COSE Working Group
Internet-Draft
Intended status: Informational
Expires: June 18, 2016

J. Schaad
August Cellars
December 16, 2015

CBOR Encoded Message Syntax
draft-ietf-cose-msg-09

Abstract

Concise Binary Object Representation (CBOR) is data format designed for small code size and small message size. There is a need for the ability to have the basic security services defined for this data format. This document specifies procesing for signatures, message authentication codes, and encryption using CBOR. This document also specifies a represention for cryptographic keys using CBOR.

Contributing to this document

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-spec>. 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

Copyright Notice

Table of Contents

1. Introduction

There has been an increased focus on the small, constrained devices that make up the Internet of Things (IOT). One of the standards that has come out of this process is the Concise Binary Object Representation (CBOR). CBOR extended the data model of the JavaScript Object Notation (JSON) by allowing for binary data among other changes. CBOR is being adopted by several of the IETF working groups dealing with the IOT world as their encoding of data structures. CBOR was designed specifically to be both small in terms of messages transport and implementation size as well having a schema free decoder. A need exists to provide basic 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] using JSON [RFC7159] that specified how to process encryption, signatures and message

authentication (MAC) operations, and how to encode keys using JSON. This document does the same work for use with the CBOR [RFC7049] document format. While there is a strong attempt to keep the flavor of the original JOSE documents, two considerations are taken into account:

- o CBOR has capabilities that are not present in JSON and should be used. One example of this is the fact that CBOR has a method of encoding binary directly without first converting it into a base64 encoded string.

- o 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 as for JOSE.

1.1. Design changes from JOSE

- o Define a top level message structure so that encrypted, signed and MACed messages can easily identified and still have a consistent view.

- o Signed messages separate the concept of protected and unprotected parameters that are for the content and the signature.

- o Recipient processing has been made more uniform. A recipient structure is required for all recipients rather than only for some.

- o MAC messages are separated from signed messages.

- o MAC messages have the ability to use all recipient algorithms on the MAC authentication key.

- o Use binary encodings for binary data rather than base64url encodings.

- o Combine the authentication tag for encryption algorithms with the ciphertext.

- o Remove the flattened mode of encoding. Forcing the use of an array of recipients at all times forces the message size to be two bytes larger, but one gets a corresponding decrease in the implementation size that should compensate for this. [CREF1]

1.2. 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 [RFC2119].

When the words appear in lower case, their natural language meaning is used.

1.3. CBOR Grammar

There currently is no standard CBOR grammar available for use by specifications. We therefore describe the CBOR structures in prose.

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 CDDL. 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).

There is a version of a CBOR grammar in the CBOR Data Definition Language (CDDL) [I-D.greevenbosch-appsawg-cbor-cddl]. Since CDDL has not be published as an RFC, this grammar may not work with the final version of CDDL when it is published. For those people who prefer using a formal language to describe the syntax of the CBOR, an informational version of the CBOR grammar is interweaved into the text as well. 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.)

```
//artwork[@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 define to make the tool quieter
Internal_Types = Sig_structure / Enc_structure / MAC_structure /
        COSE_KDF_Context
```

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.

1.4. CBOR Related Terminology

In JSON, maps are called objects and only have one kind of map key: a string. In COSE, we use both strings and integers (both negative and unsigned integers) as map keys. The integers are used for compactness of encoding and easy comparison. (Generally, in this document the value zero is going to be reserved and not used.) Since the work "key" is mainly used in its other meaning, as a cryptographic key, we use the term "label" for this usage as a map keys.

A CDDL grammar fragment is defined that defines the non-terminals 'label' as in the previous paragraph and 'values' which permits any value to be used.

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

1.5. Document Terminology

In this document we use the following terminology: [CREF2]

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].

Key management is used as a term to describe how a key at level n is obtained from level n+1 in encrypted and MACed messages. The term is also used to discuss key life cycle management, this document does not discuss key life cycle operations.

2. Basic COSE Structure

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

1. The set of protected header parameters wrapped in a bstr.

2. The set of unprotected header parameters as a map.

3. The content of the message. The content is either the plain text or the encrypted text as appropriate. (The content may be absent, but the location is still used.)

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

Identification of which message is present is done by one of three methods:

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

- The message type is identified by a CBOR tag. This document defines a CBOR tag for each of the message structures.

- 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.

Tag Value	cose-type	Data Item	Semantics
TBD1	cose-sign	COSE_Sign	COSE Signed Data Object
TBD7	cose-sign1	COSE_Sign1	COSE Single Signer Data Object
TBD2	cose-enveloped	COSE_Enveloped	COSE Enveloped Data Object
TBD3	cose-encrypted	COSE_Encrypted	COSE Encrypted Data Object
TBD4	cose-mac	COSE_Mac	COSE Mac-ed Data Object
TBD6	cose-mac0	COSE_Mac0	COSE Mac w/o Recipients Object
TBD5	N/A	COSE_Key, COSE_KeySet	COSE Key or COSE Key Set Object

Table 1: COSE Object Identification

The following CDDL fragment identifies all of the top level messages defined in this document. Separate non-terminals are defined for the tagged and the untagged versions of the messages for the convenience of applications.

```
COSE_Messages = COSE_Untagged_Message / COSE_Tagged_Message

COSE_Untagged_Message = COSE_Sign / COSE_Sign1 /
    COSE_Enveloped /
    COSE_Encrypted /
    COSE_Mac / COSE_Mac0

COSE_Tagged_Message = COSE_Sign_Tagged / COSE_Sign1_Tagged /
    COSE_Enveloped_Tagged /
    COSE_Encrypted_Tagged /
    COSE_Mac_Tagged / COSE_Mac0_Tagged
```

3. Header Parameters

The structure of COSE has been designed to have two buckets of information that are not considered to be part of the payload itself, but are used for holding information about content, algorithms, keys, or evaluation hints for the processing of the layer. These two buckets are available for use in all of the structures in this document except for keys. While these buckets can be present, they may not all be usable in all instances. For example, while the protected bucket is defined as part of recipient structures, most of the algorithms that are used for recipients do not provide the necessary functionality to provide the needed protection and thus the bucket should not be used.

Both buckets are implemented as CBOR maps. The map key is a 'label' (Section 1.4). The value portion is dependent on the definition for the label. Both maps use the same set of label/value pairs. The integer and string values for labels has been divided into several sections with 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 16.2).

Two buckets are provided for each layer:

protected: Contains parameters about the current layer that are to be 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 an empty protected map as a zero length binary object (it is shorter). Recipients MUST accept both a zero length binary value 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 finesses 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 parameters that deal with the current layer are to be placed at that layer. As an example of this, the parameter 'content type' describes the content of the message being carried in the message. As such, this 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. (The only data that should need 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 parameters that go into the buckets come from the IANA "COSE Header Parameters" (Section 16.2). Some common parameters are defined in the next section, but a number of parameters are defined throughout this document.

The following CDDL fragment represents the two header buckets. A group Headers is defined in CDDL which 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 which represents the map of header values. It uses forward references to a group definition of headers for generic and algorithms.

```
Headers = (
    protected : bstr,                 ; Contains a header_map
    unprotected : header_map
)

header_map = {
    Generic_Headers,
    ; Algorithm_Headers,
    + label => values
}
```

3.1. Common COSE Headers Parameters

This section defines a set of common header parameters. A summary of those parameters can be found in Table 2. This table should be consulted to determine the value of label used as well as the type of the value.

The set of header parameters defined in this section are:

alg This parameter is used to indicate the algorithm used for the security processing. This parameter MUST be present at each level of a signed, encrypted or authenticated message. When the algorithm supports authenticating extenral data, this parameter MUST be in the protected header bucket. The value is taken from the 'COSE Algorithm Registry' (see Section 16.4).

crit This parameter is used to ensure that applications will take appropriate action based on the values found. The parameter is used to indicate which protected header labels an application that is processing a message is required to understand. When present, this parameter MUST be placed in the protected header bucket.

* Integer labels in the range of 0 to 10 SHOULD be omitted.

* Integer labels in the range -1 to -255 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 parameters associated with that algorithm. (The algorithm range is -1 to -65536, it is assumed that the higher end will deal with more optional algorithm specific items.)

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

content type This 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. Strings are from the IANA 'Media Types' registry. Applications SHOULD provide this parameter if the content structure is potentially ambiguous.

kid This parameter one of the ways that can be used to find the key to be used. The value of this parameter is matched against the 'kid' member in a COSE_Key structure. Applications MUST NOT assume that 'kid' values are unique. There may be more than one key with the same 'kid' value, it may be required that all of the keys need to be checked to find the correct one. The internal structure of 'kid' values is not defined and generally cannot be relied on by applications. Key identifier values are hints about which key to use. They are not directly a security critical field. For this reason, they can be placed in the unprotected headers bucket.

Initialization Vector This parameter holds the Initialization Vector (IV) value. For some symmetric encryption algorithms this may be referred to as a nonce. As the IV is authenticated by encryption process, it can be placed in the unprotected header bucket.

Partial Initialization Vector This parameter holds a part of the IV value. When using the COSE_Encrypted structure, frequently a portion of the IV is part of the context associated with the key value. This field is used to carry the portion of the IV that changes for each message. As the IV is authenticated by the encryption process, this value can be placed in the unprotected header bucket.
The final IV is generated by concatenating the fixed portion of the IV, a zero string and the changing portion of the IV. The length of the zero string is computed by taking the required IV length and subtracting the lengths of the fixed and changing IV portions.

counter signature This parameter holds a counter signature value. Counter signatures provide a method of having a second party sign some data. The counter signature can occur as an unprotected attribute in any of the following structures: COSE_Sign, COSE_Sign1, COSE_Signature, COSE_Enveloped, COSE_recipient, COSE_Encrypted, COSE_Mac and COSE_Mac0. These structures all have the same basic structure so that a consistent calculation of the counter signature can be computed. Details on computing counter signatures are found in Section 4.5.

operation time This parameter provides the time the content cryptographic operation is performed. For signatures and recipient structures, this would be the time that the signature or recipient key object was created. For content structures, this would be the time that the content structure was created. The unsigned integer value is the number of seconds, excluding leap seconds, after midnight UTC, January 1, 1970. The field is primarily intended to be to be used for countersignatures, however it can additionally be used for replay detection as well.

name	label	value type	value registry	description
alg	1	int / tstr	COSE Algorithm Registry	Integers are taken from table --POINT TO REGISTRY--
crit	2	[+ label]	COSE Header Label Registry	integer values are from -- POINT TO REGISTRY --
content type	3	tstr / int	CoAP Content-Formats or Media Types registry	Value is either a Media Type or an integer from the CoAP Content Format registry
kid	4	bstr		key identifier
IV	5	bstr		Full Initialization Vector
Partial IV	6	bstr		Partial Initialization Vector
counter signature	7	COSE_Signature		CBOR encoded signature structure
operation time	8	uint		Time the COSE structure was created

Table 2: Common Header Parameters

The CDDL fragment that represents the set of headers defined in this section is given below. Each of the headers is tagged as optional because they do not need to be in every map, headers 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, ; Counter signature
    ? 8 => uint         ; Operation time
)
```

4. Signing Objects

COSE supports two different signature structures. COSE_Sign allows for one or more signers to be applied to a single content. COSE_Sign1 is restricted to a single signer.

4.1. Signing with One or More Signers

The signature structure allows for one or more signatures to be applied to a message payload. There are provisions for parameters about the content and parameters about the signature to be carried along with the signature itself. These parameters may be authenticated by the signature, or just present. Examples of parameters about the content would be the type of content, when the content was created, and who created the content. [CREF3] Examples of parameters about the signature would be the algorithm and key used to create the signature, when the signature was created, and counter-signatures.

When more than one signature is present, the successful validation of one signature associated with a given signer is usually treated as a successful signature by that signer. However, there are some application environments where other rules are needed. An application that employs a rule other than one valid signature for each signer must specify those rules. Also, where simple matching of the signer identifier is not sufficient to determine whether the signatures were generated by the same signer, the application specification must describe how to determine which signatures were generated by the same signer. Support of different communities of recipients is the primary reason that signers choose to include more than one signature. For example, the COSE_Sign structure might include signatures generated with the RSA signature algorithm and with the Elliptic Curve Digital Signature Algorithm (ECDSA) signature algorithm. This allows recipients to verify the signature associated with one algorithm or the other. (The original source of this text is [RFC5652].) More detailed information on multiple signature evaluation can be found in [RFC5752].

The signature structure can be encoded either with or without a tag depending on the context it will be used in. The signature structure is identified by the CBOR tag TBD1. The CDDL fragment that represents this is.

```
COSE_Sign_Tagged = #6.991(COSE_Sign) ; Replace 991 with TBD1
```

A COSE Signing Message is divided into 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 Signing Messages can be found in Appendix B.3.

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

protected is described in Section 3.

unprotected is described in Section 3.

payload 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, 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 message recovery algorithm is used (Section 8), 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 binary string rather than as being absent.

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

The CDDL fragment which 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 is described in Section 3.

unprotected is described in Section 3.

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

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

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

4.2. Signing with One Signer

The signature structure can be encoded either with or without a tag depending on the context it will be used in. The signature structure is identified by the CBOR tag TBD7. The CDDL fragment that represents this is:

```
COSE_Sign1_Tagged = #6.997(COSE_Sign1) ; Replace 997 with TBD7
```

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

protected is described in Section 3.

unprotected is described in Section 3.

payload is described in Section 4.1.

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

The CDDL fragment which 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 that we supply in the COSE document is the ability for applications to provide additional data to be authenticated as part of the security, 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. An example of data that can be placed in this location would be CoAP options for transaction ids and nonces to check for replay protection. If the data is in the options section, then it is available for routers to help in performing the replay detection and prevention. However, it may also be desired to protect these values so that they cannot be modified in transit. This is the purpose of the externally supplied data field.

This document describes the process for using a byte array of externally supplied authenticated data, however 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, care needs to be taken that data cannot bleed between the items. This is usually addressed by making fields fixed width and/or encoding the length of the field. 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, a defined 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.

4.4. Signing and Verification Process

In order to create a signature, a consistent byte stream is needed in order to process. This algorithm takes in the body information (COSE_Sign), the signer information (COSE_Signer), and the application data (External). A CBOR array is used to construct the byte stream to be processed. The fields of the array in order are:

1. A text string identifying the context that this signature is being used in. The context 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.

2. The protected attributes from the body structure encoded in a bstr type.

3. The protected attributes from the signer structure encoded in a bstr type. This field is omitted for the COSE_Sign1 signature structure.

4. The protected attributes from the application encoded in a bstr type. If this field is not supplied, it defaults to a zero length binary string.

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 which describes the above text is.

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

How to compute a signature:

1. Create a Sig_structure and populate it with the appropriate fields. For body_protected and sign_protected, if the map is empty, a bstr of length zero is used.

2. If the application has supplied external additional authenticated data to be included in the computation, then it is placed in the third field. If no data was supplied, then a zero length binary string is used.

3. Create the value ToBeSigned by encoding the Sig_structure to a byte string.

4. 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).

5. Place the resulting signature value in the 'signature' field of the map.

How to verify a signature:

1. Create a Sig_structure object and populate it with the appropriate fields. For body_protected and sign_protected, if the map is empty, a bstr of length zero is used.

2. If the application has supplied external additional authenticated data to be included in the computation, then it is placed in the third field. If no data was supplied, then a zero length binary string is used.

3. Create the value ToBeSigned by encoding the Sig_structure to a byte string.

4. 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, one must also perform the appropriate checks to ensure that the key is correctly paired with the signing identity and that the appropriate authorization is done.

4.5. Computing Counter Signatures

Counter signatures provide a method of having a different signature occur on some piece of content. This is normally used to provide a signature on a signature allowing for a proof that a signature existed at a given time (i.e. a Timestamp). In this document we allow for counter signatures to exist in a greater number of environments. As an example, it is possible to place a counter signature in the unprotected attributes of a COSE_Enveloped object. This would allow for an intermediary to either verify that the encrypted byte stream has not been modified, without being able to decrypt it. Or for the intermediary to assert that an encrypted byte stream either existed at a given time or passed through it in terms of routing (i.e. a proxy signature).

An example of a proxy signature on a signature can be found in Appendix B.3.3. An example of a proxy signature on an encryption object can be found in Appendix B.2.3.

The creation and validation of counter signatures over the different items relies on the fact that the structure all of our objects have the same structure. The elements are a set of protected attributes, a set of unprotected attributes and a body in that order. This means that the Sig_structure can be used for in a uniform manner to get the byte stream for processing a signature. If the counter signature is

going to be computed over a COSE_Enveloped structure, the body_protected and payload items can be mapped into the Sig_structure in the same manner as from the COSE_Sign structure.

It should be noted that only signature algorithm with appendix (see Section 8) should be used for counter signatures. This is because the body should be able to be processed without having to evaluate the countersignature, and this is not possible for signature schemes with message recovery.

5. Encryption Objects

COSE supports two different encryption structures. COSE_Enveloped is used when the key needs to be explicitly identified. This structure supports the use of recipient structures to allow for random content encryption keys to be used. COSE_Encrypted is used when a recipient structure is not needed because the key to be used is known implicitly.

5.1. Enveloped COSE Structure

The enveloped structure allows for one or more recipients of a message. There are provisions for parameters about the content and parameters about the recipient information to be carried in the message. The parameters associated with the content can be authenticated by the content encryption algorithm. The parameters associated with the recipient can be authenticated by the recipient algorithm (when the algorithm supports it). Examples of parameters about the content are the type of the content, when the content was created, and the content encryption algorithm. Examples of parameters about the recipient are the recipient's key identifier, the recipient encryption algorithm. [CREF4]

In COSE, the same techniques and structures are used for encrypting both the plain text and the keys used to protect the text. This is different from the approach used by both [RFC5652] and [RFC7516] where different structures are used for the content layer and for the recipient layer. Two structures are defined COSE_Enveloped to hold the encrypted content and COSE_recipient to hold the encrypted keys for recipients. Examples of encrypted messages can be found in Appendix B.2.

The COSE Enveloped structure can be encoded either with or without a tag depending on the context it will be used in. The COSE Enveloped structure is identified by the CBOR tag TBD2. The CDDL fragment that represents this is.

```
COSE_Enveloped_Tagged = #6.992(COSE_Enveloped) ; Replace 992 with TBD2
```

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

protected is described in Section 3.

unprotected is described in Section 3.

ciphertext contains the encrypted plain text 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 null object.

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

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

protected is described in Section 3.

unprotected is described in Section 3.

ciphertext contains the encrypted key encoded as a bstr. If there is not an encrypted key, then this field is encoded as a nil value.

recipients contains an array of recipient information structures. The type for the recipient information structure is a COSE_recipient. If there are no recipient information structures, this element is absent.

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

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

5.1.1. Recipient Algorithm Classes

A typical 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 depends on which class the recipient algorithm falls into. Specific details on each of the classes can be found in Section 12. A short summary of the five recipient algorithm classes is:

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

symmetric key-encryption keys: The CEK is encrypted using a previously distributed symmetric KEK.

key agreement: The recipient's public key and a sender's private key are used to generate a pairwise secret, a 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.

5.2. Encrypted COSE structure

The 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.

The structure defined to hold an encrypted message is COSE_Encrypted. Examples of encrypted messages can be found in Appendix B.2.

The COSE_Encrypted structure can be encoded either with or without a tag depending on the context it will be used in. The COSE_Encrypted structure is identified by the CBOR tag TBD3. The CDDL fragment that represents this is.

```
COSE_Encrypted_Tagged = #6.993(COSE_Encrypted) ; Replace 993 with TBD3
```

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

protected is described in Section 3.

unprotected is described in Section 3.

ciphertext contains the encrypted plain text. 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 null value.

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

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

5.3. Encryption Algorithm for AEAD algorithms

The encryption algorithm for AEAD algorithms is fairly simple.

1. Create a CBOR array (Enc_structure) to encode the authenticated data.

2. Place a context string in the form of a tstr in the first location to identify the data and location being encoded. The strings defined are:

   "Encrypted" for the the content encryption of an encrypted data structure.

   "Enveloped" for the first level of an enveloped data structure (i.e. for content encryption).

   "Env_Recipient" for a recipient encoding to be placed in an enveloped data structure.

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

3. Copy the protected header field from the message to be sent to the second location in the Enc_structure.

4. If the application has supplied external additional authenticated data to be included in the computation, then it is placed in the third location ('external_aad' field) of the Enc_structure. If no data was supplied, then a zero length binary value is used.

(See Section 4.3 for application guidance on constructing this field.)

5. Encode the Enc_structure using a CBOR Canonical encoding Section 14 to get the AAD value.

6. 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 level.

   Direct 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 randomly generated.

7. Call the encryption algorithm with K (the encryption key to use), P (the plain text) and AAD (the additional authenticated data). Place the returned cipher text into the 'ciphertext' field of the structure.

8. For recipients of the message, recursively perform the encryption algorithm for that recipient using the encryption key as the plain text.

The CDDL fragment which defines the Enc_structure used for the authenticated data structure is:

```
Enc_structure = [
    context : "Enveloped" / "Encrypted" / "Env_Recipient" /
        "Mac_Recipient",
    protected: bstr,
    external_aad: bstr
]
```

5.4. Encryption algorithm for AE algorithms

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

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 level.

Direct 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 randomly generated.

4. Call the encryption algorithm with K (the encryption key to use) and the P (the plain text). Place the returned cipher text into the 'ciphertext' field of the structure.

5. For recipients of the message, recursively perform the encryption algorithm for that recipient using the encryption key as the plain text.

6. MAC Objects

COSE supports two different MAC structures. COSE_MAC is used when the key needs to be explicitly identified. The structure supports the use of recipient structures to allow for random content encryption keys to be used. COSE_MAC0 is used when the a recipient structure is not needed because the key to be used is implicitly known.

6.1. MAC Message with Recipients

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 it 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 key agreement (without the key wrap step). In both of these cases, the entity hat created and sent the message MAC can be validated. (The knowledge of sender assumes that there are only two parties involved and you did not send the message yourself.)

The MAC message uses two structures, the COSE_Mac structure defined in this section for carrying the body and the COSE_recipient structure (Section 5.1) to hold the key used for the MAC computation. Examples of MAC messages can be found in Appendix B.1.

The MAC structure can be encoded either withor without a tag depending on the context it will be used in. The MAC truture is identified by the CBOR tag TBD4. The CDDL fragment that represents this is:

```
COSE_Mac_Tagged = #6.994(COSE_Mac)         ; Replace 994 with TBD4
```

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

protected is described in Section 3.

unprotected is described in Section 3.

payload 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, then a null CBOR object is placed in this location and it is the responsibility of the application to ensure that it will be transported without changes.

tag contains the MAC value.

recipients contains the recipient information. See the description under COSE_Encryption for more info.

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

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

6.2. MAC 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 MAC message uses the COSE_Mac0 structure defined in this section for carrying the body.

The MAC structure can be encoded either withor without a tag depending on the context it will be used in. The MAC structure is identified by the CBOR tag TBD6. The CDDL fragment that represents this is:

```
COSE_Mac0_Tagged = #6.996(COSE_Mac0)    ; Replace 996 with TBD6
```

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

protected is described in Section 3.

unprotected is described in Section 3.

payload 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, then a null CBOR object is placed in this location and it is the responsibility of the application to ensure that it will be transported without changes.

tag contains the MAC value.

The CDDL fragment which corresponds to the above text is:

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

6.3. How to compute 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 text string that identifies the structure that is being encoded. This string is "MAC" for the COSE_Mac structure. This 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 binary string is to be encoded.

3. If the application has supplied external authenticated data, encode it as a binary value and place in the MAC_structure. If there is no external authenticated data, then use a zero length 'bstr'. (See Section 4.3 for application guidance on constructing this field.)

4. The payload to be MAC-ed 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: bstr,
     external_aad: bstr,
     payload: bstr
]
```

The steps to compute a MAC are:

1. Create a MAC_structure and fill in the fields.

2. Encode the MAC_structure using a canonical CBOR encoder. The resulting bytes is the value to compute the MAC on.

3. Compute the MAC and place the result in the 'tag' field of the COSE_Mac structure.

4. Encrypt and encode the MAC key for each recipient of the message.

7. Key Structure

A COSE Key structure is built on a CBOR map object. The set of common parameters that can appear in a COSE Key can be found in the IANA registry 'COSE Key Common Parameter Registry' (Section 16.5). Additional parameters defined for specific key types can be found in the IANA registry 'COSE Key Type Parameters' (Section 16.6).

A COSE Key Set uses a CBOR array object as its underlying type. The values of the array elements are COSE Keys. A Key Set MUST have at least one element in the array.

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

The CDDL grammar describing a COSE_Key and COSE_KeySet is:

```
COSE_Key = {
    key_kty => tstr / int,
    ? key_ops => [+ (tstr / int) ],
    ? key_alg => tstr / int,
    ? key_kid => bstr,
    * label => values
}

COSE_KeySet = [+COSE_Key]
```

7.1. COSE Key Common Parameters

This document defines a set of common parameters for a COSE Key object. Table 3 provides a summary of the parameters defined in this section. There are also a set of parameters that are defined for a specific key type. Key type specific parameters can be found in Section 13.

name	label	CBOR type	registry	description
kty	1	tstr / int	COSE General Values	Identification of the key type
key_ops	4	[* (tstr/int)]		Restrict set of permissible operations
alg	3	tstr / int	COSE Algorithm Values	Key usage restriction to this algorithm
kid	2	bstr		Key Identification value - match to kid in message
use	*	tstr		deprecated - don't use

Table 3: 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

Table 19. 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 a trust decision process.

alg: This parameter is used to restrict the algorithms that are to be used with this 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 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 4.

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 wrapping.
unwrap key	6	The key is used for key unwrapping. Requires private key fields.
derive key	7	The key is used for deriving keys.
derive bits	8	The key is used for deriving bits.

Table 4: Key Operation Values

The following provides a CDDL fragment which duplicates the assignment labels from Table 3.

```
;key_labels
key_kty=1
key_kid=2
key_alg=3
key_ops=4
```

8. Signature Algorithms

There are two basic signature algorithm structures that can be used. The first is the common signature with appendix. In this structure, the message content is processed and a signature is produced, the signature is called the appendix. This is the message structure used by our common algorithms such as ECDSA and RSASSA-PSS. (In fact the SSA in RSASSA-PSS stands for Signature Scheme with Appendix.) The basic structure becomes:

signature = Sign(message content, key)

valid = Verification(message content, key, signature)

The second is a signature with message recovery. (An example of such an algorithm is [PVSig].) In this structure, 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 an effectively smaller signature, the signature size is still potentially large, but the message content is 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 which 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. Thirdly, 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.

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

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

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.

8.1. ECDSA

ECDSA [DSS] defines a signature algorithm using ECC.

The ECDSA signature algorithm is parameterized with a hash function (h). In the event that the length of the hash function output is greater than the group of the key, the left-most bytes of the hash output are used.

The algorithms defined in this document can be found in Table 5.

name	value	hash	description
ES256	-7	SHA-256	ECDSA w/ SHA-256
ES384	-8	SHA-384	ECDSA w/ SHA-384
ES512	-9	SHA-512	ECDSA w/ SHA-512

Table 5: ECDSA Algorithm Values

This document defines ECDSA to work only with the curves P-256, P-384 and P-521. This document requires that the curves be encoded using the 'EC2' key type. Implementations need to check that the key type and curve are correct when creating and verifying a signature. Other documents can defined it to work with other curves and points in the future.

In order to promote interoperability, it is suggested that SHA-256 be used only with curve P-256, SHA-384 be used only with curve P-384 and SHA-512 be used with curve P-521. This is aligned with the recommendation in Section 4 of [RFC5480].

The signature algorithm results in a pair of integers (R, S). These integers will be of the same order as length of the key used for the signature process. The signature is encoded by converting the integers into byte strings of the same length as the key size. The length is rounded up to the nearest byte and is left padded with zero bits to get to the correct length. The two integers are then concatenated together to form a byte string that is the resulting signature.

Using the function defined in [RFC3447] the signature is:
Signature = I2OSP(R, n) | I2OSP(S, n)
where n = ceiling(key_length / 8)

When using a COSE key for this algorithm, the following checks are made:

- The 'kty' field MUST be present and it MUST be 'EC2'.

- If the 'alg' field present, it MUST match the ECDSA signature algorithm being used.

- If the 'key_ops' field is present, it MUST include 'sign' when creating an ECDSA signature.

- o If the 'key_ops' field is present, it MUST include 'verify' when verifying an ECDSA signature.

8.1.1. Security Considerations

The security strength of the signature is no greater than the minimum of the security strength associated with the bit length of the key and the security strength of the hash function.

System which have poor random number generation can leak their keys by signing two different messages with the same value 'k' (the per-message random value). [RFC6979] provides a method to deal with this problem by making 'k' be deterministic based on the message content rather than randomly generated. Applications that specify ECDSA should evaluate the ability to get good random number generation and require this when it is not possible.

Note: Use of this technique a good idea even when good random number generation exists. Doing so both reduces the possibility of having the same value of 'k' in two signature operations, but allows for reproducible signature values which helps testing.

There are two substitution attacks that can theoretically be mounted against the ECDSA signature algorithm.

- o Changing the curve used to validate the signature: If one changes the curve used to validate the signature, then potentially one could have a two messages with the same signature each computed under a different curve. The only requirement on the new curve is that its order be the same as the old one and it be acceptable to the client. An example would be to change from using the curve secp256r1 (aka P-256) to using secp256k1. (Both are 256 bit curves.) We current do not have any way to deal with this version of the attack except to restrict the overall set of curves that can be used.

- o Change the hash function used to validate the signature: If one has either two different hash functions of the same length, or one can truncate a hash function down, then one could potentially find collisions between the hash functions rather than within a single hash function. (For example, truncating SHA-512 to 256 bits might collide with a SHA-256 bit hash value.) This attack can be mitigated by including the signature algorithm identifier in the data to be signed.

9. Message Authentication (MAC) Algorithms

Message Authentication Codes (MACs) provide data authentication and integrity protection. 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.)

MACs use the same basic structure 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 basic structure becomes:

```
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. HMAC). This document defines a MAC algorithm for each of these two constructions.

9.1. Hash-based Message Authentication Codes (HMAC)

The Hash-base Message Authentication Code algorithm (HMAC) [RFC2104][RFC4231] was designed to deal with length extension attacks. The algorithm was also designed to allow for new hash algorithms to be directly plugged in without changes to the hash function. The HMAC design process has been vindicated as, while the security of hash algorithms such as MD5 has decreased over time, the security of HMAC combined with MD5 has not yet been shown to be compromised [RFC6151].

The HMAC algorithm is parameterized by an inner and outer padding, a hash function (h) and an authentication tag value length. For this specification, the inner and outer padding are fixed to the values set in [RFC2104]. The length of the authentication tag corresponds to the difficulty of producing a forgery. For use in constrained environments, we define a set of HMAC algorithms that are truncated. There are currently no known issues when truncating, however the security strength of the message tag is correspondingly reduced in strength. When truncating, the left-most tag length bits are kept and transmitted.

The algorithm defined in this document can be found in Table 6.

name	value	Hash	Length	description
HMAC 256/64	*	SHA-256	64	HMAC w/ SHA-256 truncated to 64 bits
HMAC 256/256	4	SHA-256	256	HMAC w/ SHA-256
HMAC 384/384	5	SHA-384	384	HMAC w/ SHA-384
HMAC 512/512	6	SHA-512	512	HMAC w/ SHA-512

Table 6: HMAC Algorithm Values

Some recipient algorithms carry the key while others derive a key from secret data. For those algorithms that carry the key (i.e. RSA-OAEP and AES-KeyWrap), the size of the HMAC key SHOULD be the same size as the underlying hash function. For those algorithms that derive the key, the derived key MUST be the same size as the underlying hash function.

When using a COSE key for this algorithm, the following checks are made:

- The 'kty' field MUST be present and it MUST be 'Symmetric'.

- If the 'alg' field present, it MUST match the HMAC algorithm being used.

- If the 'key_ops' field is present, it MUST include 'sign' when creating an HMAC authentication tag.

- If the 'key_ops' field is present, it MUST include 'verify' when verifying an HMAC authentication tag.

Implementations creating and validating MAC values MUST validate that the key type, key length, and algorithm are correct and appropriate for the entities involved.

9.1.1. Security Considerations

HMAC has proved to be resistant to attack even when used with weakening hash algorithms. The current best method appears to be a

brute force attack on the key. This means that key size is going to be directly related to the security of an HMAC operation.

9.2. AES Message Authentication Code (AES-CBC-MAC)

AES-CBC-MAC is defined in [MAC].

AES-CBC-MAC is parameterized by the key length, the authentication tag length and the IV used. For all of these algorithms, the IV is fixed to all zeros. We provide an array of algorithms for various key lengths and tag lengths. The algorithms defined in this document are found in Table 7.

name	value	key length	tag length	description
AES-MAC 128/64	*	128	64	AES-MAC 128 bit key, 64-bit tag
AES-MAC 256/64	*	256	64	AES-MAC 256 bit key, 64-bit tag
AES-MAC 128/128	*	128	128	AES-MAC 128 bit key, 128-bit tag
AES-MAC 256/128	*	256	128	AES-MAC 256 bit key, 128-bit tag

Table 7: AES-MAC Algorithm Values

Keys may be obtained either from a key structure or from a recipient structure. Implementations creating and validating MAC values MUST validate that the key type, key length and algorithm are correct and appropriate for the entities involved.

When using a COSE key for this algorithm, the following checks are made:

- o The 'kty' field MUST be present and it MUST be 'Symmetric'.
- o If the 'alg' field present, it MUST match the AES-MAC algorithm being used.
- o If the 'key_ops' field is present, it MUST include 'sign' when creating an AES-MAC authentication tag.

- If the 'key_ops' field is present, it MUST include 'verify' when verifying an AES-MAC authentication tag.

9.2.1. Security Considerations

A number of attacks exist against CBC-MAC that need to be considered.

- A single key must only be used for messages of a fixed and known length. If this is not the case, an attacker will be able to generate a message with a valid tag given two message, tag pairs. This can be addressed by using different keys for different length messages. (CMAC mode also addresses this issue.)

- If the same key is used for both encryption and authentication operations, using CBC modes an attacker can produce messages with a valid authentication code.

- If the IV can be modified, then messages can be forged. This is addressed by fixing the IV to all zeros.

10. 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 symmetric key is obtained.

We restrict 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 sake, the authentication code will normally be defined as being appended to the cipher text stream. The basic structure becomes:

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

valid, message content = Decrypt(cipher text, 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.

10.1. AES GCM

The GCM mode is a generic authenticated encryption block cipher mode defined in [AES-GCM]. The GCM mode is combined with the AES block encryption algorithm to define an AEAD cipher.

The GCM mode is parameterized with by the size of the authentication tag and the size of the nonce. This document fixes the size of the nonce at 96-bits. The size of the authentication tag is limited to a small set of values. For this document however, the size of the authentication tag is fixed at 128 bits.

The set of algorithms defined in this document are in Table 8.

name	value	description
A128GCM	1	AES-GCM mode w/ 128-bit key, 128-bit tag
A192GCM	2	AES-GCM mode w/ 192-bit key, 128-bit tag
A256GCM	3	AES-GCM mode w/ 256-bit key, 128-bit tag

Table 8: Algorithm Value for AES-GCM

Keys may be obtained either from a key structure or from a recipient structure. Implementations encrypting and decrypting MUST validate that the key type, key length and algorithm are correct and appropriate for the entities involved.

When using a COSE key for this algorithm, the following checks are made:

- o The 'kty' field MUST be present and it MUST be 'Symmetric'.
- o If the 'alg' field present, it MUST match the AES-GCM algorithm being used.
- o If the 'key_ops' field is present, it MUST include 'encrypt' or 'key wrap' when encrypting.
- o If the 'key_ops' field is present, it MUST include 'decrypt' or 'key unwrap' when decrypting.

10.1.1. Security Considerations

When using AES-CCM, the following restrictions MUST be enforced:

- o The key and nonce pair MUST be unique for every message encrypted.

- o The total amount of data encrypted MUST NOT exceed 2^39 - 256 bits. An explicit check is required only in environments where it is expected that it might be exceeded.

Consideration was given to supporting smaller tag values, the constrained community would desire tag sizes in the 64-bit range. Doing show drastically changes both the maximum messages size (generally not an issue) and the number of times that a key can be used. Given that CCM is the usual mode for constrained environments restricted modes are not supported.

10.2. AES CCM

Counter with CBC-MAC (CCM) is a generic authentication encryption block cipher mode defined in [RFC3610]. The CCM mode is combined with the AES block encryption algorithm to define a commonly used content encryption algorithm used in constrained devices.

The CCM mode has two parameter choices. The first choice is M, the size of the authentication field. The choice of the value for M involves a trade-off between message expansion and the probably that an attacker can undetectably modify a message. The second choice is L, the size of the length field. This value requires a trade-off between the maximum message size and the size of the Nonce.

It is unfortunate that the specification for CCM specified L and M as a count of bytes rather than a count of bits. This leads to possible misunderstandings where AES-CCM-8 is frequently used to refer to a version of CCM mode where the size of the authentication is 64 bits and not 8 bits. These values have traditionally been specified as bit counts rather than byte counts. This document will follow the tradition of using bit counts so that it is easier to compare the different algorithms presented in this document.

We define a matrix of algorithms in this document over the values of L and M. Constrained devices are usually operating in situations where they use short messages and want to avoid doing recipient specific cryptographic operations. This favors smaller values of M and larger values of L. Less constrained devices do will want to be able to user larger messages and are more willing to generate new keys for every operation. This favors larger values of M and smaller values of L. (The use of a large nonce means that random generation

of both the key and the nonce will decrease the chances of repeating the pair on two different messages.)

The following values are used for L:

16 bits (2) limits messages to 2^16 bytes (64 KiB) in length. This sufficiently long for messages in the constrained world. The nonce length is 13 bytes allowing for 2^(13*8) possible values of the nonce without repeating.

64 bits (8) limits messages to 2^64 bytes in length. The nonce length is 7 bytes allowing for 2^56 possible values of the nonce without repeating.

The following values are used for M:

64 bits (8) produces a 64-bit authentication tag. This implies that there is a 1 in 2^64 chance that a modified message will authenticate.

128 bits (16) produces a 128-bit authentication tag. This implies that there is a 1 in 2^128 chance that a modified message will authenticate.

name	value	L	M	k	description
AES-CCM-16-64-128	10	16	64	128	AES-CCM mode 128-bit key, 64-bit tag, 13-byte nonce
AES-CCM-16-64-256	11	16	64	256	AES-CCM mode 256-bit key, 64-bit tag, 13-byte nonce
AES-CCM-64-64-128	30	64	64	128	AES-CCM mode 128-bit key, 64-bit tag, 7-byte nonce
AES-CCM-64-64-256	31	64	64	256	AES-CCM mode 256-bit key, 64-bit tag, 7-byte nonce
AES-CCM-16-128-128	12	16	128	128	AES-CCM mode 128-bit key, 128-bit tag, 13-byte nonce
AES-CCM-16-128-256	13	16	128	256	AES-CCM mode 256-bit key, 128-bit tag, 13-byte nonce
AES-CCM-64-128-128	32	64	128	128	AES-CCM mode 128-bit key, 128-bit tag, 7-byte nonce
AES-CCM-64-128-256	33	64	128	256	AES-CCM mode 256-bit key, 128-bit tag, 7-byte nonce

Table 9: Algorithm Values for AES-CCM

Keys may be obtained either from a key structure or from a recipient structure. Implementations encrypting and decrypting MUST validate that the key type, key length and algorithm are correct and appropriate for the entities involved.

When using a COSE key for this algorithm, the following checks are made:

- The 'kty' field MUST be present and it MUST be 'Symmetric'.

- If the 'alg' field present, it MUST match the AES-CCM algorithm being used.

- If the 'key_ops' field is present, it MUST include 'encrypt' or 'key wrap' when encrypting.

- If the 'key_ops' field is present, it MUST include 'decrypt' or 'key unwrap' when decrypting.

10.2.1. Security Considerations

When using AES-CCM, the following restrictions MUST be enforced:

- The key and nonce pair MUST be unique for every message encrypted.

- The total number of times the AES block cipher is used MUST NOT exceed 2^61 operations. This limitation is the sum of times the block cipher is used in computing the MAC value and in performing stream encryption operations. An explicit check is required only in environments where it is expected that it might be exceeded.

[RFC3610] additionally calls out one other consideration of note. It is possible to do a pre-computation attack against the algorithm in cases where the portions encryption content is highly predictable. This reduces the security of the key size by half. Ways to deal with this attack include adding a random portion to the nonce value and/or increasing the key size used. Using a portion of the nonce for a random value will decrease the number of messages that a single key can be used for. Increasing the key size may require more resources in the constrained device. See sections 5 and 10 of [RFC3610] for more information.

10.3. ChaCha20 and Poly1305

ChaCha20 and Poly1305 combined together is a new AEAD mode that is defined in [RFC7539]. This is a new algorithm defined to be a cipher that is not AES and thus would not suffer from any future weaknesses found in AES. These cryptographic functions are designed to be fast in software-only implementations.

The ChaCha20/Poly1305 AEAD construction defined in [RFC7539] has no parameterization. It takes a 256-bit key and a 96-bit nonce as well as the plain text and additional data as inputs and produces the

cipher text as an option. We define one algorithm identifier for this algorithm in Table 10.

name	value	description
ChaCha20/Poly1305	24	ChaCha20/Poly1305 w/ 256-bit key, 128-bit tag

Table 10: Algorithm Value for AES-GCM

Keys may be obtained either from a key structure or from a recipient structure. Implementations encrypting and decrypting MUST validate that the key type, key length and algorithm are correct and appropriate for the entities involved.

When using a COSE key for this algorithm, the following checks are made:

- o The 'kty' field MUST be present and it MUST be 'Symmetric'.
- o If the 'alg' field present, it MUST match the ChaCha algorithm being used.
- o If the 'key_ops' field is present, it MUST include 'encrypt' or 'key wrap' when encrypting.
- o If the 'key_ops' field is present, it MUST include 'decrypt' or 'key unwrap' when decrypting.

10.3.1. Security Considerations

The pair of key, nonce MUST be unique for every invocation of the algorithm. Nonce counters are considered to be an acceptable way of ensuring that they are unique.

11. Key Derivation Functions (KDF)

Key Derivation Functions (KDFs) are used to take some secret value and generate a different one. The original secret values come in three basic flavors:

- o Secrets that are uniformly random: This is the type of secret which is created by a good random number generator.
- o Secrets that are not uniformly random: This is type of secret which is created by operations like key agreement.

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

General KDF functions work well with the first type of secret, can do reasonable well with the second type of secret and generally do poorly with the last type of secret. None of the KDF functions in this section are designed to deal with the type of secrets that are used for passwords. Functions like PBSE2 [RFC2898] need to be used for that type of secret.

Many functions are going to handle the first two type of secrets differently. The KDF function defined in Section 11.1 can use different underlying constructions if the secret is uniformly random than if the secret is not uniformly random. This is reflected in the set of algorithms defined for HKDF.

When using KDF functions, one component that is generally 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. This document defines a single context structure and a single KDF function.

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

The HKDF key derivation algorithm is defined in [RFC5869].

The HKDF algorithm takes these inputs:

secret - a shared value that is secret. Secrets may be either previously shared or derived from operations like a DH key agreement.

salt - an optional public value that is used to change the generation process. If specified, the salt is carried using the 'salt' algorithm parameter. While [RFC5869] suggests that the length of the salt be the same as the length of the underlying hash value, any amount of salt will improve the security as different key values will be generated. A parameter to carry the salt is defined in Table 12. This parameter is protected by being included in the key computation and does not need to be separately authenticated.

length - the number of bytes of output that need to be generated.

context information - Information that describes the context in which the resulting value will be used. Making this information specific to the context that the material is going to be used

ensures that the resulting material will always be unique. The context structure used is encoded into the algorithm identifier.

PRF - The underlying pseudo-random function to be used in the HKDF algorithm. The PRF is encoded into the HKDF algorithm selection.

HKDF is defined to use HMAC as the underlying PRF. However, it is possible to use other functions in the same construct to provide a different KDF function that may be more appropriate in the constrained world. Specifically, one can use AES-CBC-MAC as the PRF for the expand step, but not for the extract step. When using a good random shared secret of the correct length, the extract step can be skipped. The extract cannot be skipped if the secret is not uniformly random, for example if it is the result of an ECDH key agreement step.

The algorithms defined in this document are found in Table 11.

name	PRF	Skip extract	context
HKDF SHA-256	HMAC with SHA-256	no	HKDF using HMAC SHA-256 as the PRF
HKDF SHA-512	HMAC with SHA-512	no	HKDF using HMAC SHA-512 as the PRF
HKDF AES-MAC-128	AES-CBC-128	yes	HKDF using AES-MAC as the PRF w/ 128-bit key
HKDF AES-MAC-256	AES-CBC-256	yes	HKDF using AES-MAC as the PRF w/ 256-bit key

Table 11: HKDF algorithms

name	label	type	description
salt	-20	bstr	Random salt

Table 12: HKDF Algorithm Parameters

11.2. Context Information Structure

The context information structure is used to ensure that the derived keying material is "bound" to the context of the transaction. The context information structure used here is based on that defined in [SP800-56A]. By using CBOR for the encoding of the context information structure, we automatically get the same type and length separation of fields that is obtained by the use of ASN.1. This means that there is no need to encode the lengths for the base elements as it is done by the encoding used in JOSE (Section 4.6.2 of [RFC7518]). [CREF5]

The context information structure refers to PartyU and PartyV as the two parties which are doing the key derivation. Unless the application protocol defines differently, we assign PartyU to the entity that is creating the message and PartyV to the entity that is receiving the message. By doing this association, different keys will be derived for each direction as the context information is different in each direction.

Application protocols are free to define the roles differently. For example, they could assign the PartyU role to the entity that initiates the connection and allow directly sending multiple messages over the connection in both directions without changing the role information. It is still reommended that different keys be derived in each direction to avoid reflection problems.

The context structure is built from information that is known to both entities. This information can be obtained from a variety of sources:

o Fields can be define by the application. This is commonly used to assign names to parties.

o Fields can be defined by usage of the output. Examples of this are the algorithm and key size that are being generated.

o Fields can be defined by parameters from the message. We define a set of parameters in Table 13 which can be used to carry the values associated with the context structure. Examples of this are identities and nonce values. These parameters are designed to be placed in the unprotected bucket of the recipient structure. (They do not need to be in the protected bucket since they already are included in the cryptographic computation by virtue of being included in the context structure.)

We define a CBOR object to hold the context information. This object is referred to as CBOR_KDF_Context. The object is based on a CBOR array type. The fields in the array are:

AlgorithmID This field indicates the algorithm for which the key material will be used. This field is required to be present and is a copy of the algorithm identifier in the message. The field exists in the context information so that if the same environment is used for different algorithms, then completely different keys will be generated each of those algorithms. (This practice means if algorithm A is broken and thus can is easier to find, the key derived for algorithm B will not be the same as the key for algorithm B.)

PartyUInfo This field holds information about party U. The PartyUInfo is encoded as a CBOR structure. The elements of PartyUInfo are encoded in the order presented, however if the element does not exist no element is placed in the array. The elements of the PartyUInfo array are:

identity This contains the identity information for party U. The identities can be assigned in one of two manners. Firstly, a protocol can assign identities based on roles. For example, the roles of "client" and "server" may be assigned to different entities in the protocol. Each entity would then use the correct label for the data they send or receive. The second way is for a protocol to assign identities is to use a name based on a naming system (i.e. DNS, X.509 names).
We define an algorithm parameter 'PartyU identity' that can be used to carry identity information in the message. However, identity information is often known as part of the protocol and can thus be inferred rather than made explicit. If identity information is carried in the message, applications SHOULD have a way of validating the supplied identity information. The identity information does not need to be specified and can be left as absent.

nonce This contains a one time nonce value. The nonce can either be implicit from the protocol or carried as a value in the unprotected headers.
We define an algorithm parameter 'PartyU nonce' that can be used to carry this value in the message However, the nonce value could be determined by the application and the value determined from elsewhere.
This item is optional and can be absent.

other This contains other information that is defined by the protocol.

This item is optional and can be absent.

PartyVInfo This field holds information about party V. The PartyVInfo is encoded as a CBOR structure. For store and forward environments, the party V information may be minimal or even absent. The elements of PartyVInfo are encoded in the order presented, however if the element does not exist no element is placed in the array. The elements of the PartyVInfo array are:

identity This contains the identity information for party V. The identities can be assigned in one of two manners. Firstly, a protocol can assign identities based on roles. For example, the roles of "client" and "server" may be assigned to different entities in the protocol. Each entity would then use the correct label for the data they send or receive. The second way is for a protocol to assign identities is to use a name based on a naming system (i.e. DNS, X.509 names).
We define an algorithm parameter 'PartyU identity' that can be used to carry identity information in the message. However, identity information is often known as part of the protocol and can thus be inferred rather than made explicit. If identity information is carried in the message, applications SHOULD have a way of validating the supplied identity information. The identity information does not need to be specified and can be left as absent.

nonce This contains a one time nonce value. The nonce can either be implicit from the protocol or carried as a value in the unprotected headers.
We define an algorithm parameter 'PartyU nonce' that can be used to carry this value in the message However, the nonce value could be determined by the application and the value determined from elsewhere.
This item is optional and can be absent.

other This contains other information that is defined by the protocol.
This item is optional and can be absent.

SuppPubInfo This field contains public information that is mutually known to both parties.

keyDataLength This is set to the number of bits of the desired output value. (This practice means if algorithm A can use two different key lengths, the key derived for longer key size will not contain the key for shorter key size as a prefix.)

protected This field contains the protected parameter field.

other The field other is for free form data defined by the application. An example is that an application could defined two different strings to be placed here to generate different keys for a data stream vs a control stream. This field is optional and will only be present if the application defines a structure for this information. Applications that define this SHOULD use CBOR to encode the data so that types and lengths are correctly include.

SuppPrivInfo This field contains private information that is mutually known information. An example of this information would be a pre-existing shared secret. The field is optional and will only be present if the application defines a structure for this information. Applications that define this SHOULD use CBOR to encode the data so that types and lengths are correctly included.

name	label	type	description
PartyU identity	-21	bstr	Party U identity Information
PartyU nonce	-22	bstr / int	Party U provided nonce
PartyU other	-23	bstr	Party U other provided information
PartyV identity	-24	bstr	Party V identity Information
PartyV nonce	-25	bstr / int	Party V provided nonce
PartyV other	-26	bstr	Party V other provided information

Table 13: Context Algorithm Parameters

The following CDDL fragment corresponds to the text above.

```
COSE_KDF_Context = [
    AlgorithmID : int / tstr,
    PartyUInfo : [
        ? nonce : bstr / int,
        ? identity : bstr,
        ? other : bstr,
    ],
    PartyVInfo : [
        ? nonce : bstr,
        ? identity : bstr / tstr,
        ? other : bstr
    ],
    SuppPubInfo : [
        keyDataLength : uint,
        protected : bstr,
        ? other : bstr
    ],
    ? SuppPrivInfo : bstr
]
```

12. Recipient Algorithm Classes

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 and then a set of algorithms are specified for each of the classes. The names of the recipient algorithm classes used here are the same as are defined in [RFC7516]. Other specifications use different terms for the recipient algorithm classes or do not support some of our recipient algorithm classes.

12.1. 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 content key. When direct encryption mode is used, it MUST be the only mode used on the message.

The COSE_Enveloped structure for the recipient is organized as follows:

o The 'protected' field MUST be a zero length item unless it is used in the computation of the content key.

o The 'alg' parameter MUST be present.

o A parameter identifying the shared secret SHOULD be present.

- The 'ciphertext' field MUST be a zero length item.

- The 'recipients' field MUST be absent.

12.1.1. Direct Key

This recipient algorithm is the simplest, the identified key is directly used as the key for the next layer down in the message. There are no algorithm parameters defined for this algorithm. The algorithm identifier value is assigned in Table 14.

When this algorithm is used, the protected field MUST be zero length. The key type MUST be 'Symmetric'.

name	value	description
direct	-6	Direct use of CEK

Table 14: Direct Key

12.1.1.1. Security Considerations

This recipient algorithm has several potential problems that need to be considered:

- These keys need to have some method to be regularly updated over time. All of the content encryption algorithms specified in this document have limits on how many times a key can be used without significant loss of security.

- These keys need to be dedicated to a single algorithm. There have been a number of attacks developed over time when a single key is used for multiple different algorithms. One example of this is the use of a single key both for CBC encryption mode and CBC-MAC authentication mode.

- Breaking one message means all messages are broken. If an adversary succeeds in determining the key for a single message, then the key for all messages is also determined.

12.1.2. Direct Key with KDF

These recipient algorithms take a common shared secret between the two parties and applies the HKDF function (Section 11.1) using the context structure defined in Section 11.2 to transform the shared secret into the necessary key. The 'protected' field can be of non-

zero length. The 'protected' field is copied into the SuppPubInfo.protected field of the context structure. Either the 'salt' parameter of HKDF or the partyU 'nonce' parameter of the context structure MUST be present. The salt/nonce parameter can be generated either randomly or deterministically. The requirement is that it be a unique value for the key pair in question.

If the salt/nonce value is generated randomly, then it is suggested that the length of the random value be the same length as the hash function underlying HKDF. While there is no way to guarantee that it will be unique, there is a high probability that it will be unique. If the salt/nonce value is generated deterministically, it can be guaranteed to be unique and thus there is no length requirement.

A new IV must be used if the same key is used in more than one message. The IV can be modified in a predictable manner, a random manner or an unpredictable manner. One unpredictable manner that can be used is to use the HKDF function to generate the IV. If HKDF is used for generating the IV, the algorithm identifier is set to "IV-GENERATION".

When these algorithms are used, the key type MUST be 'symmetric'.

The set of algorithms defined in this document can be found in Table 15.

name	value	KDF	description
direct+HKDF-SHA-256	*	HKDF SHA-256	Shared secret w/ HKDF and SHA-256
direct+HKDF-SHA-512	*	HKDF SHA-512	Shared secret w/ HKDF and SHA-512
direct+HKDF-AES-128	*	HKDF AES-MAC-128	Shared secret w/ AES-MAC 128-bit key
direct+HKDF-AES-256	*	HKDF AES-MAC-256	Shared secret w/ AES-MAC 256-bit key

Table 15: Direct Key

When using a COSE key for this algorithm, the following checks are made:

o The 'kty' field MUST be present and it MUST be 'Symmetric'.

- If the 'alg' field present, it MUST match the KDF algorithm being used.

- If the 'key_ops' field is present, it MUST include 'deriveKey or 'deriveBits'.

12.1.2.1. Security Considerations

The shared secret needs to have some method to be regularly updated over time. The shared secret forms the basis of trust. Although not used directly, it should still be subject to scheduled rotation.

12.2. Key Wrapping

In key wrapping 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 wrapping algorithms for COSE are AE algorithms. Key wrapping 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 has some degree of organization and may in fact be repeating the same content. The use of Key Wrapping loses the weak data origination that is provided by the direct encryption algorithms.

The COSE_Enveloped 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 recipients field present, 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 plain text to be encrypted is the key from next layer down (usually the content layer).

- At a minimum, the 'unprotected' field MUST contain the 'alg' parameter and SHOULD contain a parameter identifying the shared secret.

12.2.1. AES Key Wrapping

The AES Key Wrapping algorithm is defined in [RFC3394]. This algorithm uses an AES key to wrap a value that is a multiple of 64 bits. As such, it can be used to wrap a key for any of the content encryption algorithms defined in this document. The algorithm requires a single fixed parameter, the initial value. This is fixed to the value specified in Section 2.2.3.1 of [RFC3394]. There are no public parameters that vary on a per invocation basis.

Keys may be obtained either from a key structure or from a recipient structure. If the key obtained from a key structure, the key type MUST be 'Symmetric'. Implementations encrypting and decrypting MUST validate that the key type, key length and algorithm are correct and appropriate for the entities involved.

When using a COSE key for this algorithm, the following checks are made:

- o The 'kty' field MUST be present and it MUST be 'Symmetric'.
- o If the 'alg' field present, it MUST match the AES Key Wrap algorithm being used.
- o If the 'key_ops' field is present, it MUST include 'encrypt' or 'key wrap' when encrypting.
- o If the 'key_ops' field is present, it MUST include 'decrypt' or 'key unwrap' when decrypting.

name	value	key size	description
A128KW	-3	128	AES Key Wrap w/ 128-bit key
A192KW	-4	192	AES Key Wrap w/ 192-bit key
A256KW	-5	256	AES Key Wrap w/ 256-bit key

Table 16: AES Key Wrap Algorithm Values

12.2.1.1. Security Considerations for AES-KW

The shared secret need to have some method to be regularly updated over time. The shared secret is the basis of trust.

12.3. Key Encryption

Key Encryption mode is also called key transport mode in some standards. Key Encryption mode differs from Key Wrap mode in that it uses an asymmetric encryption algorithm rather than a symmetric encryption algorithm to protect the key. This document defines one Key Encryption mode algorithm.

When using a key encryption algorithm, the COSE_Enveloped structure for the recipient is organized as follows:

- The 'protected' field MUST be absent.

- The plain text to be encrypted is the key from next layer down (usually the content layer).

- At a minimum, the 'unprotected' field MUST contain the 'alg' parameter and SHOULD contain a parameter identifying the asymmetric key.

12.4. 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 A).

The most commonly used key agreement algorithm used is Diffie-Hellman, but other variants exist. Since COSE is designed for a store and forward environment rather than an on-line environment, many of the DH variants cannot be used as the receiver of the message cannot provide any 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 message.

Two variants of DH that are easily supported are:

Ephemeral-Static 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 DH: where a static key is used for both the sender and the recipient. The use of static keys allows for 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 is required to ensure that a different key is created for each message.

In this specification, both variants are specified. This has been done to provide the weak data origination option for use with MAC operations.

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_Enveloped structure for the recipient is organized as follows:

- The 'protected' field MUST be absent.

- At a minimum, the 'unprotected' field MUST contain the 'alg' parameter and SHOULD contain a parameter identifying the recipient's asymmetric key.

- The 'unprotected' field MUST contain the 'epk' parameter.

12.4.1. ECDH

The basic mathematics for Elliptic Curve Diffie-Hellman can be found in [RFC6090].

ECDH is parameterized by the following:

- Curve Type/Curve: The curve selected controls not only the size of the shared secret, but the mathematics for computing the shared secret. The curve selected also controls how a point in the curve is represented and what happens for the identity points on the curve. In this specification, we allow for a number of different curves to be used. A set of curves are defined in Table 20. Since the only the math is changed by changing the curve, the curve is not fixed for any of the algorithm identifiers we define. Instead, it is defined by the points used.

- Ephemeral-static or static-static: The key agreement process may be done using either a static or an ephemeral key for the sender's side. When using ephemeral keys, the sender MUST generate a new ephemeral key for every key agreement operation. The ephemeral key is placed in the 'ephemeral key' parameter and MUST be present for all algorithm identifiers that use ephemeral keys. When using static keys, the sender MUST either generate a new random value or

otherwise create a unique value to be placed in either in the KDF parameters or the context structure. For the KDF functions used, this means either in the 'salt' parameter for HKDF (Table 12) or in the 'PartyU nonce' parameter for the context structure (Table 13) MUST be present. (Both may be present if desired.) The value in the parameter MUST be unique for the key pair being used. It is acceptable to use a global counter that is incremented for every static-static operation and use the resulting value. When using static keys, the static key needs to be identified to the recipient. The static key can be identified either by providing the key ('static key') or by providing a key identifier for the static key ('static key id'). Both of these parameters are defined in Table 18

- o Key derivation algorithm: The result of an ECDH key agreement process does not provide a uniformly random secret. As such, it needs to be run through a KDF in order to produce a usable key. Processing the secret through a KDF also allows for the introduction of both context material, how the key is going to be used, and one time material in the even to of a static-static key agreement.

- o Key Wrap algorithm: A key wrap algorithm of 'none' means that the result of the KDF is going to be used as the key directly. This option, along with static-static, should be used if knowledge about the sender is desired. If 'none' is used, then the content layer encryption algorithm size is value fed to the context structure. Support is also provided for any of the key wrap algorithms defined in Section 12.2.1. If one of these options is used, the input key size to the key wrap algorithm is the value fed into the context structure as the key size.

The set of direct ECDH algorithms defined in this document are found in Table 17.

name	value	KDF	Ephemeral-Static	Key Wrap	description
ECDH-ES + HKDF-256	50	HKDF - SHA-256	yes	none	ECDH ES w/ HKDF - generate key directly
ECDH-ES + HKDF-512	51	HKDF - SHA-256	yes	none	ECDH ES w/ HKDF - generate

					key directly
ECDH-SS + HKDF-256	52	HKDF - SHA-256	no	none	ECDH ES w/ HKDF - generate key directly
ECDH-SS + HKDF-512	53	HKDF - SHA-256	no	none	ECDH ES w/ HKDF - generate key directly
ECDH-ES + A128KW	54	HKDF - SHA-256	yes	A128KW	ECDH ES w/ Concat KDF and AES Key wrap w/ 128 bit key
ECDH-ES+A192KW	55	HKDF - SHA-256	yes	A192KW	ECDH ES w/ Concat KDF and AES Key wrap w/ 192 bit key
ECDH-ES + A256KW	56	HKDF - SHA-256	yes	A256KW	ECDH ES w/ Concat KDF and AES Key wrap w/ 256 bit key
ECDH-SS + A128KW	57	HKDF - SHA-256	no	A128KW	ECDH SS w/ Concat KDF and AES Key wrap w/ 128 bit key
ECDH-SS + A192KW	58	HKDF - SHA-256	no	A192KW	ECDH SS w/ Concat KDF and AES Key wrap w/ 192 bit key
ECDH-SS + A256KW	59	HKDF - SHA-256	no	A256KW	ECDH SS w/ Concat KDF and AES Key

						wrap w/ 256 bit key

Table 17: ECDH Algorithm Values

name	label	type	algorithm	description
ephemeral key	-1	COSE_Key	ECDH-ES	Ephemeral Public key for the sender
static key	-2	COSE_Key	ECDH-ES	Static Public key for the sender
static key id	-3	bstr	ECDH-SS	Static Public key identifier for the sender

Table 18: ECDH Algorithm Parameters

This document defines these algorithms to be used with the curves P-256, P-384, P-521. Implementations MUST verify that the key type and curve are correct. Different curves are restricted to different key types. Implementations MUST verify that the curve and algorithm are appropriate for the entities involved.

When using a COSE key for this algorithm, the following checks are made:

- The 'kty' field MUST be present and it MUST be 'EC2'.

- If the 'alg' field present, it MUST match the Key Agreement algorithm being used.

- If the 'key_ops' field is present, it MUST include 'derive key' or 'derive bits' for the private key.

- If the 'key_ops' field is present, it MUST be empty for the public key.

12.5. Key Agreement with KDF

Key Agreement with Key Wrapping uses a randomly generated CEK. The CEK is then encrypted using a Key Wrapping algorithm and a key derived from the shared secret computed by the key agreement algorithm.

The COSE_Enveloped structure for the recipient is organized as follows:

- o The 'protected' field is fed into the KDF context structure.
- o The plain text to be encrypted is the key from next layer down (usually the content layer).
- o The 'alg' parameter MUST be present in the layer.
- o A parameter identifying the recipient's key SHOULD be present. A parameter identifying the sender's key SHOULD be present.

12.5.1. ECDH

These algorithms are defined in Table 17.

When using a COSE key for this algorithm, the following checks are made:

- o The 'kty' field MUST be present and it MUST be 'EC2'.
- o If the 'alg' field present, it MUST match the Key Agreement algorithm being used.
- o If the 'key_ops' field is present, it MUST include 'derive key' or 'derive bits' for the private key.
- o If the 'key_ops' field is present, it MUST be empty for the public key.

13. Keys

The COSE_Key object defines a way to hold a single key object. It is still required that the members of individual key types be defined. This section of the document is where we define an initial set of members for specific key types.

For each of the key types, we define both public and private members. The public members are what is transmitted to others for their usage. We define private members mainly for the purpose of archival of keys by individuals. However, there are some circumstances in which private keys may be distributed by various entities in a protocol. Examples include: entities that have poor random number generation, centralized key creation for multi-cast type operations, and protocols in which a shared secret is used as a bearer token for authorization purposes.

Key types are identified by the 'kty' member of the COSE_Key object. In this document, we define four values for the member:

name	value	description
EC2	2	Elliptic Curve Keys w/ X,Y Coordinate pair
Symmetric	4	Symmetric Keys
Reserved	0	This value is reserved

Table 19: Key Type Values

13.1. Elliptic Curve Keys

Two different key structures could be defined for Elliptic Curve keys. One version uses both an x and a y coordinate, potentially with point compression. This is the traditional EC point representation that is used in [RFC5480]. The other version uses only the x coordinate as the y coordinate is either to be recomputed or not needed for the key agreement operation. Currently no algorithms are defined using this key structure.

name	key type	value	description
P-256	EC2	1	NIST P-256 also known as secp256r1
P-384	EC2	2	NIST P-384 also known as secp384r1
P-521	EC2	3	NIST P-521 also known as secp521r1

Table 20: EC Curves

13.1.1. Double Coordinate Curves

The traditional way of sending EC curves has been to send either both the x and y coordinates, or the x coordinate and a sign bit for the y coordinate. The latter encoding has not been recommended in the IETF due to potential IPR issues. However, for operations in constrained environments, the ability to shrink a message by not sending the y coordinate is potentially useful.

For EC keys with both coordinates, the 'kty' member is set to 2 (EC2). The key parameters defined in this section are summarized in Table 21. The members that are defined for this key type are:

crv contains an identifier of the curve to be used with the key. The curves defined in this document for this key type can be found in Table 20. Other curves may be registered in the future and private curves can be used as well.

x contains the x coordinate for the EC point. The integer is converted to an octet string as defined in [SEC1]. Leading zero octets MUST be preserved.

y contains either the sign bit or the value of y coordinate for the EC point. When encoding the value y, the integer is converted to an octet string (as defined in [SEC1]) and encoded as a CBOR bstr. Leading zero octets MUST be preserved. The compressed point encoding is also supported. Compute the sign bit as laid out in the Elliptic-Curve-Point-to-Octet-String Conversion function of [SEC1]. If the sign bit is zero, then encode y as a CBOR false value, otherwise encode y as a CBOR true value. The encoding of the infinity point is not supported.

d contains the private key.

For public keys, it is REQUIRED that 'crv', 'x' and 'y' be present in the structure. For private keys, it is REQUIRED that 'crv' and 'd' be present in the structure. For private keys, it is RECOMMENDED that 'x' and 'y' also be present, but they can be recomputed from the required elements and omitting them saves on space.

name	key type	value	type	description
crv	2	-1	int / tstr	EC Curve identifier - Taken from the COSE General Registry
x	2	-2	bstr	X Coordinate
y	2	-3	bstr / bool	Y Coordinate
d	2	-4	bstr	Private key

Table 21: EC Key Parameters

13.2. Symmetric Keys

Occasionally it is required that a symmetric key be transported between entities. This key structure allows for that to happen.

For symmetric keys, the 'kty' member is set to 3 (Symmetric). The member that is defined for this key type is:

k contains the value of the key.

This key structure contains only private key information, care must be taken that it is never transmitted accidentally. For public keys, there are no required fields. For private keys, it is REQUIRED that 'k' be present in the structure.

name	key type	value	type	description
k	4	-1	bstr	Key Value

Table 22: Symmetric Key Parameters

14. CBOR Encoder Restrictions

There has been an attempt to limit the number of places where the document needs to impose restrictions on how the CBOR Encoder needs to work. We have managed to narrow it down to the following restrictions:

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

- The rules for Canonical CBOR (Section 3.9 of RFC 7049) MUST be used in these locations. The main rule that needs to be enforced is that all lengths in these structures MUST be encoded such that they are encoded using definite lengths and the minimum length encoding is used.

- 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.

15. Application Profiling Considerations

This document is designed to provide a set of security services, but not to provide 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 is deferred to each application.

Applications are therefore intended to profile the usage of this document. This section provides a set of guidelines and topics that applications need to consider when using this document.

- Applications need to determine the set of messages defined in this document that it 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 parameter. If the partial IV parameter is specified, then the application would also need to define how the fixed portion of the IV would be determined.

- When applications use externally defined authenticated data, they need to define how that data is to be defined. This document assumes that the data will be provided as a byte stream. 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 preconfiguration of the set of algorithms to providing a discovery method built into the protocol. S/MIME provided a number of

different ways to approach the problem that applications could follow:

* Advertising in the message (S/MIME capabilities) [RFC5751].

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

* Minimum requirements for the S/MIME, which have been updated over time [RFC2633][RFC5751].

16. IANA Considerations

16.1. CBOR Tag assignment

It is requested that IANA assign the following tags from the "Concise Binary Object Representation (CBOR) Tags" registry. It is requested that the tags be assigned in the 24 to 255 value range.

The tags to be assigned are in table Table 1.

16.2. COSE Header Parameter Registry

It is requested that IANA create a new registry entitled "COSE Header Parameters". The registery is to be created as Expert Review Required. Expert review guidelines are provided in Section 16.10

The columns of the registry are:

name The name is present to make it easier to refer to and discuss the registration entry. The value is not used in the protocol. Names are to be unique in the table.

label This is the value used for the label. The label can be either an integer or a string. Registration in the table is based on the value of the label requested. Integer values between 1 and 255 and strings of length 1 are designated as Standards Track Document required. Integer values from 256 to 65535 and strings of length 2 are designated as Specification Required. Integer values of greater than 65535 and strings of length greater than 2 are designated as first come, first served. Integer values in the range -1 to -65536 are delegated to the "COSE Header Algorithm Label" registry. Integer values beyond -65536 are marked as private use.

value This contains the CBOR type for the value portion of the label.

value registry This contains a pointer to the registry used to contain values where the set is limited.

description This contains a brief description of the header field.

specification This contains a pointer to the specification defining the header field (where public).

The initial contents of the registry can be found in Table 2. The specification column for all rows in that table should be this document.

Additionally, the label of 0 is to be marked as 'Reserved'.

16.3. COSE Header Algorithm Label Table

It is requested that IANA create a new registry entitled "COSE Header Algorithm Labels". The registery is to be created as Expert Review Required. Expert review guidelines are provided in Section 16.10

The columns of the registry are:

name The name is present to make it easier to refer to and discuss the registration entry. The value is not used in the protocol.

algorithm The algorithm(s) that this registry entry is used for. This value is taken from the "COSE Algorithm Value" registry. Multiple algorithms can be specified in this entry. For the table, the algorithm, label pair MUST be unique.

label This is the value used for the label. The label is an integer in the range of -1 to -65536.

value This contains the CBOR type for the value portion of the label.

value registry This contains a pointer to the registry used to contain values where the set is limited.

description This contains a brief description of the header field.

specification This contains a pointer to the specification defining the header field (where public).

The initial contents of the registry can be found in Table 12, Table 13, and Table 18. The specification column for all rows in that table should be this document.

16.4. COSE Algorithm Registry

It is requested that IANA create a new registry entitled "COSE Algorithm Registry". The registery is to be created as Expert Review Required. Expert review guidelines are provided in Section 16.10

The columns of the registry are:

value The value to be used to identify this algorithm. Algorithm values MUST be unique. The value can be a positive integer, a negative integer or a string. Integer values between -256 and 255 and strings of length 1 are designated as Standards Track Document required. Integer values from -65536 to 65535 and strings of length 2 are designated as Specification Required. Integer values of greater than 65535 and strings of length greater than 2 are designated as first come, first served. Integer values beyond -65536 are marked as private use.

description A short description of the algorithm.

specification A document where the algorithm is defined (if publicly available).

The initial contents of the registry can be found in Table 9, Table 8, Table 10, Table 5, Table 6, Table 7, Table 14, Table 15, Table 16, and Table 17. The specification column for all rows in that table should be this document.

16.5. COSE Key Common Parameter Registry

It is requested that IANA create a new registry entitled "COSE Key Common Parameter" Registry. The registery is to be created as Expert Review Required. Expert review guidelines are provided in Section 16.10

The columns of the registry are:

name This is a descriptive name that enables easier reference to the item. It is not used in the encoding.

label The value to be used to identify this algorithm. Key map labels MUST be unique. The label can be a positive integer, a negative integer or a string. Integer values between 0 and 255 and strings of length 1 are designated as Standards Track Document required. Integer values from 256 to 65535 and strings of length 2 are designated as Specification Required. Integer values of greater than 65535 and strings of length greater than 2 are designated as first come, first served. Integer values in the

range -1 to -65536 are used for key parameters specific to a single algorithm delegated to the "COSE Key Type Parameter Label" registry. Integer values beyond -65536 are marked as private use.

CBOR Type This field contains the CBOR type for the field

registry This field denotes the registry that values come from, if one exists.

description This field contains a brief description for the field

specification This contains a pointer to the public specification for the field if one exists

This registry will be initially populated by the values in Section 7.1. The specification column for all of these entries will be this document.

16.6. COSE Key Type Parameter Registry

It is requested that IANA create a new registry "COSE Key Type Parameters". The registery is to be created as Expert Review Required. Expert review guidelines are provided in Section 16.10

The columns of the table are:

key type This field contains a descriptive string of a key type. This should be a value that is in the COSE General Values table and is placed in the 'kty' field of a COSE Key structure.

name This is a descriptive name that enables easier reference to the item. It is not used in the encoding.

label The label is to be unique for every value of key type. The range of values is from -256 to -1. Labels are expected to be reused for different keys.

CBOR type This field contains the CBOR type for the field

description This field contains a brief description for the field

specification This contains a pointer to the public specification for the field if one exists

This registry will be initially populated by the values in Table 21 and Table 22. The specification column for all of these entries will be this document.

16.7. COSE Elliptic Curve Registry

It is requested that IANA create a new registry "COSE Elliptic Curve Parameters". The registery is to be created as Expert Review Required. Expert review guidelines are provided in Section 16.10

The columns of the table are:

name This is a descriptive name that enables easier reference to the item. It is not used in the encoding.

value This is the value used to identify the curve. These values MUST be unique. The integer values from -256 to 255 are designated as Standards Track Document Required. The integer values from 256 to 65535 and -65536 to -257 are designated as Specification Required. Integer values over 65535 are designated as first come, first served. Integer values less than -65536 are marked as private use.

key type This designates the key type(s) that can be used with this curve.

description This field contains a brief description of the curve.

specification This contains a pointer to the public specification for the curve if one exists.

This registry will be initially populated by the values in Table 19. The specification column for all of these entries will be this document.

16.8. Media Type Registrations

16.8.1. 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_MSG.

Type name: application

Subtype name: cose

Required parameters: N/A

Optional parameters: cose-type

Encoding considerations: binary

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

Interoperability considerations: N/A

Published specification: RFC TBD

Applications that use this media type: To be identified

Fragment identifier considerations: N/A

Additional information:

* 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

16.8.2. COSE Key media type

This section registers the "application/cose-key+cbor" and "application/cose-key-set+cbor" 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.

Type name: application

Subtype name: cose-key+cbor

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

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

Interoperability considerations: N/A

Published specification: RFC TBD

Applications that use this media type: To be identified

Fragment identifier considerations: N/A

Additional information:

* 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

Type name: application

Subtype name: cose-key-set+cbor

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

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

Interoperability considerations: N/A

Published specification: RFC TBD

Applications that use this media type: To be identified

Fragment identifier considerations: N/A

Additional information:

* 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

16.9. CoAP Content Format Registrations

This section registers a set of content formats for CoAP. ID assignment in the 24-255 range is requested.

Media Type	Encoding	ID	Reference
application/cose; cose-type ="cose-sign"		TBD10	[This Document]
application/cose; cose-type ="cose-sign1"		TBD11	[This Document]
application/cose; cose-type ="cose-enveloped"		TBD12	[This Document]
application/cose; cose-type ="cose-encrypted"		TBD13	[This Document]
application/cose; cose-type ="cose-mac"		TBD14	[This Document]
application/cose; cose-type ="cose-mac0"		TBD15	[This Document]
application/cose-key		TBD16	[This Document]
application/cose-key-set		TBD17	[This Document

16.10. Expert Review Instructions

All of the IANA registries established in this document are defined as expert review. This section gives some general guidelines for what the experts should be looking for, but they are being designated as experts for a reason so they should be given substantial latitude.

Expert reviewers should take into consideration the following points:

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

- o Specifications are required for the standards track range of point assignment. Specifications should exist for specification required ranges, but early assignment before a specification is

available is considered to be permissible. Specifications are needed for the first-come, first-serve range if they are expected to be used outside of closed environments in an inoperable way. When specifications are not provided, the description provided needs to have sufficient information to identify what point is being used for.

- Experts should take into account the expected usage of fields when approving point assignment. The fact that there is a range for standards track documents does not mean that a standards track document cannot have points assigned outside of that range. Some of the ranges are restricted in range, items which are not expected to be common or are not expected to be used in restricted environments should be assigned to values which will encode to longer byte strings.

- When algorithms are registered, vanity registrations should be discouraged. One way to do this is to require applications to provide additional documentation on security analysis of algorithms. Another thing that should be considered is to request for an opinion on the algorithm from the Cryptographic Forum Research Group. Algorithms which do not meet the security requirements of the community and the messages structures should not be registered.

17. Security Considerations

There are security considerations:

1. Protect private keys.

2. MAC messages with more than one recipient means one cannot figure out which party sent the message.

3. Use of a direct key with other recipient structures hands the key to the other recipients.

4. Use of direct ECDH direct encryption is easy for people to leak information on if there are other recipients in the message.

5. Considerations about protected vs unprotected header fields. WHy the algorithm parameter needs to be protected.

6. Need to verify that: 1) the kty field of the key matches the key and algorithm being used, 2) the kty field needs to be included in the trust decision as well as the other key fields, and 3) the algorithm is included in the trust decision.

18. References

18.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://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, <http://www.rfc-editor.org/info/rfc7049>.

18.2. Informative References

[AES-GCM] Dworkin, M., "NIST Special Publication 800-38D: Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC.", Nov 2007.

[DSS] U.S. National Institute of Standards and Technology, "Digital Signature Standard (DSS)", July 2013.

[I-D.greevenbosch-appsawg-cbor-cddl] Vigano, C. and H. Birkholz, "CBOR data definition language (CDDL): a notational convention to express CBOR data structures", draft-greevenbosch-appsawg-cbor-cddl-07 (work in progress), October 2015.

[MAC] NiST, N., "FIPS PUB 113: Computer Data Authentication", May 1985.

[PVSig] Brown, D. and D. Johnson, "Formal Security Proofs for a Signature Scheme with Partial Message Recover", February 2000.

[RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <http://www.rfc-editor.org/info/rfc2104>.

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

[RFC2898] Kaliski, B., "PKCS #5: Password-Based Cryptography Specification Version 2.0", RFC 2898, DOI 10.17487/RFC2898, September 2000, <http://www.rfc-editor.org/info/rfc2898>.

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

[RFC3447] Jonsson, J. and B. Kaliski, "Public-Key Cryptography Standards (PKCS) #1: RSA Cryptography Specifications Version 2.1", RFC 3447, DOI 10.17487/RFC3447, February 2003, <http://www.rfc-editor.org/info/rfc3447>.

[RFC3610] Whiting, D., Housley, R., and N. Ferguson, "Counter with CBC-MAC (CCM)", RFC 3610, DOI 10.17487/RFC3610, September 2003, <http://www.rfc-editor.org/info/rfc3610>.

[RFC4231] Nystrom, M., "Identifiers and Test Vectors for HMAC-SHA-224, HMAC-SHA-256, HMAC-SHA-384, and HMAC-SHA-512", RFC 4231, DOI 10.17487/RFC4231, December 2005, <http://www.rfc-editor.org/info/rfc4231>.

[RFC4262] Santesson, S., "X.509 Certificate Extension for Secure/Multipurpose Internet Mail Extensions (S/MIME) Capabilities", RFC 4262, DOI 10.17487/RFC4262, December 2005, <http://www.rfc-editor.org/info/rfc4262>.

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

[RFC5480] Turner, S., Brown, D., Yiu, K., Housley, R., and T. Polk, "Elliptic Curve Cryptography Subject Public Key Information", RFC 5480, DOI 10.17487/RFC5480, March 2009, <http://www.rfc-editor.org/info/rfc5480>.

[RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <http://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, <http://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, <http://www.rfc-editor.org/info/rfc5752>.

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

[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, <http://www.rfc-editor.org/info/rfc5990>.

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

[RFC6151] Turner, S. and L. Chen, "Updated Security Considerations for the MD5 Message-Digest and the HMAC-MD5 Algorithms", RFC 6151, DOI 10.17487/RFC6151, March 2011, <http://www.rfc-editor.org/info/rfc6151>.

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

[RFC7159] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", RFC 7159, DOI 10.17487/RFC7159, March 2014, <http://www.rfc-editor.org/info/rfc7159>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <http://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, <http://www.rfc-editor.org/info/rfc7515>.

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

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

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

[RFC7539] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 7539, DOI 10.17487/RFC7539, May 2015, <http://www.rfc-editor.org/info/rfc7539>.

[SEC1] Standards for Efficient Cryptography Group, "SEC 1: Elliptic Curve Cryptography", May 2009.

[SP800-56A] Barker, E., Chen, L., Roginsky, A., and M. Smid, "NIST Special Publication 800-56A: Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", May 2013.

Appendix A. Three Levels of Recipient Information

All of the currently defined recipient algorithms classes only use two levels of the COSE_Encrypt structure. The first level is the message content and the second level is the content key encryption. However, if one uses a recipient algorithm such as RSA-KEM (see Appendix A of RSA-KEM [RFC5990], then it make sense to have three levels of the COSE_Encrypt structure.

These levels would be:

- Level 0: The content encryption level. This level contains the payload of the message.
- Level 1: The encryption of the CEK by a KEK.
- Level 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:

- Level 0: Has a content encrypted with AES-GCM using a 128-bit key.
- Level 1: Uses the AES Key wrap algorithm with a 128-bit key.
- Level 2: Uses ECDH Ephemeral-Static direct to generate the level 1 key.

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

Size of binary file is 216 bytes

```
992(
  [
    / protected / h'a10101' / {
        \ alg \ 1:1 \ AES-GCM 128 \
      } / ,
    / unprotected / {
      / iv / 5:h'02d1f7e6f26c43d4868d87ce'
    },
    / ciphertext / h'64f84d913ba60a76070a9a48f26e97e863e28529bf9be9d
e3bea1788f681200d875242f6',
    / recipients / [
      [
        / protected / h'',
        / unprotected / {
          / alg / 1:-3 / A128KW /
        },
        / ciphertext / h'5a15dbf5b282ecb31a6074ee3815c252405dd7583e0
78188',
        / recipients / [
          [
            / protected / h'',
            / unprotected / {
              / alg / 1:50 / ECDH-ES + HKDF-256 /,
              / kid / 4:'meriadoc.brandybuck@buckland.example',
              / ephemeral / -1:{
                / kty / 1:2,
                / crv / -1:1,
                / x / -2:h'b2add44368ea6d641f9ca9af308b4079aeb519f11
e9b8a55a600b21233e86e68',
                / y / -3:h'1a2cf118b9ee6895c8f415b686d4ca1cef362d4a7
630a31ef6019c0c56d33de0'
              }
            },
            / ciphertext / h''
          ]
        ]
      ]
    ]
  ]
)
```

Appendix B. Examples

The examples can be found at https://github.com/cose-wg/Examples. The file names in each section correspond the same file names in the repository. I am currently still in the process of getting the

examples up there along with some control information for people to be able to check and reproduce the examples.

Examples may have some features that are in question but not yet incorporated in the document.

To make it easier to read, the examples are presented using the CBOR's diagnostic notation rather than a binary dump. A ruby based tool exists to convert between a number of formats. 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 < inputfile > outputfile
```

The examples can be extracted from the XML version of this document via an XPath expression as all of the artwork is tagged with the attribute type='CBORdiag'.

B.1. Examples of MAC messages

B.1.1. Shared Secret Direct MAC

This example users the following:

- MAC: AES-CMAC, 256-bit key, truncated to 64 bits
- Recipient class: direct shared secret
- File name: Mac-04

Size of binary file is 73 bytes

```
994(
  [
    / protected / h'a1016f4145532d434d41432d3235362f3634' / {
        \ alg \ 1:"AES-CMAC-256//64"
      } / ,
    / unprotected / {},
    / payload / 'This is the content.',
    / tag / h'5924501e17f6e852',
    / recipients / [
      [
        / protected / h'',
        / unprotected / {
          / alg / 1:-6 / direct /,
          / kid / 4:'our-secret'
        },
        / ciphertext / h''
      ]
    ]
  ]
)
```

B.1.2. 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 217 bytes

```
994(
  [
    / protected / h'a10104' / {
        \ alg \ 1:4 \ HMAC 256//256 \
      } / ,
    / unprotected / {},
    / payload / 'This is the content.',
    / tag / h'fc672c2bc7e9e811a0ec6173bdadfe3f11d71a1fc04164feea711b
330c2b2478',
    / recipients / [
      [
        / protected / h'',
        / unprotected / {
          / alg / 1:52 / ECDH-SS + HKDF-256 /,
          / kid / 4:'meriadoc.brandybuck@buckland.example',
          / static kid / -3:'peregrin.took@tuckborough.example',
          "apu":h'4d8553e7e74f3c6a3a9dd3ef286a8195cbf8a23d19558ccfec
7d34b824f42d92bd06bd2c7f0271f0214e141fb779ae2856abf585a58368b017e7f2
a9e5ce4db5'
        },
        / ciphertext / h''
      ]
    ]
  ]
)
```

B.1.3. Wrapped MAC

This example uses the following:

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

o Recipient class: AES keywrap w/ a pre-shared 256-bit key

Size of binary file is 124 bytes

```
994(
  [
    / protected / h'a1016e4145532d3132382d4d41432d3634' / {
        \ alg \ 1:"AES-128-MAC-64"
      } / ,
    / unprotected / {},
    / payload / 'This is the content.',
    / tag / h'f65bc4e5ed133779',
    / recipients / [
      [
        / protected / h'',
        / unprotected / {
          / alg / 1:-5 / A256KW /,
          / kid / 4:'018c0ae5-4d9b-471b-bfd6-eef314bc7037'
        },
        / ciphertext / h'711ab0dc2fc4585dce27effa6781c8093eba906f227
b6eb0'
      ]
    ]
  ]
)
```

B.1.4. Multi-recipient MAC 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 374 bytes

```
994(
  [
    / protected / h'a10104' / {
        \ alg \ 1:4 \ HMAC 256//256 \
      } / ,
    / unprotected / {},
    / payload / 'This is the content.',
    / tag / h'a25bff1b6251926c3b3314d9802831e9101fee82f11bec87ce622a
5c10292bce',
    / recipients / [
      [
        / protected / h'',
        / unprotected / {
          / alg / 1:54 / ECHD-ES+A128KW /,
          / kid / 4:'bilbo.baggins@hobbiton.example',
          -1:{
            1:2,
            -1:3,
            -2:h'43b12669acac3fd27898ffba0bcd2e6c366d53bc4db71f909a7
59304acfb5e18cdc7ba0b13ff8c7636271a6924b1ac63c02688075b55ef2d613574e
7dc242f79c3',
            -3:h'812dd694f4ef32b11014d74010a954689c6b6e8785b333d1ab4
4f22b9d1091ae8fc8ae40b687e5cfbe7ee6f8b47918a07bb04e9f5b1a51a334a16bc
09777434113'
          }
        },
        / ciphertext / h'70306cbce4b28f40cb7574b6928b5318c965b28a4e8
a892d71ddbab944fe68799baec290899623b1'
      ],
      [
        / protected / h'',
        / unprotected / {
          / alg / 1:-5 / A256KW /,
          / kid / 4:'018c0ae5-4d9b-471b-bfd6-eef314bc7037'
        },
        / ciphertext / h'0b2c7cfce04e98276342d6476a7723c090dfdd15f9a
518e7736549e998370695e6d6a83b4ae507bb'
      ]
    ]
  ]
)
```

B.2. Examples of Encrypted Messages

B.2.1. Direct ECDH

This example uses the following:

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

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

Size of binary file is 184 bytes

```
992(
  [
    / protected / h'a10101' / {
        \ alg \ 1:1 \ AES-GCM 128 \
      } / ,
    / unprotected / {
      / iv / 5:h'c9cf4df2fe6c632bf7886413'
    },
    / ciphertext / h'45fce2814311024d3a479e7d3eed063850f3f0b9ce550fb
62f23a0d5151c8049bed5802a',
    / recipients / [
      [
        / protected / h'',
        / unprotected / {
          / alg / 1:50 / ECDH-ES + HKDF-256 /,
          / kid / 4:'meriadoc.brandybuck@buckland.example',
          / ephemeral / -1:{
            / kty / 1:2,
            / crv / -1:1,
            / x / -2:h'98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbf
bf054e1c7b4d91d6280',
            / y / -3:h'f01400b089867804b8e9fc96c3932161f1934f4223069
170d924b7e03bf822bb'
          }
        },
        / ciphertext / h''
      ]
    ]
  ]
)
```

B.2.2. Direct plus Key Derivation

This example uses the following:

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

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

   * APU identity: "lighting-client"

   * APV identity: "lighting-server"

   * Supplementary Public Other: "Encryption Example 02"

Size of binary file is 97 bytes

```
992(
  [
    / protected / h'a1010a' / {
        \ alg \ 1:10 \ AES-CCM-16-64-128 \
      } / ,
    / unprotected / {
      / iv / 5:h'89f52f65a1c580933b5261a7'
    },
    / ciphertext / h'5e70f2058526d70d29c155015c5723a3f9c15a13a6a9f4c
ece341510',
    / recipients / [
      [
        / protected / h'',
        / unprotected / {
          / alg / 1:"dir+kdf",
          / kid / 4:'our-secret',
          -20:'aabbccddeeffgghh'
        },
        / ciphertext / h''
      ]
    ]
  ]
)
```

B.2.3. Counter Signature on Encrypted Content

This example uses the following:

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

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

Size of binary file is 357 bytes

```
992(
  [
    / protected / h'a10101' / {
        \ alg \ 1:1 \ AES-GCM 128 \
      } / ,
    / unprotected / {
      / iv / 5:h'c9cf4df2fe6c632bf7886413',
      / countersign / 7:[
        / protected / h'',
        / unprotected / {
          / alg / 1:-9 / ES512 /,
          / kid / 4:'bilbo.baggins@hobbiton.example'
        },
        / signature / h'00aa98cbfd382610a375d046a275f30266e8d0faacb9
069fde06e37825ae7825419c474f416ded0c8e3e7b55bff68f2a704135bdf99186f6
6659461c8cf929cc7fb300e4ec6cac6be6f18d92cd319dccfc354d78cbdf2b1cf293
c9d8f82449feeb4f25a24b80a08c2ddbae8507b3da7c4c869ef7c20a82e3d7b9b54f
031a76fcebca1fcb'
      ]
    },
    / ciphertext / h'45fce2814311024d3a479e7d3eed063850f3f0b9ce550fb
62f23a0d5151c8049bed5802a',
    / recipients / [
      [
        / protected / h'',
        / unprotected / {
          / alg / 1:50 / ECDH-ES + HKDF-256 /,
          / kid / 4:'meriadoc.brandybuck@buckland.example',
          / ephemeral / -1:{
            / kty / 1:2,
            / crv / -1:1,
            / x / -2:h'98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbf
bf054e1c7b4d91d6280',
            / y / -3:h'f01400b089867804b8e9fc96c3932161f1934f4223069
170d924b7e03bf822bb'
          }
        },
        / ciphertext / h''
      ]
    ]
  ]
)
```

B.2.4. Encrypted Content w/ Implicit Recipient

This example uses the following:

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

Size of binary file is 53 bytes

```
993(
  [
    / protected / h'a1010a' / {
        \ alg \ 1:10 \ AES-CCM-16-64-128 \
      } / ,
    / unprotected / {
      / iv / 5:h'89f52f65a1c580933b5261a7'
    },
    / ciphertext / h'16c4b98fe85c8e2eed1f990ef40ce02cd54aa3195d43ed6
18a9df43e'
  ]
)
```

B.2.5. Encrypted Content w/ Implicit Recipient and Partial IV

This example uses the following:

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

- o Prefix for IV is 89F52F65A1C580933B52

Size of binary file is 43 bytes

```
993(
  [
    / protected / h'a1010a' / {
        \ alg \ 1:10 \ AES-CCM-16-64-128 \
      } / ,
    / unprotected / {
      / partial iv / 6:h'61a7'
    },
    / ciphertext / h'2b2dd3406aa1e83b488d32d6852bfad387a5199c6fcc3d6
c6bbff5e2'
  ]
)
```

B.3. Examples of Signed Message

B.3.1. Single Signature

This example uses the following:

- o Signature Algorithm: ECDSA w/ SHA-256, Curve P-256-1

Size of binary file is 106 bytes

```
991(
  [
    / protected / h'',
    / unprotected / {},
    / payload / 'This is the content.',
    / signatures / [
      [
        / protected / h'a10126' / {
            \ alg \ 1:-7 \ ES256 \
          } / ,
        / unprotected / {
          / kid / 4:'11'
        },
        / signature / h'00eae868ecc176883766c5dc5ba5b8dca25dab3c2e56
a551ce5705b793914348e100d702a242d4f6428a2b6ce0bae311a1be41f3c0333ed3
d892e4d07af86f338a89'
      ]
    ]
  ]
)
```

B.3.2. Multiple Signers

This example uses the following:

- o Signature Algorithm: ECDSA w/ SHA-256, Curve P-256-1
- o Signature Algorithm: ECDSA w/ SHA-512, Curve P-521

Size of binary file is 276 bytes

```
991(
  [
    / protected / h'',
    / unprotected / {},
    / payload / 'This is the content.',
    / signatures / [
      [
        / protected / h'a10126' / {
            \ alg \ 1:-7 \ ES256 \
          } / ,
        / unprotected / {
          / kid / 4:'11'
        },
        / signature / h'0dc1c5e62719d8f3cce1468b7c881eee6a8088b46bf8
36ae956dd38fe93199193571f290e9a471cbcb3bbfd6f35ce9b22bd100621bcdbf2f
8ba16c19d86e9306'
      ],
      [
        / protected / h'',
        / unprotected / {
          / alg / 1:-9 / ES512 /,
          / kid / 4:'bilbo.baggins@hobbiton.example'
        },
        / signature / h'012ce5b1dfe8b5aa6eaa09a54c58a84ad0900e4fdf27
59ec22d1c861cccd75c7e1c4025a2da35e512fc2874d6ac8fd862d09ad07ed2deac2
97b897561e04a8d4247601bb1af26e0fce66df949d0de84627280129c9110f2ab241
217cf151b3a147215cfddc31ea02569ac927b43b6f67418e694b92a69a3363a3c1c0
0149c41c4722471c'
      ]
    ]
  ]
)
```

B.3.3. Counter Signature

This example uses the following:

- o Signature Algorithm: ECDSA w/ SHA-256, Curve P-256-1

B.4. COSE Keys

B.4.1. 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

```
[
  {
    / crv / -1:1,
    / x / -2:h'65eda5a12577c2bae829437fe338701a10aaa375e1bb5b5de108d
e439c08551d',
    / y / -3:h'1e52ed75701163f7f9e40ddf9f341b3dc9ba860af7e0ca7ca7e9e
ecd0084d19c',
    / kty / 1:2,
    / kid / 2:'meriadoc.brandybuck@buckland.example'
  },
  {
    / crv / -1:3,
    / x / -2:h'0072992cb3ac08ecf3e5c63dedec0d51a8c1f79ef2f82f94f3c73
7bf5de7986671eac625fe8257bbd0394644caaa3aaf8f27a4585fbbcad0f24576200
85e5c8f42ad',
    / y / -3:h'01dca6947bce88bc5790485ac97427342bc35f887d86d65a08937
7e247e60baa55e4e8501e2ada5724ac51d6909008033ebc10ac999b9d7f5cc2519f3
fe1ea1d9475',
    / kty / 1:2,
    / kid / 2:'bilbo.baggins@hobbiton.example'
  },
  {
    / crv / -1:1,
    / x / -2:h'98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbfbf054e1c
7b4d91d6280',
    / y / -3:h'f01400b089867804b8e9fc96c3932161f1934f4223069170d924b
7e03bf822bb',
    / kty / 1:2,
    / kid / 2:'peregrin.took@tuckborough.example'
  },
  {
    / crv / -1:1,
    / x / -2:h'bac5b11cad8f99f9c72b05cf4b9e26d244dc189f745228255a219
a86d6a09eff',
    / y / -3:h'20138bf82dc1b6d562be0fa54ab7804a3a64b6d72ccfed6b6fb6e
d28bbfc117e',
    / kty / 1:2,
    / kid / 2:'11'
  }
]
```

B.4.2. 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

```
[
  {
    / kty / 1:2,
    / kid / 2:'meriadoc.brandybuck@buckland.example',
    / crv / -1:1,
    / x / -2:h'65eda5a12577c2bae829437fe338701a10aaa375e1bb5b5de108d
e439c08551d',
    / y / -3:h'1e52ed75701163f7f9e40ddf9f341b3dc9ba860af7e0ca7ca7e9e
ecd0084d19c',
    / d / -4:h'aff907c99f9ad3aae6c4cdf21122bce2bd68b5283e6907154ad91
1840fa208cf'
  },
  {
    / kty / 1:4,
    / kid / 2:'our-secret',
    / k / -1:h'849b57219dae48de646d07dbb533566e976686457c1491be3a76d
cea6c427188'
  },
  {
    / kty / 1:2,
    / kid / 2:'bilbo.baggins@hobbiton.example',
    / crv / -1:3,
    / x / -2:h'0072992cb3ac08ecf3e5c63dedec0d51a8c1f79ef2f82f94f3c73
7bf5de7986671eac625fe8257bbd0394644caaa3aaf8f27a4585fbbcad0f24576200
85e5c8f42ad',
    / y / -3:h'01dca6947bce88bc5790485ac97427342bc35f887d86d65a08937
7e247e60baa55e4e8501e2ada5724ac51d6909008033ebc10ac999b9d7f5cc2519f3
fe1ea1d9475',
    / d / -4:h'00085138ddabf5ca975f5860f91a08e91d6d5f9a76ad4018766a4
76680b55cd339e8ab6c72b5facdb2a2a50ac25bd086647dd3e2e6e99e84ca2c3609f
df177feb26d'
  },
  {
```

```
        / kty / 1:4,
        / kid / 2:'our-secret2',
        / k / -1:h'849b5786457c1491be3a76dcea6c4271'
      },
      {
        / kty / 1:2,
        / crv / -1:1,
        / kid / 2:'peregrin.took@tuckborough.example',
        / x / -2:h'98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbfbf054e1c
   7b4d91d6280',
        / y / -3:h'f01400b089867804b8e9fc96c3932161f1934f4223069170d924b
   7e03bf822bb',
        / d / -4:h'02d1f7e6f26c43d4868d87ceb2353161740aacf1f7163647984b5
   22a848df1c3'
      },
      {
        / kty / 1:4,
        / kid / 2:'018c0ae5-4d9b-471b-bfd6-eef314bc7037',
        / k / -1:h'849b57219dae48de646d07dbb533566e976686457c1491be3a76d
   cea6c427188'
      },
      {
        / kty / 1:2,
        / kid / 2:'11',
        / crv / -1:1,
        / x / -2:h'bac5b11cad8f99f9c72b05cf4b9e26d244dc189f745228255a219
   a86d6a09eff',
        / y / -3:h'20138bf82dc1b6d562be0fa54ab7804a3a64b6d72ccfed6b6fb6e
   d28bbfc117e',
        / d / -4:h'57c92077664146e876760c9520d054aa93c3afb04e306705db609
   0308507b4d3'
      }
    ]
```

Appendix C. Document Updates

C.1. Version -08 to -09

- Integrate CDDL syntax into the text
- Define Expert review guidelines
- Expand application profiling guidelines
- Expand text around Partial IV
- Creation time becomes Operation time

- Add tagging for all structures so that they cannot be moved
- Add optional parameter to cose media type
- Add single signature and mac structures

C.2. Version -07 to -08

- Redefine sequence number into a the Partial IV.

C.3. Version -06 to -07

- Editorial Changes
- Make new IANA registries be Expert Review

C.4. Version -05 to -06

- Remove new CFRG Elliptical Curve key agreement algorithms.
- Remove RSA algorithms
- Define a creation time and sequence number for discussions.
- Remove message type field from all structures.
- Define CBOR tagging for all structures with IANA registrations.

C.5. Version -04 to -05

- Removed the jku, x5c, x5t, x5t#S256, x5u, and jwk headers.
- Add enveloped data vs encrypted data structures.
- Add counter signature parameter.

C.6. Version -03 to -04

- Change top level from map to array.
- Eliminate the term "key management" from the document.
- Point to content registries for the 'content type' attribute
- Push protected field into the KDF functions for recipients.
- Remove password based recipient information.

- o Create EC Curve Registry.

C.7. Version -02 to -03

- o Make a pass over all of the algorithm text.

- o Alter the CDDL so that Keys and KeySets are top level items and the key examples validate.

- o Add sample key structures.

- o Expand text on dealing with Externally Supplied Data.

- o Update the examples to match some of the renumbering of fields.

C.8. Version -02 to -03

- o Add a set of straw man proposals for algorithms. It is possible/ expected that this text will be moved to a new document.

- o Add a set of straw man proposals for key structures. It is possible/expected that this text will be moved to a new document.

- o Provide guidance on use of externally supplied authenticated data.

- o Add external authenticated data to signing structure.

C.9. Version -01 to -2

- o Add first pass of algorithm information

- o Add direct key derivation example.

C.10. Version -00 to -01

- o Add note on where the document is being maintained and contributing notes.

- o Put in proposal on MTI algorithms.

- o Changed to use labels rather than keys when talking about what indexes a map.

- o Moved nonce/IV to be a common header item.

- o Expand section to discuss the common set of labels used in COSE_Key maps.

- Start marking element 0 in registries as reserved.

- Update examples.

Editorial Comments

[CREF1] JLS: Need to check this list for correctness before publishing.

[CREF2] JLS: I have not gone through the document to determine what needs to be here yet. We mostly want to grab terms that are used in unusual ways or are not generally understood.

[CREF3] Hannes: Ensure that the list of examples only includes items that are implemented in this specification. Check the other places where such lists occur and ensure that they also follow this rule.

[CREF4] JLS: Restrict to the set of supported parameters.

[CREF5] Ilari: Look to see if we need to be clearer about how the fields defined in the table are transported and thus why they have labels.

Author's Address

Jim Schaad
August Cellars

Email: ietf@augustcellars.com