core to implementing the algorithm. Algorithms can assign labels outside of this range and include them in the "crit" header parameter when the ability to process the content of the label is not considered to be core functionality of the algorithm but does need to be understood to correctly process this instance. Integer labels in the range -129 to -65536 SHOULD be included, as these would be less common header parameters that might not be generally supported.

* Labels for header parameters required for an application MAY be omitted. Applications should have a statement declaring whether or not the label can be omitted.

The header parameters indicated by "crit" can be processed by either the security-library code or an application using a security library; the only requirement is that the header parameter is processed. If the "crit" value list includes a label for which the header parameter is not in the protected-header-parameters bucket, this is a fatal error in processing the message.

content type: This header parameter is used to indicate the content type of the data in the "payload" or "ciphertext" field. Integers are from the "CoAP Content-Formats" IANA registry table [COAP.Formats]. Text values follow the syntax of "<type-name>/<subtype-name>", where <type-name> and <subtype-name> are defined in Section 4.2 of [RFC6838]. Leading and trailing whitespace is not permitted. Textual content type values, along with parameters and subparameters, can be located using the IANA "Media Types" registry. Applications SHOULD provide this header parameter if the content structure is potentially ambiguous.

kid: This header parameter identifies one piece of data that can be

used as input to find the needed cryptographic key. The value of this header parameter can be matched against the "kid" member in a COSE_Key structure. Other methods of key distribution can define an equivalent field to be matched. Applications MUST NOT assume that "kid" values are unique. There may be more than one key with the same "kid" value, so all of the keys associated with this "kid" may need to be checked. The internal structure of "kid" values is not defined and cannot be relied on by applications. Key identifier values are hints about which key to use. This is not a security-critical field. For this reason, it can be placed in the unprotected-header-parameters bucket.

IV: This header parameter holds the Initialization Vector (IV) value. For some symmetric encryption algorithms, this may be referred to as a nonce. The IV can be placed in the unprotected bucket, since for AE and AEAD algorithms, modifying the IV will cause the decryption to fail.

Partial IV: This header parameter holds a part of the IV value. When using the COSE_Encrypt0 structure, a portion of the IV can be part of the context associated with the key (Context IV), while a portion can be changed with each message (Partial IV). This field is used to carry a value that causes the IV to be changed for each message. The Partial IV can be placed in the unprotected bucket, as modifying the value will cause the decryption to yield plaintext that is readily detectable as garbled. The "Initialization Vector" and "Partial Initialization Vector" header parameters MUST NOT both be present in the same security layer.

The message IV is generated by the following steps:

1. Left-pad the Partial IV with zeros to the length of IV (determined by the algorithm).

2. XOR the padded Partial IV with the Context IV.

Name	Label	Value Type	Value Registry	Description
alg	1	int / tstr	COSE Algorithms registry	Cryptographic algorithm to use
crit	2	[+ label]	COSE Header Parameters registry	Critical header parameters to be understood
content type	3	tstr / uint	CoAP Content-Formats or Media Types registries	Content type of the payload
kid	4	bstr		Key identifier
IV	5	bstr		Full Initialization Vector
Partial IV	6	bstr		Partial Initialization Vector

Table 3: Common Header Parameters

The CDDL fragment that represents the set of header parameters defined in this section is given below. Each of the header parameters is tagged as optional, because they do not need to be in every map; header parameters required in specific maps are discussed above.

```
Generic_Headers = (
```

```
    ? 1 => int / tstr,  ; algorithm identifier
    ? 2 => [+label],    ; criticality
    ? 3 => tstr / int,  ; content type
    ? 4 => bstr,        ; key identifier
    ? ( 5 => bstr //    ; IV
        6 => bstr )     ; Partial IV
)
```

4. Signing Objects

COSE supports two different signature structures. COSE_Sign allows for one or more signatures to be applied to the same content. COSE_Sign1 is restricted to a single signer. The structures cannot be converted between each other; as the signature computation includes a parameter identifying which structure is being used, the converted structure will fail signature validation.

4.1. Signing with One or More Signers

The COSE_Sign structure allows for one or more signatures to be applied to a message payload. Header parameters relating to the content and header parameters relating to the signature are carried along with the signature itself. These header parameters may be authenticated by the signature, or just be present. An example of a header parameter about the content is the content type header parameter. An example of a header parameter about the signature would be the algorithm and key used to create the signature.

[RFC5652] indicates that:

> When more than one signature is present, the successful validation of one signature associated with a given signer is usually treated as a successful signature by that signer. However, there are some application environments where other rules are needed. An application that employs a rule other than one valid signature for each signer must specify those rules. Also, where simple matching of the signer identifier is not sufficient to determine whether the signatures were generated by the same signer, the application specification must describe how to determine which signatures were generated by the same signer. Support of different communities of recipients is the primary reason that signers choose to include more than one signature.

For example, the COSE_Sign structure might include signatures generated with the Edwards-curve Digital Signature Algorithm (EdDSA) [RFC8032] and the Elliptic Curve Digital Signature Algorithm (ECDSA) [DSS]. This allows recipients to verify the signature associated with one algorithm or the other. More detailed information on multiple signature evaluations can be found in [RFC5752].

The signature structure can be encoded as either tagged or untagged, depending on the context it will be used in. A tagged COSE_Sign structure is identified by the CBOR tag 98. The CDDL fragment that represents this is:

```
COSE_Sign_Tagged = #6.98(COSE_Sign)
```

A COSE Signed Message is defined in two parts. The CBOR object that carries the body and information about the message is called the COSE_Sign structure. The CBOR object that carries the signature and information about the signature is called the COSE_Signature structure. Examples of COSE Signed Messages can be found in Appendix C.1.

The COSE_Sign structure is a CBOR array. The fields of the array, in order, are:

protected: This is as described in Section 3.

unprotected: This is as described in Section 3.

payload: This field contains the serialized content to be signed. If the payload is not present in the message, the application is required to supply the payload separately. The payload is wrapped in a bstr to ensure that it is transported without changes. If the payload is transported separately ("detached content"), then a nil CBOR object is placed in this location, and it is the responsibility of the application to ensure that it will be transported without changes.

Note: When a signature with a message recovery algorithm is used (Section 8.1), the maximum number of bytes that can be recovered is the length of the original payload. The size of the encoded payload is reduced by the number of bytes that will be recovered. If all of the bytes of the original payload are consumed, then the transmitted payload is encoded as a zero-length byte string rather than as being absent.

signatures: This field is an array of signatures. Each signature is represented as a COSE_Signature structure.

The CDDL fragment that represents the above text for COSE_Sign follows.

```
COSE_Sign = [
    Headers,
    payload : bstr / nil,
    signatures : [+ COSE_Signature]
]
```

The COSE_Signature structure is a CBOR array. The fields of the array, in order, are:

protected: This is as described in Section 3.

unprotected: This is as described in Section 3.

signature: This field contains the computed signature value. The type of the field is a bstr. Algorithms MUST specify padding if the signature value is not a multiple of 8 bits.

The CDDL fragment that represents the above text for COSE_Signature follows.

```
COSE_Signature =  [
    Headers,
    signature : bstr
]
```

4.2. Signing with One Signer

The COSE_Sign1 signature structure is used when only one signature is going to be placed on a message. The header parameters dealing with the content and the signature are placed in the same pair of buckets, rather than having the separation of COSE_Sign.

The structure can be encoded as either tagged or untagged depending on the context it will be used in. A tagged COSE_Sign1 structure is identified by the CBOR tag 18. The CDDL fragment that represents this is:

```
COSE_Sign1_Tagged = #6.18(COSE_Sign1)
```

The CBOR object that carries the body, the signature, and the information about the body and signature is called the COSE_Sign1 structure. Examples of COSE_Sign1 messages can be found in Appendix C.2.

The COSE_Sign1 structure is a CBOR array. The fields of the array, in order, are:

protected: This is as described in Section 3.

unprotected: This is as described in Section 3.

payload: This is as described in Section 4.1.

signature: This field contains the computed signature value. The type of the field is a bstr.

The CDDL fragment that represents the above text for COSE_Sign1 follows.

```
COSE_Sign1 = [
    Headers,
    payload : bstr / nil,
    signature : bstr
]
```

4.3. Externally Supplied Data

One of the features offered in COSE is the ability for applications to provide additional data that is to be authenticated but is not carried as part of the COSE object. The primary reason for supporting this can be seen by looking at the CoAP message structure [RFC7252], where the facility exists for options to be carried before the payload. Examples of data that can be placed in this location would be the CoAP code or CoAP options. If the data is in the headers of the CoAP message, then it is available for proxies to help in performing proxying operations. For example, the Accept option can be used by a proxy to determine if an appropriate value is in the proxy's cache. The sender can use the additional-data functionality to enable detection of any changes to the set of Accept values made by a proxy or an attacker. By including the field in the externally supplied data, any subsequent modification will cause the server processing of the message to result in failure.

This document describes the process for using a byte array of externally supplied authenticated data; the method of constructing the byte array is a function of the application. Applications that use this feature need to define how the externally supplied authenticated data is to be constructed. Such a construction needs to take into account the following issues:

* If multiple items are included, applications need to ensure that the same byte string cannot be produced if there are different inputs. An example of how the problematic scenario could arise would be by concatenating the text strings "AB" and "CDE" or by concatenating the text strings "ABC" and "DE". This is usually addressed by making fields a fixed width and/or encoding the length of the field as part of the output. Using options from CoAP [RFC7252] as an example, these fields use a TLV structure so they can be concatenated without any problems.

* If multiple items are included, an order for the items needs to be defined. Using options from CoAP as an example, an application could state that the fields are to be ordered by the option number.

* Applications need to ensure that the byte string is going to be the same on both sides. Using options from CoAP might give a problem if the same relative numbering is kept. An intermediate node could insert or remove an option, changing how the relative numbering is done. An application would need to specify that the relative number must be re-encoded to be relative only to the options that are in the external data.

4.4. Signing and Verification Process

In order to create a signature, a well-defined byte string is needed. The Sig_structure is used to create the canonical form. This signing and verification process takes in the body information (COSE_Sign or COSE_Sign1), the signer information (COSE_Signature), and the

application data (external source). A Sig_structure is a CBOR array. The fields of the Sig_structure, in order, are:

1. A context text string identifying the context of the signature. The context text string is:

   "Signature" for signatures using the COSE_Signature structure.

   "Signature1" for signatures using the COSE_Sign1 structure.

2. The protected attributes from the body structure, encoded in a bstr type. If there are no protected attributes, a zero-length byte string is used.

3. The protected attributes from the signer structure, encoded in a bstr type. If there are no protected attributes, a zero-length byte string is used. This field is omitted for the COSE_Sign1 signature structure.

4. The externally supplied data from the application, encoded in a bstr type. If this field is not supplied, it defaults to a zero-length byte string. (See Section 4.3 for application guidance on constructing this field.)

5. The payload to be signed, encoded in a bstr type. The full payload is used here, independent of how it is transported.

The CDDL fragment that describes the above text is:

```
Sig_structure = [
    context : "Signature" / "Signature1",
    body_protected : empty_or_serialized_map,
    ? sign_protected : empty_or_serialized_map,
    external_aad : bstr,
    payload : bstr
]
```

How to compute a signature:

1. Create a Sig_structure and populate it with the appropriate fields.

2. Create the value ToBeSigned by encoding the Sig_structure to a byte string, using the encoding described in Section 9.

3. Call the signature creation algorithm, passing in K (the key to sign with), alg (the algorithm to sign with), and ToBeSigned (the value to sign).

4. Place the resulting signature value in the correct location. This is the "signature" field of the COSE_Signature or COSE_Sign1 structure.

The steps for verifying a signature are:

1. Create a Sig_structure and populate it with the appropriate fields.

2. Create the value ToBeSigned by encoding the Sig_structure to a byte string, using the encoding described in Section 9.

3. Call the signature verification algorithm, passing in K (the key to verify with), alg (the algorithm used to sign with), ToBeSigned (the value to sign), and sig (the signature to be verified).

In addition to performing the signature verification, the application performs the appropriate checks to ensure that the key is correctly paired with the signing identity and that the signing identity is authorized before performing actions.

5. Encryption Objects

COSE supports two different encryption structures. COSE_Encrypt0 is used when a recipient structure is not needed because the key to be used is known implicitly. COSE_Encrypt is used the rest of the time. This includes cases where there are multiple recipients or a recipient algorithm other than direct (i.e., preshared secret) is used.

5.1. Enveloped COSE Structure

The enveloped structure allows for one or more recipients of a message. There are provisions for header parameters about the content and header parameters about the recipient information to be carried in the message. The protected header parameters associated with the content are authenticated by the content encryption algorithm. The protected header parameters associated with the recipient (when the algorithm supports it) are authenticated by the recipient algorithm. Examples of header parameters about the content are the type of the content and the content encryption algorithm. Examples of header parameters about the recipient are the recipient's key identifier and the recipient's encryption algorithm.

The same techniques and nearly the same structure are used for encrypting both the plaintext and the keys. This is different from the approach used by both "Cryptographic Message Syntax (CMS)" [RFC5652] and "JSON Web Encryption (JWE)" [RFC7516], where different structures are used for the content layer and the recipient layer. Two structures are defined: COSE_Encrypt to hold the encrypted content and COSE_recipient to hold the encrypted keys for recipients. Examples of enveloped messages can be found in Appendix C.3.

The COSE_Encrypt structure can be encoded as either tagged or untagged, depending on the context it will be used in. A tagged COSE_Encrypt structure is identified by the CBOR tag 96. The CDDL fragment that represents this is:

```
COSE_Encrypt_Tagged = #6.96(COSE_Encrypt)
```

The COSE_Encrypt structure is a CBOR array. The fields of the array, in order, are:

protected: This is as described in Section 3.

unprotected: This is as described in Section 3.

ciphertext: This field contains the ciphertext, encoded as a bstr. If the ciphertext is to be transported independently of the control information about the encryption process (i.e., detached content), then the field is encoded as a nil value.

recipients: This field contains an array of recipient information structures. The type for the recipient information structure is a COSE_recipient.

The CDDL fragment that corresponds to the above text is:

```
COSE_Encrypt = [
    Headers,
    ciphertext : bstr / nil,
    recipients : [+COSE_recipient]
]
```

The COSE_recipient structure is a CBOR array. The fields of the array, in order, are:

protected: This is as described in Section 3.

unprotected: This is as described in Section 3.

ciphertext: This field contains the encrypted key, encoded as a

bstr. All encoded keys are symmetric keys; the binary value of the key is the content. If there is not an encrypted key, then this field is encoded as a nil value.

recipients: This field contains an array of recipient information structures. The type for the recipient information structure is a COSE_recipient (an example of this can be found in Appendix B). If there are no recipient information structures, this element is absent.

The CDDL fragment that corresponds to the above text for COSE_recipient is:

```
COSE_recipient = [
    Headers,
    ciphertext : bstr / nil,
    ? recipients : [+COSE_recipient]
]
```

5.1.1. Content Key Distribution Methods

An encrypted message consists of an encrypted content and an encrypted CEK for one or more recipients. The CEK is encrypted for each recipient, using a key specific to that recipient. The details of this encryption depend on which class the recipient algorithm falls into. Specific details on each of the classes can be found in Section 8.5. A short summary of the five content key distribution methods is:

direct: The CEK is the same as the identified previously distributed symmetric key or is derived from a previously distributed secret. No CEK is transported in the message.

symmetric key-encryption keys (KEKs): The CEK is encrypted using a previously distributed symmetric KEK. Also known as key wrap.

key agreement: The recipient's public key and a sender's private key are used to generate a pairwise secret, a Key Derivation Function (KDF) is applied to derive a key, and then the CEK is either the derived key or encrypted by the derived key.

key transport: The CEK is encrypted with the recipient's public key.

passwords: The CEK is encrypted in a KEK that is derived from a password. As of when this document was published, no password algorithms have been defined.

5.2. Single Recipient Encrypted

The COSE_Encrypt0 encrypted structure does not have the ability to specify recipients of the message. The structure assumes that the recipient of the object will already know the identity of the key to be used in order to decrypt the message. If a key needs to be identified to the recipient, the enveloped structure ought to be used.

Examples of encrypted messages can be found in Appendix C.4.

The COSE_Encrypt0 structure can be encoded as either tagged or untagged, depending on the context it will be used in. A tagged COSE_Encrypt0 structure is identified by the CBOR tag 16. The CDDL fragment that represents this is:

```
COSE_Encrypt0_Tagged = #6.16(COSE_Encrypt0)
```

The COSE_Encrypt0 structure is a CBOR array. The fields of the array, in order, are:

protected: This is as described in Section 3.

unprotected: This is as described in Section 3.

ciphertext: This is as described in Section 5.1.

The CDDL fragment for COSE_Encrypt0 that corresponds to the above text is:

```
COSE_Encrypt0 = [
    Headers,
    ciphertext : bstr / nil,
]
```

5.3. How to Encrypt and Decrypt for AEAD Algorithms

The encryption algorithm for AEAD algorithms is fairly simple. The first step is to create a consistent byte string for the authenticated data structure. For this purpose, we use an Enc_structure. The Enc_structure is a CBOR array. The fields of the Enc_structure, in order, are:

1. A context text string identifying the context of the authenticated data structure. The context text string is:

   "Encrypt0" for the content encryption of a COSE_Encrypt0 data structure.

   "Encrypt" for the first layer of a COSE_Encrypt data structure (i.e., for content encryption).

   "Enc_Recipient" for a recipient encoding to be placed in a COSE_Encrypt data structure.

   "Mac_Recipient" for a recipient encoding to be placed in a MACed message structure.

   "Rec_Recipient" for a recipient encoding to be placed in a recipient structure.

2. The protected attributes from the body structure, encoded in a bstr type. If there are no protected attributes, a zero-length byte string is used.

3. The externally supplied data from the application encoded in a bstr type. If this field is not supplied, it defaults to a zero-length byte string. (See Section 4.3 for application guidance on constructing this field.)

The CDDL fragment that describes the above text is:

```
Enc_structure = [
    context : "Encrypt" / "Encrypt0" / "Enc_Recipient" /
        "Mac_Recipient" / "Rec_Recipient",
    protected : empty_or_serialized_map,
    external_aad : bstr
]
```

How to encrypt a message:

1. Create an Enc_structure and populate it with the appropriate fields.

2. Encode the Enc_structure to a byte string (Additional Authenticated Data (AAD)), using the encoding described in Section 9.

3. Determine the encryption key (K). This step is dependent on the class of recipient algorithm being used. For:

   No Recipients: The key to be used is determined by the algorithm and key at the current layer. Examples are key wrap keys (Section 8.5.2) and preshared secrets.

Direct Encryption and Direct Key Agreement: The key is determined by the key and algorithm in the recipient structure. The encryption algorithm and size of the key to be used are inputs into the KDF used for the recipient. (For direct, the KDF can be thought of as the identity operation.) Examples of these algorithms are found in Sections 6.1 and 6.3 of [RFC9053].

Other: The key is randomly generated.

4. Call the encryption algorithm with K (the encryption key), P (the plaintext), and AAD. Place the returned ciphertext into the "ciphertext" field of the structure.

5. For recipients of the message using non-direct algorithms, recursively perform the encryption algorithm for that recipient, using K (the encryption key) as the plaintext.

How to decrypt a message:

1. Create an Enc_structure and populate it with the appropriate fields.

2. Encode the Enc_structure to a byte string (AAD), using the encoding described in Section 9.

3. Determine the decryption key. This step is dependent on the class of recipient algorithm being used. For:

No Recipients: The key to be used is determined by the algorithm and key at the current layer. Examples are key wrap keys (Section 8.5.2) and preshared secrets.

Direct Encryption and Direct Key Agreement: The key is determined by the key and algorithm in the recipient structure. The encryption algorithm and size of the key to be used are inputs into the KDF used for the recipient. (For direct, the KDF can be thought of as the identity operation.)

Other: The key is determined by decoding and decrypting one of the recipient structures.

4. Call the decryption algorithm with K (the decryption key to use), C (the ciphertext), and AAD.

5.4. How to Encrypt and Decrypt for AE Algorithms

How to encrypt a message:

1. Verify that the "protected" field is a zero-length byte string.

2. Verify that there was no external additional authenticated data supplied for this operation.

3. Determine the encryption key. This step is dependent on the class of recipient algorithm being used. For:

No Recipients: The key to be used is determined by the algorithm and key at the current layer. Examples are key wrap keys (Section 8.5.2) and preshared secrets.

Direct Encryption and Direct Key Agreement: The key is determined by the key and algorithm in the recipient structure. The encryption algorithm and size of the key to be used are inputs into the KDF used for the recipient. (For direct, the KDF can be thought of as the identity operation.) Examples of these algorithms are found in Sections 6.1 and 6.3 of [RFC9053].

Other: The key is randomly generated.

4. Call the encryption algorithm with K (the encryption key to use) and P (the plaintext). Place the returned ciphertext into the "ciphertext" field of the structure.

5. For recipients of the message using non-direct algorithms, recursively perform the encryption algorithm for that recipient, using K (the encryption key) as the plaintext.

How to decrypt a message:

1. Verify that the "protected" field is a zero-length byte string.

2. Verify that there was no external additional authenticated data supplied for this operation.

3. Determine the decryption key. This step is dependent on the class of recipient algorithm being used. For:

   No Recipients: The key to be used is determined by the algorithm and key at the current layer. Examples are key wrap keys (Section 8.5.2) and preshared secrets.

   Direct Encryption and Direct Key Agreement: The key is determined by the key and algorithm in the recipient structure. The encryption algorithm and size of the key to be used are inputs into the KDF used for the recipient. (For direct, the KDF can be thought of as the identity operation.) Examples of these algorithms are found in Sections 6.1 and 6.3 of [RFC9053].

   Other: The key is determined by decoding and decrypting one of the recipient structures.

4. Call the decryption algorithm with K (the decryption key to use) and C (the ciphertext).

6. MAC Objects

COSE supports two different MAC structures. COSE_Mac0 is used when a recipient structure is not needed because the key to be used is implicitly known. COSE_Mac is used for all other cases. These include a requirement for multiple recipients, the key being unknown, or a recipient algorithm other than direct.

In this section, we describe the structure and methods to be used when doing MAC authentication in COSE. This document allows for the use of all of the same classes of recipient algorithms as are allowed for encryption.

There are two modes in which MAC operations can be used. The first is just a check that the content has not been changed since the MAC was computed. Any class of recipient algorithm can be used for this purpose. The second mode is to both check that the content has not been changed since the MAC was computed and use the recipient algorithm to verify who sent it. The classes of recipient algorithms that support this are those that use a preshared secret or do Static-Static (SS) key agreement (without the key wrap step). In both of these cases, the entity that created and sent the message MAC can be validated. (This knowledge of the sender assumes that there are only two parties involved and that you did not send the message to yourself.) The origination property can be obtained with both of the MAC message structures.

6.1. MACed Message with Recipients

A multiple-recipient MACed message uses two structures: the COSE_Mac structure defined in this section for carrying the body and the COSE_recipient structure (Section 5.1) to hold the key used for the MAC computation. Examples of MACed messages can be found in Appendix C.5.

The MAC structure can be encoded as either tagged or untagged depending on the context it will be used in. A tagged COSE_Mac structure is identified by the CBOR tag 97. The CDDL fragment that represents this is:

```
COSE_Mac_Tagged = #6.97(COSE_Mac)
```

The COSE_Mac structure is a CBOR array. The fields of the array, in order, are:

protected: This is as described in Section 3.

unprotected: This is as described in Section 3.

payload: This field contains the serialized content to be MACed. If the payload is not present in the message, the application is required to supply the payload separately. The payload is wrapped in a bstr to ensure that it is transported without changes. If the payload is transported separately (i.e., detached content), then a nil CBOR value is placed in this location, and it is the responsibility of the application to ensure that it will be transported without changes.

tag: This field contains the MAC value.

recipients: This is as described in Section 5.1.

The CDDL fragment that represents the above text for COSE_Mac follows.

```
COSE_Mac = [
   Headers,
   payload : bstr / nil,
   tag : bstr,
   recipients : [+COSE_recipient]
]
```

6.2. MACed Messages with Implicit Key

In this section, we describe the structure and methods to be used when doing MAC authentication for those cases where the recipient is implicitly known.

The MACed message uses the COSE_Mac0 structure defined in this section for carrying the body. Examples of MACed messages with an implicit key can be found in Appendix C.6.

The MAC structure can be encoded as either tagged or untagged, depending on the context it will be used in. A tagged COSE_Mac0 structure is identified by the CBOR tag 17. The CDDL fragment that represents this is:

```
COSE_Mac0_Tagged = #6.17(COSE_Mac0)
```

The COSE_Mac0 structure is a CBOR array. The fields of the array, in order, are:

protected: This is as described in Section 3.

unprotected: This is as described in Section 3.

payload: This is as described in Section 6.1.

tag: This field contains the MAC value.

The CDDL fragment that corresponds to the above text is:

```
COSE_Mac0 = [
   Headers,
   payload : bstr / nil,
   tag : bstr,
```

```
]
```

6.3. How to Compute and Verify a MAC

In order to get a consistent encoding of the data to be authenticated, the MAC_structure is used to create the canonical form. The MAC_structure is a CBOR array. The fields of the MAC_structure, in order, are:

1. A context text string that identifies the structure that is being encoded. This context text string is "MAC" for the COSE_Mac structure. This context text string is "MAC0" for the COSE_Mac0 structure.

2. The protected attributes from the body structure. If there are no protected attributes, a zero-length bstr is used.

3. The externally supplied data from the application, encoded as a bstr type. If this field is not supplied, it defaults to a zero-length byte string. (See Section 4.3 for application guidance on constructing this field.)

4. The payload to be MACed, encoded in a bstr type. The full payload is used here, independent of how it is transported.

The CDDL fragment that corresponds to the above text is:

```
MAC_structure = [
     context : "MAC" / "MAC0",
     protected : empty_or_serialized_map,
     external_aad : bstr,
     payload : bstr
]
```

The steps to compute a MAC are:

1. Create a MAC_structure and populate it with the appropriate fields.

2. Create the value ToBeMaced by encoding the MAC_structure to a byte string, using the encoding described in Section 9.

3. Call the MAC creation algorithm, passing in K (the key to use), alg (the algorithm to MAC with), and ToBeMaced (the value to compute the MAC on).

4. Place the resulting MAC in the "tag" field of the COSE_Mac or COSE_Mac0 structure.

5. For COSE_Mac structures, encrypt and encode the MAC key for each recipient of the message.

The steps to verify a MAC are:

1. Create a MAC_structure and populate it with the appropriate fields.

2. Create the value ToBeMaced by encoding the MAC_structure to a byte string, using the encoding described in Section 9.

3. For COSE_Mac structures, obtain the cryptographic key by decoding and decrypting one of the recipient structures.

4. Call the MAC creation algorithm, passing in K (the key to use), alg (the algorithm to MAC with), and ToBeMaced (the value to compute the MAC on).

5. Compare the MAC value to the "tag" field of the COSE_Mac or COSE_Mac0 structure.

7. Key Objects

A COSE Key structure is built on a CBOR map. The set of common parameters that can appear in a COSE Key can be found in the IANA "COSE Key Common Parameters" registry [COSE.KeyParameters] (see Section 11.2). Additional parameters defined for specific key types can be found in the IANA "COSE Key Type Parameters" registry [COSE.KeyTypes].

A COSE Key Set uses a CBOR array object as its underlying type. The values of the array elements are COSE Keys. A COSE Key Set MUST have at least one element in the array. Examples of COSE Key Sets can be found in Appendix C.7.

Each element in a COSE Key Set MUST be processed independently. If one element in a COSE Key Set is either malformed or uses a key that is not understood by an application, that key is ignored, and the other keys are processed normally.

The element "kty" is a required element in a COSE_Key map.

The CDDL grammar describing COSE_Key and COSE_KeySet is:

```
COSE_Key = {
    1 => tstr / int,          ; kty
    ? 2 => bstr,              ; kid
    ? 3 => tstr / int,        ; alg
    ? 4 => [+ (tstr / int) ], ; key_ops
    ? 5 => bstr,              ; Base IV
    * label => values
}

COSE_KeySet = [+COSE_Key]
```

7.1. COSE Key Common Parameters

This document defines a set of common parameters for a COSE Key object. Table 4 provides a summary of the parameters defined in this section. There are also parameters that are defined for specific key types. Key-type-specific parameters can be found in [RFC9053].

Name	Label	CBOR Type	Value Registry	Description
kty	1	tstr / int	COSE Key Types	Identification of the key type
kid	2	bstr		Key identification value -- match to "kid" in message
alg	3	tstr / int	COSE Algorithms	Key usage restriction to this algorithm
key_ops	4	[+ (tstr/ int)]		Restrict set of permissible operations
Base IV	5	bstr		Base IV to be xor-ed with Partial IVs

Table 4: Key Map Labels

kty: This parameter is used to identify the family of keys for this structure and, thus, the set of key-type-specific parameters to be found. The set of values defined in this document can be found in [COSE.KeyTypes]. This parameter MUST be present in a key object. Implementations MUST verify that the key type is appropriate for

the algorithm being processed. The key type MUST be included as part of the trust-decision process.

alg: This parameter is used to restrict the algorithm that is used with the key. If this parameter is present in the key structure, the application MUST verify that this algorithm matches the algorithm for which the key is being used. If the algorithms do not match, then this key object MUST NOT be used to perform the cryptographic operation. Note that the same key can be in a different key structure with a different or no algorithm specified; however, this is considered to be a poor security practice.

kid: This parameter is used to give an identifier for a key. The identifier is not structured and can be anything from a user-provided byte string to a value computed on the public portion of the key. This field is intended for matching against a "kid" parameter in a message in order to filter down the set of keys that need to be checked. The value of the identifier is not a unique value and can occur in other key objects, even for different keys.

key_ops: This parameter is defined to restrict the set of operations that a key is to be used for. The value of the field is an array of values from Table 5. Algorithms define the values of key ops that are permitted to appear and are required for specific operations. The set of values matches that in [RFC7517] and [W3C.WebCrypto].

Base IV: This parameter is defined to carry the base portion of an IV. It is designed to be used with the Partial IV header parameter defined in Section 3.1. This field provides the ability to associate a Base IV with a key that is then modified on a per-message basis with the Partial IV.

Extreme care needs to be taken when using a Base IV in an application. Many encryption algorithms lose security if the same IV is used twice.

If different keys are derived for each sender, starting at the same Base IV is likely to satisfy this condition. If the same key is used for multiple senders, then the application needs to provide for a method of dividing the IV space up between the senders. This could be done by providing a different base point to start from or a different Partial IV to start with and restricting the number of messages to be sent before rekeying.

Name	Value	Description
sign	1	The key is used to create signatures. Requires private key fields.
verify	2	The key is used for verification of signatures.
encrypt	3	The key is used for key transport encryption.
decrypt	4	The key is used for key transport decryption. Requires private key fields.
wrap key	5	The key is used for key wrap encryption.
unwrap key	6	The key is used for key wrap decryption. Requires private key fields.
derive key	7	The key is used for deriving keys. Requires private key fields.

derive bits	8	The key is used for deriving bits not to be used as a key. Requires private key fields.
MAC create	9	The key is used for creating MACs.
MAC verify	10	The key is used for validating MACs.

Table 5: Key Operation Values

8. Taxonomy of Algorithms Used by COSE

In this section, a taxonomy of the different algorithm types that can be used in COSE is laid out. This taxonomy should not be considered to be exhaustive. New algorithms will be created that will not fit into this taxonomy.

8.1. Signature Algorithms

Signature algorithms provide data-origination and data-integrity services. Data origination provides the ability to infer who originated the data based on who signed the data. Data integrity provides the ability to verify that the data has not been modified since it was signed.

There are two general signature algorithm schemes. The first is signature with appendix. In this scheme, the message content is processed and a signature is produced; the signature is called the appendix. This is the scheme used by algorithms such as ECDSA and the RSA Probabilistic Signature Scheme (RSASSA-PSS). (In fact, the SSA in RSASSA-PSS stands for Signature Scheme with Appendix.)

The signature functions for this scheme are:

signature = Sign(message content, key)

valid = Verification(message content, key, signature)

The second scheme is signature with message recovery; an example of such an algorithm is [PVSig]. In this scheme, the message content is processed, but part of it is included in the signature. Moving bytes of the message content into the signature allows for smaller signed messages; the signature size is still potentially large, but the message content has shrunk. This has implications for systems implementing these algorithms and applications that use them. The first is that the message content is not fully available until after a signature has been validated. Until that point, the part of the message contained inside of the signature is unrecoverable. The second implication is that the security analysis of the strength of the signature can be very much dependent on the structure of the message content. Finally, in the event that multiple signatures are applied to a message, all of the signature algorithms are going to be required to consume the same bytes of message content. This means that the mixing of the signature-with-message-recovery and signature-with-appendix schemes in a single message is not supported.

The signature functions for this scheme are:

signature, message sent = Sign(message content, key)

valid, message content = Verification(message sent, key, signature)

No message recovery signature algorithms have been formally defined for COSE yet. Given the new constraints arising from this scheme, while some issues have already been identified, there is a high probability that additional issues will arise when integrating message recovery signature algorithms. The first algorithm defined is going to need to make decisions about these issues, and those decisions are likely to be binding on any further algorithms defined.

We use the following terms below:

message content bytes: The byte string provided by the application to be signed.

to-be-signed bytes: The byte string passed into the signature algorithm.

recovered bytes: The bytes recovered during the signature verification process.

Some of the issues that have already been identified are:

* The to-be-signed bytes are not the same as the message content bytes. This is because we build a larger to-be-signed message during the signature processing. The length of the recovered bytes may exceed the length of the message content, but not the length of the to-be-signed bytes. This may lead to privacy considerations if, for example, the externally supplied data contains confidential information.

* There may be difficulties in determining where the recovered bytes match up with the to-be-signed bytes, because the recovered bytes contain data not in the message content bytes. One possible option would be to create a padding scheme to prevent that.

* Not all message recovery signature algorithms take the recovered bytes from the end of the to-be-signed bytes. This is a problem, because the message content bytes are at the end of the to-be-signed bytes. If the bytes to be recovered are taken from the start of the to-be-signed bytes, then, by default, none of the message content bytes may be included in the recovered bytes. One possible option to deal with this is to reverse the to-be-signed data in the event that recovered bytes are taken from the start rather than the end of the to-be-signed bytes.

Signature algorithms are used with the COSE_Signature and COSE_Sign1 structures. At the time of this writing, only signatures with appendices are defined for use with COSE; however, considerable interest has been expressed in using a signature-with-message-recovery algorithm, due to the effective size reduction that is possible.

8.2. Message Authentication Code (MAC) Algorithms

Message Authentication Codes (MACs) provide data authentication and integrity protection. They provide either no or very limited data origination. A MAC, for example, cannot be used to prove the identity of the sender to a third party.

MACs use the same scheme as signature-with-appendix algorithms. The message content is processed, and an authentication code is produced. The authentication code is frequently called a tag.

The MAC functions are:

```
tag = MAC_Create(message content, key)

valid = MAC_Verify(message content, key, tag)
```

MAC algorithms can be based on either a block cipher algorithm (i.e., AES-MAC) or a hash algorithm (i.e., a Hash-based Message Authentication Code (HMAC)). [RFC9053] defines a MAC algorithm using each of these constructions.

MAC algorithms are used in the COSE_Mac and COSE_Mac0 structures.

8.3. Content Encryption Algorithms

Content encryption algorithms provide data confidentiality for

potentially large blocks of data using a symmetric key. They provide integrity on the data that was encrypted; however, they provide either no or very limited data origination. (One cannot, for example, be used to prove the identity of the sender to a third party.) The ability to provide data origination is linked to how the CEK is obtained.

COSE restricts the set of legal content encryption algorithms to those that support authentication both of the content and additional data. The encryption process will generate some type of authentication value, but that value may be either explicit or implicit in terms of the algorithm definition. For simplicity's sake, the authentication code will normally be defined as being appended to the ciphertext stream. The encryption functions are:

ciphertext = Encrypt(message content, key, additional data)

valid, message content = Decrypt(ciphertext, key, additional data)

Most AEAD algorithms are logically defined as returning the message content only if the decryption is valid. Many, but not all, implementations will follow this convention. The message content MUST NOT be used if the decryption does not validate.

These algorithms are used in COSE_Encrypt and COSE_Encrypt0.

8.4. Key Derivation Functions (KDFs)

KDFs are used to take some secret value and generate a different one. The secret value comes in three flavors:

* Secrets that are uniformly random. This is the type of secret that is created by a good random number generator.

* Secrets that are not uniformly random. This is the type of secret that is created by operations like key agreement.

* Secrets that are not random. This is the type of secret that people generate for things like passwords.

General KDFs work well with the first type of secret, can do reasonably well with the second type of secret, and generally do poorly with the last type of secret. Functions like Argon2 [RFC9106] need to be used for nonrandom secrets.

The same KDF can be set up to deal with the first two types of secrets in different ways. The KDF defined in Section 5.1 of [RFC9053] is such a function. This is reflected in the set of algorithms defined around the HMAC-based Extract-and-Expand Key Derivation Function (HKDF).

When using KDFs, one component that is included is context information. Context information is used to allow for different keying information to be derived from the same secret. The use of context-based keying material is considered to be a good security practice.

8.5. Content Key Distribution Methods

Content key distribution methods (recipient algorithms) can be defined into a number of different classes. COSE has the ability to support many classes of recipient algorithms. In this section, a number of classes are listed. For the recipient algorithm classes defined in [RFC7516], the same names are used. Other specifications use different terms for the recipient algorithm classes or do not support some of the recipient algorithm classes.

8.5.1. Direct Encryption

The Direct Encryption class of algorithms share a secret between the sender and the recipient that is used either directly or after

manipulation as the CEK. When direct-encryption mode is used, it MUST be the only mode used on the message.

The COSE_Recipient structure for the recipient is organized as follows:

* The "protected" field MUST be a zero-length byte string unless it is used in the computation of the content key.

* The "alg" header parameter MUST be present.

* A header parameter identifying the shared secret SHOULD be present.

* The "ciphertext" field MUST be a zero-length byte string.

* The "recipients" field MUST be absent.

8.5.2. Key Wrap

In key wrap mode, the CEK is randomly generated, and that key is then encrypted by a shared secret between the sender and the recipient. All of the currently defined key wrap algorithms for COSE are AE algorithms. Key wrap mode is considered to be superior to Direct Encryption if the system has any capability for doing random-key generation. This is because the shared key is used to wrap random data rather than data that has some degree of organization and may in fact be repeating the same content. The use of key wrap loses the weak data origination that is provided by the direct-encryption algorithms.

The COSE_Recipient structure for the recipient is organized as follows:

* The "protected" field MUST be a zero-length byte string if the key wrap algorithm is an AE algorithm.

* The "recipients" field is normally absent but can be used. Applications MUST deal with a recipient field being present that has an unsupported algorithm. Failing to decrypt that specific recipient is an acceptable way of dealing with it. Failing to process the message is not an acceptable way of dealing with it.

* The plaintext to be encrypted is the key from the next layer down (usually the content layer).

* At a minimum, the "unprotected" field MUST contain the "alg" header parameter and SHOULD contain a header parameter identifying the shared secret.

8.5.3. Key Transport

Key transport mode is also called key encryption mode in some standards. Key transport mode differs from key wrap mode in that it uses an asymmetric encryption algorithm rather than a symmetric encryption algorithm to protect the key. A set of key transport algorithms is defined in [RFC8230].

When using a key transport algorithm, the COSE_Recipient structure for the recipient is organized as follows:

* The "protected" field MUST be a zero-length byte string.

* The plaintext to be encrypted is the key from the next layer down (usually the content layer).

* At a minimum, the "unprotected" field MUST contain the "alg" header parameter and SHOULD contain a parameter identifying the asymmetric key.

8.5.4. Direct Key Agreement

The Direct Key Agreement class of recipient algorithms uses a key agreement method to create a shared secret. A KDF is then applied to the shared secret to derive a key to be used in protecting the data. This key is normally used as a CEK or MAC key but could be used for other purposes if more than two layers are in use (see Appendix B).

The most commonly used key agreement algorithm is Diffie-Hellman, but other variants exist. Since COSE is designed for a store-and-forward environment rather than an online environment, many of the DH variants cannot be used, as the receiver of the message cannot provide any dynamic key material. One side effect of this is that forward secrecy (see [RFC4949]) is not achievable. A static key will always be used for the receiver of the COSE object.

Two variants of DH that are supported are:

Ephemeral-Static (ES) DH: The sender of the message creates a one-time DH key and uses a static key for the recipient. The use of the ephemeral sender key means that no additional random input is needed, as this is randomly generated for each message.

Static-Static (SS) DH: A static key is used for both the sender and the recipient. The use of static keys allows for the recipient to get a weak version of data origination for the message. When Static-Static key agreement is used, then some piece of unique data for the KDF is required to ensure that a different key is created for each message.

When direct key agreement mode is used, there MUST be only one recipient in the message. This method creates the key directly, and that makes it difficult to mix with additional recipients. If multiple recipients are needed, then the version with key wrap needs to be used.

The COSE_Recipient structure for the recipient is organized as follows:

* At a minimum, headers MUST contain the "alg" header parameter and SHOULD contain a header parameter identifying the recipient's asymmetric key.

* The headers SHOULD identify the sender's key for the Static-Static versions and MUST contain the sender's ephemeral key for the ephemeral-static versions.

8.5.5. Key Agreement with Key Wrap

Key Agreement with Key Wrap uses a randomly generated CEK. The CEK is then encrypted using a key wrap algorithm and a key derived from the shared secret computed by the key agreement algorithm. The function for this would be:

```
encryptedKey = KeyWrap(KDF(DH-Shared, context), CEK)
```

The COSE_Recipient structure for the recipient is organized as follows:

* The "protected" field is fed into the KDF context structure.

* The plaintext to be encrypted is the key from the next layer down (usually the content layer).

* The "alg" header parameter MUST be present in the layer.

* A header parameter identifying the recipient's key SHOULD be present. A header parameter identifying the sender's key SHOULD be present.

9. CBOR Encoding Restrictions

This document limits the restrictions it imposes on how the CBOR Encoder needs to work. The new encoding restrictions are aligned with the Core Deterministic Encoding Requirements specified in Section 4.2.1 of RFC 8949 [STD94]. It has been narrowed down to the following restrictions:

* The restriction applies to the encoding of the Sig_structure, the Enc_structure, and the MAC_structure.

* Encoding MUST be done using definite lengths, and the length of the (encoded) argument MUST be the minimum possible length. This means that the integer 1 is encoded as "0x01" and not "0x1801".

* Applications MUST NOT generate messages with the same label used twice as a key in a single map. Applications MUST NOT parse and process messages with the same label used twice as a key in a single map. Applications can enforce the parse-and-process requirement by using parsers that will fail the parse step or by using parsers that will pass all keys to the application, and the application can perform the check for duplicate keys.

10. Application Profiling Considerations

This document is designed to provide a set of security services but not impose algorithm implementation requirements for specific usage. The interoperability requirements are provided for how each of the individual services are used and how the algorithms are to be used for interoperability. The requirements about which algorithms and which services are needed are deferred to each application.

An example of a profile can be found in [RFC8613], where one was developed for carrying content in combination with CoAP headers.

It is intended that a profile of this document be created that defines the interoperability requirements for that specific application. This section provides a set of guidelines and topics that need to be considered when profiling this document.

* Applications need to determine the set of messages defined in this document that they will be using. The set of messages corresponds fairly directly to the needed set of security services and security levels.

* Applications may define new header parameters for a specific purpose. Applications will oftentimes select specific header parameters to use or not to use. For example, an application would normally state a preference for using either the IV or the Partial IV header parameter. If the Partial IV header parameter is specified, then the application also needs to define how the fixed portion of the IV is determined.

* When applications use externally defined authenticated data, they need to define how that data is encoded. This document assumes that the data will be provided as a byte string. More information can be found in Section 4.3.

* Applications need to determine the set of security algorithms that is to be used. When selecting the algorithms to be used as the mandatory-to-implement set, consideration should be given to choosing different types of algorithms when two are chosen for a specific purpose. An example of this would be choosing HMAC-SHA512 and AES-CMAC (Cipher-Based Message Authentication Code) as different MAC algorithms; the construction is vastly different between these two algorithms. This means that a weakening of one algorithm would be unlikely to lead to a weakening of the other algorithms. Of course, these algorithms do not provide the same level of security and thus may not be comparable for the desired security functionality. Additional guidance can be found in [BCP201].

* Applications may need to provide some type of negotiation or

discovery method if multiple algorithms or message structures are permitted. The method can range from something as simple as requiring preconfiguration of the set of algorithms to providing a discovery method built into the protocol. S/MIME provided a number of different ways to approach the problem that applications could follow:

- Advertising in the message (S/MIME capabilities) [RFC8551].

- Advertising in the certificate (capabilities extension) [RFC4262].

- Minimum requirements for the S/MIME, which have been updated over time [RFC2633] [RFC3851] [RFC5751] [RFC8551]. (Note that [RFC2633] was obsoleted by [RFC3851], which was obsoleted by [RFC5751], which was obsoleted by [RFC8551].)

11. IANA Considerations

The registries and registrations listed below were defined by RFC 8152 [RFC8152]. The majority of the following actions are to update the references to point to this document.

Note that while [RFC9053] also updates the registries and registrations originally established by [RFC8152], the requested updates are mutually exclusive. The updates requested in this document do not conflict or overlap with the updates requested in [RFC9053], and vice versa.

11.1. COSE Header Parameters Registry

The "COSE Header Parameters" registry was defined by [RFC8152]. IANA has updated the reference for this registry to point to this document instead of [RFC8152]. IANA has also updated all entries that referenced [RFC8152], except "counter signature" and "CounterSignature0", to refer to this document. The references for "counter signature" and "CounterSignature0" continue to reference [RFC8152].

11.2. COSE Key Common Parameters Registry

The "COSE Key Common Parameters" registry [COSE.KeyParameters] was defined in [RFC8152]. IANA has updated the reference for this registry to point to this document instead of [RFC8152]. IANA has also updated the entries that referenced [RFC8152] to refer to this document.

11.3. Media Type Registrations

11.3.1. COSE Security Message

IANA has registered the "application/cose" media type in the "Media Types" registry. This media type is used to indicate that the content is a COSE message.

Type name: application

Subtype name: cose

Required parameters: N/A

Optional parameters: cose-type

Encoding considerations: binary

Security considerations: See the Security Considerations section of RFC 9052.

Interoperability considerations: N/A

Published specification: RFC 9052

Applications that use this media type: IoT applications sending security content over HTTP(S) transports.

Fragment identifier considerations: N/A

Additional information:

* Deprecated alias names for this type: N/A
* Magic number(s): N/A
* File extension(s): cbor
* Macintosh file type code(s): N/A

Person & email address to contact for further information: iesg@ietf.org

Intended usage: COMMON

Restrictions on usage: N/A

Author: Jim Schaad

Change Controller: IESG

Provisional registration? No

11.3.2. COSE Key Media Type

IANA has registered the "application/cose-key" and "application/cose-key-set" media types in the "Media Types" registry. These media types are used to indicate, respectively, that the content is a COSE_Key or COSE_KeySet object.

The template for "application/cose-key" is as follows:

Type name: application

Subtype name: cose-key

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: See the Security Considerations section of RFC 9052.

Interoperability considerations: N/A

Published specification: RFC 9052

Applications that use this media type: Distribution of COSE-based keys for IoT applications.

Fragment identifier considerations: N/A

Additional information:

* Deprecated alias names for this type: N/A
* Magic number(s): N/A
* File extension(s): cbor
* Macintosh file type code(s): N/A

Person & email address to contact for further information: iesg@ietf.org

Intended usage: COMMON

Restrictions on usage: N/A

Author: Jim Schaad

Change Controller: IESG

Provisional registration? No

The template for registering "application/cose-key-set" is:

Type name: application

Subtype name: cose-key-set

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: See the Security Considerations section of RFC 9052.

Interoperability considerations: N/A

Published specification: RFC 9052

Applications that use this media type: Distribution of COSE-based keys for IoT applications.

Fragment identifier considerations: N/A

Additional information:

* Deprecated alias names for this type: N/A
* Magic number(s): N/A
* File extension(s): cbor
* Macintosh file type code(s): N/A

Person & email address to contact for further information: iesg@ietf.org

Intended usage: COMMON

Restrictions on usage: N/A

Author: Jim Schaad

Change Controller: IESG

Provisional registration? No

11.4. CoAP Content-Formats Registry

IANA added entries to the "CoAP Content-Formats" registry as indicated in [RFC8152]. IANA has updated the reference to point to this document instead of [RFC8152].

11.5. CBOR Tags Registry

IANA added entries to the "CBOR Tags" registry as indicated in [RFC8152]. IANA has updated the references to point to this document instead of [RFC8152].

11.6. Expert Review Instructions

All of the IANA registries established by [RFC8152] are, at least in

part, defined as Expert Review [RFC8126]. 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 the following into consideration:

* Point squatting should be discouraged. Reviewers are encouraged to get sufficient information for registration requests to ensure that the usage is not going to duplicate an existing registration and that the code point is likely to be used in deployments. The ranges tagged as private use are intended for testing purposes and closed environments; code points in other ranges should not be assigned for testing.

* Standards Track or BCP RFCs are required to register a code point in the Standards Action range. Specifications should exist for Specification Required ranges, but early assignment before an RFC is available is considered to be permissible. Specifications are needed for the first-come, first-served range if the points are expected to be used outside of closed environments in an interoperable way. When specifications are not provided, the description provided needs to have sufficient information to identify what the point is being used for.

* Experts should take into account the expected usage of fields when approving code point assignment. The fact that the Standards Action range is only available to Standards Track documents does not mean that a Standards Track document cannot have points assigned outside of that range. The length of the encoded value should be weighed against how many code points of that length are left and the size of device it will be used on.

* When algorithms are registered, vanity registrations should be discouraged. One way to do this is to require registrations to provide additional documentation on security analysis of the algorithm. Another thing that should be considered is requesting an opinion on the algorithm from the Crypto Forum Research Group (CFRG). Algorithms are expected to meet the security requirements of the community and the requirements of the message structures in order to be suitable for registration.

12. Security Considerations

There are a number of security considerations that need to be taken into account by implementers of this specification. While some considerations have been highlighted here, additional considerations may be found in the documents listed in the references.

Implementations need to protect the private key material for all individuals. Some cases in this document need to be highlighted with regard to this issue.

* Use of the same key for two different algorithms can leak information about the key. It is therefore recommended that keys be restricted to a single algorithm.

* Use of "direct" as a recipient algorithm combined with a second recipient algorithm exposes the direct key to the second recipient; Section 8.5 forbids combining "direct" recipient algorithms with other modes.

* Several of the algorithms in [RFC9053] have limits on the number of times that a key can be used without leaking information about the key.

The use of Elliptic Curve Diffie-Hellman (ECDH) and direct plus KDF (with no key wrap) will not directly lead to the private key being leaked; the one-way function of the KDF will prevent that. There is, however, a different issue that needs to be addressed. Having two recipients requires that the CEK be shared between two recipients.

The second recipient therefore has a CEK that was derived from material that can be used for the weak proof of origin. The second recipient could create a message using the same CEK and send it to the first recipient; the first recipient would, for either Static-Static ECDH or direct plus KDF, make an assumption that the CEK could be used for proof of origin, even though it is from the wrong entity. If the key wrap step is added, then no proof of origin is implied and this is not an issue.

Although it has been mentioned before, it bears repeating that the use of a single key for multiple algorithms has been demonstrated in some cases to leak information about a key, providing the opportunity for attackers to forge integrity tags or gain information about encrypted content. Binding a key to a single algorithm prevents these problems. Key creators and key consumers are strongly encouraged to not only create new keys for each different algorithm, but to include that selection of algorithm in any distribution of key material and strictly enforce the matching of algorithms in the key structure to algorithms in the message structure. In addition to checking that algorithms are correct, the key form needs to be checked as well. Do not use an "EC2" key where an "OKP" key is expected.

Before using a key for transmission, or before acting on information received, a trust decision on a key needs to be made. Is the data or action something that the entity associated with the key has a right to see or a right to request? A number of factors are associated with this trust decision. Some highlighted here are:

* What are the permissions associated with the key owner?

* Is the cryptographic algorithm acceptable in the current context?

* Have the restrictions associated with the key, such as algorithm or freshness, been checked, and are they correct?

* Is the request something that is reasonable, given the current state of the application?

* Have any security considerations that are part of the message been enforced (as specified by the application or "crit" header parameter)?

One area that has been getting exposure is traffic analysis of encrypted messages based on the length of the message. This specification does not provide a uniform method for providing padding as part of the message structure. An observer can distinguish between two different messages (for example, "YES" and "NO") based on the length for all of the content encryption algorithms that are defined in [RFC9053]. This means that it is up to the applications to document how content padding is to be done in order to prevent or discourage such analysis. (For example, the text strings could be defined as "YES" and "NO ".)

13. References

13.1. 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>.

[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>.

[RFC9053] Schaad, J., "CBOR Object Signing and Encryption (COSE): Initial Algorithms", RFC 9053, DOI 10.17487/RFC9053, August 2022, <https://www.rfc-editor.org/info/rfc9053>.

[STD94] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, December 2020, <https://www.rfc-editor.org/info/std94>.

13.2. Informative References

[BCP201] Housley, R., "Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms", BCP 201, RFC 7696, November 2015, <https://www.rfc-editor.org/info/bcp201>.

[COAP.Formats] IANA, "CoAP Content-Formats", <https://www.iana.org/assignments/core-parameters/>.

[CORE-GROUPCOMM] Dijk, E., Wang, C., and M. Tiloca, "Group Communication for the Constrained Application Protocol (CoAP)", Work in Progress, Internet-Draft, draft-ietf-core-groupcomm-bis-07, 11 July 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-core-groupcomm-bis-07>.

[COSE-COUNTERSIGN] Schaad, J. and R. Housley, "CBOR Object Signing and Encryption (COSE): Countersignatures", Work in Progress, Internet-Draft, draft-ietf-cose-countersign-08, 22 August 2022, <https://datatracker.ietf.org/doc/html/draft-ietf-cose-countersign-08>.

[COSE.Algorithms] IANA, "COSE Algorithms", <https://www.iana.org/assignments/cose/>.

[COSE.KeyParameters] IANA, "COSE Key Common Parameters", <https://www.iana.org/assignments/cose/>.

[COSE.KeyTypes] IANA, "COSE Key Types", <https://www.iana.org/assignments/cose/>.

[DSS] National Institute of Standards and Technology, "Digital Signature Standard (DSS)", FIPS 186-4, DOI 10.6028/NIST.FIPS.186-4, July 2013, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>.

[GitHub-Examples] "GitHub Examples of COSE", commit 3221310, 3 June 2020, <https://github.com/cose-wg/Examples>.

[PVSig] Brown, D.R.L. and D.B. Johnson, "Formal Security Proofs for a Signature Scheme with Partial Message Recovery", LNCS Volume 2020, DOI 10.1007/3-540-45353-9_11, June 2000, <https://www.certicom.com/content/dam/certicom/images/pdfs/CerticomWP-PVSigSec_login.pdf>.

[RFC2633] Ramsdell, B., Ed., "S/MIME Version 3 Message Specification", RFC 2633, DOI 10.17487/RFC2633, June 1999, <https://www.rfc-editor.org/info/rfc2633>.

[RFC3394] Schaad, J. and R. Housley, "Advanced Encryption Standard (AES) Key Wrap Algorithm", RFC 3394, DOI 10.17487/RFC3394, September 2002, <https://www.rfc-editor.org/info/rfc3394>.

[RFC3851] Ramsdell, B., Ed., "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.1 Message Specification", RFC 3851, DOI 10.17487/RFC3851, July 2004, <https://www.rfc-editor.org/info/rfc3851>.

[RFC4262] Santesson, S., "X.509 Certificate Extension for Secure/

Multipurpose Internet Mail Extensions (S/MIME) Capabilities", RFC 4262, DOI 10.17487/RFC4262, December 2005, <https://www.rfc-editor.org/info/rfc4262>.

[RFC4949] Shirey, R., "Internet Security Glossary, Version 2", FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007, <https://www.rfc-editor.org/info/rfc4949>.

[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>.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <https://www.rfc-editor.org/info/rfc5652>.

[RFC5751] Ramsdell, B. and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification", RFC 5751, DOI 10.17487/RFC5751, January 2010, <https://www.rfc-editor.org/info/rfc5751>.

[RFC5752] Turner, S. and J. Schaad, "Multiple Signatures in Cryptographic Message Syntax (CMS)", RFC 5752, DOI 10.17487/RFC5752, January 2010, <https://www.rfc-editor.org/info/rfc5752>.

[RFC5990] Randall, J., Kaliski, B., Brainard, J., and S. Turner, "Use of the RSA-KEM Key Transport Algorithm in the Cryptographic Message Syntax (CMS)", RFC 5990, DOI 10.17487/RFC5990, September 2010, <https://www.rfc-editor.org/info/rfc5990>.

[RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <https://www.rfc-editor.org/info/rfc6838>.

[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>.

[RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <https://www.rfc-editor.org/info/rfc7515>.

[RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <https://www.rfc-editor.org/info/rfc7516>.

[RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <https://www.rfc-editor.org/info/rfc7517>.

[RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <https://www.rfc-editor.org/info/rfc7518>.

[RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <https://www.rfc-editor.org/info/rfc8032>.

[RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <https://www.rfc-editor.org/info/rfc8126>.

[RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <https://www.rfc-editor.org/info/rfc8152>.

[RFC8230] Jones, M., "Using RSA Algorithms with CBOR Object Signing and Encryption (COSE) Messages", RFC 8230, DOI 10.17487/RFC8230, September 2017, <https://www.rfc-editor.org/info/rfc8230>.

[RFC8551] Schaad, J., Ramsdell, B., and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification", RFC 8551, DOI 10.17487/RFC8551, April 2019, <https://www.rfc-editor.org/info/rfc8551>.

[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>.

[RFC9054] Schaad, J., "CBOR Object Signing and Encryption (COSE): Hash Algorithms", RFC 9054, DOI 10.17487/RFC9054, August 2022, <https://www.rfc-editor.org/info/rfc9054>.

[RFC9106] Biryukov, A., Dinu, D., Khovratovich, D., and S. Josefsson, "Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications", RFC 9106, DOI 10.17487/RFC9106, September 2021, <https://www.rfc-editor.org/info/rfc9106>.

[STD90] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, December 2017, <https://www.rfc-editor.org/info/std90>.

[W3C.WebCrypto] Watson, M., Ed., "Web Cryptography API", W3C Recommendation, 26 January 2017, <https://www.w3.org/TR/WebCryptoAPI/>.

Appendix A. Guidelines for External Data Authentication of Algorithms

During development of COSE, the requirement that the algorithm identifier be located in the protected attributes was relaxed from a must to a should. Two basic reasons have been advanced to support this position. First, the resulting message will be smaller if the algorithm identifier is omitted from the most common messages in a CoAP environment. Second, there is a potential bug that will arise if full checking is not done correctly between the different places that an algorithm identifier could be placed (the message itself, an application statement, the key structure that the sender possesses, and the key structure the recipient possesses).

This appendix lays out how such a change can be made and the details that an application needs to specify in order to use this option. Two different sets of details are specified: those needed to omit an algorithm identifier and those needed to use the variant on the countersignature attribute that contains no attributes about itself.

Three sets of recommendations are laid out. The first set of recommendations applies to having an implicit algorithm identified for a single layer of a COSE object. The second set of recommendations applies to having multiple implicit algorithms identified for multiple layers of a COSE object. The third set of recommendations applies to having implicit algorithms for multiple COSE object constructs.

The key words from BCP 14 ([RFC2119] and [RFC8174]) are deliberately not used here. This specification can provide recommendations, but it cannot enforce them.

This set of recommendations applies to the case where an application is distributing a fixed algorithm along with the key information for use in a single COSE object. This normally applies to the smallest of the COSE objects -- specifically, COSE_Sign1, COSE_Mac0, and COSE_Encrypt0 -- but could apply to the other structures as well.

The following items should be taken into account:

* Applications need to list the set of COSE structures that implicit algorithms are to be used in. Applications need to require that the receipt of an explicit algorithm identifier in one of these structures will lead to the message being rejected. This requirement is stated so that there will never be a case where there is any ambiguity about the question of which algorithm should be used, the implicit or the explicit one. This applies even if the transported algorithm identifier is a protected attribute. This applies even if the transported algorithm is the same as the implicit algorithm.

* Applications need to define the set of information that is to be considered to be part of a context when omitting algorithm identifiers. At a minimum, this would be the key identifier (if needed), the key, the algorithm, and the COSE structure it is used with. Applications should restrict the use of a single key to a single algorithm. As noted for some of the algorithms in [RFC9053], the use of the same key in different, related algorithms can lead to leakage of information about the key, leakage about the data, or the ability to perform forgeries.

* In many cases, applications that make the algorithm identifier implicit will also want to make the context identifier implicit for the same reason. That is, omitting the context identifier will decrease the message size (potentially significantly, depending on the length of the identifier). Applications that do this will need to describe the circumstances where the context identifier is to be omitted and how the context identifier is to be inferred in these cases. (An exhaustive search over all of the keys would normally not be considered to be acceptable.) An example of how this can be done is to tie the context to a transaction identifier. Both would be sent on the original message, but only the transaction identifier would need to be sent after that point, as the context is tied into the transaction identifier. Another way would be to associate a context with a network address. All messages coming from a single network address can be assumed to be associated with a specific context. (In this case, the address would normally be distributed as part of the context.)

* Applications cannot rely on key identifiers being unique unless they take significant efforts to ensure that they are computed in such a way as to create this guarantee. Even when an application does this, the uniqueness might be violated if the application is run in different contexts (i.e., with a different context provider) or if the system combines the security contexts from different applications together into a single store.

* Applications should continue the practice of protecting the algorithm identifier. Since this is not done by placing it in the protected attributes field, applications should define an application-specific external data structure that includes this value. This external data field can be used as such for content encryption, MAC, and signature algorithms. It can be used in the SuppPrivInfo field for those algorithms that use a KDF to derive a key value. Applications may also want to protect other information that is part of the context structure as well. It should be noted that those fields, such as the key or a Base IV, that are protected by virtue of being used in the cryptographic computation do not need to be included in the external data field.

The second case is having multiple implicit algorithm identifiers

specified for a multiple-layer COSE object. An example of how this would work is the encryption context that an application specifies, which contains a content encryption algorithm, a key wrap algorithm, a key identifier, and a shared secret. The sender omits sending the algorithm identifier for both the content layer and the recipient layer, leaving only the key identifier. The receiver then uses the key identifier to get the implicit algorithm identifiers.

The following additional items need to be taken into consideration:

* Applications that want to support this will need to define a structure that allows for, and clearly identifies, both the COSE structure to be used with a given key and the structure and algorithm to be used for the secondary layer. The key for the secondary layer is computed as normal from the recipient layer.

The third case is having multiple implicit algorithm identifiers, but targeted at potentially unrelated layers or different COSE objects. There are a number of different scenarios where this might be applicable. Some of these scenarios are:

* Two contexts are distributed as a pair. Each of the contexts is for use with a COSE_Encrypt message. Each context will consist of distinct secret keys and IVs and potentially even different algorithms. One context is for sending messages from party A to party B, and the second context is for sending messages from party B to party A. This means that there is no chance for a reflection attack to occur, as each party uses different secret keys to send its messages; a message that is reflected back to it would fail to decrypt.

* Two contexts are distributed as a pair. The first context is used for encryption of the message, and the second context is used to place a countersignature on the message. The intention is that the second context can be distributed to other entities independently of the first context. This allows these entities to validate that the message came from an individual without being able to decrypt the message and see the content.

* Two contexts are distributed as a pair. The first context contains a key for dealing with MACed messages, and the second context contains a different key for dealing with encrypted messages. This allows for a unified distribution of keys to participants for different types of messages that have different keys, but where the keys may be used in a coordinated manner.

For these cases, the following additional items need to be considered:

* Applications need to ensure that the multiple contexts stay associated. If one of the contexts is invalidated for any reason, all of the contexts associated with it should also be invalidated.

Appendix B. Two Layers of Recipient Information

All of the currently defined recipient algorithm classes only use two layers of the COSE structure. The first layer (COSE_Encrypt) is the message content, and the second layer (COSE_Recipient) is the content key encryption. However, if one uses a recipient algorithm such as the RSA Key Encapsulation Mechanism (RSA-KEM) (see Appendix A of RSA-KEM [RFC5990]), then it makes sense to have two layers of the COSE_Recipient structure.

These layers would be:

* Layer 0: The content encryption layer. This layer contains the payload of the message.

* Layer 1: The encryption of the CEK by a KEK.

* Layer 2: The encryption of a long random secret using an RSA key

and a key derivation function to convert that secret into the KEK.

This is an example of what a triple-layer message would look like. To make it easier to read, it is presented using the extended CBOR diagnostic notation (defined in [RFC8610]) rather than as a binary dump. The message has the following layers:

* Layer 0: Has content encrypted with AES-GCM using a 128-bit key.

* Layer 1: Uses the AES Key Wrap algorithm with a 128-bit key.

* Layer 2: Uses ECDH Ephemeral-Static direct to generate the Layer 1 key.

In effect, this example is a decomposed version of using the ECDH-ES+A128KW algorithm.

Size of binary file is 183 bytes

```
96(
  [ / COSE_Encrypt /
    / protected h'a10101' / << {
        / alg / 1:1 / AES-GCM 128 /
      } >>,
    / unprotected / {
      / iv / 5:h'02d1f7e6f26c43d4868d87ce'
    },
    / ciphertext / h'64f84d913ba60a76070a9a48f26e97e863e2852948658f0
811139868826e89218a75715b',
    / recipients / [
      [ / COSE_Recipient /
        / protected / h'',
        / unprotected / {
          / alg / 1:-3 / A128KW /
        },
        / ciphertext / h'dbd43c4e9d719c27c6275c67d628d493f090593db82
18f11',
        / recipients / [
          [ / COSE_Recipient /
            / protected h'a1013818' / << {
                / alg / 1:-25 / ECDH-ES + HKDF-256 /
              } >> ,
            / unprotected / {
              / ephemeral / -1:{
                / kty / 1:2,
                / crv / -1:1,
                / x / -2:h'b2add44368ea6d641f9ca9af308b4079aeb519f11
e9b8a55a600b21233e86e68',
                / y / -3:false
              },
              / kid / 4:'meriadoc.brandybuck@buckland.example'
            },
            / ciphertext / h''
          ]
        ]
      ]
    ]
  ]
)
```

Appendix C. Examples

This appendix includes a set of examples that show the different features and message types that have been defined in this document. To make the examples easier to read, they are presented using the extended CBOR diagnostic notation (defined in [RFC8610]) rather than as a binary dump.

A GitHub project has been created at [GitHub-Examples] that contains not only the examples presented in this document, but a more complete set of testing examples as well. Each example is found in a JSON

file that contains the inputs used to create the example, some of the intermediate values that can be used in debugging the example, and the output of the example presented both as a hex dump and in CBOR diagnostic notation format. Some of the examples at the site are designed to be failure-testing cases; these are clearly marked as such in the JSON file. If errors in the examples in this document are found, the examples on GitHub will be updated, and a note to that effect will be placed in the JSON file.

As noted, the examples are presented using CBOR's diagnostic notation. A Ruby-based tool exists that can convert between the diagnostic notation and binary. This tool can be installed with the command line:

```
gem install cbor-diag
```

The diagnostic notation can be converted into binary files using the following command line:

```
diag2cbor.rb < inputfile > outputfile
```

The examples can be extracted from the XML version of this document via an XPath expression, as all of the source code is tagged with the attribute type='cbor-diag'. (Depending on the XPath evaluator one is using, it may be necessary to deal with > as an entity.)

```
//sourcecode[@type='cbor-diag']/text()
```

C.1. Examples of Signed Messages

C.1.1. Single Signature

This example uses the following:

* Signature Algorithm: ECDSA w/ SHA-256, Curve P-256

Size of binary file is 103 bytes

```
98(
  [
    / protected / h'',
    / unprotected / {},
    / payload / 'This is the content.',
    / signatures / [
      [
        / protected h'a10126' / << {
            / alg / 1:-7 / ECDSA 256 /
          } >>,
        / unprotected / {
          / kid / 4:'11'
        },
        / signature / h'e2aeafd40d69d19dfe6e52077c5d7ff4e408282cbefb
5d06cbf414af2e19d982ac45ac98b8544c908b4507de1e90b717c3d34816fe926a2b
98f53afd2fa0f30a'
      ]
    ]
  ]
)
```

C.1.2. Multiple Signers

This example uses the following:

* Signature Algorithm: ECDSA w/ SHA-256, Curve P-256

* Signature Algorithm: ECDSA w/ SHA-512, Curve P-521

Size of binary file is 277 bytes

```
98(
  [
```

```
    / protected / h'',
    / unprotected / {},
    / payload / 'This is the content.',
    / signatures / [
      [
        / protected h'a10126' / << {
            / alg / 1:-7 / ECDSA 256 /
          } >>,
        / unprotected / {
          / kid / 4:'11'
        },
        / signature / h'e2aeafd40d69d19dfe6e52077c5d7ff4e408282cbefb
5d06cbf414af2e19d982ac45ac98b8544c908b4507de1e90b717c3d34816fe926a2b
98f53afd2fa0f30a'
      ],
      [
        / protected h'a1013823' / << {
            / alg / 1:-36 / ECDSA 521 /
          } >> ,
        / unprotected / {
          / kid / 4:'bilbo.baggins@hobbiton.example'
        },
        / signature / h'00a2d28a7c2bdb1587877420f65adf7d0b9a06635dd1
de64bb62974c863f0b160dd2163734034e6ac003b01e8705524c5c4ca479a952f024
7ee8cb0b4fb7397ba08d009e0c8bf482270cc5771aa143966e5a469a09f613488030
c5b07ec6d722e3835adb5b2d8c44e95ffb13877dd2582866883535de3bb03d01753f
83ab87bb4f7a0297'
      ]
    ]
  ]
)
```

C.1.3. Signature with Criticality

This example uses the following:

* Signature Algorithm: ECDSA w/ SHA-256, Curve P-256

* There is a criticality marker on the "reserved" header parameter.

Size of binary file is 125 bytes

```
98(
  [
    / protected h'a2687265736572766564f4028168726573657276656 4' /
    << {
        "reserved":false,
        / crit / 2:[
          "reserved"
        ]
      } >>,
    / unprotected / {},
    / payload / 'This is the content.',
    / signatures / [
      [
        / protected h'a10126' / << {
            / alg / 1:-7 / ECDSA 256 /
          } >>,
        / unprotected / {
          / kid / 4:'11'
        },
        / signature / h'3fc54702aa56e1b2cb20284294c9106a63f91bac658d
69351210a031d8fc7c5ff3e4be39445b1a3e83e1510d1aca2f2e8a7c081c7645042b
18aba9d1fad1bd9c'
      ]
    ]
  ]
)
```

C.2. Single Signer Examples

C.2.1. Single ECDSA Signature

This example uses the following:

* Signature Algorithm: ECDSA w/ SHA-256, Curve P-256

Size of binary file is 98 bytes

```
18(
  [
    / protected h'a10126' / << {
        / alg / 1:-7 / ECDSA 256 /
      } >>,
    / unprotected / {
      / kid / 4:'11'
    },
    / payload / 'This is the content.',
    / signature / h'8eb33e4ca31d1c465ab05aac34cc6b23d58fef5c083106c4
d25a91aef0b0117e2af9a291aa32e14ab834dc56ed2a223444547e01f11d3b0916e5
a4c345cacb36'
  ]
)
```

C.3. Examples of Enveloped Messages

C.3.1. Direct ECDH

This example uses the following:

* CEK: AES-GCM w/ 128-bit key

* Recipient class: ECDH Ephemeral-Static, Curve P-256

Size of binary file is 151 bytes

```
96(
  [
    / protected h'a10101' / << {
        / alg / 1:1 / AES-GCM 128 /
      } >>,
    / unprotected / {
      / iv / 5:h'c9cf4df2fe6c632bf7886413'
    },
    / ciphertext / h'7adbe2709ca818fb415f1e5df66f4e1a51053ba6d65a1a0
c52a357da7a644b8070a151b0',
    / recipients / [
      [
        / protected h'a1013818' / << {
            / alg / 1:-25 / ECDH-ES + HKDF-256 /
          } >>,
        / unprotected / {
          / ephemeral / -1:{
            / kty / 1:2,
            / crv / -1:1,
            / x / -2:h'98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbf
bf054e1c7b4d91d6280',
            / y / -3:true
          },
          / kid / 4:'meriadoc.brandybuck@buckland.example'
        },
        / ciphertext / h''
      ]
    ]
  ]
)
```

C.3.2. Direct Plus Key Derivation

This example uses the following:

* CEK: AES-CCM w/ 128-bit key, truncate the tag to 64 bits

* Recipient class: Use HKDF on a shared secret with the following implicit fields as part of the context.

  - salt: "aabbccddeeffgghh"

  - PartyU identity: "lighting-client"

  - PartyV identity: "lighting-server"

  - Supplementary Public Other: "Encryption Example 02"

Size of binary file is 91 bytes

```
96(
  [
    / protected h'a1010a' / << {
        / alg / 1:10 / AES-CCM-16-64-128 /
      } >>,
    / unprotected / {
      / iv / 5:h'89f52f65a1c580933b5261a76c'
    },
    / ciphertext / h'753548a19b1307084ca7b2056924ed95f2e3b17006dfe93
1b687b847',
    / recipients / [
      [
        / protected h'a10129' / << {
            / alg / 1:-10
          } >>,
        / unprotected / {
          / salt / -20:'aabbccddeeffgghh',
          / kid / 4:'our-secret'
        },
        / ciphertext / h''
      ]
    ]
  ]
)
```

C.3.3. Encrypted Content with External Data

This example uses the following:

* CEK: AES-GCM w/ 128-bit key

* Recipient class: ECDH Static-Static, Curve P-256 with AES Key Wrap

* Externally Supplied AAD: h'0011bbcc22dd44ee55ff660077'

Size of binary file is 173 bytes

```
96(
  [
    / protected h'a10101' / << {
        / alg / 1:1 / AES-GCM 128 /
      } >> ,
    / unprotected / {
      / iv / 5:h'02d1f7e6f26c43d4868d87ce'
    },
    / ciphertext / h'64f84d913ba60a76070a9a48f26e97e863e28529d8f5335
e5f0165eee976b4a5f6c6f09d',
    / recipients / [
      [
        / protected / h'a101381f' / {
            \ alg \ 1:-32 \ ECDH-SS+A128KW \
          } / ,
        / unprotected / {
          / static kid / -3:'peregrin.took@tuckborough.example',
          / kid / 4:'meriadoc.brandybuck@buckland.example',
          / U nonce / -22:h'0101'
        },
```

```
        / ciphertext / h'41e0d76f579dbd0d936a662d54d8582037de2e366fd
   e1c62'
      ]
    ]
  ]
)
```

C.4. Examples of Encrypted Messages

C.4.1. Simple Encrypted Message

This example uses the following:

* CEK: AES-CCM w/ 128-bit key and a 64-bit tag

Size of binary file is 52 bytes

```
16(
  [
    / protected h'a1010a' / << {
        / alg / 1:10 / AES-CCM-16-64-128 /
      } >> ,
    / unprotected / {
      / iv / 5:h'89f52f65a1c580933b5261a78c'
    },
    / ciphertext / h'5974e1b99a3a4cc09a659aa2e9e7fff161d38ce71cb45ce
   460ffb569'
  ]
)
```

C.4.2. Encrypted Message with a Partial IV

This example uses the following:

* CEK: AES-CCM w/ 128-bit key and a 64-bit tag

* Prefix for IV is 89F52F65A1C580933B52

Size of binary file is 41 bytes

```
16(
  [
    / protected h'a1010a' / << {
        / alg / 1:10 / AES-CCM-16-64-128 /
      } >> ,
    / unprotected / {
      / partial iv / 6:h'61a7'
    },
    / ciphertext / h'252a8911d465c125b6764739700f0141ed09192de139e05
   3bd09abca'
  ]
)
```

C.5. Examples of MACed Messages

C.5.1. Shared Secret Direct MAC

This example uses the following:

* MAC: AES-CMAC, 256-bit key, truncated to 64 bits

* Recipient class: direct shared secret

Size of binary file is 57 bytes

```
97(
  [
    / protected h'a1010f' / << {
        / alg / 1:15 / AES-CBC-MAC-256//64 /
      } >> ,
    / unprotected / {},
```

```
      / payload / 'This is the content.',
      / tag / h'9e1226ba1f81b848',
      / recipients / [
        [
          / protected / h'',
          / unprotected / {
            / alg / 1:-6 / direct /,
            / kid / 4:'our-secret'
          },
          / ciphertext / h''
        ]
      ]
    ]
  )
```

C.5.2. ECDH Direct MAC

This example uses the following:

* MAC: HMAC w/SHA-256, 256-bit key

* Recipient class: ECDH key agreement, two static keys, HKDF w/ context structure

Size of binary file is 214 bytes

```
  97(
    [
      / protected h'a10105' / << {
          / alg / 1:5 / HMAC 256//256 /
        } >> ,
      / unprotected / {},
      / payload / 'This is the content.',
      / tag / h'81a03448acd3d305376eaa11fb3fe416a955be2cbe7ec96f012c99
  4bc3f16a41',
      / recipients / [
        [
          / protected h'a101381a' / << {
              / alg / 1:-27 / ECDH-SS + HKDF-256 /
            } >> ,
          / unprotected / {
            / static kid / -3:'peregrin.took@tuckborough.example',
            / kid / 4:'meriadoc.brandybuck@buckland.example',
            / U nonce / -22:h'4d8553e7e74f3c6a3a9dd3ef286a8195cbf8a23d
  19558ccfec7d34b824f42d92bd06bd2c7f0271f0214e141fb779ae2856abf585a583
  68b017e7f2a9e5ce4db5'
          },
          / ciphertext / h''
        ]
      ]
    ]
  )
```

C.5.3. Wrapped MAC

This example uses the following:

* MAC: AES-MAC, 128-bit key, truncated to 64 bits

* Recipient class: AES Key Wrap w/ a preshared 256-bit key

Size of binary file is 109 bytes

```
  97(
    [
      / protected h'a1010e' / << {
          / alg / 1:14 / AES-CBC-MAC-128//64 /
        } >> ,
      / unprotected / {},
      / payload / 'This is the content.',
      / tag / h'36f5afaf0bab5d43',
```

```
      / recipients / [
        [
          / protected / h'',
          / unprotected / {
            / alg / 1:-5 / A256KW /,
            / kid / 4:'018c0ae5-4d9b-471b-bfd6-eef314bc7037'
          },
          / ciphertext / h'711ab0dc2fc4585dce27effa6781c8093eba906f227
   b6eb0'
        ]
      ]
    ]
   )
```

C.5.4. Multi-Recipient MACed Message

This example uses the following:

* MAC: HMAC w/ SHA-256, 128-bit key

* Recipient class: Uses two different methods.

  1. ECDH Ephemeral-Static, Curve P-521, AES Key Wrap w/ 128-bit key

  2. AES Key Wrap w/ 256-bit key

Size of binary file is 309 bytes

```
   97(
     [
       / protected h'a10105' / << {
           / alg / 1:5 / HMAC 256//256 /
         } >> ,
       / unprotected / {},
       / payload / 'This is the content.',
       / tag / h'bf48235e809b5c42e995f2b7d5fa13620e7ed834e337f6aa43df16
   1e49e9323e',
       / recipients / [
         [
           / protected h'a101381c' / << {
               / alg / 1:-29 / ECDH-ES+A128KW /
             } >> ,
           / unprotected / {
             / ephemeral / -1:{
               / kty / 1:2,
               / crv / -1:3,
               / x / -2:h'0043b12669acac3fd27898ffba0bcd2e6c366d53bc4db
   71f909a759304acfb5e18cdc7ba0b13ff8c7636271a6924b1ac63c02688075b55ef2
   d613574e7dc242f79c3',
               / y / -3:true
             },
             / kid / 4:'bilbo.baggins@hobbiton.example'
           },
           / ciphertext / h'339bc4f79984cdc6b3e6ce5f315a4c7d2b0ac466fce
   a69e8c07dfbca5bb1f661bc5f8e0df9e3eff5'
         ],
         [
           / protected / h'',
           / unprotected / {
             / alg / 1:-5 / A256KW /,
             / kid / 4:'018c0ae5-4d9b-471b-bfd6-eef314bc7037'
           },
           / ciphertext / h'0b2c7cfce04e98276342d6476a7723c090dfdd15f9a
   518e7736549e998370695e6d6a83b4ae507bb'
         ]
       ]
     ]
   )
```

C.6. Examples of MAC0 Messages

C.6.1. Shared-Secret Direct MAC

This example uses the following:

* MAC: AES-CMAC, 256-bit key, truncated to 64 bits

* Recipient class: direct shared secret

Size of binary file is 37 bytes

```
17(
  [
    / protected h'a1010f' / << {
       / alg / 1:15 / AES-CBC-MAC-256//64 /
     } >> ,
    / unprotected / {},
    / payload / 'This is the content.',
    / tag / h'726043745027214f'
  ]
)
```

Note that this example uses the same inputs as Appendix C.5.1.

C.7. COSE Keys

C.7.1. Public Keys

This is an example of a COSE Key Set. This example includes the public keys for all of the previous examples.

In order, the keys are:

* An EC key with a kid of "meriadoc.brandybuck@buckland.example"

* An EC key with a kid of "11"

* An EC key with a kid of "bilbo.baggins@hobbiton.example"

* An EC key with a kid of "peregrin.took@tuckborough.example"

Size of binary file is 481 bytes

```
[
  {
    -1:1,
    -2:h'65eda5a12577c2bae829437fe338701a10aaa375e1bb5b5de108de439c0
8551d',
    -3:h'1e52ed75701163f7f9e40ddf9f341b3dc9ba860af7e0ca7ca7e9eecd008
4d19c',
    1:2,
    2:'meriadoc.brandybuck@buckland.example'
  },
  {
    -1:1,
    -2:h'bac5b11cad8f99f9c72b05cf4b9e26d244dc189f745228255a219a86d6a
09eff',
    -3:h'20138bf82dc1b6d562be0fa54ab7804a3a64b6d72ccfed6b6fb6ed28bbf
c117e',
    1:2,
    2:'11'
  },
  {
    -1:3,
    -2:h'0072992cb3ac08ecf3e5c63dedec0d51a8c1f79ef2f82f94f3c737bf5de
7986671eac625fe8257bbd0394644caaa3aaf8f27a4585fbbcad0f2457620085e5c8
f42ad',
    -3:h'01dca6947bce88bc5790485ac97427342bc35f887d86d65a089377e247e
60baa55e4e8501e2ada5724ac51d6909008033ebc10ac999b9d7f5cc2519f3fe1ea1
d9475',
    1:2,
```

```
      2:'bilbo.baggins@hobbiton.example'
    },
    {
      -1:1,
      -2:h'98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbfbf054e1c7b4d91
  d6280',
      -3:h'f01400b089867804b8e9fc96c3932161f1934f4223069170d924b7e03bf
  822bb',
      1:2,
      2:'peregrin.took@tuckborough.example'
    }
  ]
```

C.7.2. Private Keys

This is an example of a COSE Key Set. This example includes the private keys for all of the previous examples.

In order the keys are:

* An EC key with a kid of "meriadoc.brandybuck@buckland.example"
* An EC key with a kid of "11"
* An EC key with a kid of "bilbo.baggins@hobbiton.example"
* A shared-secret key with a kid of "our-secret"
* An EC key with a kid of "peregrin.took@tuckborough.example"
* A shared-secret key with kid "our-secret2"
* A shared-secret key with a kid of "018c0ae5-4d9b-471b-bfd6-eef314bc7037"

Size of binary file is 816 bytes

```
  [
    {
      1:2,
      2:'meriadoc.brandybuck@buckland.example',
      -1:1,
      -2:h'65eda5a12577c2bae829437fe338701a10aaa375e1bb5b5de108de439c0
  8551d',
      -3:h'1e52ed75701163f7f9e40ddf9f341b3dc9ba860af7e0ca7ca7e9eecd008
  4d19c',
      -4:h'aff907c99f9ad3aae6c4cdf21122bce2bd68b5283e6907154ad911840fa
  208cf'
    },
    {
      1:2,
      2:'11',
      -1:1,
      -2:h'bac5b11cad8f99f9c72b05cf4b9e26d244dc189f745228255a219a86d6a
  09eff',
      -3:h'20138bf82dc1b6d562be0fa54ab7804a3a64b6d72ccfed6b6fb6ed28bbf
  c117e',
      -4:h'57c92077664146e876760c9520d054aa93c3afb04e306705db609030850
  7b4d3'
    },
    {
      1:2,
      2:'bilbo.baggins@hobbiton.example',
      -1:3,
      -2:h'0072992cb3ac08ecf3e5c63dedec0d51a8c1f79ef2f82f94f3c737bf5de
  7986671eac625fe8257bbd0394644caaa3aaf8f27a4585fbbcad0f2457620085e5c8
  f42ad',
      -3:h'01dca6947bce88bc5790485ac97427342bc35f887d86d65a089377e247e
  60baa55e4e8501e2ada5724ac51d6909008033ebc10ac999b9d7f5cc2519f3fe1ea1
  d9475',
      -4:h'00085138ddabf5ca975f5860f91a08e91d6d5f9a76ad4018766a476680b
```

```
   55cd339e8ab6c72b5facdb2a2a50ac25bd086647dd3e2e6e99e84ca2c3609fdf177f
   eb26d'
     },
     {
       1:4,
       2:'our-secret',
       -1:h'849b57219dae48de646d07dbb533566e976686457c1491be3a76dcea6c4
   27188'
     },
     {
       1:2,
       -1:1,
       2:'peregrin.took@tuckborough.example',
       -2:h'98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbfbf054e1c7b4d91
   d6280',
       -3:h'f01400b089867804b8e9fc96c3932161f1934f4223069170d924b7e03bf
   822bb',
       -4:h'02d1f7e6f26c43d4868d87ceb2353161740aacf1f7163647984b522a848
   df1c3'
     },
     {
       1:4,
       2:'our-secret2',
       -1:h'849b5786457c1491be3a76dcea6c4271'
     },
     {
       1:4,
       2:'018c0ae5-4d9b-471b-bfd6-eef314bc7037',
       -1:h'849b57219dae48de646d07dbb533566e976686457c1491be3a76dcea6c4
   27188'
     }
   ]
```

Acknowledgments

This document is a product of the COSE Working Group of the IETF.

The following individuals are to blame for getting me started on this project in the first place: Richard Barnes, Matt Miller, and Martin Thomson.

The initial draft version of the specification was based to some degree on the outputs of the JOSE and S/MIME Working Groups.

The following individuals provided input into the final form of the document: Carsten Bormann, John Bradley, Brian Campbell, Michael B. Jones, Ilari Liusvaara, Francesca Palombini, Ludwig Seitz, and GÃ¶ran Selander.

Author's Address

Jim Schaad
August Cellars