Workgroup: COSE Working Group
Internet-Draft:
draft-ietf-cose-rfc8152bis-struct-09
Obsoletes: 8152 (if approved)
Published: 14 May 2020
Intended Status: Standards Track
Expires: 15 November 2020
Authors: J. Schaad
August Cellars

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 for the ability to have basic security services defined 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 [I-D.ietf-cose-rfc8152bis-algs] obsoletes RFC8152.

Contributing to this document

This note is to be removed before publishing as an RFC.

The source for this draft is being maintained in GitHub. Suggested changes should be submitted as pull requests at https://github.com/cose-wg/cose-rfc8152bis. Instructions are on that page as well. Editorial changes can be managed in GitHub, but any substantial issues need to be discussed on the COSE mailing list.

Status of This Memo

at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on 15 November 2020.

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)" [RFC7049]. CBOR extended the data model of the JavaScript Object Notation (JSON) [RFC8259] 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 encoding of data structures. CBOR was designed specifically both to be small in terms of messages transported and implementation size and be 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 along with [I-D.ietf-cose-rfc8152bis-algs] defines the CBOR Object Signing and Encryption (COSE) standard, which does the same thing for the CBOR encoding format. 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 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 which are transmitted over the wire. Two objects 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 starting 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 [I-D.ietf-cose-rfc8152bis-algs] and [RFC8230]. Details for additional algorithms are expected to be defined in future documents.

One feature that is present in CMS [RFC5652] 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 document 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. A common structure is going to include a URI or other pointer to where the data that is being hashed is kept, allowing this to be application-specific.

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 RFC8152

* Split the original document into this document and [I-D.ietf-cose-rfc8152bis-algs].

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

* Rearrange the text around counter signatures and define a CBOR Tag for a standalone counter signature.

* Text clarifications and changes in terminology.

1.3. Design Changes from JOSE

* Define a single top message structure 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 from 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.

* Use binary encodings, rather than base64url encodings, to encode binary data.

* Combine the authentication tag for encryption algorithms with the ciphertext.

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

1.4. CBOR Grammar

There was not a standard CBOR grammar available when COSE was originally written. For that reason the CBOR data objects defined here are described in prose. Since that time CBOR Data Definition Language (CDDL) [RFC8610] has been published as an RFC. The CBOR grammar presented in this document is compatible with CDDL.

The 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 CBOR Data Definition Language (CDDL) [RFC8610]. In this specification, the following primitive types are used:

any -- non-specific 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).

Two 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.

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 version of a CBOR grammar is presented in CDDL. 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 following 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 non-terminal 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 non-terminal Internal_Types is defined for dealing with the automated validation tools used during the writing of this document. It references those non-terminals 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.

The presence in a CBOR map of a label that is not a text string or 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 non-terminal '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 is a synonym for octet.

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

Authenticated Encryption (AE) [RFC5116] algorithms are those encryption algorithms that provide an authentication check of the contents algorithm with the encryption service.

Authenticated Encryption with Associated Data (AEAD) [RFC5116] algorithms provide the same content authentication service as AE algorithms, but they additionally provide for authentication of non-encrypted data as well.

Context is used throughout the document to represent information that is not part of the COSE message. Information which 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 configuration; 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 6.1). This message type is broken into two layers: the content layer and the recipient layer. In the content layer, the plaintext is encrypted and information about the encrypted message is placed. In the recipient layer, the content encryption key (CEK) is encrypted and information about how it is encrypted for each recipient is placed. A single layer version of the encryption message COSE_Encrypt0 (Section 6.2) is provided for cases where the CEK is pre-shared.

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
TBD0	cose-countersign	COSE_Countersignature	COSE standalone counter signature

Table 1: COSE Message Identification

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

Table 2: CoAP Content-Formats for COSE

The following CDDL fragment identifies all of the top messages defined in this document. Separate non-terminals are defined for the tagged and the 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_Countersignature

COSE_Tagged_Message = COSE_Sign_Tagged / COSE_Sign1_Tagged /
    COSE_Encrypt_Tagged / COSE_Encrypt0_Tagged /
    COSE_Mac_Tagged / COSE_Mac0_Tagged / COSE_Countersignature_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 all 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 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 12.2).

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 binary encoding is preferred because it is both shorter and the version used in the serialization structures for cryptographic computation. After encoding the map, the value is wrapped in the binary object. Recipients **MUST** accept both a zero-length byte string and a zero-length map encoded in the binary value. The wrapping allows for the encoding of the

protected map to be transported with a greater chance that it will not be altered 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.

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 12.2). Some common 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 before they are 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 label and the type of the value.

The set of header parameters defined in this section are:

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 (COSE_Sign, COSE_Sign1, COSE_Mac, and COSE_Mac0). This authentication can be done either by placing the header parameter in the protected header parameter bucket or as part of the externally supplied data. 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. When present, this the 'crit' header parameter **MUST** be placed in the protected header parameter 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 as they are algorithm dependent. If an application can

correctly process an algorithm, it can be assumed that it will correctly process all of the common header parameters associated with that algorithm. 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 if 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 fields. Integers are from the "CoAP Content-Formats" IANA registry table [COAP.Formats]. Text values following 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 also omitted. Textual content 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 as modifying the IV will cause the decryption to yield plaintext that is readily detectable as garbled.

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.

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

counter signature: This header parameter holds one or more counter signature values. Counter signatures provide a method of having a second party sign some data. The counter signature header parameter can occur as an unprotected attribute in any of the following structures: COSE_Sign1, COSE_Signature, COSE_Encrypt, COSE_recipient, COSE_Encrypt0, COSE_Mac, and COSE_Mac0. These structures all have the same beginning elements, so that a consistent calculation of the counter signature can be computed. Details on counter signatures are found in Section 5.

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		

Name	Label	Value Type	Value Registry	Description
				Partial Initialization Vector
counter signature	7	COSE_Signature / [+ COSE_Signature]		CBOR-encoded signature structure

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
    ? 7 => COSE_Signature / [+COSE_Signature] ; Counter signature
)
```

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 present. An example of header a parameter about the content is the content type. Examples of header parameters about the signature would be the algorithm and key used to create the signature and counter signatures.

RFC 5652 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 for 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 with 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 body 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 9.1), the maximum number of bytes that can be recovered is the length of the payload. The size of the payload is reduced by the number of bytes that will be recovered. If all of the bytes of the payload are consumed, then the 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 the COSE document is the ability for applications to provide additional data to be authenticated, but that 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 its operations. For example, the Accept Option can be used by a proxy to determine if an appropriate value is in the proxy's cache. But the sender can cause a failure at the server if a proxy, or an attacker, changes the set of accept values by including the field in the application-supplied data.

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. This would occur by appending the text strings 'AB' and 'CDE' or by appending 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 number 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.

"CounterSignature" for signatures used as counter signature attributes.

"CounterSignature0" for signatures used as CounterSignature0 attributes.

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 and CounterSignature0 attributes.

4. The protected attributes 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 payload is placed here independent of how it is transported.

The CDDL fragment that describes the above text is:

```
Sig_structure = [
    context : "Signature" / "Signature1" / "CounterSignature" /
              "CounterSignature0",
    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 10.

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, COSE_Sign1 or COSE_Countersignature structures. This is the value of the Countersignature0 attribute.

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 10.

3. Call the signature verification algorithm passing in K (the key to verify with), alg (the algorithm used 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. Counter Signatures

COSE supports two different forms for counter signatures. Full counter signatures use the structure COSE_Countersign. This is same structure as COSE_Signature and thus it can have protected attributes, chained counter signatures and information about identifying the key. Abbreviated counter signatures use the structure COSE_Countersign1. This structure only contains the signature value and nothing else. 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.

COSE was designed for uniformity in how the data structures are specified. One result of this is that for COSE one can expand the concept of counter signatures beyond just the idea of signing a signature to being able to sign most of the structures without having to create a new signing layer. When creating a counter signature, one needs to be clear about the security properties that result. When done on a COSE_Signature, the normal counter signature semantics are preserved. That is the counter signature makes a statement about the existence of a signature and, when used as a timestamp, a time point at which the signature exists. When done on a COSE_Mac or a COSE_Mac0, one effectively upgrades the MAC operation to a signature operation. When done on a COSE_Encrypt or COSE_Encrypt0, the existence of the encrypted data is attested to. It should be noted that there is a big difference between attesting to the encrypted data as opposed to attesting to the unencrypted

data. If the latter is what is desired, then one needs to apply a signature to the data and then encrypt that. It is always possible to construct cases where the use of two different keys will appear to result in a successful decryption (the tag check success), but which produce two completely different plaintexts. This situation is not detectable by a counter signature on the encrypted data.

5.1. Full Counter Signatures

The COSE_Countersignature structure allows for the same set of capabilities of a COSE_Signature. This means that all of the capabilities of a signature are duplicated with this structure. Specifically, the counter signer does not need to be related to the producer of what is being counter signed as key and algorithm identification can be placed in the counter signature attributes. This also means that the counter signature can itself be counter signed. This is a feature required by protocols such as long-term archiving services. More information on how this is used can be found in the evidence record syntax described in [RFC4998].

The full counter signature structure can be encoded as either tagged or untagged depending on the context it is used in. A tagged COSE_Countersign structure is identified by the CBOR tag TBD0. The CDDL fragment for full counter signatures is:

```
COSE_CounterSignature_Tagged = #6.98(COSE_CounterSignature)
COSE_CounterSignature = COSE_Signature
```

The details of the fields of a counter signature can be found in Section 4.1. The process of creating and validating abbreviated counter signatures is defined in Section 4.4.

An example of a counter signature on a signature can be found in Appendix C.1.3. An example of a counter signature in an encryption object can be found in Appendix C.3.3.

It should be noted that only a signature algorithm with appendix (see Section 9.1) can be used for counter signatures. This is because the body should be able to be processed without having to evaluate the counter signature, and this is not possible for signature schemes with message recovery.

5.2. Abbreviated Counter Signatures

Abbreviated counter signatures were designed primarily to deal with the problem of having encrypted group messaging, but still needing to know who originated the message. The objective was to keep the counter signature as small as possible while still providing the

needed security. For abbreviated counter signatures, there is no provision for any protected attributes related to the signing operation. Instead, the parameters for computing or verifying the abbreviated counter signature are inferred from the same context used to describe the encryption, signature, or MAC processing.

The byte string representing the signature value is placed in the CounterSignature0 attribute. This attribute is then encoded as an unprotected header parameter. The attribute is defined below.

The process of creating and validating abbreviated counter signatures is defined in Section 4.4.

Name	Label	Value Type	Value	Description
CounterSignature0	9	bstr		Abbreviated Counter Signature

Table 4: Header Parameter for CounterSignature0

6. 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. pre-shared secret) is used.

6.1. 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 are authenticated by the recipient algorithm (when the algorithm supports it). 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 for 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 encrypted 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]
]
```

6.1.1. 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 9.5. 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 (KEK): 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.

6.2. 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.3.

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 6.1.

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

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

6.3. 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 an 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 protected attributes 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 10.

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 transport keys (Section 9.5.3), key wrap keys (Section 9.5.2), or pre-shared 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 6.1.2 and 6.3 of [I-D.ietf-cose-rfc8152bis-algs].

   Other: The key is randomly or pseudo-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, 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 10.

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 transport keys (Section 9.5.3), key wrap keys (Section 9.5.2), or pre-shared 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.

6.4. How to Encrypt and Decrypt for AE Algorithms

How to encrypt a message:

1. Verify that the 'protected' field is empty.

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 transport keys (Section 9.5.3), key wrap keys (Section 9.5.2), or pre-shared 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 6.1.2 and 6.3 of [I-D.ietf-cose-rfc8152bis-algs].

   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, 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 empty.

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 transport keys (Section 9.5.3), key wrap keys (Section 9.5.2), or pre-shared 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 6.1.2 and 6.3 of [I-D.ietf-cose-rfc8152bis-algs].

   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).

7. 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 of 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.

When using MAC operations, there are two modes in which they 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 to use the recipient algorithm to verify who sent it. The classes of recipient algorithms that support this are those that use a pre-shared 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.

7.1. MACed Message with Recipients

The 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 6.1) 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 6.1.

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

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

7.2. 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 7.1.

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,
]
```

7.3. 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 have a 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 COSE_MAC structure. If there are no protected attributes, a zero-length bstr is used.

3. The protected attributes 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 payload is placed 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 10.

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 10.

3. For COSE_Mac structures, obtain the cryptographic key from one of the recipients of the message.

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.

8. 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 (Section 12.4). Additional parameters defined for specific key types can be found in the IANA "COSE Key Type Parameters" registry ([COSE.KeyParameters]).

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]
```

8.1. COSE Key Common Parameters

This document defines a set of common parameters for a COSE Key object. Table 5 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 [I-D.ietf-cose-rfc8152bis-algs].

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 5: 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.

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 6. 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 Partial 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, using the same Base IV with Partial IVs starting at zero is likely to ensure that the IV would not be used twice for a single key. 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 6: Key Operation Values

9. 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 which

will not fit into this taxonomy. If this occurs, then new documents addressing this new algorithms are going to be needed.

9.1. 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 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 signatures; the signature size is still potentially large, but the message content has shrunk. This has implications for systems implementing these algorithms and for 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 is that the security analysis of the strength of the signature is very much based on the structure of the message content. Messages that are highly predictable require additional randomness to be supplied as part of the signature process. In the worst case, it becomes the same as doing a signature with appendix. 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 number of bytes of message content. This means that the mixing of the different schemes in a single message is not supported, and if a recovery signature scheme is used, then the same amount of content needs to be consumed by all of the signatures.

The signature functions for this scheme are:

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

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

Signature algorithms are used with the COSE_Signature and COSE_Sign1 structures. At this time, only signatures with appendixes 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. Implementations will need to keep this in mind for later possible integration.

9.2. 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)). [I-D.ietf-cose-rfc8152bis-algs] defines a MAC algorithm using each of these constructions.

MAC algorithms are used in the COSE_Mac and COSE_Mac0 structures.

9.3. 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.

9.4. 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 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 PBES2 [RFC8018] need to be used for non-random secrets.

The same KDF can be set up to deal with the first two types of secrets in a different way. The KDF defined in section 5.1 of [I-D.ietf-cose-rfc8152bis-algs] 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.

9.5. 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. The names of the recipient algorithm classes used here are the same as those defined in [RFC7516]. Other specifications use different terms for the recipient algorithm classes or do not support some of the recipient algorithm classes.

9.5.1. Direct Encryption

The direct encryption class 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.

9.5.2. 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_Encrypt structure for the recipient is organized as follows:

- The 'protected' field **MUST** be absent 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, not being able to decrypt that

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 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.

9.5.3. 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 are defined in [RFC8230].

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

* The 'protected' field **MUST** be absent.

* 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.

9.5.4. 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 perfect 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: where 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: where 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_Encrypt 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.

9.5.5. 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_Encrypt 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.

10. CBOR Encoding Restrictions

The document limits the restrictions it imposes on the CBOR Encoder needs to work. We have managed to narrow it 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 values **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.

11. 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 set of security services that are needed and to the security levels needed.

- Applications may define new header parameters for a specific purpose. Applications will often times 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 are 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 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.

*Applications may need to provide some type of negotiation or discovery method if multiple algorithms or message structures are permitted. The method can be as simple as requiring pre-configuration 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) [RFC5751].

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

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

12. IANA Considerations

The registries and registrations listed below were created during processing of RFC 8152 [RFC8152]. The only known action at this time is to update the references.

12.1. CBOR Tag Assignment

IANA assigned tags in the "CBOR Tags" registry as part of processing [RFC8152]. IANA is requested to update the references from [RFC8152] to this document.

IANA is requested to register a new tag for the CounterSignature type.

  *Tag: TBD0

*Data Item: COSE_Signature

*Semantics: COSE standalone counter signature

*Reference: [[this document]]

12.2. COSE Header Parameters Registry

IANA created a registry titled "COSE Header Parameters" as part of processing [RFC8152]. The registry has been created to use the "Expert Review Required" registration procedure [RFC8126].

IANA is requested to update the reference for entries in the table from [RFC8152] to this document. This document does not update the expert review guidelines provided in [RFC8152].

12.3. COSE Header Algorithm Parameters Registry

IANA created a registry titled "COSE Header Algorithm Parameters" as part of processing [RFC8152]. The registry has been created to use the "Expert Review Required" registration procedure [RFC8126].

IANA is requested to update the references from [RFC8152] to this document. This document does not update the expert review guidelines provided in [RFC8152].

12.4. COSE Key Common Parameters Registry

IANA created a registry titled "COSE Key Common Parameters" as part of the processing of [RFC8152]. The registry has been created to use the "Expert Review Required" registration procedure [RFC8126].

IANA is requested to update the reference for entries in the table from [RFC8152] to this document. This document does not update the expert review guidelines provided in [RFC8152].

12.5. Media Type Registrations

12.5.1. COSE Security Message

This section registers the 'application/cose' media type in the "Media Types" registry. These media types are 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 [[This Document]].

Interoperability considerations: N/A

Published specification: [[this document]]

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, ietf@augustcellars.com

Change Controller: IESG

Provisional registration? No

12.5.2. COSE Key Media Type

This section registers 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 content is a COSE_Key or COSE_KeySet object.

The template for registering 'application/cose-key' is:

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 [[This Document]].

Interoperability considerations: N/A

Published specification: [[this document]]

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, ietf@augustcellars.com

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 [[This Document]].

Interoperability considerations: N/A

Published specification: [[this document]]

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, ietf@augustcellars.com

Change Controller: IESG

Provisional registration? No

12.6. CoAP Content-Formats Registry

IANA added the following entries to the "CoAP Content-Formats" registry while processing [RFC8152]. IANA is requested to update the reference value from [RFC8152] to [[This Document]].

13. Security Considerations

There are a number of security considerations that need to be taken into account by implementers of this specification. The security considerations that are specific to an individual algorithm are placed next to the description of the algorithm. 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 any individuals. There are some cases that need to be highlighted on this issue.

- Using 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.

- Several of the algorithms in [I-D.ietf-cose-rfc8152bis-algs] have limits on the number of times that a key can be used without leaking information about the key.

The use of 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, the use of a single key for multiple algorithms has been demonstrated in some cases to leak information about that key, provide 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 not only to 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 of the ones that are 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)?

There are a large number of algorithms presented in [I-D.ietf-cose-rfc8152bis-algs] that use nonce values. Nonces generally have some type of restriction on their values. Generally a nonce needs to be a unique value either for a key or for some other conditions. In all of these cases, there is no known requirement on the nonce being both unique and unpredictable; under these circumstances, it's reasonable to use a counter for creation of the nonce. In cases where one wants the pattern of the nonce to be unpredictable as well as unique, one can use a key created for that purpose and encrypt the counter to produce the nonce value.

One area that has been starting to get exposure is doing traffic analysis of encrypted messages based on the length of the message. This specification does not provide for a uniform method of 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 [I-D.ietf-cose-rfc8152bis-algs] document. 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 '.)

14. Implementation Status

This section is to be removed before publishing as an RFC.

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual

implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations may exist.

According to [RFC7942], "this will allow reviewers and working groups to assign due consideration to documents that have the benefit of running code, which may serve as evidence of valuable experimentation and feedback that have made the implemented protocols more mature. It is up to the individual working groups to use this information as they see fit".

14.1. Author's Versions

There are three different implementations that have been created by the author of the document both to create the examples that are included in the document and to validate the structures and methodology used in the design of COSE.

*Implementation Location: https://github.com/cose-wg

*Primary Maintainer: Jim Schaad

*Languages: There are three different languages that are currently supported: Java, C# and C.

*Cryptography: The Java and C# libraries use Bouncy Castle to provide the required cryptography. The C version uses OPENSSL Version 1.0 for the cryptography.

*Coverage: The C version currently does not have full counter sign support. The other two versions do. They do have support to allow for implicit algorithm support as they allow for the application to set attributes that are not to be sent in the message.

*Testing: All of the examples in the example library are generated by the C# library and then validated using the Java and C libraries. All three libraries have tests to allow for the creating of the same messages that are in the example library followed by validating them. These are not compared against the example library. The Java and C# libraries have unit testing included. Not all of the **MUST** statements in the document have been implemented as part of the libraries. One such statement is the requirement that unique labels be present.

*Licensing: Revised BSD License

14.2. JavaScript Version

*Implementation Location: https://github.com/erdtman/cose-js

*Primary Maintainer: Samuel Erdtman

*Languages: JavaScript

*Cryptography: TBD

*Coverage: Full Encrypt, Signature and MAC objects are supported.

*Testing: Basic testing against the common example library.

*Licensing: Apache License 2.0

14.3. Python Version

*Implementation Location: https://github.com/TimothyClaeys/COSE-PYTHON

*Primary Maintainer: Timothy Claeys

*Languages: Python

*Cryptography: pyecdsak, crypto python libraries

*Coverage: TBD

*Testing: Basic testing plus running against the common example library.

*Licensing: BSD 3-Clause License

14.4. COSE Testing Library

*Implementation Location: https://github.com/cose-wg/Examples

*Primary Maintainer: Jim Schaad

*Description: A set of tests for the COSE library is provided as part of the implementation effort. Both success and fail tests have been provided. All of the examples in this document are part of this example set.

*Coverage: An attempt has been made to have test cases for every message type and algorithm in the document. Currently examples dealing with counter signatures, and ECDH with Curve24459 and Goldilocks are missing.

*Licensing: Public Domain

15. References

15.1. Normative References

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

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

[COSE.KeyParameters] IANA, "COSE Key Parameters", , <https://www.iana.org/assignments/cose/cose.xhtml#key-common-parameters>.

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

[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>.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <https://www.rfc-editor.org/info/rfc7049>.

[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>.

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

[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>.

[I-D.ietf-cose-rfc8152bis-algs] Schaad, J., "CBOR Object Signing and Encryption (COSE): Initial Algorithms", Work in Progress, Internet-Draft, draft-ietf-cose-rfc8152bis-algs-07, 9 March 2020, <https://tools.ietf.org/html/draft-ietf-cose-rfc8152bis-algs-07>.

15.2. Informative References

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

[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>.

[RFC8018] Moriarty, K., Ed., Kaliski, B., and A. Rusch, "PKCS #5: Password-Based Cryptography Specification Version 2.1", RFC 8018, DOI 10.17487/RFC8018, January 2017, <https://www.rfc-editor.org/info/rfc8018>.

[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>.

[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>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[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>.

[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>.

[PVSig] Brown, D. and D. Johnson, "Formal Security Proofs for a Signature Scheme with Partial Message Recovery", DOI

10.1007/3-540-45353-9_11, LNCS Volume 2020, June 2000, <https://doi.org/10.1007/3-540-45353-9_11>.

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

[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>.

[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>.

[RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <https://www.rfc-editor.org/info/rfc7942>.

[RFC4998] Gondrom, T., Brandner, R., and U. Pordesch, "Evidence Record Syntax (ERS)", RFC 4998, DOI 10.17487/RFC4998, August 2007, <https://www.rfc-editor.org/info/rfc4998>.

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. There were two basic reasons that 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 a variant on the counter signature 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 [RFC2119] 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 [I-D.ietf-cose-rfc8152bis-algs], 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, are protected by virtue of being used in the cryptographic computation and 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 counter signature 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_Encrypt structure. The first layer is the message content, and the second layer 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 three layers of the COSE_Encrypt 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. The message has the following layers:

- Layer 0: Has a 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(
  [
    / protected / h'a10101' / {
        \ alg \ 1:1 \ AES-GCM 128 \
      } / ,
    / unprotected / {
      / iv / 5:h'02d1f7e6f26c43d4868d87ce'
    },
    / ciphertext / h'64f84d913ba60a76070a9a48f26e97e863e2852948658f0
811139868826e89218a75715b',
    / recipients / [
      [
        / protected / h'',
        / unprotected / {
          / alg / 1:-3 / A128KW /
        },
        / ciphertext / h'dbd43c4e9d719c27c6275c67d628d493f090593db82
18f11',
        / recipients / [
          [
            / 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 <https://github.com/cose-wg/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 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 the 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 sourcecode is tagged with the attribute type='CBORdiag'. (Depending on the XPath evaluator one is using, it may be necessary to deal with > as an entity.)

```
//sourcecode[@type='CDDL']/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
          } / ,
        / unprotected / {
          / kid / 4:'bilbo.baggins@hobbiton.example'
        },
        / signature / h'00a2d28a7c2bdb1587877420f65adf7d0b9a06635dd1
de64bb62974c863f0b160dd2163734034e6ac003b01e8705524c5c4ca479a952f024
7ee8cb0b4fb7397ba08d009e0c8bf482270cc5771aa143966e5a469a09f613488030
c5b07ec6d722e3835adb5b2d8c44e95ffb13877dd2582866883535de3bb03d01753f
83ab87bb4f7a0297'
      ]
    ]
  ]
)
```

C.1.3. Counter Signature

This example uses the following:

- Signature Algorithm: ECDSA w/ SHA-256, Curve P-256
- The same header parameters are used for both the signature and the counter signature.

Size of binary file is 180 bytes

```
98(
  [
    / protected / h'',
    / unprotected / {
      / countersign / 7:[
        / protected / h'a10126' / {
            \ alg \ 1:-7 \ ECDSA 256 \
          } / ,
        / unprotected / {
          / kid / 4:'11'
        },
        / signature / h'5ac05e289d5d0e1b0a7f048a5d2b643813ded50bc9e4
9220f4f7278f85f19d4a77d655c9d3b51e805a74b099e1e085aacd97fc29d72f887e
8802bb6650cceb2c'
      ]
    },
    / payload / 'This is the content.',
    / signatures / [
      [
        / protected / h'a10126' / {
            \ alg \ 1:-7 \ ECDSA 256 \
          } / ,
        / unprotected / {
          / kid / 4:'11'
        },
        / signature / h'e2aeafd40d69d19dfe6e52077c5d7ff4e408282cbefb
5d06cbf414af2e19d982ac45ac98b8544c908b4507de1e90b717c3d34816fe926a2b
98f53afd2fa0f30a'
      ]
    ]
  ]
)
```

C.1.4. 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'a2687265736572766564f4028168726573657665726564' /
{
        "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. Counter Signature on Encrypted Content

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 326 bytes

```
96(
  [
    / protected / h'a10101' / {
        \ alg \ 1:1 \ AES-GCM 128 \
      } / ,
    / unprotected / {
      / iv / 5:h'c9cf4df2fe6c632bf7886413',
      / countersign / 7:[
        / protected / h'a1013823' / {
            \ alg \ 1:-36
          } / ,
        / unprotected / {
          / kid / 4:'bilbo.baggins@hobbiton.example'
        },
        / signature / h'00929663c8789bb28177ae28467e66377da12302d7f9
594d2999afa5dfa531294f8896f2b6cdf1740014f4c7f1a358e3a6cf57f4ed6fb02f
cf8f7aa989f5dfd07f0700a3a7d8f3c604ba70fa9411bd10c2591b483e1d2c31de00
3183e434d8fba18f17a4c7e3dfa003ac1cf3d30d44d2533c4989d3ac38c38b71481c
c3430c9d65e7ddff'
      ]
    },
    / 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.3.4. 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 \ ECHD-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 pre-shared 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 three 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 \ ECHD-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 "peregrin.took@tuckborough.example"
- *An EC key with a kid of "bilbo.baggins@hobbiton.example"
- *An EC key with a kid of "11"

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"

  *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 a kid of "018c0ae5-4d9b-471b-bfd6-
   eef314bc7037"

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

  *An EC key with a kid of "11"

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 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, Brain Campbell, Michael B. Jones, Ilari Liusvaara, Francesca Palombini, Ludwig Seitz, and Goran Selander.

Author's Address

Jim Schaad
August Cellars

Email: ietf@augustcellars.com