

COSE Working Group
Internet-Draft
Updates: 8152 (if approved)
Intended status: Standards Track
Expires: 25 December 2021

J. Schaad
August Cellars
R. Housley, Ed.
Vigil Security
23 June 2021

CBOR Object Signing and Encryption (COSE): Countersignatures
draft-ietf-cose-countersign-05

Abstract

Concise Binary Object Representation (CBOR) is a data format designed for small code size and small message size. CBOR Object Signing and Encryption (COSE) defines a set of security services for CBOR. This document defines a countersignature algorithm along with the needed header parameters and CBOR tags for COSE.

Contributing to this document

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

The source for this draft is being maintained in GitHub. Suggested changes should be submitted as pull requests at <https://github.com/cose-wg/countersign>. 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

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

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

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

This Internet-Draft will expire on 25 December 2021.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Requirements Terminology	4
1.2. CBOR Grammar	4
1.3. Document Terminology	4
2. Countersignature Header Parameters	5
3. Version 2 Countersignatures	6
3.1. Full Countersignatures	7
3.2. Abbreviated Countersignatures	8
3.3. Signing and Verification Process	8
4. CBOR Encoding Restrictions	10
5. IANA Considerations	10
5.1. CBOR Tag Assignment	10
5.2. COSE Header Parameters Registry	11
6. Security Considerations	11
7. Implementation Status	13
7.1. Author's Versions	14
7.2. COSE Testing Library	14
8. References	15
8.1. Normative References	15
8.2. Informative References	15
Appendix A. Examples	16
A.1. Use of Early Code Points	17
A.2. Examples of Signed Messages	17
A.2.1. Countersignature	17
A.3. Examples of Signed1 Messages	18
A.3.1. Countersignature	18
A.4. Examples of Enveloped Messages	19
A.4.1. Countersignature on Encrypted Content	19
A.5. Examples of Encrypted Messages	20
A.5.1. Countersignature on Encrypted Content	21
A.6. Examples of MACed Messages	21
A.6.1. Countersignature on MAC Content	21

A.7. Examples of MAC0 Messages	22
A.7.1. Countersignature on MAC0 Content	22
Acknowledgments	23
Authors' Addresses	23

1. Introduction

There has been an increased focus on small, constrained devices that make up the Internet of Things (IoT). One of the standards that has come out of this process is "Concise Binary Object Representation (CBOR)" [RFC8949]. CBOR extended the data model of the JavaScript Object Notation (JSON) [STD90] by allowing for binary data, among other changes. CBOR has been adopted by several of the IETF working groups dealing with the IoT world as their encoding of data structures. CBOR was designed specifically both to be small in terms of messages transported and implementation size and to be a schema-free decoder. A need exists to provide message security services for IoT, and using CBOR as the message-encoding format makes sense.

During the process of advancing COSE to an Internet Standard, it was noticed the description of the security properties of countersignatures was incorrect for the COSE_Sign1 structure. Since the security properties that were described, those of a true countersignature, were those that the working group desired, the decision was made to remove all of the countersignature text from [I-D.ietf-cose-rfc8152bis-struct] and create a new document to both deprecate the old countersignature algorithm and to define a new one with the desired security properties.

The problem with the previous countersignature algorithm was that the cryptographically computed value was not always included. The initial assumption that the cryptographic value was in the third slot of the array was known not to be true at the time, but in the case of the MAC structures this was not deemed to be an issue. The new algorithm is more aggressive about the set of values included in the countersignature computation so that the cryptographic computed values is included. The exception to this is the COSE_Signature structure where there is no cryptographic computed value.

The new algorithm is designed to produce the same countersignature value in those cases where the cryptographic computed value was already included. This means that for those structures the only thing that would need to be done is to change the value of the header parameter.

1.1. Requirements Terminology

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

1.2. CBOR Grammar

CBOR grammar in the document is presented using CBOR Data Definition Language (CDDL) [RFC8610].

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

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

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

```
start = COSE_Countersignature_Tagged / Internal_Types
```

```
; This is defined to make the tool quieter:
```

```
Internal_Types = Countersign_structure / COSE_Countersignature0
```

The non-terminal Internal_Types is defined for dealing with the automated validation tools used during the writing of this document. It references those non-terminals that are used for security computations but are not emitted for transport.

1.3. Document Terminology

In this document, we use the following terminology:

Byte is a synonym for octet.

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

Context is used throughout the document to represent information that is not part of the COSE message. Information which is part of the context can come from several different sources including: Protocol interactions, associated key structures, and program configuration. The context to use can be implicit, identified using the 'kid

context' header parameter defined in [RFC8613], or identified by a protocol-specific identifier. Context should generally be included in the cryptographic construction; for more details see Section 4.3 of [I-D.ietf-cose-rfc8152bis-struct].

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

2. Countersignature Header Parameters

This section defines a set of common header parameters. A summary of these header parameters can be found in Table 1. This table should be consulted to determine the value of label and the type of the value.

The set of header parameters defined in this section are:

V2 countersignature: This header parameter holds one or more countersignature values. Countersignatures provide a method of having a second party sign some data. The countersignature header parameter can occur as an unprotected attribute in any of the following structures: COSE_Sign1, COSE_Signature, COSE_Encrypt, COSE_recipient, COSE_Encrypt0, COSE_Mac, and COSE_Mac0. Details on version 2 countersignatures are found in Section 3.

Name	Label	Value Type	Value Registry	Description
counter signature version 2	TBD10	COSE_Countersignature / [+ COSE_Countersignature]		V2 counter signature attribute
counter signature 0 version 2	TBD11	COSE_Countersignature0		Abbreviated Counter signature version 2

Table 1: Common Header Parameters

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

```
Generic_Headers /= (  
  ? TBD10 => COSE_Countersignature / [+COSE_Countersignature]  
  ; V2 Countersignature  
  ? TBD11 => COSE_Countersignature0 ; V2 Countersignature0  
)
```

3. Version 2 Countersignatures

A countersignature is normally defined as a second signature that confirms a primary signature. A normal example of a countersignature is the signature that a notary public places on a document as witnessing that you have signed the document. Thus applying a countersignature to either the COSE_Signature or COSE_Sign1 objects match this traditional definition. This document extends the context of a countersignature to allow it to be applied to all of the security structures defined. It needs to be noted that the countersignature needs to be treated as a separate operation from the initial operation even if it is applied by the same user as is done in [I-D.ietf-core-oscore-groupcomm].

COSE supports two different forms for countersignatures. Full countersignatures use the structure COSE_Countersignature. This is same structure as COSE_Signature and thus it can have protected and unprotected attributes, including chained countersignatures. Abbreviated countersignatures use the structure COSE_Countersignature0. This structure only contains the signature value and nothing else. The structures cannot be converted between each other; as the signature computation includes a parameter identifying which structure is being used, the converted structure will fail signature validation.

The version 2 countersignature changes the algorithm used for computing the signature from the original version Section 4.5 of [RFC8152]. The new version now includes the cryptographic material generated for all of the structures rather than just for a subset.

COSE was designed for uniformity in how the data structures are specified. One result of this is that for COSE one can expand the concept of countersignatures beyond just the idea of signing a signature to being able to sign most of the structures without having to create a new signing layer. When creating a countersignature, one needs to be clear about the security properties that result. When done on a COSE_Signature or COSE_Sign1, the normal countersignature semantics are preserved. That is the countersignature makes a statement about the existence of a signature and, when used as a timestamp, a time point at which the signature exists. When done on a COSE_Sign, this is the same as applying a second signature to the payload and adding a parallel signature as a new COSE_Signature is

the preferred method. When done on a COSE_Mac or COSE_Mac0, the payload is included as well as the MAC value. When done on a COSE_Encrypt or COSE_Encrypt0, the existence of the encrypted data is attested to. It should be noted that there is a big difference between attesting to the encrypted data as opposed to attesting to the plaintext data. Usually, the signer wishes to countersign the plaintext data, and then encrypt the data along with the countersignature. This approach prevents an attacker from stripping countersignatures. In addition, this approach prevents an observer from linking the public keys needed to verify the countersignatures across different payloads. It is always possible to construct cases where the use of two different keys will appear to result in a successful decryption (the tag check success), but which produce two completely different plaintexts. This situation is not detectable by a countersignature on the encrypted data.

3.1. Full Countersignatures

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

The full countersignature structure can be encoded as either tagged or untagged depending on the context it is used in. A tagged COSE_Countersignature structure is identified by the CBOR tag TBD0. The countersignature structure is the same as that used for a signer on a signed object. The CDDL fragment for full countersignatures is:

```
COSE_Countersignature_Tagged = #6.9999(COSE_Countersignature)
COSE_Countersignature = COSE_Signature
```

The details of the fields of a countersignature can be found in Section 4.1 of [I-D.ietf-cose-rfc8152bis-struct].

An example of a countersignature on a signature can be found in Appendix A.2.1. An example of a countersignature in an encryption object can be found in Appendix A.4.1.

It should be noted that only a signature algorithm with appendix (see Section 8 of [I-D.ietf-cose-rfc8152bis-struct]) can be used for countersignatures. 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.

3.2. Abbreviated Countersignatures

Abbreviated countersignatures were designed primarily to deal with the problem of encrypted group messaging, but where it is required to know who originated the message. The objective was to keep the countersignature as small as possible while still providing the needed security. For abbreviated countersignatures, there is no provision for any protected attributes related to the signing operation. Instead, the parameters for computing or verifying the abbreviated countersignature are provided by the same context used to describe the encryption, signature, or MAC processing.

The CDDL fragment for the abbreviated countersignatures is:

```
COSE_Countersignature0 = bstr
```

The byte string representing the signature value is placed in the Countersignature0 attribute. This attribute is then encoded as an unprotected header parameter. The attribute is defined below.

3.3. Signing and Verification Process

In order to create a signature, a well-defined byte string is needed. The Countersign_structure is used to create the canonical form. This signing and verification process takes in countersignature target structure, the signer information (COSE_Signature), and the application data (external source). A Countersign_structure is a CBOR array. The target structure of the countersignature needs to have all of its cryptographic functions finalized before the computing the signature. The fields of the Countersign_structure in order are:

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

"CounterSignature" for signatures using the
COSE_Countersignature structure when other_fields is absent.

"CounterSignature0" for signatures using the
COSE_Countersignature0 structure when other_fields is absent.

"CounterSignatureV2" for signatures using the COSE_Countersignature structure when other_fields is present.

"CounterSignature0V2" for signatures using the COSE_Countersignature0 structure when other_fields is present.

2. The protected attributes from the target structure encoded in a bstr type. If there are no protected attributes, a zero-length byte string is used.
3. The protected attributes from the signer structure encoded in a bstr type. If there are no protected attributes, a zero-length byte string is used. This field is omitted for the Countersignature0V2 attribute.
4. The externally supplied data from the application encoded in a bstr type. If this field is not supplied, it defaults to a zero-length byte string. (See Section 4.3 of [I-D.ietf-cose-rfc8152bis-struct] for application guidance on constructing this field.)
5. The payload to be signed encoded in a bstr type. The payload is placed here independent of how it is transported.
6. If there are only two bstr fields in the target structure, this field is omitted. The field is an array of all bstr fields after the second. As an example, this would be an array of one element for the COSE_Sign1 structure containing the signature value.

The CDDL fragment that describes the above text is:

```
Countersign_structure = [  
  context : "CounterSignature" / "CounterSignature0" /  
            "CounterSignatureV2" / "CounterSignature0V2" /,  
  body_protected : empty_or_serialized_map,  
  ? sign_protected : empty_or_serialized_map,  
  external_aad : bstr,  
  payload : bstr,  
  ? other_fields : [ + bstr ]  
]
```

How to compute a countersignature:

1. Create a Countersign_structure and populate it with the appropriate fields.
2. Create the value ToBeSigned by encoding the Countersign_structure to a byte string, using the encoding described in Section 4.

3. Call the signature creation algorithm passing in K (the key to sign with), alg (the algorithm to sign with), and ToBeSigned (the value to sign).
4. Place the resulting signature value in the correct location. This is the 'signature' field of the COSE_Countersignature structure. This is the value of the Countersignature0 attribute.

The steps for verifying a countersignature are:

1. Create a Countersign_structure and populate it with the appropriate fields.
2. Create the value ToBeSigned by encoding the Countersign_structure to a byte string, using the encoding described in Section 4.
3. Call the signature verification algorithm passing in K (the key to verify with), alg (the algorithm used sign with), ToBeSigned (the value to sign), and sig (the signature to be verified).

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

4. CBOR Encoding Restrictions

In order to always regenerate the same byte string for the "to be signed" value, the core deterministic encoding rules defined in Section 4.2.1 of [RFC8949]. These rules match the ones laid out in Section 9 of [I-D.ietf-cose-rfc8152bis-struct].

5. IANA Considerations

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

5.1. CBOR Tag Assignment

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

- * Tag: TBD0
- * Data Item: COSE_Countersignature
- * Semantics: COSE standalone V2 countersignature

- * Reference: [[this document]]

5.2. COSE Header Parameters Registry

IANA created a registry titled "COSE Header Parameters" as part of processing [RFC8152].

IANA is requested to register the following new items in the registry. For these entries, the Value Registry column will be blank and the Reference column will be [[This Document]].

Name	Label	Value Type	Description
counter signature version 2	TBD10	COSE_Countersignature / [+ COSE_Countersignature]	V2 countersignature attribute
Countersignature0 version 2	TBD11	COSE_Countersignature0	Abbreviated Counter signature vesion 2

Table 2: New Common Header Parameters

IANA is requested to modify the Description field for "counter signature" and "CounterSignature0" to include the words "(Deprecated by [[This Document]])".

6. Security Considerations

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

Implementations need to protect the private key material for any individuals. There are some cases that need to be highlighted on this issue.

- * Using the same key for two different algorithms can leak information about the key. It is therefore recommended that keys be restricted to a single algorithm.
- * Use of 'direct' as a recipient algorithm combined with a second recipient algorithm exposes the direct key to the second recipient.

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

The use of ECDH and direct plus KDF (with no key wrap) will not directly lead to the private key being leaked; the one way function of the KDF will prevent that. There is, however, a different issue that needs to be addressed. Having two recipients requires that the CEK be shared between two recipients. The second recipient therefore has a CEK that was derived from material that can be used for the weak proof of origin. The second recipient could create a message using the same CEK and send it to the first recipient; the first recipient would, for either static-static ECDH or direct plus KDF, make an assumption that the CEK could be used for proof of origin even though it is from the wrong entity. If the key wrap step is added, then no proof of origin is implied and this is not an issue.

Although it has been mentioned before, the use of a single key for multiple algorithms has been demonstrated in some cases to leak information about that key, provide the opportunity for attackers to forge integrity tags, or gain information about encrypted content. Binding a key to a single algorithm prevents these problems.

Key creators and key consumers are strongly encouraged not only to create new keys for each different algorithm, but to include that selection of algorithm in any distribution of key material and strictly enforce the matching of algorithms in the key structure to algorithms in the message structure. In addition to checking that algorithms are correct, the key form needs to be checked as well. Do not use an 'EC2' key where an 'OKP' key is expected.

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

- * What are the permissions associated with the key owner?
- * Is the cryptographic algorithm acceptable in the current context?
- * Have the restrictions associated with the key, such as algorithm or freshness, been checked and are they correct?
- * Is the request something that is reasonable, given the current state of the application?

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

Analysis of the size of encrypted messages can provide information about the plaintext messages. This specification does not provide a uniform method for padding messages prior to encryption. An observer can distinguish between two different messages (for example, 'YES' and 'NO') based on the length for all of the content encryption algorithms that are defined in [I-D.ietf-cose-rfc8152bis-algs]. This means that it is up to the applications to specify how content padding is to be done to prevent or discourage such analysis. (For example, the text strings could be defined as 'YES' and 'NO'.)

When either COSE_Encrypt and COSE_Mac is used and more than two parties share the key, data origin authentication is not provided. Any party that knows the message-authentication key can compute a valid authentication tag; therefore, the contents could originate from any one of the parties that share the key.

Countersignatures of COSE_Encrypt and COSE_Mac with short authentication tags do not provide the security properties associated with the same algorithm used in COSE_Sign. To provide 128-bit security against collision attacks, the tag length MUST be at least 256-bits. A countersignature of a COSE_Mac with AES-MAC 256/128 provides at most 64 bits of integrity protection. Similarly, a countersignature of a COSE_Encrypt with AES-CCM-16-64-128 provides at most 32 bits of integrity protection.

7. Implementation Status

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

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations may exist.

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

7.1. Author's Versions

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

- * Implementation Location: <https://github.com/cose-wg>
- * Primary Maintainer: Jim Schaad
- * Languages: There are three different languages that are currently supported: Java and C#.
- * Cryptography: The Java and C# libraries use Bouncy Castle to provide the required cryptography.
- * Coverage: Both implementations can produce and consume both the old and new countersignatures.
- * Testing: All of the examples in the example library are generated by the C# library and then validated using the Java and C libraries. Both libraries have tests to allow for the creating of the same messages that are in the example library followed by validating them. These are not compared against the example library. The Java and C# libraries have unit testing included. Not all of the MUST statements in the document have been implemented as part of the libraries. One such statement is the requirement that unique labels be present.
- * Licensing: Revised BSD License

7.2. COSE Testing Library

- * Implementation Location: <https://github.com/cose-wg/Examples>
- * Primary Maintainer: Jim Schaad

- * **Description:** A set of tests for the COSE library is provided as part of the implementation effort. Both success and fail tests have been provided. All of the examples in this document are part of this example set.
- * **Coverage:** An attempt has been made to have test cases for every message type and algorithm in the document. Currently examples dealing with countersignatures, and ECDH with Curve25519 and Goldilocks are missing.
- * **Licensing:** Public Domain

8. References

8.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC8949] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", STD 94, RFC 8949, DOI 10.17487/RFC8949, December 2020, <<https://www.rfc-editor.org/info/rfc8949>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [I-D.ietf-cose-rfc8152bis-algs] Schaad, J., "CBOR Object Signing and Encryption (COSE): Initial Algorithms", Work in Progress, Internet-Draft, draft-ietf-cose-rfc8152bis-algs-12, 24 September 2020, <<https://tools.ietf.org/html/draft-ietf-cose-rfc8152bis-algs-12>>.

8.2. Informative References

- [RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <<https://www.rfc-editor.org/info/rfc8610>>.
- [RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <<https://www.rfc-editor.org/info/rfc8152>>.

- [STD90] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, December 2017.
<<https://www.rfc-editor.org/info/std90>>
- [RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014,
<<https://www.rfc-editor.org/info/rfc7252>>.
- [RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016,
<<https://www.rfc-editor.org/info/rfc7942>>.
- [RFC4998] Gondrom, T., Brandner, R., and U. Pordesch, "Evidence Record Syntax (ERS)", RFC 4998, DOI 10.17487/RFC4998, August 2007, <<https://www.rfc-editor.org/info/rfc4998>>.
- [I-D.ietf-core-oscore-groupcomm]
Tiloca, M., Selander, G., Palombini, F., Mattsson, J. P., and J. Park, "Group OSCORE - Secure Group Communication for CoAP", Work in Progress, Internet-Draft, draft-ietf-core-oscore-groupcomm-11, 22 February 2021,
<<https://tools.ietf.org/html/draft-ietf-core-oscore-groupcomm-11>>.
- [I-D.ietf-cose-rfc8152bis-struct]
Schaad, J., "CBOR Object Signing and Encryption (COSE): Structures and Process", Work in Progress, Internet-Draft, draft-ietf-cose-rfc8152bis-struct-15, 1 February 2021,
<<https://tools.ietf.org/html/draft-ietf-cose-rfc8152bis-struct-15>>.
- [RFC8613] Selander, G., Mattsson, J., Palombini, F., and L. Seitz, "Object Security for Constrained RESTful Environments (OSCORE)", RFC 8613, DOI 10.17487/RFC8613, July 2019,
<<https://www.rfc-editor.org/info/rfc8613>>.

Appendix A. Examples

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

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

```
gem install cbor-diag
```

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

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

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

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

A.1. Use of Early Code Points

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

The examples in this Appendix use code points proposed for early allocation by IANA. When IANA makes the allocation, these examples will be updated as needed.

A.2. Examples of Signed Messages

A.2.1. Countersignature

This example uses the following:

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

Size of binary file is 180 bytes

```

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

```

A.3. Examples of Signed Messages

A.3.1. Countersignature

This example uses the following:

- * Signature Algorithm: ECDSA w/ SHA-256, Curve P-256
- * Countersignature Algorithm: ECDSA w/ SHA-512, Curve P-521

Size of binary file is 275 bytes

```

18(
  [
    / protected h'A201260300' / << {
      / alg / 1:-7, / ECDSA 256 /
      / ctyp / 3:0
    } >>,
    / unprotected / {
      / kid / 4: "11",
      / countersign / 11: [
        / protected h'A1013823' / << {
          / alg / 1:-36 / ECDSA 512 /
        } >>,
        / unprotected / {
          / kid / 4: "bilbo.baggins@hobbiton.example"
        },
        / signature / h'01B1291B0E60A79C459A4A9184A0D393E034B34AF069
A1CCA34F5A913AFFFF698002295FA9F8FCBFB6FDFF59132FC0C406E98754A98F1FBF
E81C03095F481856BC470170227206FA5BEE3C0431C56A66824E7AAF692985952E31
271434B2BA2E47A335C658B5E995AEB5D63CF2D0CED367D3E4CC8FFFD53B70D115BA
A9E86961FBD1A5CF'
      ]
    },
    / payload / 'This is the content.',
    / signature / h'BB587D6B15F47BFD54D2CBFCECEF75451E92B08A514BD439
FA3AA65C6AC92DF0D7328C4A47529B32ADD3DD1B4E940071C021E9A8F2641F1D8E3B
053DDD65AE52'
  ]
)

```

A.4. Examples of Enveloped Messages

A.4.1. Countersignature on Encrypted Content

This example uses the following:

- * CEK: AES-GCM w/ 128-bit key
- * Recipient class: ECDH Ephemeral-Static, Curve P-256
- * Countersignature Algorithm: ECDSA w/ SHA-512, Curve P-521

Size of binary file is 326 bytes

```

96(
  [
    / protected h'a10101' / << {
      / alg / 1:1 / AES-GCM 128 /
    } >>,
    / unprotected / {
      / iv / 5:h'c9cf4df2fe6c632bf7886413',
      / countersign / 11:[
        / protected h'a1013823' / << {
          / alg / 1:-36 / ES512 /
        } >> ,
        / unprotected / {
          / kid / 4:'bilbo.baggins@hobbiton.example'
        },
        / signature / h'00929663c8789bb28177ae28467e66377da12302d7f9
594d2999afa5dfa531294f8896f2b6cdf1740014f4c7f1a358e3a6cf57f4ed6fb02f
cf8f7aa989f5dfd07f0700a3a7d8f3c604ba70fa9411bd10c2591b483eld2c31de00
3183e434d8fba18f17a4c7e3dfa003ac1cf3d30d44d2533c4989d3ac38c38b71481c
c3430c9d65e7ddff'
      ],
    },
    / ciphertext / h'7adbe2709ca818fb415f1e5df66f4e1a51053ba6d65a1a0
c52a357da7a644b8070a151b0',
    / recipients / [
      [
        / protected h'a1013818' / << {
          / alg / 1:-25 / ECDH-ES + HKDF-256 /
        } >> ,
        / unprotected / {
          / ephemeral / -1:{
            / kty / 1:2,
            / crv / -1:1,
            / x / -2:h'98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbf
bf054e1c7b4d91d6280',
            / y / -3:true
          },
          / kid / 4:'meriadoc.brandybuck@buckland.example'
        },
        / ciphertext / h''
      ]
    ]
  ]
)

```

A.5. Examples of Encrypted Messages

A.5.1. Countersignature on Encrypted Content

This example uses the following:

- * CEK: AES-GCM w/ 128-bit key
- * Countersignature algorithm: EdDSA with Curve Ed25519

Size of binary file is 136 bytes

```
16(
  [
    / protected h'A10101' / << {
      / alg / 1:1 / AES-GCM 128 /
    } >>,
    / unprotected / {
      / iv / 5: h'02D1F7E6F26C43D4868D87CE',
      / countersign / 11: [
        / protected h'A10127' / << {
          / alg / 1:-8 / EdDSA with Ed25519 /
        } >>,
        / unprotected / {
          / kid / 4: '11'
        },
        / signature / h'E10439154CC75C7A3A5391491F88651E0292FD0FE0E0
2CF740547EAF6677B4A4040B8ECA16DB592881262F77B14C1A086C02268B17171CA1
6BE4B8595F8C0A08'
      ],
    / ciphertext / h'60973A94BB2898009EE52ECFD9AB1DD25867374B162E2C0
3568B41F57C3CC16F9166250A'
  ]
)
```

A.6. Examples of MACed Messages

A.6.1. Countersignature on MAC Content

This example uses the following:

- * MAC algorithm: HMAC/SHA-256 with 256-bit key
- * Countersignature algorithm: EdDSA with Curve Ed25519

Size of binary file is 159 bytes

```

97(
  [
    / protected h'A10105' / << {
      / alg / 1:5 / HS256 /
    } >>,
    / unprotected / {
      / countersign / 11: [
        / protected h'A10127' / << {
          / alg / 1:-8 / EdDSA /
        } >>,
        / unprotected / {
          / kid / 4: '11'
        },
        / signature / h'602566F4A311DC860740D2DF54D4864555E85BC036EA
5A6CF7905B96E499C5F66B01C4997F6A20C37C37543ADEA1D705347D38A5B13594B2
9583DD741F455101'
      ],
    },
    / payload / 'This is the content.',
    / tag / h'2BDCC89F058216B8A208DDC6D8B54AA91F48BD63484986565105C9
AD5A6682F6',
    / recipients / [
      [
        / protected / h'',
        / unprotected / {
          / alg / 1: -6, / direct /
          / kid / 4: 'our-secret'
        },
        / ciphertext / h''
      ]
    ]
  ]
)

```

A.7. Examples of MAC0 Messages

A.7.1. Countersignature on MAC0 Content

This example uses the following:

- * MAC algorithm: HMAC/SHA-256 with 256-bit key
- * Countersignature algorithm: EdDSA with Curve Ed25519

Size of binary file is 159 bytes

```

17(
  [
    / protected h'A10105' / << {
      / alg / 1:5 / HS256 /
    } >>,
    / unprotected / {
      / countersign / 11: [
        / protected h'A10127' / << {
          / alg / 1:-8 / EdDSA /
        } >>,
        / unprotected / {
          / kid / 4: '11'
        },
        / signature / h'968A315DF6B4F26362E11F4CFD2F2F4E76232F39657B
F1598837FF9332CDDD7581E248116549451F81EF823DA5974F885B681D3D6E38FC41
42D8F8E9E7DC8F0D'
      ],
    },
    / payload / 'This is the content.',
    / tag / h'A1A848D3471F9D61EE49018D244C824772F223AD4F935293F1789F
C3A08D8C58'
  ]
)

```

Acknowledgments

This document is a product of the COSE working group of the IETF.

The initial version of the specification was based to some degree on the outputs of the JOSE and S/MIME working groups.

Jim Schaad passed on 3 October 2020. This document is primarily his work. Russ Housley served as the document editor after Jim's untimely death, mostly helping with the approval and publication processes. Jim deserves all credit for the technical content.

Jim Schaad and Jonathan Hammell provided the examples in the Appendix.

{{{ RFC EDITOR: Please remove Russ Housley as an author of this document at publication. Jim Schaad should be listed as the sole author. }}}}

Authors' Addresses

Jim Schaad
August Cellars

Email: ietf@augustcellars.com

Russ Housley (editor)
Vigil Security, LLC

Email: housley@vigilsec.com

Network Working Group
Internet-Draft
Intended status: Informational
Expires: March 18, 2021

J. Schaad
August Cellars
September 14, 2020

CBOR Object Signing and Encryption (COSE): Hash Algorithms
draft-ietf-cose-hash-algs-09

Abstract

The CBOR Object Signing and Encryption (COSE) syntax [I-D.ietf-cose-rfc8152bis-struct] does not define any direct methods for using hash algorithms. There are, however, circumstances where hash algorithms are used, such as indirect signatures where the hash of one or more contents are signed, and X.509 certificate or other object identification by the use of a fingerprint. This document defines a set of hash algorithms that are identified by COSE Algorithm Identifiers.

Contributing to this document

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

The source for this draft is being maintained in GitHub. Suggested changes should be submitted as pull requests at <https://github.com/cose-wg/X509> Editorial changes can be managed in GitHub, but any substantial issues need to be discussed on the COSE mailing list.

Status of This Memo

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

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

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

This Internet-Draft will expire on March 18, 2021.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Requirements Terminology	3
2. Hash Algorithm Usage	3
2.1. Example CBOR hash structure	4
3. Hash Algorithm Identifiers	5
3.1. SHA-1 Hash Algorithm	6
3.2. SHA-2 Hash Algorithms	6
3.3. SHAKE Algorithms	8
4. IANA Considerations	9
4.1. COSE Algorithm Registry	9
5. Security Considerations	10
6. Normative References	10
7. Informative References	11
Author's Address	12

1. Introduction

The CBOR Object Signing and Encryption (COSE) syntax does not define any direct methods for the use of hash algorithms. It also does not define a structure syntax that is used to encode a digested object structure along the lines of the DigestedData ASN.1 structure in [CMS]. This omission was intentional, as a structure consisting of just a digest identifier, the content, and a digest value does not, by itself, provide any strong security service. Additionally, an application is going to be better off defining this type of structure so that it can include any additional data that needs to be hashed, as well as methods of obtaining the data.

While the above is true, there are some cases where having some standard hash algorithms defined for COSE with a common identifier makes a great deal of sense. Two of the cases where these are going to be used are:

- * Indirect signing of content, and
- * Object identification.

Indirect signing of content is a paradigm where the content is not directly signed, but instead a hash of the content is computed and that hash value, along with an identifier for the hash algorithm, is included in the content that will be signed. Doing indirect signing allows for a signature to be validated without first downloading all of the content associated with the signature. Rather the signature can be validated on all of the hash values and pointers to the associated contents, then those associated parts can be downloaded, the hash value of that part computed, and then compared to the hash value in the signed content. This capability can be of even greater importance in a constrained environment as not all of the content signed may be needed by the device. An example of how this is used can be found in [I-D.ietf-suit-manifest].

The use of hashes to identify objects is something that has been very common. One of the primary things that has been identified by a hash function in a secure message is a certificate. Two examples of this can be found in [ESS] and the COSE equivalents in [I-D.ietf-cose-x509].

1.1. Requirements Terminology

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

2. Hash Algorithm Usage

As noted in the previous section, hash functions can be used for a variety of purposes. Some of these purposes require that a hash function be cryptographically strong. These include direct and indirect signatures. That is, using the hash as part of the signature or using the hash as part of the body to be signed. Other uses of hash functions may not require the same level of strength.

This document contains some hash functions that are not designed to be used for cryptographic operations. An application that is using a hash function needs to carefully evaluate exactly what hash properties are needed and which hash functions are going to provide them. Applications should also make sure that the ability to change hash functions is part of the base design, as cryptographic advances are sure to reduce the strength of a hash function [BCP201].

A hash function is a map from one, normally large, bit string to a second, usually smaller, bit string. As the number of possible input values is far greater than the number of possible output values, it is inevitable that there are going to be collisions. The trick is to make sure that it is difficult to find two values that are going to map to the same output value. A "Collision Attack" is one where an attacker can find two different messages that have the same hash value. A hash function that is susceptible to practical collision attacks, SHOULD NOT be used for a cryptographic purpose. The discovery of theoretical collision attacks against a given hash function SHOULD trigger protocol maintainers and users to do a review of the continued suitability of the algorithm if alternatives are available and migration is viable. The only reason why such a hash function is used is when there is absolutely no other choice (e.g. a Hardware Security Module (HSM) that cannot be replaced), and only after looking at the possible security issues. Cryptographic purposes would include the creation of signatures or the use of hashes for indirect signatures. These functions may still be usable for non-cryptographic purposes.

An example of a non-cryptographic use of a hash is for filtering from a collection of values to find a set of possible candidates; the candidates can then be checked to see if they can successfully be used. A simple example of this is the classic fingerprint of a certificate. If the fingerprint is used to verify that it is the correct certificate, then that usage is a cryptographic one and is subject to the warning above about collision attack. If, however, the fingerprint is used to sort through a collection of certificates to find those that might be used for the purpose of verifying a signature, a simple filter capability is sufficient. In this case, one still needs to confirm that the public key validates the signature (and the certificate is trusted), and all certificates that don't contain a key that validates the signature can be discarded as false positives.

To distinguish between these two cases, a new value in the recommended column of the COSE Algorithms registry is to be added. "Filter Only" indicates that the only purpose of a hash function should be to filter results and it is not intended for applications which require a cryptographically strong algorithm.

2.1. Example CBOR hash structure

[COSE] did not provide a default structure for holding a hash value not only because no separate hash algorithms were defined, but because how the structure is setup is frequently application specific. There are four fields that are often included as part of a hash structure:

- * The hash algorithm identifier.
- * The hash value.
- * A pointer to the value that was hashed. This could be a pointer to a file, an object that can be obtained from the network, or a pointer to someplace in the message, or something very application specific.
- * Additional data; this can be something as simple as a random value (i.e. salt) to make finding hash collisions slightly harder (as the payload handed to the application could have been selected to have a collision), or as complicated as a set of processing instructions that are used with the object that is pointed to. The additional data can be dealt with in a number of ways, prepending or appending to the content, but it is strongly suggested that it either be a fixed known size, or the lengths of the pieces being hashed be included. (Encoding as a CBOR array accomplishes this requirement.)

An example of a structure which permits all of the above fields to exist would look like the following.

```
COSE_Hash_V = (  
  1 : int / tstr, # Algorithm identifier  
  2 : bstr, # Hash value  
  ? 3 : tstr, # Location of object that was hashed  
  ? 4 : any # object containing other details and things  
)
```

Below is an alternative structure that could be used in situations where one is searching a group of objects for a matching hash value. In this case, the location would not be needed and adding extra data to the hash would be counterproductive. This results in a structure that looks like this:

```
COSE_Hash_Find = [  
  hashAlg : int / tstr,  
  hashValue : bstr  
]
```

3. Hash Algorithm Identifiers

3.1. SHA-1 Hash Algorithm

The SHA-1 hash algorithm [RFC3174] was designed by the United States National Security Agency and published in 1995. Since that time a large amount of cryptographic analysis has been applied to this algorithm and a successful collision attack has been created ([SHA-1-collision]). The IETF formally started discouraging the use of SHA-1 with the publishing of [RFC6194].

Despite the above, there are still times where SHA-1 needs to be used and therefore it makes sense to assign a codepoint for the use of this hash algorithm. Some of these situations are with historic HSMS where only SHA-1 is implemented; other situations are where the SHA-1 value is used for the purpose of filtering and thus the collision resistance property is not needed.

Because of the known issues for SHA-1 and the fact that it should no longer be used, the algorithm will be registered with the recommendation of "Filter Only". This provides guidance about when the algorithm is safe for use, while discouraging usage where it is not safe.

The COSE capabilities for this algorithm is an empty array.

Name	Value	Description	Capabilities	Reference	Recommended
SHA-1	-14	SHA-1 Hash	[]	[This Document]	Filter Only

Table 1: SHA-1 Hash Algorithm

3.2. SHA-2 Hash Algorithms

The family of SHA-2 hash algorithms [FIPS-180-4] was designed by the United States National Security Agency and published in 2001. Since that time some additional algorithms have been added to the original set to deal with length extension attacks and some performance issues. While the SHA-3 hash algorithms have been published since that time, the SHA-2 algorithms are still broadly used.

There are a number of different parameters for the SHA-2 hash functions. The set of hash functions which have been chosen for inclusion in this document are based on those different parameters and some of the trade-offs involved.

- * *SHA-256/64* provides a truncated hash. The length of the truncation is designed to allow for smaller transmission size. The trade-off is that the odds that a collision will occur increase proportionally. Use of this hash function needs analysis of the potential problems with having a collision occur, or must be limited to where the function of the hash is non-cryptographic.

The latter is the case for [I-D.ietf-cose-x509]. The hash value is used to select possible certificates and, if there are multiple choices remaining then, each choice can be tested by using the public key.

- * *SHA-256* is probably the most common hash function used currently. SHA-256 is an efficient hash algorithm for 32-bit hardware.
- * *SHA-384* and *SHA-512* hash functions are efficient for 64-bit hardware.
- * *SHA-512/256* provides a hash function that runs more efficiently on 64-bit hardware, but offers the same security levels as SHA-256.

The COSE capabilities array for these algorithms is empty.

Name	Value	Description	Capabilities	Reference	Recommended
SHA-256/64	-15	SHA-2 256-bit Hash truncated to 64-bits	[]	[This Document]	Filter Only
SHA-256	-16	SHA-2 256-bit Hash	[]	[This Document]	Yes
SHA-384	-43	SHA-2 384-bit Hash	[]	[This Document]	Yes
SHA-512	-44	SHA-2 512-bit Hash	[]	[This Document]	Yes
SHA-512/256	-17	SHA-2 512-bit Hash truncated to 256-bits	[]	[This Document]	Yes

Table 2: SHA-2 Hash Algorithms

3.3. SHAKE Algorithms

The family of SHA-3 hash algorithms [FIPS-202] was the result of a competition run by NIST. The pair of algorithms known as SHAKE-128 and SHAKE-256 are the instances of SHA-3 that are currently being standardized in the IETF. This is the reason for including these algorithms in this document.

The SHA-3 hash algorithms have a significantly different structure than the SHA-2 hash algorithms.

Unlike the SHA-2 hash functions, no algorithm identifier is created for shorter lengths. The length of the hash value stored is 256-bits for SHAKE-128 and 512-bits for SHAKE-256.

The COSE capabilities array for these algorithms is empty.

Name	Value	Description	Capabilities	Reference	Recommended
SHAKE128	-18	SHAKE-128 256-bit Hash Value	[]	[This Document]	Yes
SHAKE256	-45	SHAKE-256 512-bit Hash Value	[]	[This Document]	Yes

Table 3: SHAKE Hash Functions

4. IANA Considerations

The IANA actions in [I-D.ietf-cose-rfc8152bis-struct] and [I-D.ietf-cose-rfc8152bis-algs] need to be executed before the actions in this document. Where early allocation of codepoints has been made, these should be preserved.

4.1. COSE Algorithm Registry

IANA is requested to register the following algorithms in the "COSE Algorithms" registry.

- * The SHA-1 hash function found in Table 1.
- * The set of SHA-2 hash functions found in Table 2.
- * The set of SHAKE hash functions found in Table 3.

Many of the hash values produced are relatively long and as such the use of a two byte algorithm identifier seems reasonable. SHA-1 is tagged as 'Filter Only' and thus a longer algorithm identifier is appropriate even though it is a shorter hash value.

IANA is requested to add the value of 'Filter Only' to the set of legal values for the 'Recommended' column. This value is only to be used for hash functions and indicates that it is not to be used for purposes which require collision resistance. IANA is requested to add this document to the reference section for this table due to this addition.

5. Security Considerations

Protocols need to perform a careful analysis of the properties of a hash function that are needed and how they map onto the possible attacks. In particular, one needs to distinguish between those uses that need the cryptographic properties, such as collision resistance, and properties that correspond to possible object identification. The different attacks correspond to who or what is being protected: is it the originator that is the attacker or a third party? This is the difference between collision resistance and second pre-image resistance. As a general rule, longer hash values are "better" than short ones, but trade-offs of transmission size, timeliness, and security all need to be included as part of this analysis. In many cases the value being hashed is a public value and, as such, pre-image resistance is not part of this analysis.

Algorithm agility needs to be considered a requirement for any use of hash functions [BCP201]. As with any cryptographic function, hash functions are under constant attack and the cryptographic strength of hash algorithms will be reduced over time.

6. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [I-D.ietf-cose-rfc8152bis-struct] Schaad, J., "CBOR Object Signing and Encryption (COSE): Structures and Process", Work in Progress, Internet-Draft, draft-ietf-cose-rfc8152bis-struct-12, August 24, 2020, <<https://tools.ietf.org/html/draft-ietf-cose-rfc8152bis-struct-12>>.
- [FIPS-180-4] National Institute of Standards and Technology, "Secure Hash Standard", FIPS PUB 180-4, August 2015.
- [FIPS-202] National Institute of Standards and Technology, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", FIPS PUB 202, August 2015.

- [RFC3174] Eastlake 3rd, D. and P. Jones, "US Secure Hash Algorithm 1 (SHA1)", RFC 3174, DOI 10.17487/RFC3174, September 2001, <<https://www.rfc-editor.org/info/rfc3174>>.

7. Informative References

- [CMS] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <<https://www.rfc-editor.org/info/rfc5652>>.
- [ESS] Hoffman, P., Ed., "Enhanced Security Services for S/MIME", RFC 2634, DOI 10.17487/RFC2634, June 1999, <<https://www.rfc-editor.org/info/rfc2634>>.
- [I-D.ietf-cose-x509] Schaad, J., "CBOR Object Signing and Encryption (COSE): Header parameters for carrying and referencing X.509 certificates", Work in Progress, Internet-Draft, draft-ietf-cose-x509-06, March 9, 2020, <<https://tools.ietf.org/html/draft-ietf-cose-x509-06>>.
- [RFC6194] Polk, T., Chen, L., Turner, S., and P. Hoffman, "Security Considerations for the SHA-0 and SHA-1 Message-Digest Algorithms", RFC 6194, DOI 10.17487/RFC6194, March 2011, <<https://www.rfc-editor.org/info/rfc6194>>.
- [I-D.ietf-cose-rfc8152bis-algs] Schaad, J., "CBOR Object Signing and Encryption (COSE): Initial Algorithms", Work in Progress, Internet-Draft, draft-ietf-cose-rfc8152bis-algs-11, July 1, 2020, <<https://tools.ietf.org/html/draft-ietf-cose-rfc8152bis-algs-11>>.
- [I-D.ietf-suit-manifest] Moran, B., Tschofenig, H., Birkholz, H., and K. Zandberg, "A Concise Binary Object Representation (CBOR)-based Serialization Format for the Software Updates for Internet of Things (SUIT) Manifest", Work in Progress, Internet-Draft, draft-ietf-suit-manifest-09, July 13, 2020, <<https://tools.ietf.org/html/draft-ietf-suit-manifest-09>>.
- [BCP201] Housley, R., "Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms", BCP 201, RFC 7696, November 2015. <<https://www.rfc-editor.org/info/bcp201>>

[SHA-1-collision]

Stevens, M., Bursztein, E., Karpman, P., Albertini, A.,
and Y. Markov, "The first collision for full SHA-1",
February 2017,
<<https://shattered.io/static/shattered.pdf>>.

[COSE]

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

Author's Address

Jim Schaad
August Cellars

Email: ietf@augustcellars.com

COSE Working Group
Internet-Draft
Obsoletes: 8152 (if approved)
Intended status: Informational
Expires: 28 March 2021

J. Schaad
August Cellars
24 September 2020

CBOR Object Signing and Encryption (COSE): Initial Algorithms
draft-ietf-cose-rfc8152bis-algs-12

Abstract

Concise Binary Object Representation (CBOR) is a data format designed for small code size and small message size. There is a need for the ability to have basic security services defined for this data format. This document defines a set of algorithms that can be used with the CBOR Object Signing and Encryption (COSE) protocol RFC XXXX.

Contributing to this document

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

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 28 March 2021.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	3
1.1. Requirements Terminology	4
1.2. Changes from RFC8152	4
1.3. Document Terminology	4
1.4. CBOR Grammar	5
1.5. Examples	5
2. Signature Algorithms	5
2.1. ECDSA	5
2.1.1. Security Considerations for ECDSA	7
2.2. Edwards-Curve Digital Signature Algorithms (EdDSAs)	8
2.2.1. Security Considerations for EdDSA	9
3. Message Authentication Code (MAC) Algorithms	9
3.1. Hash-Based Message Authentication Codes (HMACs)	9
3.1.1. Security Considerations for HMAC	11
3.2. AES Message Authentication Code (AES-CBC-MAC)	11
3.2.1. Security Considerations AES-CBC_MAC	12
4. Content Encryption Algorithms	12
4.1. AES GCM	12
4.1.1. Security Considerations for AES-GCM	13
4.2. AES CCM	14
4.2.1. Security Considerations for AES-CCM	17
4.3. ChaCha20 and Poly1305	18
4.3.1. Security Considerations for ChaCha20/Poly1305	19
5. Key Derivation Functions (KDFs)	19
5.1. HMAC-Based Extract-and-Expand Key Derivation Function (HKDF)	19
5.2. Context Information Structure	21
6. Content Key Distribution Methods	26
6.1. Direct Encryption	27
6.1.1. Direct Key	27
6.1.2. Direct Key with KDF	28
6.2. Key Wrap	29

6.2.1. AES Key Wrap	30
6.3. Direct Key Agreement	31
6.3.1. Direct ECDH	31
6.4. Key Agreement with Key Wrap	34
6.4.1. ECDH with Key Wrap	35
7. Key Object Parameters	37
7.1. Elliptic Curve Keys	37
7.1.1. Double Coordinate Curves	38
7.2. Octet Key Pair	39
7.3. Symmetric Keys	40
8. COSE Capabilities	41
8.1. Assignments for Existing Algorithms	42
8.2. Assignments for Existing Key Types	42
8.3. Examples	42
9. CBOR Encoding Restrictions	45
10. IANA Considerations	45
10.1. Changes to "COSE Key Types" registry.	45
10.2. Changes to "COSE Algorithms" registry	46
10.3. Changes to the "COSE Key Type Parameters" registry	46
10.4. Expert Review Instructions	46
11. Security Considerations	47
12. References	49
12.1. Normative References	49
12.2. Informative References	51
Acknowledgments	54
Author's Address	54

1. Introduction

There has been an increased focus on small, constrained devices that make up the Internet of Things (IoT). One of the standards that has come out of this process is "Concise Binary Object Representation (CBOR)" [RFC7049]. CBOR extended the data model of JavaScript Object Notation (JSON) [STD90] 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 transported and implementation size and be a schema-free decoder. A need exists to provide message security services for IoT, and using CBOR as the message-encoding format makes sense.

The core COSE specification consists of two documents. [I-D.ietf-cose-rfc8152bis-struct] contains the serialization structures and the procedures for using the different cryptographic algorithms. This document provides an initial set of algorithms for use with those structures.

1.1. Requirements Terminology

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

1.2. Changes from RFC8152

- * Extract the sections dealing with specific algorithms into this document. The sections dealing with structure and general processing rules are placed in [I-D.ietf-cose-rfc8152bis-struct].
- * Text clarifications and changes in terminology.

1.3. Document Terminology

In this document, we use the following terminology:

Byte is a synonym for octet.

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

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

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

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

The tables for algorithms contain the following columns:

- * A name for use in documents for the algorithms.
- * The value used on the wire for the algorithm. One place this is used is the algorithm header parameter of a message.

- * A short description so that the algorithm can be easily identified when scanning the IANA registry.

Additional columns may be present in the table depending on the algorithms.

1.4. CBOR Grammar

At the time that [RFC8152] was initially published, the CBOR Data Definition Language (CDDL) [RFC8610] had not yet been published. This document uses a variant of CDDL which is described in [I-D.ietf-cose-rfc8152bis-struct].

1.5. Examples

A GitHub project has been created at [GitHub-Examples] that contains a set of testing examples as well. Each example is found in a JSON file that contains the inputs used to create the example, some of the intermediate values that can be used for debugging, and the output of the example. The results are encoded using both hexadecimal and CBOR diagnostic notation format.

Some of the examples are designed to test failure case; these are clearly marked as such in the JSON file. If errors in the examples in this document are found, the examples on GitHub will be updated, and a note to that effect will be placed in the JSON file.

2. Signature Algorithms

Section 9.1 of [I-D.ietf-cose-rfc8152bis-struct] contains a generic description of signature algorithms. The document defines signature algorithm identifiers for two signature algorithms.

2.1. ECDSA

ECDSA [DSS] defines a signature algorithm using ECC. Implementations SHOULD use a deterministic version of ECDSA such as the one defined in [RFC6979]. The use of a deterministic signature algorithm allows for systems to avoid relying on random number generators in order to avoid generating the same value of 'k' (the per-message random value). Biased generation of the value 'k' can be attacked, and collisions of this value leads to leaked keys. It additionally allows for doing deterministic tests for the signature algorithm. The use of deterministic ECDSA does not lessen the need to have good random number generation when creating the private key.

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 leftmost bytes of the hash output are used.

The algorithms defined in this document can be found in Table 1.

Name	Value	Hash	Description
ES256	-7	SHA-256	ECDSA w/ SHA-256
ES384	-35	SHA-384	ECDSA w/ SHA-384
ES512	-36	SHA-512	ECDSA w/ SHA-512

Table 1: 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' (two coordinate elliptic curve) key type. Implementations need to check that the key type and curve are correct when creating and verifying a signature. Future documents may define 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 the same length as the 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 [RFC8017], the signature is:

Signature = I2OSP(R, n) | I2OSP(S, n)

where $n = \text{ceiling}(\text{key_length} / 8)$

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

2.1.1.1. Security Considerations for ECDSA

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.

Note: Use of a deterministic signature technique is 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 and allows for reproducible signature values, which helps testing. There have been recent attacks involving faulting the device in order to extract the key. This can be addressed by combining both randomness and determinism [I-D.mattsson-cfrg-det-sigs-with-noise].

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

- * Changing the curve used to validate the signature: If one changes the curve used to validate the signature, then potentially one could have 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 currently 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.

- * Change the hash function used to validate the signature: If one either has two different hash functions of the same length or can truncate a hash function, 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). As the hash algorithm is part of the signature algorithm identifier, this attack is mitigated by including a signature algorithm identifier in the protected header bucket.

2.2. Edwards-Curve Digital Signature Algorithms (EdDSAs)

[RFC8032] describes the elliptic curve signature scheme Edwards-curve Digital Signature Algorithm (EdDSA). In that document, the signature algorithm is instantiated using parameters for edwards25519 and edwards448 curves. The document additionally describes two variants of the EdDSA algorithm: Pure EdDSA, where no hash function is applied to the content before signing, and HashEdDSA, where a hash function is applied to the content before signing and the result of that hash function is signed. For EdDSA, the content to be signed (either the message or the pre-hash value) is processed twice inside of the signature algorithm. For use with COSE, only the pure EdDSA version is used. This is because it is not expected that extremely large contents are going to be needed and, based on the arrangement of the message structure, the entire message is going to need to be held in memory in order to create or verify a signature. This means that there does not appear to be a need to be able to do block updates of the hash, followed by eliminating the message from memory. Applications can provide the same features by defining the content of the message as a hash value and transporting the COSE object (with the hash value) and the content as separate items.

The algorithms defined in this document can be found in Table 2. A single signature algorithm is defined, which can be used for multiple curves.

Name	Value	Description
EdDSA	-8	EdDSA

Table 2: EdDSA Algorithm Values

[RFC8032] describes the method of encoding the signature value.

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

- * The 'kty' field MUST be present, and it MUST be 'OKP' (Octet Key Pair).
- * The 'crv' field MUST be present, and it MUST be a curve defined for this signature algorithm.
- * If the 'alg' field is present, it MUST match 'EdDSA'.
- * If the 'key_ops' field is present, it MUST include 'sign' when creating an EdDSA signature.
- * If the 'key_ops' field is present, it MUST include 'verify' when verifying an EdDSA signature.

2.2.1. Security Considerations for EdDSA

How public values are computed is not the same when looking at EdDSA and Elliptic Curve Diffie-Hellman (ECDH); for this reason, the public key should not be used with the other algorithm.

If batch signature verification is performed, a well-seeded cryptographic random number generator is REQUIRED (Section 8.2 of [RFC8032]). Signing and non-batch signature verification are deterministic operations and do not need random numbers of any kind.

3. Message Authentication Code (MAC) Algorithms

Section 9.2 of [I-D.ietf-cose-rfc8152bis-struct] contains a generic description of MAC algorithms. This section defines the conventions for two MAC algorithms.

3.1. Hash-Based Message Authentication Codes (HMACs)

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 shown as solid since, 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 one HMAC algorithm that is truncated. There are currently no known issues with truncation; however, the security

strength of the message tag is correspondingly reduced in strength. When truncating, the leftmost tag length bits are kept and transmitted.

The algorithms defined in this document can be found in Table 3.

Name	Value	Hash	Tag Length	Description
HMAC 256/64	4	SHA-256	64	HMAC w/ SHA-256 truncated to 64 bits
HMAC 256/256	5	SHA-256	256	HMAC w/ SHA-256
HMAC 384/384	6	SHA-384	384	HMAC w/ SHA-384
HMAC 512/512	7	SHA-512	512	HMAC w/ SHA-512

Table 3: HMAC Algorithm Values

Some recipient algorithms transport the key, while others derive a key from secret data. For those algorithms that transport the key (such as AES Key Wrap), the size of the HMAC key SHOULD be the same size as the output of the underlying hash function. For those algorithms that derive the key (such as ECDH), 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 is present, it MUST match the HMAC algorithm being used.
- * If the 'key_ops' field is present, it MUST include 'MAC create' when creating an HMAC authentication tag.
- * If the 'key_ops' field is present, it MUST include 'MAC 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.

3.1.1. Security Considerations for HMAC

HMAC has proved to be resistant to attack even when used with weakened hash algorithms. The current best known attack is to brute force the key. This means that key size is going to be directly related to the security of an HMAC operation.

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

AES-CBC-MAC is defined in [MAC]. (Note that this is not the same algorithm as AES Cipher-Based Message Authentication Code (AES-CMAC) [RFC4493].)

AES-CBC-MAC is parameterized by the key length, the authentication tag length, and the Initialization Vector (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 4.

Name	Value	Key Length	Tag Length	Description
AES-MAC 128/64	14	128	64	AES-MAC 128-bit key, 64-bit tag
AES-MAC 256/64	15	256	64	AES-MAC 256-bit key, 64-bit tag
AES-MAC 128/128	25	128	128	AES-MAC 128-bit key, 128-bit tag
AES-MAC 256/128	26	256	128	AES-MAC 256-bit key, 128-bit tag

Table 4: 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:

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

- * If the 'alg' field is present, it MUST match the AES-MAC algorithm being used.
- * If the 'key_ops' field is present, it MUST include 'MAC create' when creating an AES-MAC authentication tag.
- * If the 'key_ops' field is present, it MUST include 'MAC verify' when verifying an AES-MAC authentication tag.

3.2.1. Security Considerations AES-CBC_MAC

A number of attacks exist against Cipher Block Chaining Message Authentication Code (CBC-MAC) that need to be considered.

- * A single key must only be used for messages of a fixed or known length. If this is not the case, an attacker will be able to generate a message with a valid tag given two message and tag pairs. This can be addressed by using different keys for messages of different lengths. The current structure mitigates this problem, as a specific encoding structure that includes lengths is built and signed. (CMAC also addresses this issue.)
- * In cipher Block Chaining (CBC) mode, if the same key is used for both encryption and authentication operations, 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.

4. Content Encryption Algorithms

Section 9.3 of [I-D.ietf-cose-rfc8152bis-struct] contains a generic description of Content Encryption algorithms. This document defines the identifier and usages for three content encryption algorithms.

4.1. AES GCM

The Galois/Counter Mode (GCM) mode is a generic AEAD 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 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 5.

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 5: 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:

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

4.1.1. Security Considerations for AES-GCM

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

- * The key and nonce pair MUST be unique for every message encrypted.
- * The total number of messages encrypted for a single key MUST NOT exceed 2^{32} [SP800-38d]. An explicit check is required only in environments where it is expected that it might be exceeded.
- * A more recent analysis in [ROBUST] indicates that the the number of failed decryptions needs to be taken into account as part determining when a key roll-over is to be done. Following the recommendation of for DTLS, the number of failed message decryptions should be limited to 2^{36} .

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

4.2. AES CCM

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 growth (from the tag) and the probability 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 convention 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 both L and M. Less-constrained devices will want to be able to use larger messages and are more willing to generate new keys for every operation. This favors larger values of L and M.

The following values are used for L:

16 bits (2): This limits messages to 2^{16} bytes (64 KiB) in length. This is sufficiently long for messages in the constrained world. The nonce length is 13 bytes allowing for 2^{104} possible values of the nonce without repeating.

64 bits (8): This 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): This 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): This 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	Key Length	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	12	64	64	128	AES-CCM mode 128-bit key, 64-bit tag, 7-byte nonce
AES-CCM-64-64-256	13	64	64	256	AES-CCM mode 256-bit key, 64-bit tag, 7-byte nonce
AES-CCM-16-128-128	30	16	128	128	AES-CCM mode 128-bit key, 128-bit tag, 13-byte nonce
AES-CCM-16-128-256	31	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 6: 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 is present, it MUST match the AES-CCM algorithm being used.
- * If the 'key_ops' field is present, it MUST include 'encrypt' or 'wrap key' when encrypting.
- * If the 'key_ops' field is present, it MUST include 'decrypt' or 'unwrap key' when decrypting.

4.2.1. Security Considerations for AES-CCM

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

- * The key and nonce pair MUST be unique for every message encrypted. Note that the value of L influences the number of unique nonces.
- * 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.
- * [I-D.ietf-quic-tls] contains an analysis on the use of AES-CCM in that environment. Based on that recommendation, one should restrict the number of messages encrypted to 2^{23} . If one is using the 64-bit tag, then the limits are significantly smaller if one wants to keep the same integrity limits. A protocol recommending this needs to analyze what level of integrity is acceptable for the smaller tag size. It may be that to keep the desired integrity one needs to re-key as often as every 2^7 messages.
- * In addition to the number of messages successfully decrypted, the number of failed decryptions needs to be kept as well. If the number of failed decryptions exceeds 2^{23} then a rekeying operation should occur.

[RFC3610] additionally calls out one other consideration of note. It is possible to do a pre-computation attack against the algorithm in cases where portions of the plaintext are 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.

4.3. ChaCha20 and Poly1305

ChaCha20 and Poly1305 combined together is an AEAD mode that is defined in [RFC8439]. This is an 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 [RFC8439] has no parameterization. It takes a 256-bit key and a 96-bit nonce, as well as the plaintext and additional data as inputs and produces the ciphertext as an option. We define one algorithm identifier for this algorithm in Table 7.

Name	Value	Description
ChaCha20/Poly1305	24	ChaCha20/Poly1305 w/ 256-bit key, 128-bit tag

Table 7: Algorithm Value for ChaCha20/Poly1305

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 is present, it MUST match the ChaCha20/Poly1305 algorithm being used.

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

4.3.1. Security Considerations for ChaCha20/Poly1305

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

A more recent analysis in [ROBUST] indicates that the the number of failed decryptions needs to be taken into account as part determining when a key roll-over is to be done. Following the recommendation of for DTLS, the number of failed message decryptions should be limited to 2^{36} .

[I-D.ietf-quic-tls] recommends that no more than $2^{24.5}$ messages be encrypted under a single key.

5. Key Derivation Functions (KDFs)

Section 9.4 of [I-D.ietf-cose-rfc8152bis-struct] contains a generic description of Key Derivation Functions. This document defines a single context structure and a single KDF. These elements are used for all of the recipient algorithms defined in this document that require a KDF process. These algorithms are defined in Sections 6.1.2, 6.3.1, and 6.4.1.

5.1. HMAC-Based Extract-and-Expand Key Derivation Function (HKDF)

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

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 Diffie-Hellman (DH) key agreement.

salt -- an optional value that is used to change the generation process. The salt value can be either public or private. If the salt is public and carried in the message, then the 'salt' algorithm header parameter defined in Table 9 is used. While [RFC5869] suggests that the length of the salt be the same as the length of the underlying hash value, any positive salt length will improve the security as different key values will be generated. This parameter is protected by being included in the key computation and does not need to be separately authenticated. The salt value does not need to be unique for every message sent.

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 in which the material is going to be used ensures that the resulting material will always be tied to that usage. The context structure defined in Section 5.2 is used by the KDFs in this document.

PRF -- The underlying pseudorandom 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 that is 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. For the AES algorithm versions, the extract step is always skipped.

The extract step cannot be skipped if the secret is not uniformly random, for example, if it is the result of an ECDH key agreement step. This implies that the AES HKDF version cannot be used with ECDH. If the extract step is skipped, the 'salt' value is not used as part of the HKDF functionality.

The algorithms defined in this document are found in Table 8.

Name	PRF	Description
HKDF SHA-256	HMAC with SHA-256	HKDF using HMAC SHA-256 as the PRF
HKDF SHA-512	HMAC with SHA-512	HKDF using HMAC SHA-512 as the PRF
HKDF AES-MAC-128	AES-CBC-MAC-128	HKDF using AES-MAC as the PRF w/ 128-bit key
HKDF AES-MAC-256	AES-CBC-MAC-256	HKDF using AES-MAC as the PRF w/ 256-bit key

Table 8: HKDF Algorithms

Name	Label	Type	Algorithm	Description
salt	-20	bstr	direct+HKDF-SHA-256, direct+HKDF-SHA-512, direct+HKDF-AES-128, direct+HKDF-AES-256, ECDH-ES+HKDF-256, ECDH-ES+HKDF-512, ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-ES+A128KW, ECDH-ES+A192KW, ECDH-ES+A256KW, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	Random salt

Table 9: HKDF Algorithm Parameters

5.2. 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]).

The context information structure refers to PartyU and PartyV as the two parties that 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.

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

- * Fields can be defined by the application. This is commonly used to assign fixed names to parties, but it can be used for other items such as nonces.
- * Fields can be defined by usage of the output. Examples of this are the algorithm and key size that are being generated.
- * Fields can be defined by parameters from the message. We define a set of header parameters in Table 10 that can be used to carry the values associated with the context structure. Examples of this are identities and nonce values. These header 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.

Name	Label	Type	Algorithm	Description
PartyU identity	-21	bstr	direct+HKDF-SHA-256, direct+HKDF-SHA-512, direct+HKDF-AES-128, direct+HKDF-AES-256, ECDH-ES+HKDF-256, ECDH-ES+HKDF-512, ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-ES+A128KW, ECDH-ES+A192KW, ECDH-ES+A256KW, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	Party U identity information
PartyU nonce	-22	bstr /	direct+HKDF-SHA-256, direct+HKDF-SHA-512,	Party U provided

		int	direct+HKDF-AES-128, direct+HKDF-AES-256, ECDH-ES+HKDF-256, ECDH-ES+HKDF-512, ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-ES+A128KW, ECDH-ES+A192KW, ECDH-ES+A256KW, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	nonce
PartyU other	-23	bstr	direct+HKDF-SHA-256, direct+HKDF-SHA-512, direct+HKDF-AES-128, direct+HKDF-AES-256, ECDH-ES+HKDF-256, ECDH-ES+HKDF-512, ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-ES+A128KW, ECDH-ES+A192KW, ECDH-ES+A256KW, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	Party U other provided information
PartyV identity	-24	bstr	direct+HKDF-SHA-256, direct+HKDF-SHA-512, direct+HKDF-AES-128, direct+HKDF-AES-256, ECDH-ES+HKDF-256, ECDH-ES+HKDF-512, ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-ES+A128KW, ECDH-ES+A192KW, ECDH-ES+A256KW, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	Party V identity information
PartyV nonce	-25	bstr / int	direct+HKDF-SHA-256, direct+HKDF-SHA-512, direct+HKDF-AES-128, direct+HKDF-AES-256, ECDH-ES+HKDF-256,	Party V provided nonce

			ECDH-ES+HKDF-512, ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-ES+A128KW, ECDH-ES+A192KW, ECDH-ES+A256KW, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	
PartyV other	-26	bstr	direct+HKDF-SHA-256, direct+HKDF-SHA-512, direct+HKDF-AES-128, direct+HKDF-AES-256, ECDH-ES+HKDF-256, ECDH-ES+HKDF-512, ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-ES+A128KW, ECDH-ES+A192KW, ECDH-ES+A256KW, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	Party V other provided information

Table 10: Context Algorithm Parameters

We define a CBOR object to hold the context information. This object is referred to as COSE_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 normally is either a key wrap algorithm identifier or a content encryption algorithm identifier. The values are from the "COSE Algorithms" registry. This field is required to be present. The field exists in the context information so that a different key is generated for each algorithm even if all of the other context information is the same. In practice, this means if algorithm A is broken and thus finding the key is relatively easy, the key derived for algorithm B will not be the same as the key derived for algorithm A.

PartyUInfo: This field holds information about party U. The PartyUInfo is encoded as a CBOR array. The elements of PartyUInfo are encoded in the order presented below. 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. First, 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 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 is set to nil in that case.

nonce: This contains a nonce value. The nonce can either be implicit from the protocol or be carried as a value in the unprotected header bucket.

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 option does not need to be specified and is set to nil in that case.

other: This contains other information that is defined by the protocol. This option does not need to be specified and is set to nil in that case.

PartyVInfo: This field holds information about party V. The content of the structure is the same as for the PartyUInfo but for party V.

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. If

there are no elements in the protected field, then use a zero-length bstr.

other: This field is for free form data defined by the application. An example is that an application could define two different byte strings to be placed here to generate different keys for a data stream versus 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 included.

SuppPrivInfo: This field contains private information that is mutually known private information. An example of this information would be a preexisting shared secret. (This could, for example, be used in combination with an ECDH key agreement to provide a secondary proof of identity.) 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.

The following CDDL fragment corresponds to the text above.

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

6. Content Key Distribution Methods

Section 9.5 of [I-D.ietf-cose-rfc8152bis-struct] contains a generic description of content key distribution methods. This document defines the identifiers and usage for a number of content key distribution methods.

6.1. Direct Encryption

Direct encryption algorithm is defined in Section 9.5.1 of [I-D.ietf-cose-rfc8152bis-struct]. Information about how to fill in the COSE_Recipient structure are detailed there.

6.1.1. 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 11.

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 11: Direct Key

6.1.1.1. Security Considerations for Direct Key

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 for both the CBC encryption mode and the 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.

6.1.2. Direct Key with KDF

These recipient algorithms take a common shared secret between the two parties and applies the HKDF function (Section 5.1), using the context structure defined in Section 5.2 to transform the shared secret into the CEK. The 'protected' field can be of non-zero length. 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 shared secret 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 output of 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 for each message if the same key is used. The IV can be modified in a predictable manner, a random manner, or an unpredictable manner (i.e., encrypting a counter).

The IV used for a key can also be generated from the same HKDF functionality as the key is generated. If HKDF is used for generating the IV, the algorithm identifier is set to "IV-GENERATION".

The set of algorithms defined in this document can be found in Table 12.

Name	Value	KDF	Description
direct+HKDF-SHA-256	-10	HKDF SHA-256	Shared secret w/ HKDF and SHA-256
direct+HKDF-SHA-512	-11	HKDF SHA-512	Shared secret w/ HKDF and SHA-512
direct+HKDF-AES-128	-12	HKDF AES- MAC-128	Shared secret w/ AES-MAC 128-bit key
direct+HKDF-AES-256	-13	HKDF AES- MAC-256	Shared secret w/ AES-MAC 256-bit key

Table 12: Direct Key with KDF

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 is present, it MUST match the algorithm being used.
- * If the 'key_ops' field is present, it MUST include 'deriveKey' or 'deriveBits'.

6.1.2.1. Security Considerations for Direct Key with KDF

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.

While these methods do not provide for perfect forward secrecy, as the same shared secret is used for all of the keys generated, if the key for any single message is discovered, only the message (or series of messages) using that derived key are compromised. A new key derivation step will generate a new key that requires the same amount of work to get the key.

6.2. Key Wrap

Key wrap is defined in Section 9.5.1 of [I-D.ietf-cose-rfc8152bis-struct]. Information about how to fill in the COSE_Recipient structure is detailed there.

6.2.1. AES Key Wrap

The AES Key Wrap 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 key parameters that vary on a per-invocation basis. The protected header bucket MUST be empty.

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 is present, it MUST match the AES Key Wrap algorithm being used.
- * If the 'key_ops' field is present, it MUST include 'encrypt' or 'wrap key' when encrypting.
- * If the 'key_ops' field is present, it MUST include 'decrypt' or 'unwrap key' 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 13: AES Key Wrap Algorithm Values

6.2.1.1. Security Considerations for AES-KW

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

6.3. Direct Key Agreement

Key Transport is defined in Section 9.5.4 of [I-D.ietf-cose-rfc8152bis-struct]. Information about how to fill in the COSE_Recipient structure is detailed there.

6.3.1. Direct ECDH

The mathematics for ECDH can be found in [RFC6090]. In this document, the algorithm is extended to be used with the two curves defined in [RFC7748].

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 18.

The math used to obtain the computed secret is based on the curve selected and not on the ECDH algorithm. For this reason, a new algorithm does not need to be defined for each of the curves.

- * **Computed Secret to Shared Secret:** Once the computed secret is known, the resulting value needs to be converted to a byte string to run the KDF. The x-coordinate is used for all of the curves defined in this document. For curves X25519 and X448, the resulting value is used directly as it is a byte string of a known length. For the P-256, P-384, and P-521 curves, the x-coordinate is run through the I2OSP function defined in [RFC8017], using the same computation for n as is defined in Section 2.1.
- * **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 create a unique value. For the KDFs used, this means either the 'salt' parameter for HKDF (Table 9) or the 'PartyU nonce' parameter for the context structure (Table 10) MUST be present (both can be present if desired). The value in the parameter MUST be unique for the pair of keys being used. It is acceptable to use a global counter that is incremented for every static-static operation and use the resulting value. Care must be taken that

the counter is saved to permanent storage in a way to avoid reuse of that counter value. When using static keys, the static key should 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 header parameters are defined in Table 15.

- * 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 context material: how the key is going to be used and one-time material for static-static key agreement. All of the algorithms defined in this document use one of the HKDF algorithms defined in Section 5.1 with the context structure defined in Section 5.2.
- * Key Wrap Algorithm: No key wrap algorithm is used. This is represented in Table 14 as 'none'. The key size for the context structure is the content layer encryption algorithm size.

COSE does not have an Ephemeral-Ephemeral version defined. The reason for this is that COSE is not an online protocol by itself and thus does not have a method to establish ephemeral secrets on both sides. The expectation is that a protocol would establish the secrets for both sides, and then they would be used as static-static for the purposes of COSE, or that the protocol would generate a shared secret and a direct encryption would be used.

The set of direct ECDH algorithms defined in this document are found in Table 14.

Name	Value	KDF	Ephemeral-Static	Key Wrap	Description
ECDH-ES + HKDF-256	-25	HKDF - SHA-256	yes	none	ECDH ES w/ HKDF - generate key directly
ECDH-ES + HKDF-512	-26	HKDF - SHA-512	yes	none	ECDH ES w/ HKDF - generate key directly
ECDH-SS + HKDF-256	-27	HKDF - SHA-256	no	none	ECDH SS w/ HKDF - generate key directly
ECDH-SS + HKDF-512	-28	HKDF - SHA-512	no	none	ECDH SS w/ HKDF - generate key directly

Table 14: ECDH Algorithm Values

Name	Label	Type	Algorithm	Description
ephemeral key	-1	COSE_Key	ECDH-ES+HKDF-256, ECDH-ES+HKDF-512, ECDH-ES+A128KW, ECDH-ES+A192KW, ECDH-ES+A256KW	Ephemeral public key for the sender
static key	-2	COSE_Key	ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	Static public key for the sender
static key id	-3	bstr	ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	Static public key identifier for the sender

Table 15: ECDH Algorithm Parameters

This document defines these algorithms to be used with the curves P-256, P-384, P-521, X25519, and X448. 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' or 'OKP'.
- * If the 'alg' field is 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.

6.3.1.1. Security Considerations for ECDH

There is a method of checking that points provided from external entities are valid. For the 'EC2' key format, this can be done by checking that the x and y values form a point on the curve. For the 'OKP' format, there is no simple way to do point validation.

Consideration was given to requiring that the public keys of both entities be provided as part of the key derivation process (as recommended in Section 6.4 of [RFC7748]). This was not done as COSE is used in a store and forward format rather than in online key exchange. In order for this to be a problem, either the receiver public key has to be chosen maliciously or the sender has to be malicious. In either case, all security evaporates anyway.

A proof of possession of the private key associated with the public key is recommended when a key is moved from untrusted to trusted (either by the end user or by the entity that is responsible for making trust statements on keys).

6.4. Key Agreement with Key Wrap

Key Agreement with Key Wrap is defined in Section 9.5.5 of [I-D.ietf-cose-rfc8152bis-struct]. Information about how to fill in the COSE_Recipient structure are detailed there.

6.4.1. ECDH with Key Wrap

These algorithms are defined in Table 16.

ECDH with Key Agreement is parameterized by the same header parameters as for ECDH; see Section 6.3.1, with the following modifications:

- * Key Wrap Algorithm: Any of the key wrap algorithms defined in Section 6.2 are supported. The size of the key used for the key wrap algorithm is fed into the KDF. The set of identifiers are found in Table 16.

Name	Value	KDF	Ephemeral-Static	Key Wrap	Description
ECDH-ES + A128KW	-29	HKDF - SHA-256	yes	A128KW	ECDH ES w/ Concat KDF and AES Key Wrap w/ 128-bit key
ECDH-ES + A192KW	-30	HKDF - SHA-256	yes	A192KW	ECDH ES w/ Concat KDF and AES Key Wrap w/ 192-bit key
ECDH-ES + A256KW	-31	HKDF - SHA-256	yes	A256KW	ECDH ES w/ Concat KDF and AES Key Wrap w/ 256-bit key
ECDH-SS + A128KW	-32	HKDF - SHA-256	no	A128KW	ECDH SS w/ Concat KDF and AES Key Wrap w/ 128-bit key
ECDH-SS + A192KW	-33	HKDF - SHA-256	no	A192KW	ECDH SS w/ Concat KDF and AES Key Wrap w/ 192-bit key
ECDH-SS + A256KW	-34	HKDF - SHA-256	no	A256KW	ECDH SS w/ Concat KDF and AES Key Wrap w/ 256-bit key

Table 16: ECDH Algorithm Values with Key Wrap

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

- * The 'kty' field MUST be present, and it MUST be 'EC2' or 'OKP'.
- * If the 'alg' field is 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.

7. Key Object Parameters

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. Private members allow for the archival of keys by individuals. However, there are some circumstances in which private keys may be distributed to 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
OKP	1	Octet Key Pair
EC2	2	Elliptic Curve Keys w/ x- and y-coordinate pair
Symmetric	4	Symmetric Keys
Reserved	0	This value is reserved

Table 17: Key Type Values

7.1. Elliptic Curve Keys

Two different key structures are defined for elliptic curve keys. One version uses both an x-coordinate and a y-coordinate, potentially with point compression ('EC2'). 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 ('OKP').

Applications MUST check that the curve and the key type are consistent and reject a key if they are not.

Name	Value	Key Type	Description
P-256	1	EC2	NIST P-256 also known as secp256r1
P-384	2	EC2	NIST P-384 also known as secp384r1
P-521	3	EC2	NIST P-521 also known as secp521r1
X25519	4	OKP	X25519 for use w/ ECDH only
X448	5	OKP	X448 for use w/ ECDH only
Ed25519	6	OKP	Ed25519 for use w/ EdDSA only
Ed448	7	OKP	Ed448 for use w/ EdDSA only

Table 18: Elliptic Curves

7.1.1. Double Coordinate Curves

The traditional way of sending ECs has been to send either both the x-coordinate and y-coordinate 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 19. The members that are defined for this key type are:

- crv: This 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 18. Other curves may be registered in the future, and private curves can be used as well.
- x: This contains the x-coordinate for the EC point. The integer is converted to a byte string as defined in [SEC1]. Leading zero octets MUST be preserved.
- y: This contains either the sign bit or the value of the y-coordinate for the EC point. When encoding the value y, the integer is converted to a byte 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 [SECl]. 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: This 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.

Key Type	Name	Label	CBOR Type	Description
2	crv	-1	int / tstr	EC identifier - Taken from the "COSE Elliptic Curves" registry
2	x	-2	bstr	x-coordinate
2	y	-3	bstr / bool	y-coordinate
2	d	-4	bstr	Private key

Table 19: EC Key Parameters

7.2. Octet Key Pair

A new key type is defined for Octet Key Pairs (OKP). Do not assume that keys using this type are elliptic curves. This key type could be used for other curve types (for example, mathematics based on hyper-elliptic surfaces).

The key parameters defined in this section are summarized in Table 20. The members that are defined for this key type are:

- crv: This 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 18. Other curves may be registered in the future and private curves can be used as well.
- x: This contains the public key. The byte string contains the public key as defined by the algorithm. (For X25519, internally it is a little-endian integer.)

d: This contains the private key.

For public keys, it is REQUIRED that 'crv' and 'x' 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' also be present, but it can be recomputed from the required elements and omitting it saves on space.

Name	Key Type	Label	Type	Description
crv	1	-1	int / tstr	EC identifier - Taken from the "COSE Elliptic Curves" registry
x	1	-2	bstr	Public Key
d	1	-4	bstr	Private key

Table 20: Octet Key Pair Parameters

7.3. 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 4 ('Symmetric'). The member that is defined for this key type is:

k: This contains the value of the key.

This key structure does not have a form that contains only public members. As it is expected that this key structure is going to be transmitted, care must be taken that it is never transmitted accidentally or insecurely. For symmetric keys, it is REQUIRED that 'k' be present in the structure.

Name	Key Type	Label	Type	Description
k	4	-1	bstr	Key Value

Table 21: Symmetric Key Parameters

8. COSE Capabilities

There are some situations that have been identified where identification of capabilities of an algorithm or a key type need to be specified. One example of this is in [I-D.ietf-core-oscore-groupcomm] where the capabilities of the counter signature algorithm are mixed into the traffic key derivation process. This has a counterpart in the S/MIME specifications where `SMIMECapabilities` is defined in Section 2.5a.2 of [RFC8551]. This document defines the same concept for COSE.

The algorithm identifier is not included in the capabilities data as it should be encoded elsewhere in the message. The key type identifier is included in the capabilities data as it is not expected to be encoded elsewhere.

Two different types of capabilities are defined: capabilities for algorithms and capabilities for key type. Once defined by registration with IANA, the list of capabilities for an algorithm or key type is immutable. If it is later found that a new capability is needed for a key type or an algorithm, it will require that a new code point be assigned to deal with that. As a general rule, the capabilities are going to map to algorithm-specific header parameters or key parameters, but they do not need to do so. An example of this is the HSS-LMS key capabilities defined below where the hash algorithm used is included.

The capability structure is an array of values, the values included in the structure are dependent on a specific algorithm or key type. For algorithm capabilities, the first element should always be a key type value if applicable, but the items that are specific to a key (for example a curve) should not be included in the algorithm capabilities. This means that if one wishes to enumerate all of the capabilities for a device which implements ECDH, it requires that all of the combinations of algorithms and key pairs to be specified. The last example of Section 8.3 provides a case where this is done by allowing for a cross product to be specified between an array of algorithm capabilities and key type capabilities (see ECDH-ES+A25KW element). For a key, the first element should be the key type value. While this means that the key type value will be duplicated if both an algorithm and key capability are used, the key type is needed in order to understand the rest of the values.

8.1. Assignments for Existing Algorithms

For the current set of algorithms in the registry, those in this document as well as those in [RFC8230] and [I-D.ietf-cose-hash-sig], the capabilities list is an array with one element, the key type (from the "COSE Key Types" Registry). It is expected that future registered algorithms could have zero, one, or multiple elements.

8.2. Assignments for Existing Key Types

There are a number of pre-existing key types, the following deals with creating the capability definition for those structures:

- * OKP, EC2: The list of capabilities is:
 - The key type value. (1 for OKP or 2 for EC2.)
 - One curve for that key type from the "COSE Elliptic Curve" registry.
- * RSA: The list of capabilities is:
 - The key type value (3).
- * Symmetric: The list of capabilities is:
 - The key type value (4).
- * HSS-LMS: The list of capabilities is:
 - The key type value (5),
 - Algorithm identifier for the underlying hash function from the "COSE Algorithms" registry.

8.3. Examples

Capabilities can be use in a key derivation process to make sure that both sides are using the same parameters. This is the approach that is being used by the group communication KDF in [I-D.ietf-core-oscore-groupcomm]. The three examples below show different ways that one might include things:

- * Just an algorithm capability: This is useful if the protocol wants to require a specific algorithm such as ECDSA, but it is agnostic about which curve is being used. This does require that the algorithm identifier be specified in the protocol. See the first example.

- * Just a key type capability: This is useful if the protocol wants to require a specific a specific key type and curve, such as P-256, but will accept any algorithm using that curve (e.g. both ECDSA and ECDH). See the second example.
- * Both an algorithm and a key type capability: This is used if the protocol needs to nail down all of the options surrounding an algorithm E.g. EdDSA with the curve X25519. As with the first example, the algorithm identifier needs to be specified in the protocol. See the third example which just concatenates the two capabilities together.

Algorithm ECDSA

0x8102 / [2 \ EC2 \] /

Key type EC2 with P-256 curve:

0x820201 / [2 \ EC2 \, 1 \ P-256 \] /

ECDH-ES + A256KW with an X25519 curve:

0x8101820104 / [1 \ OKP \],[1 \ OKP \, 4 \ X25519 \] /

The capabilities can also be used by an entity to advertise what it is capable of doing. The decoded example below is one of many encoding that could be used for that purpose. Each array element includes three fields: the algorithm identifier, one or more algorithm capabilities, and one or more key type capabilities.

```
[
  [-8 / EdDSA /,
    [1 / OKP key type /],
    [
      [1 / OKP /, 6 / Ed25519 / ],
      [1 /OKP/, 7 /Ed448 /]
    ]
  ],
  [-7 / ECDSA with SHA-256/,
    [2 /EC2 key type/],
    [
      [2 /EC2/, 1 /P-256/],
      [2 /EC2/, 3 /P-521/]
    ]
  ],
  [-31 / ECDH-ES+A256KW/,
    [
      [ 2 /EC2/],
      [1 /OKP/ ]
    ],
    [
      [2 /EC2/, 1 /P-256/],
      [1 /OKP/, 4 / X25519/ ]
    ]
  ],
  [ 1 / A128GCM /,
    [ 4 / Symmetric / ],
    [ 4 / Symmetric /]
  ]
]
```

Examining the above:

- * The first element indicates that the entity supports EdDSA with curves Ed25519 and Ed448.
- * The second element indicates that the entity supports ECDSA with curves P-256 and P-521.
- * The third element indicates that the entity support ephemeral-static ECDH using AES256 key wrap. The entity can support the P-256 curve with an EC2 key type and the X25519 curve with an OKP key type.
- * The last element indicates that the entity supports AES-GCM of 128 bits for content encryption.

The entity does not advertise that it supports any MAC algorithms.

9. CBOR Encoding Restrictions

This document limits the restrictions it imposes on how the CBOR Encoder needs to work. It has been narrowed down to the following restrictions:

- * The restriction applies to the encoding of the COSE_KDF_Context.
- * Encoding MUST be done using definite lengths and the length of the MUST be the minimum possible length. This means that the integer 1 is encoded as "0x01" and not "0x1801".
- * Applications MUST NOT generate messages with the same label used twice as a key in a single map. Applications MUST NOT parse and process messages with the same label used twice as a key in a single map. Applications can enforce the parse and process requirement by using parsers that will fail the parse step or by using parsers that will pass all keys to the application, and the application can perform the check for duplicate keys.

10. IANA Considerations

IANA is requested to update all COSE registries except for "COSE Header Parameters" and "COSE Key Common Parameters" from [RFC8152] to [[This document]].

10.1. Changes to "COSE Key Types" registry.

IANA is requested to create a new column in the "COSE Key Types" registry. The new column is to be labeled "Capabilities". The new column is to be populated according to the entries in Table 22.

Value	Name	Capabilities
1	OKP	[kty(1), crv]
2	EC2	[kty(2), crv]
3	RSA	[kty(3)]
4	Symmetric	[kty(4)]
5	HSS-IMS	[kty(5), hash algorithm]

Table 22: Key Type Capabilities

10.2. Changes to "COSE Algorithms" registry

IANA is requested to create a new column in the "COSE Algorithms" registry. The new column is to be labeled "Capabilities". The new column is populated with "[kty]" for all current, non-provisional, registrations. It is expected that the documents which define those algorithms will be expanded to include this registration. If this is not done then the Designated Expert should be consulted before final registration for this document is done.

IANA is requested to update the reference column in the "COSE Algorithms" registry to include [[This Document]] as a reference for all rows where it is not already present.

IANA is requested to add a new row to the "COSE Algorithms" registry.

Name	Value	Description	Reference	Recommended
IV Generation	IV-GENERATION	For doing IV generation for symmetric algorithms.	[[THIS DOCUMENT]]	No

Table 23

The capabilities column for this registration is to be empty.

10.3. Changes to the "COSE Key Type Parameters" registry

IANA is requested to modify the description to "Public Key" for the line with "Key Type" of 2 and the "Name" of "x". See Table 20 which has been modified with this change.

10.4. Expert Review Instructions

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

- * 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.

- * 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 interoperable way. When specifications are not provided, the description provided needs to have sufficient information to identify what the point is being used for.
- * Experts should take into account the expected usage of fields when approving 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. The length of the encoded value should be weighed against how many code points of that length are left, the size of device it will be used on, and the number of code points left that encode to that size.
- * When algorithms are registered, vanity registrations should be discouraged. One way to do this is to require registrations to provide additional documentation on security analysis of the algorithm. Another thing that should be considered is requesting an opinion on the algorithm from the Crypto Forum Research Group (CFRG). Algorithms that do not meet the security requirements of the community and the messages structures should not be registered.

11. Security Considerations

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

Implementations need to protect the private key material for any individuals. There are some cases in this document that need to be highlighted on this issue.

- * Using the same key for two different algorithms can leak information about the key. It is therefore recommended that keys be restricted to a single algorithm.
- * Use of 'direct' as a recipient algorithm combined with a second recipient algorithm exposes the direct key to the second recipient.
- * Several of the algorithms in this document have limits on the number of times that a key can be used without leaking information about the key.

The use of ECDH and direct plus KDF (with no key wrap) will not directly lead to the private key being leaked; the one way function of the KDF will prevent that. There is, however, a different issue that needs to be addressed. Having two recipients requires that the CEK be shared between two recipients. The second recipient therefore has a CEK that was derived from material that can be used for the weak proof of origin. The second recipient could create a message using the same CEK and send it to the first recipient; the first recipient would, for either static-static ECDH or direct plus KDF, make an assumption that the CEK could be used for proof of origin even though it is from the wrong entity. If the key wrap step is added, then no proof of origin is implied and this is not an issue.

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

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

- * What are the permissions associated with the key owner?
- * Is the cryptographic algorithm acceptable in the current context?

- * Have the restrictions associated with the key, such as algorithm or freshness, been checked and are they correct?
- * Is the request something that is reasonable, given the current state of the application?
- * Have any security considerations that are part of the message been enforced (as specified by the application or 'crit' parameter)?

There are a large number of algorithms presented in this document that use nonce values. For all of the nonces defined in this document, there is some type of restriction on the nonce being a unique value either for a key or for some other conditions. In all of these cases, there is no known requirement on the nonce being both unique and unpredictable; under these circumstances, it's reasonable to use a counter for creation of the nonce. In cases where one wants the pattern of the nonce to be unpredictable as well as unique, one can use a key created for that purpose and encrypt the counter to produce the nonce value.

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

The analysis done in [I-D.ietf-quic-tls] is based on the number of records/packets that are sent. This should map well to the number of messages sent when use COSE so that analysis should hold here as well. It needs to be noted that the limits are based on the number of messages, but QUIC and DTLS are always pair-wise based endpoints, [I-D.ietf-core-oscore-groupcomm] use COSE in a group communication. Under these circumstances it may be that no one single entity will see all of the messages that are encrypted and therefore no single entity can trigger the rekey operation.

12. References

12.1. Normative References

[I-D.ietf-cose-rfc8152bis-struct]
Schaad, J., "CBOR Object Signing and Encryption (COSE): Structures and Process", Work in Progress, Internet-Draft,

draft-ietf-cose-rfc8152bis-struct-13, 4 September 2020,
<<https://tools.ietf.org/html/draft-ietf-cose-rfc8152bis-struct-13>>.

- [RFC2104] Krawczyk, H., Bellare, M., and R. Canetti, "HMAC: Keyed-Hashing for Message Authentication", RFC 2104, DOI 10.17487/RFC2104, February 1997, <<https://www.rfc-editor.org/info/rfc2104>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC3394] Schaad, J. and R. Housley, "Advanced Encryption Standard (AES) Key Wrap Algorithm", RFC 3394, DOI 10.17487/RFC3394, September 2002, <<https://www.rfc-editor.org/info/rfc3394>>.
- [RFC3610] Whiting, D., Housley, R., and N. Ferguson, "Counter with CBC-MAC (CCM)", RFC 3610, DOI 10.17487/RFC3610, September 2003, <<https://www.rfc-editor.org/info/rfc3610>>.
- [RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <<https://www.rfc-editor.org/info/rfc5869>>.
- [RFC6090] McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms", RFC 6090, DOI 10.17487/RFC6090, February 2011, <<https://www.rfc-editor.org/info/rfc6090>>.
- [RFC6979] Pornin, T., "Deterministic Usage of the Digital Signature Algorithm (DSA) and Elliptic Curve Digital Signature Algorithm (ECDSA)", RFC 6979, DOI 10.17487/RFC6979, August 2013, <<https://www.rfc-editor.org/info/rfc6979>>.
- [RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <<https://www.rfc-editor.org/info/rfc7049>>.
- [RFC8439] Nir, Y. and A. Langley, "ChaCha20 and Poly1305 for IETF Protocols", RFC 8439, DOI 10.17487/RFC8439, June 2018, <<https://www.rfc-editor.org/info/rfc8439>>.
- [RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <<https://www.rfc-editor.org/info/rfc7748>>.

- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [AES-GCM] National Institute of Standards and Technology, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", DOI 10.6028/NIST.SP.800-38D, NIST Special Publication 800-38D, November 2007, <<https://csrc.nist.gov/publications/nistpubs/800-38D/SP-800-38D.pdf>>.
- [DSS] National Institute of Standards and Technology, "Digital Signature Standard (DSS)", DOI 10.6028/NIST.FIPS.186-4, FIPS PUB 186-4, July 2013, <<http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>>.
- [MAC] Menees, A., van Oorschot, P., and S. Vanstone, "Handbook of Applied Cryptography", 1996.
- [SEC1] Certicom Research, "SEC 1: Elliptic Curve Cryptography", May 2009, <<http://www.secg.org/sec1-v2.pdf>>.
- [RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <<https://www.rfc-editor.org/info/rfc8032>>.
- [RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <<https://www.rfc-editor.org/info/rfc8017>>.

12.2. Informative References

- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.
- [RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <<https://www.rfc-editor.org/info/rfc8610>>.

- [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, <<https://www.rfc-editor.org/info/rfc4231>>.
- [RFC4493] Song, JH., Poovendran, R., Lee, J., and T. Iwata, "The AES-CMAC Algorithm", RFC 4493, DOI 10.17487/RFC4493, June 2006, <<https://www.rfc-editor.org/info/rfc4493>>.
- [RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <<https://www.rfc-editor.org/info/rfc5116>>.
- [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, <<https://www.rfc-editor.org/info/rfc5480>>.
- [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, <<https://www.rfc-editor.org/info/rfc6151>>.
- [STD90] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, December 2017.
<<https://www.rfc-editor.org/info/std90>>
- [RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <<https://www.rfc-editor.org/info/rfc7252>>.
- [RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <<https://www.rfc-editor.org/info/rfc7518>>.
- [RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <<https://www.rfc-editor.org/info/rfc8152>>.
- [RFC8551] Schaad, J., Ramsdell, B., and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification", RFC 8551, DOI 10.17487/RFC8551, April 2019, <<https://www.rfc-editor.org/info/rfc8551>>.

- [RFC8230] Jones, M., "Using RSA Algorithms with CBOR Object Signing and Encryption (COSE) Messages", RFC 8230, DOI 10.17487/RFC8230, September 2017, <<https://www.rfc-editor.org/info/rfc8230>>.
- [I-D.ietf-core-oscore-groupcomm]
Tiloca, M., Selander, G., Palombini, F., and J. Park,
"Group OSCORE - Secure Group Communication for CoAP", Work in Progress, Internet-Draft, draft-ietf-core-oscore-groupcomm-09, 23 June 2020, <<https://tools.ietf.org/html/draft-ietf-core-oscore-groupcomm-09>>.
- [I-D.ietf-cose-hash-sig]
Housley, R., "Use of the HSS/LMS Hash-based Signature Algorithm with CBOR Object Signing and Encryption (COSE)", Work in Progress, Internet-Draft, draft-ietf-cose-hash-sig-09, 11 December 2019, <<https://tools.ietf.org/html/draft-ietf-cose-hash-sig-09>>.
- [SP800-38d]
Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST Special Publication 800-38D , November 2007, <<https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-38d.pdf>>.
- [SP800-56A]
Barker, E., Chen, L., Roginsky, A., and M. Smid,
"Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", DOI 10.6028/NIST.SP.800-56Ar2, NIST Special Publication 800-56A, Revision 2, May 2013, <<http://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-56Ar2.pdf>>.
- [GitHub-Examples]
"GitHub Examples of COSE", <<https://github.com/cose-wg/Examples>>.
- [I-D.mattsson-cfrg-det-sigs-with-noise]
Mattsson, J., Thormarker, E., and S. Ruohomaa,
"Deterministic ECDSA and EdDSA Signatures with Additional Randomness", Work in Progress, Internet-Draft, draft-mattsson-cfrg-det-sigs-with-noise-02, 11 March 2020, <<https://tools.ietf.org/html/draft-mattsson-cfrg-det-sigs-with-noise-02>>.

- [HKDF] Krawczyk, H., "Cryptographic Extraction and Key Derivation: The HKDF Scheme", 2010, <<https://eprint.iacr.org/2010/264.pdf>>.
- [ROBUST] Fischlin, M., Günther, F., and C. Janson, "Robust Channels: Handling Unreliable Networks in the Record Layers of QUIC and DTLS", February 2020, <https://www.felixguenther.info/docs/QUIP2020_RobustChannels.pdf>.
- [I-D.ietf-quic-tls] Thomson, M. and S. Turner, "Using TLS to Secure QUIC", Work in Progress, Internet-Draft, draft-ietf-quic-tls-30, 9 September 2020, <<https://tools.ietf.org/html/draft-ietf-quic-tls-30>>.

Acknowledgments

This document is a product of the COSE working group of the IETF.

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

The initial version of the specification was based to some degree on the outputs of the JOSE and S/MIME working groups.

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

Author's Address

Jim Schaad
August Cellars

Email: ietf@augustcellars.com

COSE Working Group
Internet-Draft
Obsoletes: 8152 (if approved)
Intended status: Standards Track
Expires: 5 August 2021

J. Schaad
August Cellars
1 February 2021

CBOR Object Signing and Encryption (COSE): Structures and Process
draft-ietf-cose-rfc8152bis-struct-15

Abstract

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

This document along with [I-D.ietf-cose-rfc8152bis-algs] obsoletes RFC8152.

Contributing to this document

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

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 5 August 2021.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	4
1.1. Requirements Terminology	6
1.2. Changes from RFC8152	6
1.3. Design Changes from JOSE	6
1.4. CBOR Grammar	7
1.5. CBOR-Related Terminology	8
1.6. Document Terminology	9
2. Basic COSE Structure	10
3. Header Parameters	13
3.1. Common COSE Header Parameters	15
4. Signing Objects	18
4.1. Signing with One or More Signers	18
4.2. Signing with One Signer	20
4.3. Externally Supplied Data	21
4.4. Signing and Verification Process	22
5. Encryption Objects	24
5.1. Enveloped COSE Structure	24
5.1.1. Content Key Distribution Methods	26
5.2. Single Recipient Encrypted	26
5.3. How to Encrypt and Decrypt for AEAD Algorithms	27
5.4. How to Encrypt and Decrypt for AE Algorithms	29
6. MAC Objects	30
6.1. MACed Message with Recipients	31
6.2. MACed Messages with Implicit Key	32
6.3. How to Compute and Verify a MAC	33
7. Key Objects	34
7.1. COSE Key Common Parameters	35
8. Taxonomy of Algorithms used by COSE	37
8.1. Signature Algorithms	38
8.2. Message Authentication Code (MAC) Algorithms	40

8.3.	Content Encryption Algorithms	40
8.4.	Key Derivation Functions (KDFs)	41
8.5.	Content Key Distribution Methods	41
8.5.1.	Direct Encryption	42
8.5.2.	Key Wrap	42
8.5.3.	Key Transport	43
8.5.4.	Direct Key Agreement	43
8.5.5.	Key Agreement with Key Wrap	44
9.	CBOR Encoding Restrictions	45
10.	Application Profiling Considerations	45
11.	IANA Considerations	46
11.1.	COSE Header Parameters Registry	47
11.2.	COSE Key Common Parameters Registry	47
11.3.	Media Type Registrations	47
11.3.1.	COSE Security Message	47
11.3.2.	COSE Key Media Type	48
11.4.	CoAP Content-Formats Registry	50
11.5.	CBOR Tags Registry	50
11.6.	Expert Review Instructions	51
12.	Security Considerations	52
13.	Implementation Status	53
13.1.	Author's Versions	54
13.2.	JavaScript Version	55
13.3.	Python Version	55
13.4.	COSE Testing Library	55
14.	References	56
14.1.	Normative References	56
14.2.	Informative References	56
Appendix A.	Guidelines for External Data Authentication of Algorithms	60
Appendix B.	Two Layers of Recipient Information	63
Appendix C.	Examples	65
C.1.	Examples of Signed Messages	66
C.1.1.	Single Signature	66
C.1.2.	Multiple Signers	67
C.1.3.	Signature with Criticality	68
C.2.	Single Signer Examples	69
C.2.1.	Single ECDSA Signature	69
C.3.	Examples of Enveloped Messages	70
C.3.1.	Direct ECDH	70
C.3.2.	Direct Plus Key Derivation	71
C.3.3.	Encrypted Content with External Data	72
C.4.	Examples of Encrypted Messages	73
C.4.1.	Simple Encrypted Message	73
C.4.2.	Encrypted Message with a Partial IV	74
C.5.	Examples of MACed Messages	74
C.5.1.	Shared Secret Direct MAC	74
C.5.2.	ECDH Direct MAC	75

C.5.3. Wrapped MAC	76
C.5.4. Multi-Recipient MACed Message	77
C.6. Examples of MAC0 Messages	78
C.6.1. Shared Secret Direct MAC	78
C.7. COSE Keys	79
C.7.1. Public Keys	79
C.7.2. Private Keys	80
Acknowledgments	82
Author's Address	83

1. Introduction

There has been an increased focus on small, constrained devices that make up the Internet of Things (IoT). One of the standards that has come out of this process is "Concise Binary Object Representation (CBOR)" [I-D.ietf-cbor-7049bis]. CBOR extended the data model of the JavaScript Object Notation (JSON) [STD90] by allowing for binary data, among other changes. CBOR has been adopted by several of the IETF working groups dealing with the IoT world as their encoding of data structures. CBOR was designed specifically both to be small in terms of messages transported and implementation size and to be a schema-free decoder. A need exists to provide message security services for IoT, and using CBOR as the message-encoding format makes sense.

The JOSE working group produced a set of documents [RFC7515] [RFC7516] [RFC7517] [RFC7518] that specified how to process encryption, signatures, and Message Authentication Code (MAC) operations and how to encode keys using JSON. This document defines the CBOR Object Signing and Encryption (COSE) standard, which does the same thing for the CBOR encoding format. This document is combined with [I-D.ietf-cose-rfc8152bis-algs] which provides an initial set of algorithms. While there is a strong attempt to keep the flavor of the original JSON Object Signing and Encryption (JOSE) documents, two considerations are taken into account:

- * CBOR has capabilities that are not present in JSON and are appropriate to use. One example of this is the fact that CBOR has a method of encoding binary directly without first converting it into a base64-encoded text string.
- * COSE is not a direct copy of the JOSE specification. In the process of creating COSE, decisions that were made for JOSE were re-examined. In many cases, different results were decided on as the criteria were not always the same.

This document contains:

- * The description of the structure for the CBOR objects which are transmitted over the wire. Two objects are defined for each of encryption, signing and message authentication. One object is defined for transporting keys and one for transporting groups of keys.
- * The procedures used to build the inputs to the cryptographic functions required for each of the structures.
- * A set of attributes that apply to the different security objects.

This document does not contain the rules and procedures for using specific cryptographic algorithms. Details on specific algorithms can be found in [I-D.ietf-cose-rfc8152bis-algs] and [RFC8230]. Details for additional algorithms are expected to be defined in future documents.

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

One feature that is present in CMS that is not present in this standard is a digest structure. This omission is deliberate. It is better for the structure to be defined in each protocol as different protocols will want to include a different set of fields as part of the structure. While an algorithm identifier and the digest value are going to be common to all applications, the two values may not always be adjacent as the algorithm could be defined once with multiple values. Applications may additionally want to define additional data fields as part of the structure. One such application-specific element would be to include a URI or other pointer to where the data that is being hashed can be obtained. [I-D.ietf-cose-hash-algs] contains one such possible structure along with defining a set of digest algorithms.

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

1.1. Requirements Terminology

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

1.2. Changes from RFC8152

- * Split the original document into this document and [I-D.ietf-cose-rfc8152bis-algs].
- * Add some text describing why there is no digest structure defined by COSE.
- * Text clarifications and changes in terminology.
- * All of the details relating to countersignatures have been removed and placed in [I-D.ietf-cose-countersign].

1.3. Design Changes from JOSE

- * Define a single overall message structure so that encrypted, signed, and MACed messages can easily be identified and still have a consistent view.
- * Signed messages distinguish between the protected and unprotected header parameters that relate to the content and those that relate to the signature.
- * MACed messages are separated from signed messages.
- * MACed messages have the ability to use the same set of recipient algorithms as enveloped messages for obtaining the MAC authentication key.

- * Use binary encodings, rather than base64url encodings, to encode binary data.
- * Combine the authentication tag for encryption algorithms with the ciphertext.
- * The set of cryptographic algorithms has been expanded in some directions and trimmed in others.

1.4. CBOR Grammar

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

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

any -- non-specific value that permits all CBOR values to be placed here.

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

bstr -- byte string (major type 2).

int -- an unsigned integer or a negative integer.

nil -- a null value (major type 7, value 22).

nint -- a negative integer (major type 1).

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

uint -- an unsigned integer (major type 0).

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

FOO / BAR -- indicates that either FOO or BAR can appear here.

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

* FOO -- indicates that the type FOO appears zero or more times.

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

type1 .cbor type2 -- indicates that the contents of type1, usually bstr, contains a value of type2.

type1 .size integer -- indicates that the contents of type1 is integer bytes long

As well as the prose description, a version of a CBOR grammar is presented in CDDL. The CDDL grammar is informational; the prose description is normative.

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

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

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

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

The non-terminal Internal_Types is defined for dealing with the automated validation tools used during the writing of this document. It references those non-terminals that are used for security computations but are not emitted for transport.

1.5. CBOR-Related Terminology

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

The presence a label that is neither a text string or an integer, in a CBOR map, is an error. Applications can either fail processing or process messages by ignoring incorrect labels; however, they MUST NOT create messages with incorrect labels.

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

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

1.6. Document Terminology

In this document, we use the following terminology:

Byte is a synonym for octet.

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

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

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

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

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

2. Basic COSE Structure

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

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

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

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

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

1. The specific message type is known from the context. This may be defined by a marker in the containing structure or by restrictions specified by the application protocol.
2. The message type is identified by a CBOR tag. Messages with a CBOR tag are known in this specification as tagged messages, while those without the CBOR tag are known as untagged messages. This document defines a CBOR tag for each of the message structures. These tags can be found in Table 1.
3. When a COSE object is carried in a media type of 'application/cose', the optional parameter 'cose-type' can be used to identify the embedded object. The parameter is OPTIONAL if the tagged version of the structure is used. The parameter is REQUIRED if

the untagged version of the structure is used. The value to use with the parameter for each of the structures can be found in Table 1.

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

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

Table 1: COSE Message Identification

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

Table 2: CoAP Content-Formats for COSE

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

```
COSE_Messages = COSE_Untagged_Message / COSE_Tagged_Message
```

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

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

3. Header Parameters

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

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

The two buckets are:

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

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

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

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

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

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

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

```
Headers = (  
    protected : empty_or_serialized_map,  
    unprotected : header_map  
)  
  
header_map = {  
    Generic_Headers,  
    * label => values  
}  
  
empty_or_serialized_map = bstr .cbor header_map / bstr .size 0
```


3.1. Common COSE Header Parameters

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

The set of header parameters defined in this section are:

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

crit: This header parameter is used to indicate which protected header parameters an application that is processing a message is required to understand. Header parameters defined in this document do not need to be included as they should be understood by all implementations. When present, this the 'crit' header parameter **MUST** be placed in the protected header parameter bucket. The array **MUST** have at least one value in it.

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

- * Integer labels in the range of 0 to 7 **SHOULD** be omitted.
- * Integer labels in the range -1 to -128 can be omitted. Algorithms can assign labels in this range where the ability to process the content of the label is considered to be core to implementing the algorithm. Algorithms can assign labels outside of this range where the ability to process the content of the label is not considered to be core, but needs to be understood to correctly process this instance. Integer labels in the range -129 to -65536 **SHOULD** be included as these would be less common header parameters that might not be generally supported.
- * Labels for header parameters required for an application **MAY** be omitted. Applications should have a statement if the label can be omitted.

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

content type: This header parameter is used to indicate the content type of the data in the payload or ciphertext fields. Integers are from the "CoAP Content-Formats" IANA registry table [COAP.Formats]. Text values following the syntax of "<type-name>/<subtype-name>" where <type-name> and <subtype-name> are defined in Section 4.2 of [RFC6838]. Leading and trailing whitespace is also omitted. Textual content values along with parameters and subparameters can be located using the IANA "Media Types" registry. Applications SHOULD provide this header parameter if the content structure is potentially ambiguous.

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

IV: This header parameter holds the Initialization Vector (IV) value. For some symmetric encryption algorithms, this may be referred to as a nonce. The IV can be placed in the unprotected bucket as modifying the IV will cause the decryption to yield plaintext that is readily detectable as garbled.

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

The message IV is generated by the following steps:

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

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

Table 3: Common Header Parameters

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

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

4. Signing Objects

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

4.1. Signing with One or More Signers

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

RFC 5652 indicates that:

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

For example, the COSE_Sign structure might include signatures generated with the Edwards-curve Digital Signature Algorithm (EdDSA) [RFC8032] and with the Elliptic Curve Digital Signature Algorithm (ECDSA) [DSS]. This allows recipients to verify the signature

associated with one algorithm or the other. More-detailed information on multiple signature evaluations can be found in [RFC5752].

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

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

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

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

protected: This is as described in Section 3.

unprotected: This is as described in Section 3.

payload: This field contains the serialized content to be signed.

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

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

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

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

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

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

protected: This is as described in Section 3.

unprotected: This is as described in Section 3.

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

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

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

4.2. Signing with One Signer

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

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

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

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

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

protected: This is as described in Section 3.

unprotected: This is as described in Section 3.

payload: This is as described in Section 4.1.

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

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

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

4.3. Externally Supplied Data

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

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

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

- * If multiple items are included, an order for the items needs to be defined. Using options from CoAP as an example, an application could state that the fields are to be ordered by the option number.
- * Applications need to ensure that the byte string is going to be the same on both sides. Using options from CoAP might give a problem if the same relative numbering is kept. An intermediate node could insert or remove an option, changing how the relative number is done. An application would need to specify that the relative number must be re-encoded to be relative only to the options that are in the external data.

4.4. Signing and Verification Process

In order to create a signature, a well-defined byte string is needed. The Sig_structure is used to create the canonical form. This signing and verification process takes in the body information (COSE_Sign or COSE_Sign1), the signer information (COSE_Signature), and the application data (external source). A Sig_structure is a CBOR array. The fields of the Sig_structure in order are:

1. A context text string identifying the context of the signature. The context text string is:
 - "Signature" for signatures using the COSE_Signature structure.
 - "Signature1" for signatures using the COSE_Sign1 structure.
2. The protected attributes from the body structure encoded in a bstr type. If there are no protected attributes, a zero-length byte string is used.
3. The protected attributes from the signer structure encoded in a bstr type. If there are no protected attributes, a zero-length byte string is used. This field is omitted for the COSE_Sign1 signature structure.
4. The externally supplied data from the application encoded in a bstr type. If this field is not supplied, it defaults to a zero-length byte string. (See Section 4.3 for application guidance on constructing this field.)
5. The payload to be signed encoded in a bstr type. The payload is placed here independent of how it is transported.

The CDDL fragment that describes the above text is:


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

How to compute a signature:

1. Create a Sig_structure and populate it with the appropriate fields.
2. Create the value ToBeSigned by encoding the Sig_structure to a byte string, using the encoding described in Section 9.
3. Call the signature creation algorithm passing in K (the key to sign with), alg (the algorithm to sign with), and ToBeSigned (the value to sign).
4. Place the resulting signature value in the correct location. This is the 'signature' field of the COSE_Signature or COSE_Sign1 structure.

The steps for verifying a signature are:

1. Create a Sig_structure and populate it with the appropriate fields.
2. Create the value ToBeSigned by encoding the Sig_structure to a byte string, using the encoding described in Section 9.
3. Call the signature verification algorithm passing in K (the key to verify with), alg (the algorithm used sign with), ToBeSigned (the value to sign), and sig (the signature to be verified).

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

5. Encryption Objects

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

5.1. Enveloped COSE Structure

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

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

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

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

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

protected: This is as described in Section 3.

unprotected: This is as described in Section 3.

ciphertext: This field contains the ciphertext encoded as a bstr.

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

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

The CDDL fragment that corresponds to the above text is:

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

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

protected: This is as described in Section 3.

unprotected: This is as described in Section 3.

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

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

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

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

5.1.1. Content Key Distribution Methods

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

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

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

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

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

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

5.2. Single Recipient Encrypted

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

Examples of encrypted messages can be found in Appendix C.3.

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

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

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

protected: This is as described in Section 3.

unprotected: This is as described in Section 3.

ciphertext: This is as described in Section 5.1.

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

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

5.3. How to Encrypt and Decrypt for AEAD Algorithms

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

1. A context text string identifying the context of the authenticated data structure. The context text string is:
 - "Encrypt0" for the content encryption of a COSE_Encrypt0 data structure.
 - "Encrypt" for the first layer of a COSE_Encrypt data structure (i.e., for content encryption).
 - "Enc_Recipient" for a recipient encoding to be placed in an COSE_Encrypt data structure.
 - "Mac_Recipient" for a recipient encoding to be placed in a MACed message structure.
 - "Rec_Recipient" for a recipient encoding to be placed in a recipient structure.
2. The protected attributes from the body structure encoded in a bstr type. If there are no protected attributes, a zero-length byte string is used.
3. The externally supplied data from the application encoded in a bstr type. If this field is not supplied, it defaults to a zero-length byte string. (See Section 4.3 for application guidance on constructing this field.)

The CDDL fragment that describes the above text is:

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

How to encrypt a message:

1. Create an Enc_structure and populate it with the appropriate fields.
2. Encode the Enc_structure to a byte string (Additional Authenticated Data (AAD)), using the encoding described in Section 9.
3. Determine the encryption key (K). This step is dependent on the class of recipient algorithm being used. For:

No Recipients: The key to be used is determined by the algorithm and key at the current layer. Examples are key transport keys (Section 8.5.3), key wrap keys (Section 8.5.2), or pre-shared secrets.

Direct Encryption and Direct Key Agreement: The key is determined by the key and algorithm in the recipient structure. The encryption algorithm and size of the key to be used are inputs into the KDF used for the recipient. (For direct, the KDF can be thought of as the identity operation.) Examples of these algorithms are found in Sections 6.1.2 and 6.3 of [I-D.ietf-cose-rfc8152bis-algs].

Other: The key is randomly or pseudo-randomly generated.

4. Call the encryption algorithm with K (the encryption key), P (the plaintext), and AAD. Place the returned ciphertext into the 'ciphertext' field of the structure.
5. For recipients of the message, recursively perform the encryption algorithm for that recipient, using K (the encryption key) as the plaintext.

How to decrypt a message:

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

2. Encode the Enc_structure to a byte string (AAD), using the encoding described in Section 9.
3. Determine the decryption key. This step is dependent on the class of recipient algorithm being used. For:

No Recipients: The key to be used is determined by the algorithm and key at the current layer. Examples are key transport keys (Section 8.5.3), key wrap keys (Section 8.5.2), or pre-shared secrets.

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

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

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

5.4. How to Encrypt and Decrypt for AE Algorithms

How to encrypt a message:

1. Verify that the 'protected' field is empty.
2. Verify that there was no external additional authenticated data supplied for this operation.
3. Determine the encryption key. This step is dependent on the class of recipient algorithm being used. For:

No Recipients: The key to be used is determined by the algorithm and key at the current layer. Examples are key transport keys (Section 8.5.3), key wrap keys (Section 8.5.2), or pre-shared secrets.

Direct Encryption and Direct Key Agreement: The key is determined by the key and algorithm in the recipient structure. The encryption algorithm and size of the key to be used are inputs into the KDF used for the recipient. (For direct, the KDF can be thought of as the identity operation.) Examples of these algorithms are found in Sections 6.1.2 and 6.3 of [I-D.ietf-cose-rfc8152bis-algs].

Other: The key is randomly generated.

4. Call the encryption algorithm with K (the encryption key to use) and P (the plaintext). Place the returned ciphertext into the 'ciphertext' field of the structure.
5. For recipients of the message, recursively perform the encryption algorithm for that recipient, using K (the encryption key) as the plaintext.

How to decrypt a message:

1. Verify that the 'protected' field is empty.
2. Verify that there was no external additional authenticated data supplied for this operation.
3. Determine the decryption key. This step is dependent on the class of recipient algorithm being used. For:

No Recipients: The key to be used is determined by the algorithm and key at the current layer. Examples are key transport keys (Section 8.5.3), key wrap keys (Section 8.5.2), or pre-shared secrets.

Direct Encryption and Direct Key Agreement: The key is determined by the key and algorithm in the recipient structure. The encryption algorithm and size of the key to be used are inputs into the KDF used for the recipient. (For direct, the KDF can be thought of as the identity operation.) Examples of these algorithms are found in Sections 6.1.2 and 6.3 of [I-D.ietf-cose-rfc8152bis-algs].

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

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

COSE supports two different MAC structures. COSE_MAC0 is used when a recipient structure is not needed because the key to be used is implicitly known. COSE_MAC is used for all other cases. These include a requirement for multiple recipients, the key being unknown, or a recipient algorithm of other than direct.

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

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

6.1. MACed Message with Recipients

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

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

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

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

protected: This is as described in Section 3.

unprotected: This is as described in Section 3.

payload: This field contains the serialized content to be MACed. If

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

tag: This field contains the MAC value.

recipients: This is as described in Section 5.1.

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

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

6.2. MACed Messages with Implicit Key

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

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

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

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

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

protected: This is as described in Section 3.

unprotected: This is as described in Section 3.

payload: This is as described in Section 6.1.

tag: This field contains the MAC value.

The CDDL fragment that corresponds to the above text is:

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

6.3. How to Compute and Verify a MAC

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

1. A context text string that identifies the structure that is being encoded. This context text string is "MAC" for the COSE_Mac structure. This context text string is "MAC0" for the COSE_Mac0 structure.
2. The protected attributes from the COSE_MAC structure. If there are no protected attributes, a zero-length bstr is used.
3. The externally supplied data from the application encoded as a bstr type. If this field is not supplied, it defaults to a zero-length byte string. (See Section 4.3 for application guidance on constructing this field.)
4. The payload to be MACed encoded in a bstr type. The payload is placed here independent of how it is transported.

The CDDL fragment that corresponds to the above text is:

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

The steps to compute a MAC are:

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

2. Create the value ToBeMaced by encoding the MAC_structure to a byte string, using the encoding described in Section 9.
3. Call the MAC creation algorithm passing in K (the key to use), alg (the algorithm to MAC with), and ToBeMaced (the value to compute the MAC on).
4. Place the resulting MAC in the 'tag' field of the COSE_Mac or COSE_Mac0 structure.
5. For COSE_Mac structures, encrypt and encode the MAC key for each recipient of the message.

The steps to verify a MAC are:

1. Create a MAC_structure and populate it with the appropriate fields.
2. Create the value ToBeMaced by encoding the MAC_structure to a byte string, using the encoding described in Section 9.
3. For COSE_Mac structures, obtain the cryptographic key from one of the recipients of the message.
4. Call the MAC creation algorithm passing in K (the key to use), alg (the algorithm to MAC with), and ToBeMaced (the value to compute the MAC on).
5. Compare the MAC value to the 'tag' field of the COSE_Mac or COSE_Mac0 structure.

7. Key Objects

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

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

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

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

The CDDL grammar describing COSE_Key and COSE_KeySet is:

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

```
COSE_KeySet = [+COSE_Key]
```

7.1. COSE Key Common Parameters

This document defines a set of common parameters for a COSE Key object. Table 4 provides a summary of the parameters defined in this section. There are also parameters that are defined for specific key types. Key-type-specific parameters can be found in [I-D.ietf-cose-rfc8152bis-algs].

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

Table 4: Key Map Labels

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

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

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

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

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

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

If different keys are derived for each sender, starting at the same Base IV is likely to satisfy this condition. If the same key is used for multiple senders, then the application needs to provide for a method of dividing the IV space up between the

senders. This could be done by providing a different base point to start from or a different Partial IV to start with and restricting the number of messages to be sent before rekeying.

Name	Value	Description
sign	1	The key is used to create signatures. Requires private key fields.
verify	2	The key is used for verification of signatures.
encrypt	3	The key is used for key transport encryption.
decrypt	4	The key is used for key transport decryption. Requires private key fields.
wrap key	5	The key is used for key wrap encryption.
unwrap key	6	The key is used for key wrap decryption. Requires private key fields.
derive key	7	The key is used for deriving keys. Requires private key fields.
derive bits	8	The key is used for deriving bits not to be used as a key. Requires private key fields.
MAC create	9	The key is used for creating MACs.
MAC verify	10	The key is used for validating MACs.

Table 5: Key Operation Values

8. Taxonomy of Algorithms used by COSE

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

8.1. Signature Algorithms

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

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

The signature functions for this scheme are:

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

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

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

The signature functions for this scheme are:

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

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

No message recovery signature algorithms have been formally defined for COSE yet, and given the new constraints arising from this schemes, while some of these issues have already been identified

there is a high probability that additional issues will arise when integrating message recovery signature algorithms. The first algorithm defined is going to need to make decisions about these issues and those decisions are likely to be binding on any further algorithms defined.

We use the following terms below:

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

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

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

Some of the issues that have already been identified are:

- * The to-be-signed bytes are not the same as the message content bytes. This is because we build a larger to-be-signed message during the signature processing. The recovered bytes length may exceed the message content length, but not the length of the to-be-signed bytes. This may lead to privacy considerations if, for example, the externally supplied data contains confidential information.
- * There may be difficulties in determining where the recovered bytes match up with the to-be-signed bytes, because the recovered bytes contains data not in the message content bytes. One possible option would be to create a padding scheme to prevent that.
- * Not all message recovery signature algorithms take the recovered bytes from the end of the to-be-signed bytes. This is a problem because the message content bytes are at the end of the to-be-signed bytes. If the bytes to be recovered are taken from the start of the to-be-signed bytes then, by default, none of the message content bytes may be included in the recovered bytes. One possible option to deal with this is to reverse the to-be-signed data in the event that recovered bytes are taken from the start rather than end of the to-be-signed bytes.

Signature algorithms are used with the COSE_Signature and COSE_Sign1 structures. At the time of this writing, only signatures with 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.

8.2. Message Authentication Code (MAC) Algorithms

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

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

The MAC functions are:

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

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

MAC algorithms can be based on either a block cipher algorithm (i.e., AES-MAC) or a hash algorithm (i.e., a Hash-based Message Authentication Code (HMAC)). [I-D.ietf-cose-rfc8152bis-algs] defines a MAC algorithm using each of these constructions.

MAC algorithms are used in the COSE_Mac and COSE_Mac0 structures.

8.3. Content Encryption Algorithms

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

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

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

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

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

These algorithms are used in COSE_Encrypt and COSE_Encrypt0.

8.4. Key Derivation Functions (KDFs)

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

- * Secrets that are uniformly random: This is the type of secret that is created by a good random number generator.
- * Secrets that are not uniformly random: This is type of secret that is created by operations like key agreement.
- * Secrets that are not random: This is the type of secret that people generate for things like passwords.

General KDFs work well with the first type of secret, can do reasonably well with the second type of secret, and generally do poorly with the last type of secret. Functions like Argon2 [I-D.irtf-cfrg-argon2] need to be used for non-random secrets.

The same KDF can be set up to deal with the first two types of secrets in a different way. The KDF defined in section 5.1 of [I-D.ietf-cose-rfc8152bis-algs] is such a function. This is reflected in the set of algorithms defined around the HMAC-based Extract-and-Expand Key Derivation Function (HKDF).

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

8.5. Content Key Distribution Methods

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

8.5.1. Direct Encryption

The direct encryption class algorithms share a secret between the sender and the recipient that is used either directly or after manipulation as the CEK. When direct encryption mode is used, it MUST be the only mode used on the message.

The COSE_Recipient structure for the recipient is organized as follows:

- * The 'protected' field MUST be a zero-length byte string unless it is used in the computation of the content key.
- * The 'alg' header parameter MUST be present.
- * A header parameter identifying the shared secret SHOULD be present.
- * The 'ciphertext' field MUST be a zero-length byte string.
- * The 'recipients' field MUST be absent.

8.5.2. Key Wrap

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

The COSE_Recipient structure for the recipient is organized as follows:

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

- * The plaintext to be encrypted is the key from next layer down (usually the content layer).
- * At a minimum, the 'unprotected' field MUST contain the 'alg' header parameter and SHOULD contain a header parameter identifying the shared secret.

8.5.3. Key Transport

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

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

- * The 'protected' field MUST be absent.
- * The plaintext to be encrypted is the key from the next layer down (usually the content layer).
- * At a minimum, the 'unprotected' field MUST contain the 'alg' header parameter and SHOULD contain a parameter identifying the asymmetric key.

8.5.4. Direct Key Agreement

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

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

Two variants of DH that are supported are:

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

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

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

The COSE_Recipient structure for the recipient is organized as follows:

- * At a minimum, headers MUST contain the 'alg' header parameter and SHOULD contain a header parameter identifying the recipient's asymmetric key.
- * The headers SHOULD identify the sender's key for the static-static versions and MUST contain the sender's ephemeral key for the ephemeral-static versions.

8.5.5. Key Agreement with Key Wrap

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

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

The COSE_Recipient structure for the recipient is organized as follows:

- * The 'protected' field is fed into the KDF context structure.
- * The plaintext to be encrypted is the key from the next layer down (usually the content layer).
- * The 'alg' header parameter MUST be present in the layer.

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

9. CBOR Encoding Restrictions

This document limits the restrictions it imposes on how the CBOR Encoder needs to work. It has been narrowed down to the following restrictions:

- * The restriction applies to the encoding of the Sig_structure, the Enc_structure, and the MAC_structure.
- * Encoding MUST be done using definite lengths and the value's length MUST be the minimum possible length. This means that the integer 1 is encoded as "0x01" and not "0x1801".
- * Applications MUST NOT generate messages with the same label used twice as a key in a single map. Applications MUST NOT parse and process messages with the same label used twice as a key in a single map. Applications can enforce the parse and process requirement by using parsers that will fail the parse step or by using parsers that will pass all keys to the application, and the application can perform the check for duplicate keys.

10. Application Profiling Considerations

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

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

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

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

- * Applications may define new header parameters for a specific purpose. Applications will often times select specific header parameters to use or not to use. For example, an application would normally state a preference for using either the IV or the Partial IV header parameter. If the Partial IV header parameter is specified, then the application also needs to define how the fixed portion of the IV is determined.
- * When applications use externally defined authenticated data, they need to define how that data is encoded. This document assumes that the data will be provided as a byte string. More information can be found in Section 4.3.
- * Applications need to determine the set of security algorithms that are to be used. When selecting the algorithms to be used as the mandatory-to-implement set, consideration should be given to choosing different types of algorithms when two are chosen for a specific purpose. An example of this would be choosing HMAC-SHA512 and AES-CMAC as different MAC algorithms; the construction is vastly different between these two algorithms. This means that a weakening of one algorithm would be unlikely to lead to a weakening of the other algorithms. Of course, these algorithms do not provide the same level of security and thus may not be comparable for the desired security functionality. Additional guidance can be found in [BCP201].
- * Applications may need to provide some type of negotiation or discovery method if multiple algorithms or message structures are permitted. The method can be as simple as requiring pre-configuration of the set of algorithms to providing a discovery method built into the protocol. S/MIME provided a number of different ways to approach the problem that applications could follow:
 - Advertising in the message (S/MIME capabilities) [RFC5751].
 - Advertising in the certificate (capabilities extension) [RFC4262].
 - Minimum requirements for the S/MIME, which have been updated over time [RFC2633] [RFC5751] (note that [RFC2633] has been obsoleted by [RFC5751]).

11. IANA Considerations

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

Note that while [I-D.ietf-cose-rfc8152bis-algs] also updates the registries and registrations originally established by [RFC8152], the requested updates are mutually exclusive. The updates requested in this document do not conflict or overlap with the updates requested in [I-D.ietf-cose-rfc8152bis-algs], and vice versa.

11.1. COSE Header Parameters Registry

IANA created a registry titled "COSE Header Parameters" as part of processing [RFC8152]. IANA is requested to update the reference for this registry from [RFC8152] to this document. IANA is also requested to update the reference for all entries, except "counter signature" and "CounterSignature0", in the table from [RFC8152] to this document. The reference for "counter signature" and "CounterSignature0" are to be left as-is.

11.2. COSE Key Common Parameters Registry

IANA created a registry titled "COSE Key Common Parameters" as part of the processing of [RFC8152]. IANA is requested to update the reference for this registry from [RFC8152] to this document. IANA is also requested to update the reference for entries in the table from [RFC8152] to this document.

11.3. Media Type Registrations

11.3.1. COSE Security Message

This section registers the 'application/cose' media type in the "Media Types" registry. These media types are used to indicate that the content is a COSE message.

Type name: application

Subtype name: cose

Required parameters: N/A

Optional parameters: cose-type

Encoding considerations: binary

Security considerations: See the Security Considerations section of [[This Document]].

Interoperability considerations: N/A

Published specification: [[this document]]

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

Fragment identifier considerations: N/A

Additional information:

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

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

Intended usage: COMMON

Restrictions on usage: N/A

Author: Jim Schaad, ietf@augustcellars.com

Change Controller: IESG

Provisional registration? No

11.3.2. COSE Key Media Type

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

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

Type name: application

Subtype name: cose-key

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: See the Security Considerations section of [[This Document]].

Interoperability considerations: N/A

Published specification: [[this document]]

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

Fragment identifier considerations: N/A

Additional information:

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

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

Intended usage: COMMON

Restrictions on usage: N/A

Author: Jim Schaad, ietf@augustcellars.com

Change Controller: IESG

Provisional registration? No

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

Type name: application

Subtype name: cose-key-set

Required parameters: N/A

Optional parameters: N/A

Encoding considerations: binary

Security considerations: See the Security Considerations section of [[This Document]].

Interoperability considerations: N/A

Published specification: [[this document]]

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

Fragment identifier considerations: N/A

Additional information:

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

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

Intended usage: COMMON

Restrictions on usage: N/A

Author: Jim Schaad, ietf@augustcellars.com

Change Controller: IESG

Provisional registration? No

11.4. CoAP Content-Formats Registry

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

11.5. CBOR Tags Registry

IANA is requested to update the references from [RFC8152] to [[This Document]].

11.6. Expert Review Instructions

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

- * 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.
- * 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 interoperable way. When specifications are not provided, the description provided needs to have sufficient information to identify what the point is being used for.
- * Experts should take into account the expected usage of fields when approving 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. The length of the encoded value should be weighed against how many code points of that length are left, the size of device it will be used on, and the number of code points left that encode to that size.
- * When algorithms are registered, vanity registrations should be discouraged. One way to do this is to require registrations to provide additional documentation on security analysis of the algorithm. Another thing that should be considered is requesting an opinion on the algorithm from the Crypto Forum Research Group (CFRG). Algorithms that do not meet the security requirements of the community and the messages structures should not be registered.

12. Security Considerations

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

Implementations need to protect the private key material for any individuals. There are some cases that need to be highlighted on this issue.

- * Using the same key for two different algorithms can leak information about the key. It is therefore recommended that keys be restricted to a single algorithm.
- * Use of 'direct' as a recipient algorithm combined with a second recipient algorithm exposes the direct key to the second recipient.
- * Several of the algorithms in [I-D.ietf-cose-rfc8152bis-algs] have limits on the number of times that a key can be used without leaking information about the key.

The use of ECDH and direct plus KDF (with no key wrap) will not directly lead to the private key being leaked; the one way function of the KDF will prevent that. There is, however, a different issue that needs to be addressed. Having two recipients requires that the CEK be shared between two recipients. The second recipient therefore has a CEK that was derived from material that can be used for the weak proof of origin. The second recipient could create a message using the same CEK and send it to the first recipient; the first recipient would, for either static-static ECDH or direct plus KDF, make an assumption that the CEK could be used for proof of origin even though it is from the wrong entity. If the key wrap step is added, then no proof of origin is implied and this is not an issue.

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

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

- * What are the permissions associated with the key owner?
- * Is the cryptographic algorithm acceptable in the current context?
- * Have the restrictions associated with the key, such as algorithm or freshness, been checked and are they correct?
- * Is the request something that is reasonable, given the current state of the application?
- * Have any security considerations that are part of the message been enforced (as specified by the application or 'crit' header parameter)?

One area that has been getting exposure is traffic analysis of encrypted messages based on the length of the message. This specification does not provide for a uniform method of providing padding as part of the message structure. An observer can distinguish between two different messages (for example, 'YES' and 'NO') based on the length for all of the content encryption algorithms that are defined in [I-D.ietf-cose-rfc8152bis-algs] document. This means that it is up to the applications to document how content padding is to be done in order to prevent or discourage such analysis. (For example, the text strings could be defined as 'YES' and 'NO '.)

13. Implementation Status

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

This section records the status of known implementations of the protocol defined by this specification at the time of posting of this Internet-Draft, and is based on a proposal described in [RFC7942]. The description of implementations in this section is intended to assist the IETF in its decision processes in progressing drafts to RFCs. Please note that the listing of any individual implementation here does not imply endorsement by the IETF. Furthermore, no effort has been spent to verify the information presented here that was supplied by IETF contributors. This is not intended as, and must not be construed to be, a catalog of available implementations or their features. Readers are advised to note that other implementations may exist.

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

13.1. Author's Versions

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

- * Implementation Location: <https://github.com/cose-wg>
- * Primary Maintainer: Jim Schaad
- * Languages: There are three different languages that are currently supported: Java, C# and C.
- * Cryptography: The Java and C# libraries use Bouncy Castle to provide the required cryptography. The C version uses OpenSSL Version 1.1 for the cryptography.
- * Coverage: All versions have support to allow for implicit algorithm support as they allow for the application to set attributes that are not to be sent in the message.
- * Testing: All of the examples in the example library are generated by the C# library and then validated using the Java and C libraries. All three libraries have tests to allow for the creating of the same messages that are in the example library followed by validating them. These are not compared against the example library. The Java and C# libraries have unit testing

included. Not all of the MUST statements in the document have been implemented as part of the libraries. One such statement is the requirement that unique labels be present.

- * Licensing: Revised BSD License

13.2. JavaScript Version

- * Implementation Location: <https://github.com/erdtman/cose-js>
- * Primary Maintainer: Samuel Erdtman
- * Languages: JavaScript
- * Cryptography: TBD
- * Coverage: Full Encrypt, Signature and MAC objects are supported.
- * Testing: Basic testing against the common example library.
- * Licensing: Apache License 2.0

13.3. Python Version

- * Implementation Location: <https://github.com/TimothyClaeys/COSE-PYTHON>
- * Primary Maintainer: Timothy Claeys
- * Languages: Python
- * Cryptography: pyecdsak, crypto python libraries
- * Coverage: TBD
- * Testing: Basic testing plus running against the common example library.
- * Licensing: BSD 3-Clause License

13.4. COSE Testing Library

- * Implementation Location: <https://github.com/cose-wg/Examples>
- * Primary Maintainer: Jim Schaad

- * **Description:** A set of tests for the COSE library is provided as part of the implementation effort. Both success and fail tests have been provided. All of the examples in this document are part of this example set.
- * **Coverage:** An attempt has been made to have test cases for every message type and algorithm in the document. Currently examples dealing with ECDH with Goldilocks are missing.
- * **Licensing:** Public Domain

14. References

14.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [I-D.ietf-cbor-7049bis] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", Work in Progress, Internet-Draft, draft-ietf-cbor-7049bis-16, 30 September 2020, <<https://tools.ietf.org/html/draft-ietf-cbor-7049bis-16>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [I-D.ietf-cose-rfc8152bis-algs] Schaad, J., "CBOR Object Signing and Encryption (COSE): Initial Algorithms", Work in Progress, Internet-Draft, draft-ietf-cose-rfc8152bis-algs-12, 24 September 2020, <<https://tools.ietf.org/html/draft-ietf-cose-rfc8152bis-algs-12>>.

14.2. Informative References

- [RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <<https://www.rfc-editor.org/info/rfc8152>>.
- [RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <<https://www.rfc-editor.org/info/rfc8610>>.

- [RFC2633] Ramsdell, B., Ed., "S/MIME Version 3 Message Specification", RFC 2633, DOI 10.17487/RFC2633, June 1999, <<https://www.rfc-editor.org/info/rfc2633>>.
- [RFC4262] Santesson, S., "X.509 Certificate Extension for Secure/Multipurpose Internet Mail Extensions (S/MIME) Capabilities", RFC 4262, DOI 10.17487/RFC4262, December 2005, <<https://www.rfc-editor.org/info/rfc4262>>.
- [RFC4949] Shirey, R., "Internet Security Glossary, Version 2", FYI 36, RFC 4949, DOI 10.17487/RFC4949, August 2007, <<https://www.rfc-editor.org/info/rfc4949>>.
- [RFC5116] McGrew, D., "An Interface and Algorithms for Authenticated Encryption", RFC 5116, DOI 10.17487/RFC5116, January 2008, <<https://www.rfc-editor.org/info/rfc5116>>.
- [RFC5652] Housley, R., "Cryptographic Message Syntax (CMS)", STD 70, RFC 5652, DOI 10.17487/RFC5652, September 2009, <<https://www.rfc-editor.org/info/rfc5652>>.
- [RFC5751] Ramsdell, B. and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification", RFC 5751, DOI 10.17487/RFC5751, January 2010, <<https://www.rfc-editor.org/info/rfc5751>>.
- [RFC5752] Turner, S. and J. Schaad, "Multiple Signatures in Cryptographic Message Syntax (CMS)", RFC 5752, DOI 10.17487/RFC5752, January 2010, <<https://www.rfc-editor.org/info/rfc5752>>.
- [RFC5990] Randall, J., Kaliski, B., Brainard, J., and S. Turner, "Use of the RSA-KEM Key Transport Algorithm in the Cryptographic Message Syntax (CMS)", RFC 5990, DOI 10.17487/RFC5990, September 2010, <<https://www.rfc-editor.org/info/rfc5990>>.
- [RFC6838] Freed, N., Klensin, J., and T. Hansen, "Media Type Specifications and Registration Procedures", BCP 13, RFC 6838, DOI 10.17487/RFC6838, January 2013, <<https://www.rfc-editor.org/info/rfc6838>>.
- [STD90] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, December 2017.

<<https://www.rfc-editor.org/info/std90>>

- [BCP201] Housley, R., "Guidelines for Cryptographic Algorithm Agility and Selecting Mandatory-to-Implement Algorithms", BCP 201, RFC 7696, November 2015.
- <<https://www.rfc-editor.org/info/bcp201>>
- [RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <<https://www.rfc-editor.org/info/rfc7252>>.
- [RFC7515] Jones, M., Bradley, J., and N. Sakimura, "JSON Web Signature (JWS)", RFC 7515, DOI 10.17487/RFC7515, May 2015, <<https://www.rfc-editor.org/info/rfc7515>>.
- [RFC7516] Jones, M. and J. Hildebrand, "JSON Web Encryption (JWE)", RFC 7516, DOI 10.17487/RFC7516, May 2015, <<https://www.rfc-editor.org/info/rfc7516>>.
- [RFC7517] Jones, M., "JSON Web Key (JWK)", RFC 7517, DOI 10.17487/RFC7517, May 2015, <<https://www.rfc-editor.org/info/rfc7517>>.
- [RFC7518] Jones, M., "JSON Web Algorithms (JWA)", RFC 7518, DOI 10.17487/RFC7518, May 2015, <<https://www.rfc-editor.org/info/rfc7518>>.
- [RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <<https://www.rfc-editor.org/info/rfc8032>>.
- [DSS] National Institute of Standards and Technology, "Digital Signature Standard (DSS)", DOI 10.6028/NIST.FIPS.186-4, FIPS PUB 186-4, July 2013, <<http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf>>.
- [PVSig] Brown, D. and D. Johnson, "Formal Security Proofs for a Signature Scheme with Partial Message Recovery", DOI 10.1007/3-540-45353-9_11, LNCS Volume 2020, June 2000, <https://doi.org/10.1007/3-540-45353-9_11>.
- [W3C.WebCrypto] Watson, M., "Web Cryptography API", W3C Recommendation, January 2017, <<https://www.w3.org/TR/WebCryptoAPI/>>.

- [RFC8230] Jones, M., "Using RSA Algorithms with CBOR Object Signing and Encryption (COSE) Messages", RFC 8230, DOI 10.17487/RFC8230, September 2017, <<https://www.rfc-editor.org/info/rfc8230>>.
- [RFC7942] Sheffer, Y. and A. Farrel, "Improving Awareness of Running Code: The Implementation Status Section", BCP 205, RFC 7942, DOI 10.17487/RFC7942, July 2016, <<https://www.rfc-editor.org/info/rfc7942>>.
- [RFC3394] Schaad, J. and R. Housley, "Advanced Encryption Standard (AES) Key Wrap Algorithm", RFC 3394, DOI 10.17487/RFC3394, September 2002, <<https://www.rfc-editor.org/info/rfc3394>>.
- [I-D.ietf-cose-hash-algs]
Schaad, J., "CBOR Object Signing and Encryption (COSE): Hash Algorithms", Work in Progress, Internet-Draft, draft-ietf-cose-hash-algs-09, 14 September 2020, <<https://tools.ietf.org/html/draft-ietf-cose-hash-algs-09>>.
- [I-D.ietf-core-groupcomm-bis]
Dijk, E., Wang, C., and M. Tiloca, "Group Communication for the Constrained Application Protocol (CoAP)", Work in Progress, Internet-Draft, draft-ietf-core-groupcomm-bis-02, 2 November 2020, <<https://tools.ietf.org/html/draft-ietf-core-groupcomm-bis-02>>.
- [RFC8613] Selander, G., Mattsson, J., Palombini, F., and L. Seitz, "Object Security for Constrained RESTful Environments (OSCORE)", RFC 8613, DOI 10.17487/RFC8613, July 2019, <<https://www.rfc-editor.org/info/rfc8613>>.
- [I-D.irtf-cfrg-argon2]
Biryukov, A., Dinu, D., Khovratovich, D., and S. Josefsson, "The memory-hard Argon2 password hash and proof-of-work function", Work in Progress, Internet-Draft, draft-irtf-cfrg-argon2-12, 8 September 2020, <<https://tools.ietf.org/html/draft-irtf-cfrg-argon2-12>>.
- [COAP.Formats]
IANA, "CoAP Content-Formats", <<https://www.iana.org/assignments/core-parameters/core-parameters.xhtml#content-formats>>.

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

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

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

[I-D.ietf-cose-countersign]
    Schaad, J., "CBOR Object Signing and Encryption (COSE):
    Countersignatures".
```

Appendix A. Guidelines for External Data Authentication of Algorithms

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

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

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

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

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

The following items should be taken into account:

- * Applications need to list the set of COSE structures that implicit algorithms are to be used in. Applications need to require that the receipt of an explicit algorithm identifier in one of these structures will lead to the message being rejected. This requirement is stated so that there will never be a case where there is any ambiguity about the question of which algorithm should be used, the implicit or the explicit one. This applies even if the transported algorithm identifier is a protected attribute. This applies even if the transported algorithm is the same as the implicit algorithm.
- * Applications need to define the set of information that is to be considered to be part of a context when omitting algorithm identifiers. At a minimum, this would be the key identifier (if needed), the key, the algorithm, and the COSE structure it is used with. Applications should restrict the use of a single key to a single algorithm. As noted for some of the algorithms in [I-D.ietf-cose-rfc8152bis-algs], the use of the same key in different related algorithms can lead to leakage of information about the key, leakage about the data or the ability to perform forgeries.
- * In many cases, applications that make the algorithm identifier implicit will also want to make the context identifier implicit for the same reason. That is, omitting the context identifier will decrease the message size (potentially significantly depending on the length of the identifier). Applications that do this will need to describe the circumstances where the context identifier is to be omitted and how the context identifier is to be inferred in these cases. (An exhaustive search over all of the keys would normally not be considered to be acceptable.) An example of how this can be done is to tie the context to a transaction identifier. Both would be sent on the original message, but only the transaction identifier would need to be sent after that point as the context is tied into the transaction identifier. Another way would be to associate a context with a

network address. All messages coming from a single network address can be assumed to be associated with a specific context. (In this case, the address would normally be distributed as part of the context.)

- * Applications cannot rely on key identifiers being unique unless they take significant efforts to ensure that they are computed in such a way as to create this guarantee. Even when an application does this, the uniqueness might be violated if the application is run in different contexts (i.e., with a different context provider) or if the system combines the security contexts from different applications together into a single store.
- * Applications should continue the practice of protecting the algorithm identifier. Since this is not done by placing it in the protected attributes field, applications should define an application-specific external data structure that includes this value. This external data field can be used as such for content encryption, MAC, and signature algorithms. It can be used in the SuppPrivInfo field for those algorithms that use a KDF to derive a key value. Applications may also want to protect other information that is part of the context structure as well. It should be noted that those fields, such as the key or a Base IV, are protected by virtue of being used in the cryptographic computation and do not need to be included in the external data field.

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

The following additional items need to be taken into consideration:

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

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

- * Two contexts are distributed as a pair. Each of the contexts is for use with a COSE_Encrypt message. Each context will consist of distinct secret keys and IVs and potentially even different algorithms. One context is for sending messages from party A to party B, and the second context is for sending messages from party B to party A. This means that there is no chance for a reflection attack to occur as each party uses different secret keys to send its messages; a message that is reflected back to it would fail to decrypt.
- * Two contexts are distributed as a pair. The first context is used for encryption of the message, and the second context is used to place a countersignature on the message. The intention is that the second context can be distributed to other entities independently of the first context. This allows these entities to validate that the message came from an individual without being able to decrypt the message and see the content.
- * Two contexts are distributed as a pair. The first context contains a key for dealing with MACed messages, and the second context contains a different key for dealing with encrypted messages. This allows for a unified distribution of keys to participants for different types of messages that have different keys, but where the keys may be used in a coordinated manner.

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

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

Appendix B. Two Layers of Recipient Information

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

These layers would be:

- * Layer 0: The content encryption layer. This layer contains the payload of the message.
- * Layer 1: The encryption of the CEK by a KEK.

- * Layer 2: The encryption of a long random secret using an RSA key and a key derivation function to convert that secret into the KEK.

This is an example of what a triple layer message would look like. The message has the following layers:

- * Layer 0: Has a content encrypted with AES-GCM using a 128-bit key.
- * Layer 1: Uses the AES Key Wrap algorithm with a 128-bit key.
- * Layer 2: Uses ECDH Ephemeral-Static direct to generate the layer 1 key.

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

Size of binary file is 183 bytes

```

96(
  [ / COSE_Encrypt /
    / protected h'a10101' / << {
      / alg / 1:1 / AES-GCM 128 /
    } >>,
    / unprotected / {
      / iv / 5:h'02dlf7e6f26c43d4868d87ce'
    },
    / ciphertext / h'64f84d913ba60a76070a9a48f26e97e863e2852948658f0
811139868826e89218a75715b',
    / recipients / [
      [ / COSE_Recipient /
        / protected / h'',
        / unprotected / {
          / alg / 1:-3 / A128KW /
        },
        / ciphertext / h'dbd43c4e9d719c27c6275c67d628d493f090593db82
18f11',
        / recipients / [
          [ / COSE_Recipient /
            / protected h'a1013818' / << {
              / alg / 1:-25 / ECDH-ES + HKDF-256 /
            } >> ,
            / unprotected / {
              / ephemeral / -1:{
                / kty / 1:2,
                / crv / -1:1,
                / x / -2:h'b2add44368ea6d641f9ca9af308b4079aeb519f11
e9b8a55a600b21233e86e68',
                / y / -3:false
              },
              / kid / 4:'meriadoc.brandybuck@buckland.example'
            },
            / ciphertext / h''
          ]
        ]
      ]
    ]
  )

```

Appendix C. Examples

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

A GitHub project has been created at <https://github.com/cose-wg/Examples> that contains not only the examples presented in this document, but a more complete set of testing examples as well. Each example is found in a JSON file that contains the inputs used to create the example, some of the intermediate values that can be used in debugging the example and the output of the example presented both as a hex dump and in CBOR diagnostic notation format. Some of the examples at the site are designed failure testing cases; these are clearly marked as such in the JSON file. If errors in the examples in this document are found, the examples on GitHub will be updated, and a note to that effect will be placed in the JSON file.

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

```
gem install cbor-diag
```

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

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

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

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

C.1. Examples of Signed Messages

C.1.1. Single Signature

This example uses the following:

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

Size of binary file is 103 bytes

```

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

```

C.1.2. Multiple Signers

This example uses the following:

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

Size of binary file is 277 bytes

```

98(
  [
    / protected / h'',
    / unprotected / {},
    / payload / 'This is the content.',
    / signatures / [
      [
        / protected h'a10126' / << {
          / alg / 1:-7 / ECDSA 256 /
        } >>,
        / unprotected / {
          / kid / 4:'11'
        },
        / signature / h'e2aeafd40d69d19dfe6e52077c5d7ff4e408282cbefb
5d06cbf414af2e19d982ac45ac98b8544c908b4507de1e90b717c3d34816fe926a2b
98f53afd2fa0f30a'
      ],
      [
        / protected h'a1013823' / << {
          / alg / 1:-36 / ECDSA 521 /
        } >> ,
        / unprotected / {
          / kid / 4:'bilbo.baggins@hobbiton.example'
        },
        / signature / h'00a2d28a7c2bdb1587877420f65adf7d0b9a06635dd1
de64bb62974c863f0b160dd2163734034e6ac003b01e8705524c5c4ca479a952f024
7ee8cb0b4fb7397ba08d009e0c8bf482270cc5771aa143966e5a469a09f613488030
c5b07ec6d722e3835adb5b2d8c44e95fffb13877dd2582866883535de3bb03d01753f
83ab87bb4f7a0297'
      ]
    ]
  ]
)

```

C.1.3. Signature with Criticality

This example uses the following:

- * Signature Algorithm: ECDSA w/ SHA-256, Curve P-256
- * There is a criticality marker on the "reserved" header parameter

Size of binary file is 125 bytes

```

98(
  [
    / protected h'a2687265736572766564f40281687265736572766564' /
    << {
      "reserved":false,
      / crit / 2:[
        "reserved"
      ]
    } >>,
    / unprotected / {},
    / payload / 'This is the content.',
    / signatures / [
      [
        / protected h'a10126' / << {
          / alg / 1:-7 / ECDSA 256 /
        } >>,
        / unprotected / {
          / kid / 4:'11'
        },
        / signature / h'3fc54702aa56e1b2cb20284294c9106a63f91bac658d
69351210a031d8fc7c5ff3e4be39445b1a3e83e1510dlaca2f2e8a7c081c7645042b
18aba9d1fad1bd9c'
      ]
    ]
  ]
)

```

C.2. Single Signer Examples

C.2.1. Single ECDSA Signature

This example uses the following:

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

Size of binary file is 98 bytes

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

C.3. Examples of Enveloped Messages

C.3.1. Direct ECDH

This example uses the following:

- * CEK: AES-GCM w/ 128-bit key
- * Recipient class: ECDH Ephemeral-Static, Curve P-256

Size of binary file is 151 bytes


```

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

```

C.3.2. Direct Plus Key Derivation

This example uses the following:

- * CEK: AES-CCM w/ 128-bit key, truncate the tag to 64 bits
- * Recipient class: Use HKDF on a shared secret with the following implicit fields as part of the context.
 - salt: "aabbccddeeffgghh"
 - PartyU identity: "lighting-client"
 - PartyV identity: "lighting-server"
 - Supplementary Public Other: "Encryption Example 02"

Size of binary file is 91 bytes

```

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

```

C.3.3. Encrypted Content with External Data

This example uses the following:

- * CEK: AES-GCM w/ 128-bit key
- * Recipient class: ECDH static-Static, Curve P-256 with AES Key Wrap
- * Externally Supplied AAD: h'0011bbcc22dd44ee55ff660077'

Size of binary file is 173 bytes

```

96(
  [
    / protected h'a10101' / << {
      / alg / 1:1 / AES-GCM 128 /
    } >> ,
    / unprotected / {
      / iv / 5:h'02d1f7e6f26c43d4868d87ce'
    },
    / ciphertext / h'64f84d913ba60a76070a9a48f26e97e863e28529d8f5335
e5f0165eee976b4a5f6c6f09d',
    / recipients / [
      [
        / protected / h'a101381f' / {
          \ alg \ 1:-32 \ ECHD-SS+A128KW \
        } / ,
        / unprotected / {
          / static kid / -3:'peregrin.took@tuckborough.example',
          / kid / 4:'meriadoc.brandybuck@buckland.example',
          / U nonce / -22:h'0101'
        },
        / ciphertext / h'41e0d76f579dbd0d936a662d54d8582037de2e366fd
e1c62'
      ]
    ]
  ]
)

```

C.4. Examples of Encrypted Messages

C.4.1. Simple Encrypted Message

This example uses the following:

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

Size of binary file is 52 bytes

```

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

```

C.4.2. Encrypted Message with a Partial IV

This example uses the following:

- * CEK: AES-CCM w/ 128-bit key and a 64-bit tag
- * Prefix for IV is 89F52F65A1C580933B52

Size of binary file is 41 bytes

```

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

```

C.5. Examples of MACed Messages

C.5.1. Shared Secret Direct MAC

This example uses the following:

- * MAC: AES-CMAC, 256-bit key, truncated to 64 bits
- * Recipient class: direct shared secret

Size of binary file is 57 bytes

```

97(
  [
    / protected h'a1010f' / << {
      / alg / 1:15 / AES-CBC-MAC-256//64 /
    } >> ,
    / unprotected / {},
    / payload / 'This is the content.',
    / tag / h'9e1226balf81b848',
    / recipients / [
      [
        / protected / h'',
        / unprotected / {
          / alg / 1:-6 / direct /,
          / kid / 4:'our-secret'
        },
        / ciphertext / h''
      ]
    ]
  ]
)

```

C.5.2. ECDH Direct MAC

This example uses the following:

- * MAC: HMAC w/SHA-256, 256-bit key
- * Recipient class: ECDH key agreement, two static keys, HKDF w/ context structure

Size of binary file is 214 bytes

```

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

```

C.5.3. Wrapped MAC

This example uses the following:

- * MAC: AES-MAC, 128-bit key, truncated to 64 bits
- * Recipient class: AES Key Wrap w/ a pre-shared 256-bit key

Size of binary file is 109 bytes

```

97(
  [
    / protected h'a1010e' / << {
      / alg / 1:14 / AES-CBC-MAC-128//64 /
    } >> ,
    / unprotected / {},
    / payload / 'This is the content.',
    / tag / h'36f5afaf0bab5d43',
    / recipients / [
      [
        / protected / h'',
        / unprotected / {
          / alg / 1:-5 / A256KW /,
          / kid / 4:'018c0ae5-4d9b-471b-bfd6-eef314bc7037'
        },
        / ciphertext / h'711ab0dc2fc4585dce27effa6781c8093eba906f227
b6eb0'
      ]
    ]
  ]
)

```

C.5.4. Multi-Recipient MACed Message

This example uses the following:

- * MAC: HMAC w/ SHA-256, 128-bit key
- * Recipient class: Uses three different methods
 1. ECDH Ephemeral-Static, Curve P-521, AES Key Wrap w/ 128-bit key
 2. AES Key Wrap w/ 256-bit key

Size of binary file is 309 bytes

```

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

```

C.6. Examples of MAC0 Messages

C.6.1. Shared Secret Direct MAC

This example uses the following:

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

* Recipient class: direct shared secret

Size of binary file is 37 bytes

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

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

C.7. COSE Keys

C.7.1. Public Keys

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

In order the keys are:

- * An EC key with a kid of "meriadoc.brandybuck@buckland.example"
- * An EC key with a kid of "peregrin.took@tuckborough.example"
- * An EC key with a kid of "bilbo.baggins@hobbiton.example"
- * An EC key with a kid of "11"

Size of binary file is 481 bytes

```
[
  {
    -1:1,
    -2:h'65eda5a12577c2bae829437fe338701a10aaa375e1bb5b5de108de439c0
8551d',
    -3:h'1e52ed75701163f7f9e40ddf9f341b3dc9ba860af7e0ca7ca7e9eecd008
4d19c',
    1:2,
    2:'meriadoc.brandybuck@buckland.example'
  },
  {
    -1:1,
    -2:h'bac5b11cad8f99f9c72b05cf4b9e26d244dc189f745228255a219a86d6a
09eff',
    -3:h'20138bf82dc1b6d562be0fa54ab7804a3a64b6d72ccfed6b6fb6ed28bbf
c117e',
    1:2,
    2:'11'
  },
  {
    -1:3,
    -2:h'0072992cb3ac08ecf3e5c63dedec0d51a8clf79ef2f82f94f3c737bf5de
7986671eac625fe8257bbd0394644caaa3aaf8f27a4585fbbcad0f2457620085e5c8
f42ad',
    -3:h'01dca6947bce88bc5790485ac97427342bc35f887d86d65a089377e247e
60baa55e4e8501e2ada5724ac51d6909008033ebc10ac999b9d7f5cc2519f3fe1ea1
d9475',
    1:2,
    2:'bilbo.baggins@hobbiton.example'
  },
  {
    -1:1,
    -2:h'98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbfbf054e1c7b4d91
d6280',
    -3:h'f01400b089867804b8e9fc96c3932161f1934f4223069170d924b7e03bf
822bb',
    1:2,
    2:'peregrin.took@tuckborough.example'
  }
]
```

C.7.2. Private Keys

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

In order the keys are:

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

Size of binary file is 816 bytes

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

```

    -4:h'00085138ddabf5ca975f5860f91a08e91d6d5f9a76ad4018766a476680b
55cd339e8ab6c72b5facdb2a2a50ac25bd086647dd3e2e6e99e84ca2c3609fdf177f
eb26d'
  },
  {
    1:4,
    2:'our-secret',
    -1:h'849b57219dae48de646d07dbb533566e976686457c1491be3a76dcea6c4
27188'
  },
  {
    1:2,
    -1:1,
    2:'peregrin.took@tuckborough.example',
    -2:h'98f50a4ff6c05861c8860d13a638ea56c3f5ad7590bbfbf054e1c7b4d91
d6280',
    -3:h'f01400b089867804b8e9fc96c3932161f1934f4223069170d924b7e03bf
822bb',
    -4:h'02d1f7e6f26c43d4868d87ceb2353161740aacf1f7163647984b522a848
df1c3'
  },
  {
    1:4,
    2:'our-secret2',
    -1:h'849b5786457c1491be3a76dcea6c4271'
  },
  {
    1:4,
    2:'018c0ae5-4d9b-471b-bfd6-eef314bc7037',
    -1:h'849b57219dae48de646d07dbb533566e976686457c1491be3a76dcea6c4
27188'
  }
]

```

Acknowledgments

This document is a product of the COSE working group of the IETF.

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

The initial version of the specification was based to some degree on the outputs of the JOSE and S/MIME working groups.

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

Author's Address

Jim Schaad
August Cellars

Email: ietf@augustcellars.com

Network Working Group
Internet-Draft
Intended status: Standards Track
Expires: 16 June 2021

J. Schaad
August Cellars
13 December 2020

CBOR Object Signing and Encryption (COSE): Header parameters for
carrying and referencing X.509 certificates
draft-ietf-cose-x509-08

Abstract

The CBOR Signing And Encrypted Message (COSE) structure uses references to keys in general. For some algorithms, additional properties are defined which carry parameters relating to keys as needed. The COSE Key structure is used for transporting keys outside of COSE messages. This document extends the way that keys can be identified and transported by providing attributes that refer to or contain X.509 certificates.

Contributing to this document

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

The source for this draft is being maintained in GitHub. Suggested changes should be submitted as pull requests at <https://github.com/cose-wg/X509>. 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

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

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

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

This Internet-Draft will expire on 16 June 2021.

Copyright Notice

Copyright (c) 2020 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1. Introduction	2
1.1. Requirements Terminology	3
2. X.509 COSE Header Parameters	3
3. X.509 certificates and static-static ECDH	7
4. IANA Considerations	8
4.1. COSE Header Parameter Registry	8
4.2. COSE Header Algorithm Parameter Registry	8
5. Security Considerations	9
6. References	9
6.1. Normative References	9
6.2. Informative References	10
Author's Address	11

1. Introduction

In the process of writing [RFC8152], the working group discussed X.509 certificates [RFC5280] and decided that no use cases were presented that showed a need to support certificates. Since that time, a number of cases have been defined in which X.509 certificate support is necessary, and by implication, applications will need a documented and consistent way to handle such certificates. This document defines a set of attributes that will allow applications to transport and refer to X.509 certificates in a consistent manner.

In some of these cases, a constrained device is being deployed in the context of an existing X.509 PKI: for example, in the 6TiSCH environment, [I-D.richardson-enrollment-roadmap] describes a device enrollment solution that relies on the presence of a factory-installed certificate on the device. The [I-D.ietf-lake-edhoc] draft was also written with the idea that long term certificates could be used to provide for authentication of devices, and uses them to establish session keys. Another possible scenario is the use of COSE

as the basis for a secure messaging application. This scenario assumes the presence of long term keys and a central authentication authority. Basing such an application on public key certificates allows it to make use of well established key management disciplines.

1.1. Requirements Terminology

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

2. X.509 COSE Header Parameters

The use of X.509 certificates allows for an existing trust infrastructure to be used with COSE. This includes the full suite of enrollment protocols, trust anchors, trust chaining and revocation checking that have been defined over time by the IETF and other organizations. The key structures that have been defined in COSE currently do not support all of these properties although some may be found in COSE Web Tokens (CWT) [RFC8392].

It is not necessarily expected that constrained devices themselves will evaluate and process X.509 certificates: it is perfectly reasonable for a constrained device to be provisioned with a certificate that it subsequently provides to a relying party - along with a signature or encrypted message - on the assumption that the relying party is not a constrained device, and is capable of performing the required certificate evaluation and processing. It is also reasonable that a constrained device would have the hash of a certificate associated with a public key and be configured to use a public key for that thumbprint, but without performing the certificate evaluation or even having the entire certificate. In any case, there still needs to be an entity that is responsible for handling the possible certificate revocation.

Parties that intend to rely on the assertions made by a certificate obtained from any of these methods still need to validate it. This validation can be done according to the PKIX rules in [RFC5280] or by using a different trust structure, such as a trusted certificate distributor for self-signed certificates. The PKIX validation includes matching against the trust anchors configured for the application. These rules apply when the validation succeeds in a single step as well as when certificate chains need to be built. If the application cannot establish trust in the certificate, the public key contained in the certificate cannot be used for cryptographic operations.

The header parameters defined in this document are:

x5bag: This header parameter contains a bag of X.509 certificates. The set of certificates in this header parameter is unordered and may contain self-signed certificates. Note that there could be duplicating certificates. The certificate bag can contain certificates which are completely extraneous to the message. (An example of this would be where a signed message is being used to transport a certificate containing a key agreement key.) As the certificates are unordered, the party evaluating the signature will need to be capable of building the certificate path as necessary. That party will also have to take into account that the bag may not contain the full set of certificates needed to build any particular chain.

The trust mechanism MUST process any certificates in this parameter as untrusted input. The presence of a self-signed certificate in the parameter MUST NOT cause the update of the set of trust anchors without some out-of-band confirmation. As the contents of this header parameter are untrusted input, the header parameter can be in either the protected or unprotected header bucket.

This header parameter allows for a single X.509 certificate or a bag of X.509 certificates to be carried in the message.

- * If a single certificate is conveyed, it is placed in a CBOR byte string.
- * If multiple certificates are conveyed, a CBOR array of byte strings is used, with each certificate being in its own byte string.

x5chain: This header parameter contains an ordered array of X.509 certificates. The certificates are to be ordered starting with the certificate containing the end-entity key followed by the certificate which signed it and so on. There is no requirement for the entire chain to be present in the element if there is reason to believe that the relying party already has, or can locate the missing certificates. This means that the relying party is still required to do path building, but that a candidate path is proposed in this header parameter.

The trust mechanism MUST process any certificates in this parameter as untrusted input. The presence of a self-signed certificate in the parameter MUST NOT cause the update of the set of trust anchors without some out-of-band confirmation. As the contents of this header parameter are untrusted input, the header parameter can be in either the protected or unprotected header bucket.

This header parameter allows for a single X.509 certificate or a chain of X.509 certificates to be carried in the message.

- * If a single certificate is conveyed, it is placed in a CBOR byte string.
- * If multiple certificates are conveyed, a CBOR array of byte strings is used, with each certificate being in its own byte string.

x5t: This header parameter provides the ability to identify an X.509 certificate by a hash value (a thumbprint). The 'x5t' header parameter can be represented as an array of two elements. The first element is an algorithm identifier which is an integer or a string containing the hash algorithm identifier corresponding to either the Value (integer) or Name (string) column of the algorithm registered in the "COSE Algorithms" registry. The second element is a binary string containing the hash value computed over the DER encoded certificate.

As this header parameter does not provide any trust, the header parameter can be in either a protected or unprotected header bucket.

For interoperability, applications which use this header parameter MUST support the hash algorithm 'SHA-256', but can use other hash algorithms. This requirement allows for different implementations to be configured to use an interoperable algorithm, but does not preclude the use (by prior agreement) of other algorithms.

RFC Editor please remove the following two paragraphs:

During AD review, a question was raised about how effective the previous statement is in terms of dealing with a MTI algorithm. There needs to be some type of arrangement between the parties to agree that a specific hash algorithm is going to be used in computing the thumbprint. Making it a MUST use would make that true, but it then means that agility is going to be very difficult.

The worry is that while SHA-256 may be mandatory, if a sender supports SHA-256 but only sends SHA-512 then the recipient which only does SHA-256 would not be able to use the thumbprint. In that case both applications would conform to the specification, but still not be able to inter-operate.

x5u: This header parameter provides the ability to identify an X.509 certificate by a URI [RFC3986]. It contains a CBOR text string. The referenced resource can be any of the following media types:

- * application/pkix-cert [RFC2585]
- * application/pkcs7-mime; smime-type="certs-only" [RFC8551]

As this header parameter implies a trust relationship between the party generating the x5u parameter and the party hosting the referred-to resource, this header parameter MUST be in the protected attribute bucket.

The URI provided MUST provide integrity protection and server authentication. For example, an HTTP or CoAP GET request to retrieve a certificate MUST use TLS [RFC8446] or DTLS [I-D.ietf-tls-dtls13]. If the retrieved certificate does not chain to an existing trust anchor, the certificate MUST NOT be trusted unless the server is configured as trusted to provide new trust anchors or if an out-of-band confirmation can be received for trusting the retrieved certificate.

The header parameters are used in the following locations:

- * COSE_Signature and COSE_Sign1 objects: in these objects they identify the certificate to be used for validating the signature.
- * COSE_recipient objects: in this location they identify the certificate for the recipient of the message.

The labels assigned to each header parameter can be found in the following table.

Name	Label	Value Type	Description
x5bag	TBD4	COSE_X509	An unordered bag of X.509 certificates
x5chain	TBD3	COSE_X509	An ordered chain of X.509 certificates
x5t	TBD1	COSE_CertHash	Hash of an X.509 certificate
x5u	TBD2	uri	URI pointing to an X.509 certificate

Table 1: X.509 COSE Header Parameters

Below is an equivalent CDDL [RFC8610] description of the text above.

```
COSE_X509 = bstr / [ 2*certs: bstr ]
COSE_CertHash = [ hashAlg: (int / tstr), hashValue: bstr ]
```

The content of the bstr are the bytes of a DER encoded certificate.

3. X.509 certificates and static-static ECDH

The header parameters defined in the previous section are used to identify the recipient certificates for the ECDH key agreement algorithms. In this section we define the algorithm specific parameters that are used for identifying or transporting the sender's key for static-static key agreement algorithms.

These attributes are defined analogously to those in the previous section. There is no definition for the certificate bag, as the same attribute would be used for both the sender and recipient certificates.

x5chain-sender: This header parameter contains the chain of certificates starting with the sender's key exchange certificate. The structure is the same as 'x5chain'.

x5t-sender: This header parameter contains the hash value for the sender's key exchange certificate. The structure is the same as 'x5t'.

x5u-sender: This header parameter contains a URI for the sender's

key exchange certificate. The structure and processing are the same as 'x5u'.

Name	Label	Type	Algorithm	Description
x5t-sender	TBD	COSE_CertHash	ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	Thumbprint for the senders X.509 certificate
x5u-sender	TBD	uri	ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	URI for the senders X.509 certificate
x5chain-sender	TBD	COSE_X509	ECDH-SS+HKDF-256, ECDH-SS+HKDF-512, ECDH-SS+A128KW, ECDH-SS+A192KW, ECDH-SS+A256KW	static key X.509 certificate chain

Table 2: Static ECDH Algorithm Values

4. IANA Considerations

4.1. COSE Header Parameter Registry

IANA is requested to register the new COSE Header parameters in Table 1 in the "COSE Header Parameters" registry. The "Value Registry" field is empty for all of the items. For each item, the 'Reference' field points to this document.

4.2. COSE Header Algorithm Parameter Registry

IANA is requested to register the new COSE Header Algorithm parameters in Table 2 in the "COSE Header Algorithm Parameters" registry. For each item, the 'Reference' field points to this document.

5. Security Considerations

Establishing trust in a certificate is a vital part of processing. A major component of establishing trust is determining what the set of trust anchors are for the process. A new self-signed certificate appearing on the client cannot be a trigger to modify the set of trust anchors, because a well defined trust-establishment process is required. One common way for a new trust anchor to be added (or removed) from a device is by doing a new firmware upgrade.

In constrained systems, there is a trade-off between the order of checking the signature and checking the certificate for validity. Validating certificates can require that network resources be accessed in order to get revocation information or retrieve certificates during path building. The resulting network access can consume power and network bandwidth. On the other hand, if the certificates are validated after the signature is validated, an oracle can potentially be built based on detecting the network resources which is only done if the signature validation passes. In any event, both the signature and certificate validation **MUST** be completed successfully before acting on any requests.

Before using the key in a certificate, the key **MUST** be checked against the algorithm to be used and any algorithm specific checks need to be made. These checks can include validating that points are on curves for elliptical curve algorithms, and that sizes of RSA keys are of an acceptable size. The use of unvalidated keys can lead either to loss of security or excessive consumption of resources (for example using a 200K RSA key).

When processing x5u header parameter the security considerations of [RFC3986] and specifically those defined in Section 7.1 also apply.

Regardless of the source, certification path validation is an important part of establishing trust in a certificate. Section 6 of [RFC5280] provides guidance for the path validation. The security considerations of [RFC5280] are also important for the correct usage of this document.

The security of the algorithm used for 'x5t' does not affect the security of the system as this header parameter selects which certificate that is already present on the system should be used, but it does not provide any trust.

6. References

6.1. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <<https://www.rfc-editor.org/info/rfc5280>>.
- [RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <<https://www.rfc-editor.org/info/rfc8152>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

6.2. Informative References

- [RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <<https://www.rfc-editor.org/info/rfc8446>>.
- [I-D.ietf-tls-dtls13]
Rescorla, E., Tschofenig, H., and N. Modadugu, "The Datagram Transport Layer Security (DTLS) Protocol Version 1.3", Work in Progress, Internet-Draft, draft-ietf-tls-dtls13-39, 2 November 2020, <<https://tools.ietf.org/html/draft-ietf-tls-dtls13-39>>.
- [RFC8551] Schaad, J., Ramsdell, B., and S. Turner, "Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 4.0 Message Specification", RFC 8551, DOI 10.17487/RFC8551, April 2019, <<https://www.rfc-editor.org/info/rfc8551>>.
- [RFC2585] Housley, R. and P. Hoffman, "Internet X.509 Public Key Infrastructure Operational Protocols: FTP and HTTP", RFC 2585, DOI 10.17487/RFC2585, May 1999, <<https://www.rfc-editor.org/info/rfc2585>>.
- [I-D.ietf-lake-edhoc]
Selandier, G., Mattsson, J., and F. Palombini, "Ephemeral Diffie-Hellman Over COSE (EDHOC)", Work in Progress, Internet-Draft, draft-ietf-lake-edhoc-02, 2 November 2020, <<https://tools.ietf.org/html/draft-ietf-lake-edhoc-02>>.

- [RFC8392] Jones, M., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "CBOR Web Token (CWT)", RFC 8392, DOI 10.17487/RFC8392, May 2018, <<https://www.rfc-editor.org/info/rfc8392>>.
- [RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", RFC 8610, DOI 10.17487/RFC8610, June 2019, <<https://www.rfc-editor.org/info/rfc8610>>.
- [RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <<https://www.rfc-editor.org/info/rfc3986>>.
- [I-D.richardson-enrollment-roadmap]
Richardson, M., "Device Enrollment in IETF protocols -- A Roadmap", Work in Progress, Internet-Draft, draft-richardson-enrollment-roadmap-03, 7 October 2020, <<https://tools.ietf.org/html/draft-richardson-enrollment-roadmap-03>>.

Author's Address

Jim Schaad
August Cellars

Email: ietf@augustcellars.com