



INTERNATIONAL TELECOMMUNICATION UNION

ITU-T

TELECOMMUNICATION
STANDARDIZATION SECTOR
OF ITU

X.894

(10/2018)

SERIES X: DATA NETWORKS, OPEN SYSTEM
COMMUNICATIONS AND SECURITY

OSI applications – Generic applications of ASN.1

Generic applications of ASN.1 Cryptographic Message Syntax

CAUTION !

PREPUBLISHED RECOMMENDATION

This prepublication is an unedited version of a recently approved Recommendation. It will be replaced by the published version after editing. Therefore, there will be differences between this prepublication and the published version.

FOREWORD

The International Telecommunication Union (ITU) is the United Nations specialized agency in the field of telecommunications, information and communication technologies (ICTs). The ITU Telecommunication Standardization Sector (ITU-T) is a permanent organ of ITU. ITU-T is responsible for studying technical, operating and tariff questions and issuing Recommendations on them with a view to standardizing telecommunications on a worldwide basis.

The World Telecommunication Standardization Assembly (WTSA), which meets every four years, establishes the topics for study by the ITU-T study groups which, in turn, produce Recommendations on these topics.

The approval of ITU-T Recommendations is covered by the procedure laid down in WTSA Resolution 1.

In some areas of information technology which fall within ITU-T's purview, the necessary standards are prepared on a collaborative basis with ISO and IEC.

NOTE

In this Recommendation, the expression "Administration" is used for conciseness to indicate both a telecommunication administration and a recognized operating agency.

Compliance with this Recommendation is voluntary. However, the Recommendation may contain certain mandatory provisions (to ensure, e.g., interoperability or applicability) and compliance with the Recommendation is achieved when all of these mandatory provisions are met. The words "shall" or some other obligatory language such as "must" and the negative equivalents are used to express requirements. The use of such words does not suggest that compliance with the Recommendation is required of any party.

INTELLECTUAL PROPERTY RIGHTS

ITU draws attention to the possibility that the practice or implementation of this Recommendation may involve the use of a claimed Intellectual Property Right. ITU takes no position concerning the evidence, validity or applicability of claimed Intellectual Property Rights, whether asserted by ITU members or others outside of the Recommendation development process.

As of the date of approval of this Recommendation, ITU [had/had not] received notice of intellectual property, protected by patents, which may be required to implement this Recommendation. However, implementers are cautioned that this may not represent the latest information and are therefore strongly urged to consult the TSB patent database at <http://www.itu.int/ITU-T/ipr/>.

© ITU 2018

All rights reserved. No part of this publication may be reproduced, by any means whatsoever, without the prior written permission of ITU.

Draft Recommendation ITU-T X.894 (X.cms-prof)

Generic applications of ASN.1

Cryptographic Message Syntax

Summary

This Recommendation | International Standard provides Abstract Syntax Notation One (ASN.1) modules for using cryptographic syntax in ITU-T Recommendations. Cryptographic message syntax (CMS) provides data integrity, confidentiality, origin authenticity, and non-repudiation services needed for reliable information exchange and for strong authentication. It also brings together a set of cryptographic key management techniques to support flexible key establishment mechanisms, such as constructive key management, key agreement, key exchange, and password-based encryption. These techniques can be used to prevent fraud, and to protect personally identifiable and other sensitive information. This Recommendation | International Standard supports digital signature, encryption, and signcryption techniques based on the public-key technology defined in the ITU-T X.500 series | ISO/IEC 9594 multipart standard. All of the standardized encoding rules for ASN.1 are supported.

Keywords

AuthenticatedData,

Cryptographic Message Syntax,

DigestedData,

EncryptedData,

EnvelopedData,

SigncryptData,

SignedData

Draft Recommendation ITU-T X.894 (X.cms-prof)

Generic applications of ASN.1

Cryptographic Message Syntax

1 Scope

This Recommendation | International Standard enhances the existing cryptographic message syntax protocol by adding signcryption techniques and providing a new Abstract Syntax Notation One (ASN.1) module which conforms to the latest edition of the ASN.1 standard which can be used with all standardized encoding rules of ASN.1.

2 Normative references

The following Recommendations and International Standards contain provisions which, through reference in this text, constitute provisions of this Recommendation | International Standard. At the time of publication, the editions indicated were valid. All Recommendations and Standards are subject to revision, and parties to agreements based on this Recommendation | International Standard are encouraged to investigate the possibility of applying the most recent edition of the Recommendations and Standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards. The Telecommunication Standardization Bureau of the ITU maintains a list of currently valid ITU-T Recommendations.

2.1 Identical Recommendations | International Standards

- Recommendation ITU-T X.509 (2016) | ISO/IEC 9594-8:2017, *Information technology – Open Systems Interconnection – The Directory: Public-key and attribute certificate frameworks*.
- Recommendation ITU-T X.680 (2015) | ISO/IEC 8824-1:2015, *Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation*.
- Recommendation ITU-T X.681 (2015) | ISO/IEC 8824-2:2015, *Information technology – Abstract Syntax Notation One (ASN.1): Information object specification*.
- Recommendation ITU-T X.682 (2015) | ISO/IEC 8824-3:2015, *Information technology – Abstract Syntax Notation One (ASN.1): Constraint specification*.
- Recommendation ITU-T X.683 (2015) | ISO/IEC 8824-4:2015, *Information technology – Abstract Syntax Notation One (ASN.1): Parameterization of ASN.1 specifications*.
- Recommendation ITU-T X.690 (2015) | ISO/IEC 8825-1:2015, *Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)*.
- Recommendation ITU-T X.691 (2015) | ISO/IEC 8825-2:2015, *Information technology – ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)*.
- Recommendation ITU-T X.693 (2015) | ISO/IEC 8825-4:2015, *Information technology – ASN.1 encoding rules: XML Encoding Rules (XER)*.
- Recommendation ITU-T X.696 (2015) | ISO/IEC 8825-7:2015, *Information technology – Specification of Octet Encoding Rules (OER)*.
- ISO 11568-1:2005 Banking -- Key management (retail) -- Part 1: Principles

- ISO/IEC 18033-2:2006 Information technology -- Security techniques -- Encryption algorithms -- Part 2: Asymmetric ciphers.
- ISO/IEC 29150:2011, *Information technology – Security techniques – Signcryption*

2.2 Additional References

- IETF RFC 5652 (2009) *Cryptographic Message Syntax (CMS)*
- IETF RFC 5912 (2010) *New ASN.1 Modules for the Public Key Infrastructure Using X.509 (PKIX)*
- IETF RFC 6268 (2011) *Additional New ASN.1 Modules for the Cryptographic Message Syntax (CMS) and the Public Key Infrastructure Using X.509 (PKIX)*

3 Definitions

For the purposes of this Recommendation | International Standard, the following definitions apply:

The following terms are defined in Rec. ITU-T X.509 | ISO/IEC 9594-8:

- attribute certificate
- certificate
- certificate revocation list

The following term is defined in ISO/IEC 29150:2011

signcryption

4 Abbreviations

For the purposes of this Recommendation | International Standard, the following abbreviations apply:

ASN.1	Abstract Syntax Notation One
CMS	Cryptographic Message Syntax
CRL	Certificate Revocation List
DBEKM	Database Encryption Key Management
DK	Data Encryption Key
HK	HMAC Key
HMAC	Hashed Message Authentication
ID	Identifier
KDF	Key Derivation Function
MK	Master Key Encryption Key
PBKDF	Password-Based Key Derivation Function
SCD	Secure Cryptographic Device

5 Conventions

6 Cryptographic Message Syntax

Cryptographic Message Syntax (CMS) has been defined by IETF. The base text is in RFC 5652. ASN.1 modules have been revised to conform to the current ASN.1 standard and have been published in RFC 6268.

CMS defines the following content types:

- data: used to transfer data defined string of octets;
- signed data: used to transfer data with zero or more signatures;
- enveloped data: used to transfer encrypted data with one or more content-encryption keys;
- digested data: used to transfer data with a message digest;
- encrypted data: used to transfer encrypted data;
- authenticated data: used to transfer data with a message authentication code and one or more encrypted authentication keys;

Each of these content types is uniquely identified by an object identifier:

- for data:


```
id-data OBJECT IDENTIFIER ::= {iso(1) member-body(2) us(840)
  rsadsi(113549) pkcs(1) pkcs7(7) 1}
```
- for signed data:


```
id-signedData OBJECT IDENTIFIER ::= {iso(1) member-body(2)
  us(840) rsadsi(113549) pkcs(1) pkcs7(7) 2}
```
- for enveloped data:


```
id-envelopedData OBJECT IDENTIFIER ::= {iso(1) member-body(2) us(840)
  rsadsi(113549) pkcs(1) pkcs7(7) 3}
```
- for digested data:


```
id-digestedData OBJECT IDENTIFIER ::= {iso(1) member-body(2) us(840)
  rsadsi(113549) pkcs(1) pkcs7(7) 5}
```
- for encrypted data:


```
id-encryptedData OBJECT IDENTIFIER ::= {iso(1) member-body(2) us(840)
  rsadsi(113549) pkcs(1) pkcs7(7) 6}
```
- for authenticated data:


```
id-ct-authData OBJECT IDENTIFIER ::= {iso(1) member-body(2) us(840)
  rsadsi(113549) pkcs(1) pkcs-9(9) smime(16) ct(1) 2}
```

Data transferred with Cryptographic Message Syntax use the following ASN.1 type:

```
ContentInfo ::= SEQUENCE {
  contentType      CONTENT-TYPE.&id({ContentSet}),
  content          [0] EXPLICIT CONTENT-
  TYPE.&Type({ContentSet}{@contentType}) }
```

The **CONTENT-TYPE** information object class is defined as **TYPE-IDENTIFIER** and is used to assign one of the previous object identifiers to the corresponding ASN.1 type.

CONTENT-TYPE ::= TYPE-IDENTIFIER

ContentType ::= CONTENT-TYPE.&id

```
ContentSet CONTENT-TYPE ::= {  
    -- Define the set of content types to be recognized  
    ct-Data |  
    ct-SignedData |  
    ct-EnvelopedData |  
    ct-DigestedData |  
    ct-EncryptedData |  
    ct-AuthenticatedData,  
    ...}
```

ct-Data **CONTENT-TYPE ::= {OCTET STRING IDENTIFIED BY id-data}**

ct-SignedData **CONTENT-TYPE ::= {SignedData IDENTIFIED BY id-signedData}**

ct-EnvelopedData **CONTENT-TYPE ::= {EnvelopedData IDENTIFIED BY id-envelopedData}**

ct-DigestedData **CONTENT-TYPE ::= {DigestedData IDENTIFIED BY id-digestedData}**

ct-EncryptedData **CONTENT-TYPE ::= {EncryptedData IDENTIFIED BY id-encryptedData}**

ct-AuthenticatedData **CONTENT-TYPE ::= {AuthenticatedData IDENTIFIED BY id-ct-authData}**

Other content types can be defined by creation of new information objects of **CONTENT-TYPE** information object class using unique object identifiers.

The **ct-SignedCryptedData** defined in clause 7 is an example.

7 Signcryption

The **SigncryptedData** uses the signcryption technique defined in ISO/IEC 29150. The signcryption technique simultaneously signs and encrypts the data to achieve origin authentication, data integrity and confidentiality. Signcryption can be used in CMS in four different modes:

- a) **signcrypted-content**: content of any type or format is signcrypted using the signcryption algorithm;
- b) **signcrypted-attributes**: content of any type or format and a collection of attributes of any type or format are together signcrypted;
- c) **signcrypted-components**: elements of content of any type or format are signcrypted for one or more message recipients using the public-private keys of the sender and the public key of each recipient.
- d) **signcrypted-envelope**: a fresh symmetric key is used to encrypt content of any type or format and the resulting ciphertext is transmitted as an octet string.

7.1 The SigncryptedData type

```
id-signcryptedData OBJECT IDENTIFIER ::= {  
    {itu-t recommendation(0) x(24) cms-profile(894) signcryption(0)  
    data(0) }
```

```
ct-SigncryptData      CONTENT-TYPE ::= {
    TYPE SigncryptData IDENTIFIED BY id-signcryptData}
```

```
SigncryptData ::= SEQUENCE {
    version          CMSVersion,
    contentInformation ContentInformation,
    certificates      CertificateSet OPTIONAL,
    crls              RevocationInfoChoices OPTIONAL,
    signcrypters      Signcrypters
}
```

```
Signcrypters ::= SEQUENCE SIZE(1..MAX) OF Signcrypter
```

- a) **version** is version number.
- b) **contentInformation** identifies the signcrypt data processing mode.
- c) **certificates** is a collection of public key and/or attribute certificates to facilitate the validation of the public keys of the signcrypters. This collection may contain more certificates than necessary or fewer and, in the latter case, recipients have to use other means to get other certificates.
- d) **crls** is a collection of public key and/or attribute certificate revocation lists to facilitate the validation of the certificates of the signcrypters. As for **certificates**, this collection may contain more certificate revocation lists than necessary or fewer and, in the latter case, recipients have to use other means to get other certificate revocation lists.
- e) **signcrypters** is a collection of all signcrypter specific information.

7.2 The ContentInformation type

```
ContentInformation ::= SEQUENCE {
    mode          Mode,
    content        Content OPTIONAL
}
```

The modes are defined using the following information object class:

```
MODE ::= CLASS {
    &Type          OPTIONAL,
    &id             OBJECT IDENTIFIER UNIQUE
}
WITH SYNTAX {[WITH SYNTAX &Type] ID &id}
```

Four modes are currently defined:

- a) **signcryptAttributes**: content and attributes of any type or format are signcrypt.
- b) **signcryptComponents**: specified components of content of any type or format are signcrypt, and the resulting cryptogram is signed along with a manifest of signcrypt locations and other attributes.
- c) **signcryptContent**: content of any type or format is signcrypt.
- d) **signcryptEnveloped**: content of any type or format is encrypted under a symmetric key to create ciphertext for sharing with one or more message recipients. That symmetric key and a message digest of the ciphertext are signcrypt for each message recipient using the public-private keys of the sender and the public key of each recipient.

```
Mode ::= MODE.&id({ProcessingModes})
```

```
ProcessingModes MODE ::= {
    signcryptAttributes |
```

```

    signcryptComponents |
    signcryptContent      |
    signcryptEnveloped,
    ...-- Expect additional processing modes --
}

```

Content ::= OCTET STRING(SIZE(1..MAX))

7.2.1 The signcryptContent mode

In the signcrypt-content mode, content of any type or format is signcrypt using the signcrypt algorithm identified in the **signcryptDataAlgorithm** component of the **signcrypters** component of type **SigncryptData**. The message sender applies this signcrypt algorithm to the content using the sender public and private keys and the recipient public key. These keys are identified in the message as values of type **SigncrypterIDs** and the **signcryptValue** component of type **Signcrypter** contains the results of signcrypt.

In this processing mode of **SigncryptData**, there are no signcrypt attributes and the optional **signatureInformation** component is not present. The message sender at their option may include values in the **unsigncryptAttributes** component of the the **Signcrypter** component for any recipient. The signcrypt-content mode is indicated in a message by the following information object identifier:

signcryptContent MODE ::= { ID signcrypt-content }

A value of type **SigncryptData** using signcrypt-content mode is created as follows:

- a. The value of the **version** component of type **SigncryptData** is set to 1.
- b. In the **contentInformation** component of type **SigncryptData**, set the value of mode to the signcrypt-content object identifier. The optional **content** component should not be present.
- c. Optionally, include values in the **certificates** component of type **SigncryptData**.
- d. Optionally, include values in the **crls** component of type **SigncryptData**.
- e. For each message recipient, include a value of type **Signcrypter** in the **signcrypters** component of type **SigncryptData** as follows:
 - i. The value of the **version** component of type **Signcrypter** is set to 1.
 - ii. In the **sids** component of type **Signcrypter** set the components of a value of type **SigncrypterIDs** to identify the public-private key pairs of the message sender and recipient.
 - iii. Set the value of the **signcryptDataAlgorithm** component of type **Signcrypter** to identify the signcrypt algorithm and any associated parameters.
 - iv. Include the results of **signcrypting** content of any type or format encoded as a value of type OCTET STRING in the **signcryptValue** component of type **Signcrypter**.
 - v. The optional **signatureInformation** component of type **Signcrypter** should not be present.
 - vi. Optionally, include values in the **unsigncryptAttributes** component of type **Signcrypter**.

To recover the plaintext from the **SigncrypteData** message, a message recipient should perform the following steps:

1. Search the list of per recipient values of type **Signcrypter** in the **signcrypters** component of type **SigncrypteData** to locate the recipient public-private key pair in the **sids** component of type **Signcrypter**.
2. Designcrypt the value in the **signcryptioValue** component of type **Signcrypter** for a given recipient using their public-private key pair of the recipient and the public key of the sender identified in the **sids** component of type **Signcrypter**, and the signcryptio algorithm and associated parameters provided in the **signcrypteDataAlgorithm** component of type **Signcrypter**.

Perform certificate path validation to gain assurance that the sender public key certificate is trusted.

7.2.2 The signcrypteAttributes mode

In the signcrypte-attributes mode, content of any type or format and a collection of attributes of any type or format in the **signcryptioAttributes** component of type **SigncrypterInfo** are together signcrypte as specified for the signcrypte-content mode. At least three attributes shall be present: the messageDigest, the **contentType**, and the **signcrypteAttributes** attribute.

The signcrypte-attributes mode is indicated in a message by the following information object identifier:

signcrypteAttributes MODE ::= { ID signcrypte-attributes }

A value of type **SigncrypteData** using signcrypte-attributes mode is created as follows:

- a) The value of the **version** component of type **SigncrypteData** is set to 1.
- b) In the **contentInformation** component of type **SigncrypteData**, set the value of mode to the **signcrypte-attributes** object identifier. The optional **content** component should not be present.
- c) Optionally, include values in the **certificates** component of type **SigncrypteData**.
- d) Optionally, include values in the **crls** component of type **SigncrypteData**.
- e) For each message recipient, include a value of type **Signcrypter** in the **signcrypters** component of type **SigncrypteData** as follows:
 - i. The value of the **version** component of type **Signcrypter** is set to 1.
 - ii. In the **sids** component of type **Signcrypter** set the components of a value of type **SigncrypterIDs** to indentify the public-private key pairs of the message sender and recipient.
 - iii. Set the value of the **signcrypteDataAlgorithm** component of type **Signcrypter** to identify the signcryptio algorithm and any associated parameters.
 - iv. Create a value of type **ToBeSigncrypte** by setting its **content** component to a value of type **Content** and its **attributes** component to a value of type **SigncrypteAttributes**.

- v. Signcrypt an encoded value of type **ToBeSigncrypted** and include the results of the processing in the **signcryptValue** component of type **Signcrypter**.
- vi. The optional **signatureInformation** component of type **SigncrypterInfo** should not be present.
- vii. Optionally, include values in the **unsigncryptAttributes** component of type **Signcrypter**.

To recover the plaintext from the **SigncryptData** message, a message recipient should perform the following steps:

- a) Search the list of per recipient values of type **Signcrypter** in the **signcrypters** component of type **SigncryptData** to locate the recipient public-private key pair in the **sids** component of type **Signcrypter**.
- b) Designcrypt the value in the **signcryptValue** component of type **Signcrypter** for the recipient using the signcrypt algorithm and any associated parameters provided in the **signcryptDataAlgorithm** component of type **Signcrypter** to recover a value of type **ToBeSigncrypted**.
- c) Perform certificate path validation to gain assurance that the sender public key certificate is trusted.

7.2.3 The signcryptComponents mode

In signcrypt-components mode elements of content of any type or format are signcrypt for one or more message recipients using the public-private keys of the sender and the public key of each recipient. A content-type specific manifest of signcrypt element locations in the content are bound to the partially signcrypt content under a digital signature.

The signcrypt-components mode is indicated in a message by the following information object identifier:

signcryptComponents MODE ::= { ID signcrypt-components }

A value of type **SigncryptPartsManifest** is defined in terms of the **&id** and **&Type** fields of the **SIGNCRYPTED** information object class. This type definition uses a parametrized type, **Signcrypted{}**, whose sole parameter is the **Manifest** information object set:

SigncryptPartsManifest ::= Signcrypted{{Manifest}}

Signcrypted{SIGNCRYPTED:IOSet} ::= SEQUENCE {
 Name **SIGNCRYPTED.&id({IOSet})** ,
 Parts **SIGNCRYPTED.&Type({IOSet}){@name}** **OPTIONAL**
}

SIGNCRYPTED ::= CLASS {
 &id **OBJECT IDENTIFIER UNIQUE** ,
 &Type **OPTIONAL**
}

WITH SYNTAX {OID &id [PARMS &Type]}

Manifest SIGNCRYPTED ::= {
 xPathManifest ,
 ...
}

The type of manifest required to identify the signcrypt components of a particular object varies with the type, structure, and format of the object. For content in the form of an image, a **signcrypt** components manifest might be composed of a list of ((x, y), (x, y)) coordinate pairs that define a rectangular area of the image to be signcrypt and thereby redacted.

In this standard, a single manifest object is defined to support component identification in an XML document. The **xPathManifest** object uses location paths based on the XPath query language to specify a set of one or more signcrypt XML document components and is defined as follows:

```
xPathManifest SIGNCRYPTED ::= {
    OID id-cms-XPath  PARMS XPathSet
}

XPathSet ::= SEQUENCE SIZE(1..MAX) OF XPath

XPath ::= UTF8String (CONSTRAINED BY { -- XML Path Query Language 2.0 -- })
```

A value of type **SigncryptData** using signcrypt-components mode is created as follows:

- a) The value of the **version** component of type **SigncryptData** is set to 1.
- b) In the **contentInformation** component of type **SigncryptData**, set the value of **mode** to the signcrypt-components object identifier. The optional **content** component should not be present.
- c) Optionally, include values in the **certificates** component of type **SigncryptData**.
- d) Optionally, include values in the **crls** component of type **SigncryptData**.
- e) For each message recipient, include a value of type **Signcrypter** in the **signcrypters** component of type **SigncryptData** as follows:
 - i. The value of the **version** component of type **Signcrypter** is set to 1.
 - ii. In the **sids** component of type **Signcrypter** set the components of a value of type **SigncrypterIDs** to identify the public-private key pairs of the message sender and recipient.
 - iii. Signcrypt one or more elements of the content and include the partially signcrypt results as the value of the **signcryptValue** component of type **Signcrypter**.
 - iv. In the **signatureInformation** component of type **Signcrypter**, identify the signing key in the optional **signerIdentifier** component if this key differs from the key used to signcrypt.
 - v. If the signature algorithm is not a system default or is not indicated in the signcrypt data algorithm identifier, identify the signature algorithm object identifier and any associated parameters in the **signatureAlgorithm** component of type **Signcrypter**.
 - vi. Prepare a manifest value in the **signcryptPartsManifest** component of type **ToBeSigned** that indicates the location of each signcrypt element in the **signcryptValue** component of type **Signcrypter**.
 - vii. Include the **contentType** and **messageDigest** attributes in the **signedAttributes** component of type **ToBeSigned**.

- viii. Place the results of signing a value of type **ToBeSigned** in the **signatureValue** component of type **SignatureInformation**.
- ix. Optionally, include values in the **unsigncryptAttributes** component of type **Signcrypter**.

To recover the plaintext from the **SigncryptData** message, a message recipient should perform the following steps for each message recipient:

- a. Search the list of per recipient values of type **Signcrypter** in the **signcrypters** component of type **SigncryptData** to locate the recipient public-private key pair in the **sids** component of type **Signcrypter**.
- b. In the **signatureInformation** component of type **Signcrypter**, verify the signature in the **signatureValue** component over a value of type **ToBeSigned** using the signcryption algorithm and any associated parameters provided in the **signcryptDataAlgorithm** component of type **Signcrypter**.
- c. Using the manifest in the **signcryptPartsManifest** component of type **ToBeSigned**, designcrypt each of the indicated signcrypt elements listed in the manifest to recover the partially signcrypt components of the content.
- d. Perform certificate path validation to gain assurance that the sender certificates are trusted.

7.2.4 The signcryptEnvelope mode

In the signcrypt-envelope mode, a fresh symmetric key is used to encrypt content of any type or format, and the resulting ciphertext is placed in the **encryptedContentInfo** component of a value of type **NamedKeyEncryptedData**, a type defined in X9.73 [ANSI X9.73-2017] that may include an optional key name and optional extensions.

```
NamedKeyEncryptedData ::= SEQUENCE {
    Version                CMSVersion,
    keyName                 [0] OCTET STRING OPTIONAL,
    encryptedContentInfo    EncryptedContentInfo,
    unprotectedAttrs       [1] UnprotectedEncAttributes OPTIONAL
}
```

The **EncryptedContentInfo** type is defined in CMS protocol as:

```
EncryptedContentInfo ::=
    EncryptedContentInfoType { ContentEncryptionAlgorithmIdentifier }

EncryptedContentInfoType { AlgorithmIdentifierType } ::= SEQUENCE {
    contentType            CONTENT-TYPE.&id({ContentSet}),
    contentEncryptionAlgorithm AlgorithmIdentifierType,
    encryptedContent        [0] IMPLICIT OCTET STRING OPTIONAL
}
```

The **encryptedContentInfo** component of **NamedKeyEncryptedData** type also contains the required algorithm object identifier and any associated parameters of the symmetric encryption algorithm used to encrypt the content. This prepared value of type **NamedKeyEncryptedData** is then encoded as an **OCTET STRING** and placed in the **content** component of the **contentInfo** component of type **SigncryptData**. The **contentType** component of type **EncryptedContentInfo** is set to the information object identifier value signcrypt-envelope.

The **SigncryptData** message contains one value of type **Signcrypter** for each message recipient. The **Signcrypter** type contains a **signcryptAttributes** component that

must contain the symmetric encryption key signcrypted for each message recipient in a **signcryptedEnvelope** attribute.

The **signcryptedEnvelope** attribute is defined as follows:

```
signcryptedEnvelope ATTRIBUTE ::= {  
    WITH SYNTAX SigncryptKey ID signcrypted-envelope  
}
```

```
SigncryptKey ::= OCTET STRING
```

When the **signcryptedEnvelope** attribute is included in the **signcryptionAttributes** component of type **Signcrypter**, both the **messageDigest** and **contentType** attributes shall be present. To complete construction of the message, the **eContent** is concatenated together with all **signcryptionAttributes** and signed by the message sender. The completed message can then be transmitted to all recipients.

On receiving the message, a recipient first verifies the signature on the message. To recover the payload encrypted in the **signcryptedEnvelope** attribute, a message recipient must first designcrypt the content in the value of type **EncapsulatedContentInfo** to recover the symmetric key, then use the recovered key to decrypt the payload.

A value of type **SigncryptData** using signcrypted-envelope mode is created as follows:

- a. The value of the **version** component of type **SigncryptData** is set to 1.
- b. In the **contentInformation** component of type **SigncryptData**, set the value of mode to the signcrypted-envelope object identifier.
- c. Using a fresh symmetric key, encrypt content of any type or format and place the encrypted content in the optional **encryptedContentInfo** component of a value of type **NamedKeyEncryptedData**.
- d. Complete preparation of a value of type **NamedKeyEncryptedData**, by optionally providing a name for the symmetric key in the **keyName** component, and identifying the symmetric content encryption algorithm and any parameters in the **encryptedContent** component.
- e. Embed the prepared value of type **NamedKeyEncryptedData** in the optional content component in the **contentInformation** component of type **SigncryptData**.
- f. Prepare a **messageDigest** attribute that identifies a message digest algorithm and contains a digest of the **encryptedContent** component of type **NamedKeyEncryptedData**.
- g. Prepare a value of type **ToBeSigncrypted** that includes this **messageDigest** attribute in the **attributes** component, and the symmetric content encryption key in the **contents** component, a value of type **ToBeSigncrypted** that will be signcrypted for each message recipient.
- h. Optionally, include values in the **certificates** component of type **SigncryptData**.
- i. Optionally, include values in the **crls** component of type **SigncryptData**.
- j. For each message recipient, include a value of type **Signcrypter** in the **signcrypters** component of type **SigncryptData** as follows:

- i. The value of the **version** component of type **Signcrypter** is set to 1.
- ii. In the **sids** component of type **Signcrypter** set the components of a value of type **SigncrypterIDs** to identify the public-private key pairs of the message sender and recipient.
- iii. Set the value of the **signcrypteDataAlgorithm** component of type **Signcrypter** to identify the signcryption algorithm and any associated parameters.
- iv. Include the results of signcrypting the prepared value of type **ToBeSigncrypteD** in the **signcryptionValue** component of type **Signcrypter**.
- v. The optional **signatureInformation** component of type **Signcrypter** should not be present.
- vi. Optionally, include values in the **unsigncryptionAttributes** component of type **Signcrypter**.

To recover the plaintext from the **SigncrypteData** message, a message recipient should perform the following steps:

- a. Search the list of per recipient values of type **Signcrypter** in the **signcrypters** component of type **SigncrypteData** to locate the recipient public-private key pair in the **sids** component of type **Signcrypter**.
- b. Decode the optional **content** component in the **contentInformation** component of type **SigncrypteData**, to recover a value of type **NamedKeyEncrypteDData** from the octet string.
- c. Designcrypt the value in the **signcryptionValue** component of type **Signcrypter** for a given recipient using their public-private key pair of the recipient and the public key of the sender to recover the signcrypteD value of type **ToBeSigncrypteD** that contains a symmetric key and a message digest of the content encrypteD using the symmetric key.
- d. Perform certificate path validation to gain assurance that the sender public key certificate is trusted.
- e. Verify that the symmetric key encrypteD content has not been modified by computing a digest of the **encrypteDContentInfo** component of type **NamedKeyEncrypteDData**, using the algorithm in the message digest attribute recovered from the signcrypteD value of type **ToBeSigncrypteD** and comparing the two digests for equivalency.
- f. Use the symmetric key recovered from designcrypting the value of type **ToBeSigncrypteD** to decrypt the content value of the **contentInformation** component of type **SigncrypteData**.

7.3 The Signcrypter type

```

Signcrypter ::= SEQUENCE {
    Version                Version,
    Sids                    SigncrypterIDs,
    signcrypteDataAlgorithm
        SigncrypteDataAlgorithmIdentifier,
    signcryptionValue       SigncryptionValue,
    signatureInformation     SignatureInformation OPTIONAL,

```

```

    unsigncryptAttributes UnSigncryptAttributes
OPTIONAL
}

```

The public-private key pairs of the message sender and recipient are identified as they are in the **SigncryptData** message as values of type **SigncrypterIDs** defined as follows:

```

SigncrypterIDs ::= SEQUENCE {
    Sender      KeyPairIdentifier,
    Recipient    KeyPairIdentifier
}

```

```

KeyPairIdentifier ::= SignerIdentifier

```

A message sender uses their own public and private keys along with the public key of the recipient to signcrypt content. The message recipient uses the public key of the sender with their own public and private keys to designcrypt the content signcrypt by the sender.

The **SigncryptDataAlgorithmIdentifier** type contains the signcryption algorithm identifier and any associated parameters. For all of the signcryption algorithms defined in the ISO/IEC 29150 standard, the parameters include one of the hash functions and one of the key derivation functions defined in ISO/IEC 18033.

```

SigncryptDataAlgorithmIdentifier ::=
    AlgorithmIdentifier {{SigncryptAlgorithms}}

SigncryptAlgorithms ALGORITHM ::= {
    SigncryptionMechanism, -- ISO/IEC 29150 Signcryption
    algorithms --
    ... -- Expect additional algorithm objects --
}

```

A **signcryptionValue** component of type **Signcrypter** contains the results of signcryption, and is defined as follows:

```

SigncryptionValue ::= OCTET STRING (SIZE(1..MAX))

```

An optional **signatureInformation** component of type **Signcrypter** contains information needed to support the use of a digital signature when the signcrypt-components mode of processing is used. The definition of type **signatureInformation** is provided in the detailed processing description that follows.

```

SignatureInformation ::= SEQUENCE {
    signerIdentifier      SignerIdentifier OPTIONAL,
    signatureAlgorithm     SignatureAlgorithmIdentifier
OPTIONAL,
    toBeSigned            ToBeSigned,
    signatureValue        SignatureValue
}

ToBeSigned ::= SEQUENCE {
    signcryptPartsManifest    SigncryptPartsManifest,
    signedAttributes          SignedAttributes
}

SigncryptPartsManifest ::= Signcrypt{{Manifest}}

```

```

Manifest SIGNCRYPTED ::= {
    xPathManifest,
    ... -- Expect additional manifest types --
}

XPathManifest SIGNCRYPTED ::= {
    OID XPath PARMS XPathSet
}

XPathSet ::= SEQUENCE (SIZE(1..MAX)) OF XPath

XPath ::= UTF8String(CONSTRAINED BY { -- XML Path Language 2.0 --
})

```

An optional **unsigncryptAttributes** component of type **Signcrypter** contains any attribute values that may be defined by or required by the message sender. These attributes are not protected by signcrypt data processing. Type **UnSigncryptAttributes** is defined as follows:

```

UnSigncryptAttributes ::=
    SEQUENCE SIZE(1..MAX) OF Attribute
    {{UnSigncryptAttributes}}

UnSigncryptAttributes ATTRIBUTE ::= {
    ... -- Expect additional attributes --
}

```

8 Quantum safe SignedData signatures

Anticipated attacks on current digital signature algorithms using quantum computers will subject documents signed today and requiring long term protection to increasing security risk over time. This problem is not limited solely to quantum computing risks, such as risk of repudiation by the signer. Long termed signed documents such as a thirty-year mortgage are subject to similar risks.

These risks arise as new attacks on signature algorithms are discovered and key length requirements grow with computing power advances. An increase in the number of attacks, the continuing rise in legal and regulatory risks, and changes to the security policies of organizations all add to these risks. It is possible to mitigate some of these security risks using a quantum-safe countersignature over signed content created using current digital signature algorithms.

A countersignature that relies on a quantum-safe signature can be implemented in a SignedData message using an optional signed attribute. The schema of a **SignedData** message supports a series of signers represented in a value of type **SignerInfos**. Each signer is represented in this series as a value of type **SignerInfo**.

Type **SignerInfo** is defined as:

```

SignerInfo ::= SEQUENCE {
    version          CMSVersion,
    sid              SignerIdentifier,
    digestAlgorithm  DigestAlgorithmIdentifier,
    signedAttrs     [0] IMPLICIT SignedAttributes OPTIONAL,
    signatureAlgorithm SignatureAlgorithmIdentifier,
    signature        SignatureValue,
}

```

```

unsignedAttrs [1] IMPLICIT
Attributes{{UnsignedAttributes}} OPTIONAL }

```

Type **SignerInfo** allows each signer to use a different signing key, message digest, and signature algorithm. Each signer can include their own set of attributes that will be cryptographically bound under their signature. A **signerInfos** attribute collects this series of values into an attribute that can be included in the signed attributes of a counter signer.

The **signerInfos** attribute as follows:

```

aa-signerInfos ATTRIBUTE ::=
    {TYPE SignerInfos IDENTIFIED BY id-signerInfos}

id-signerInfos OBJECT IDENTIFIER ::= {itu-t recommendation(0)
x(24)
    cms-profile(894) attribute(2) signerInfos(1)}

```

The **signerInfos** attribute contains a value of type **SignerInfos**, a series of values of type **SignerInfo**, one value for each signer of the **SignedData** content. Each cosigner shall sign content using their choice of signature and message digest algorithm. The signing key of each cosigner shall be included in the **SignerInfo** value of the cosigner using any of the choice alternatives defined in the **SignerIdentifier** type.

The optional **signedAttrs** component of type **SignerInfo** shall be present in the message. At a minimum, each cosigner shall include a **contentType** attribute and a **messageDigest** attribute in the **signedAttrs** component of their **SignerInfo** value. Additional signed attributes of any type or format may also be included by each cosigner. Any number or type of unsigned attributes may also be included by each cosigner in the **unsignedAttrs** component of their **SignerInfo** value.

The values of type **SignerInfo** created by all signers shall follow the other defined processing. During signature verification of a **SignedData** message, a relying party may treat the **signerInfos** attribute as an opaque string. Applications that recognize this attribute may choose to defer signature verification processing. Failure of one or more cosigner **SignerInfo** values shall be handled as defined by the application.

8.1 Detached content consideration

When the detached form of **SignedData** is used, the content of the message is not present in the **SignedData** type. This message content must be available during signing and signature verification operations so that a message digest of the signed content can be calculated.

When default content location is not known to the communicating parties, content signers can include a **contentLocation** attribute in their signed attributes. This attribute can also be used when it is necessary for a cosigner to indicate a different detached object, such as a language-specific version of a contract.

The **contentLocation** attribute is defined as follows:

```

aa-contentLocation ATTRIBUTE ::=
    {TYPE URI IDENTIFIED BY id-contentLocation}

```

```
URI ::= UTF8String(SIZE(1..MAX))
```

```
id-contentLocation OBJECT IDENTIFIER ::= {itu-t recommendation(0)
x(24)
    cms-profile(894) attribute(2) contentLocation(2)}
```

A value of type URI is a Uniform Resource Identifier (URI) value that points to a location of detached **SignedData** content. A **contentLocation** attribute can be included in a **SignedAttributes** component of a **SignerInfo** component of type **SignedData** of any signer of the message. In some applications, it may be convenient to include a single content location attribute in the signed attributes of the counter signer.

8.2 Time stamp consideration

Time stamps included in **SignedData** can be used to demonstrate that a the validity period of a signer certificate included the time of signing a message. Long term signatures may need to be verified after the validity period of a signing certificate has expired. A time stamp attribute that is included in the **SignedAttributes** component can be compared by a relying party to the validity period of the signer certificate to ensure the certificate was valid for use when the message was signed.

The **timeStamped** attribute is defined as follows:

```
aa-timeStamped ATTRIBUTE ::=
    {TYPE TimeStamped IDENTIFIED BY id-timeStamped}

TimeStamped ::= SEQUENCE {
    timeStampValue      TimeStamp,
    timeStampService    URI OPTIONAL
}

TimeStamp ::= CHOICE {
    timeStampToken      TimeStampToken,
    localTimeStamp      GeneralizedTime,
    ... -- Expect additional time types --
}

id-timeStamped OBJECT IDENTIFIER ::= {itu-t recommendation(0)
x(24)
    cms-profile(894) attribute(2) timeStamped(5)}
```

The type **TimeStamped** contains two components, a required **timeStampValue** and an optional **timeStampService** that indicates the location of a time stamp service provider that can validate the time stamp. A **timeStampValue** component is a value of type **TimeStamp**, a choice between two alternatives, an X9.95 trusted timestamp token or a value from a local time source.

```
TimeStamp ::= CHOICE {
    TimeStampToken      TimeStampToken, -- X9.95 Trusted Time
    Stamp --
    localTimeStamp      GeneralizedTime
}
```

The first choice alternative of type **TimeStamp** may be any of the four types of tokens defined in the X9.95 standard.

8.3 The **tokenizedParts** attribute

The **tokenizedParts** attribute carries a manifest, a list of the locations of elements within content of any type or format that have been tokenized. Tokenization techniques provide confidentiality services for the elements of the content specified in the manifest. A manifest can be constructed on a per-message recipient basis to achieve granular message redaction. The format and the information contained in the manifest varies with the type of content and user requirements. Only one manifest is defined in this standard, a simple manifest for XML content, though the set of possible manifests is extensible and unbounded.

XML Path (XPath) expressions can be used to locate any tokenized element in any XML-instance document. To identify the tokenization components in an XML-instance document, a set of XPath expressions can be used to identify the location of each tokenized element. When the contents of an XML element are tokenized, the sender includes the outer markup tags in the tokenization operation.

These outer tags are not removed from the document, since they are used to locate the element using XPath and compared for assurance to the bounding tags in a detokenized value. The markup between these outer tags is replaced with a character string representation of the tokenization results, a value of XML type `base64Binary`. This XML type is used to represent arbitrary Base64-encoded binary data in a human-readable, 'mail-safe' format. A message recipient uses the list of XPath expressions to locate the tags in an XML-instance document that contain tokenized data.

The entire document containing tokenized components and the associated manifest attribute are signed to cryptographically bind them under a digital signature. The signature prevents any part of the document or attributes from being substituted or modified without detection.

Successful signature verification provides a relying party assurance of the authenticity of the document and the manifest attribute. A tokenization manifest attribute is protected under the signature and indicates that the tokenization components of the document have integrity and confidentiality. The signature on the partially tokenized content can then be verified and its plaintext content recovered. The recovered plaintext can then be used to replace the tokenized value and surrounding tags with the recovered XML markup prior to performing XML schema validation

9 Other key management techniques

9.1 The Constructive Key Management

The Constructive Key Management (CKM) is a dynamic symmetric key management framework described in ANSI X9.69 [ANSI X9.69-2017]. It is a method of establishing a key, whereby several components of keying material, both symmetric and asymmetric type of keys, where each component is used for a specific purpose, are combined together using a mathematical function to produce an object key (content key).

Internal keys associated with CKM meet criteria established in ISO 11568. The mathematical function inherent in CKM is a two-part process for binding an analog representation like a naming convention to an asymmetric or symmetric type of key and having that part of the process used to provide integrity to a second part which includes a random number and other symmetric key functionality which together results in a content encryption key. The first part of the process establishes labels (or attributes) that define characteristics of protected data through a naming

convention to create a cryptographic enforced access control for content (as objects) resulting from the encryption action related to the content encryption key. Protecting the data and access control for the data is enforced by the framework. Data peerage can be done through combinations of logic relationships. Each encryption event results in a new random number. A key recovery capability is inherent in the framework and an administrative capability identified in ANSI X9.69 [ANSI X9.69-2017].

```
ckmRecipientInfo KEY-MANAGEMENT ::=
    { KeyConstructRecipientInfo IDENTIFIED BY id-ckm-
      recip-info }
```

is the information object that is used to populate the components of type **OtherRecipientInfo**. The object identifier value in component **oriType** is the value **id-ckm-recip-info**, and the associated type is a value in component **oriValue** of type **KeyConstructRecipientInfo** in its encoded format.

In an XML encoding, a CKM value of type **OtherRecipientInfo** is

```
<OtherRecipientInfo>
  <oriType>
    1.2.840.10060.2.1    <!-- id-ckm-recip-info -->
  </oriType>
  <oriValue>
    <KeyConstructRecipientInfo>
      <version> 1 </version>
      <did>
        <domainName>domainName</domainName>
        <domainMaintenanceLevel>1</domainMaintenanceLevel>
      </did>
      <ckmid>
        <unencrypted> 1 2 3</unencrypted>
      </ckmid>
      <ukm>some-base64-text</ukm>
      <keyConstructionAlgorithm>
        <algorithm>algorithmIdentifierOid</algorithm>
      </keyConstructionAlgorithm>
      <encryptedRandom>some-base64-text</encryptedRandom>
    </KeyConstructRecipientInfo>
  </oriValue>
</OtherRecipientInfo>
```

In this example, ukm and encryptedRandom values contain octet string values encoded in base 64. The algorithmIdentifier value contains an object identifier value.

9.2 Database Encryption Key Management

Database Encryption Key Management (DBEKM) is a key management technique defined in X9.73 [ANSI X9.73-2017] for managing symmetric keys in a database environment. This technique relies on the secure exchange of information between a secure cryptographic device (SCD) and a server. The dbekmRecipientInfo object of class KEY-MANAGEMENT is defined as follows:

```
dbekmRecipientInfo KEY-MANAGEMENT ::=
```

```
{ DBEKMRecipientInfo IDENTIFIED BY id-dbekm-recv-info
}
```

This information object that is used to populate the components of type **OtherRecipientInfo**. The object identifier value in component **oriType** is the value **id-dbekm-recv-info**, and the associated type is a value in component **oriValue** of type **DBEKMRecipientInfo** in its encoded format.

The DBEKM method is designed to support using a data encryption key in cryptographic software and using a master key in cryptographic hardware, also referred to as a secure cryptographic device (SCD) without violating the cryptographic boundary of the SCD or spoofing the SCD by presenting the SCD with a data object that is intended for use as a cryptographic key. Further, DBEKM eliminates the need of relying on a software object using a password-based key derivation function (PBDF) to protect data encryption keys.

The DBEKM method employs three key types:

1. Master Key Encryption Key (MK) used to protect the HMAC Key (HK)
2. Keyed Hash Message Authentication (HMAC) Key (HK) used to generate a seed
3. Data Encryption Key (DK) used to protect data

The DBEKM method employs four cryptographic functions:

1. Algorithms for data and key encryption, i.e. AES
2. Algorithms for Keyed Hash Message Authentication (HMAC)
3. Algorithms for hashing, e.g. Secure Hash Algorithm 2 (SHA-2), Secure Hash Algorithm 3 (SHA-3), etc.
4. Algorithms for key derivation functions (KDF), i.e. ISO/IEC 11770-6

The DBEKM method has the following benefits:

1. The cryptographic hardware generates the Master Key Encryption Key (MK) and the HMAC Key (HK)
2. The cryptographic hardware retains the MK but not the HK
3. The cryptographic hardware generates a seed per the unique database identifier
4. The cryptographic hardware does not know the KDF and cannot generate the Data Key Encryption Key (DK)
5. The server does not store any cleartext keys or passwords protecting keys
6. The server manages its own unique database identifier

The DBEKM method has the following security considerations:

1. The seed is securely transmitted from the cryptographic hardware to the database server
2. The server cryptographic software generates a DK using a KDF based on the seed
3. The server memory contains the cleartext data key (DK) for use with cryptographic software

The DBEKM method can be used for generating a DK as a Content Encryption Key (CEK), for changing a DK, using multiple DK with a single server when protecting different data elements, and using multiple DK or MK with multiple servers.

9.2.1 Single server initial data encryption key

For a single server initial data encryption key, a secure connection is first established between the SCD and the server storage. The SCD generates the MK and HK, encrypts the HK using the MK, and sends the HK cryptogram to the server over a secured channel. The server stores the HK cryptogram and at some previous or subsequent point in time generates a unique identifier (ID) that represents a meaningful database name. Meanwhile, the SCD destroys the HK but retains the MK.

To obtain a seed for the DK the server sends a request to the SCD containing the HK cryptogram and the unique ID over a secured channel. The SCD decrypts the HK using the MK, generates a seed using the HMAC algorithm with the HK and the unique ID, and sends the seed to the server over a secured channel. The server generates the DK using a KDF with the seed and installs the DK into memory for data encryption and decryption. Meanwhile, the SCD destroys the HK and the seed.

9.2.2 Single server data encryption key change

For a single server data encryption key change, a secure connection is first established between the SCD and the server storage. The server can change the DK at any time by changing the unique ID. The data was previously encrypted by sending a request to the SCD with the HK cryptogram and unique ID, generating the DK using the KDF with the seed received from the SCD, and encrypting the data using the DK in memory.

To change the DK the server decrypts the data, generates and stores a new ID, sends a request to the SCD with the HK cryptogram and the new ID, generates a new DK using the new seed, and encrypts the data using the new DK in memory. The server can decrypt with the old DK and re-encrypt with the new DK all of the data at once, or manage a gradual migration between the two DK.

The SCD only retains the MK, it destroys the old and new DK, the old and new ID, and the old and new seeds. As long as the server can manage its IDs it can manage its DKs accordingly but it cannot generate the DK without obtaining the seed from the SCD. The server can change the DK at any time by managing its ID, and if needed can recover old DK as long as it archives the HK cryptogram and the associated ID.

9.2.3 Multiple data encryption keys with single server

A single server might need multiple encryption keys to protect different information on a single server instead of using the same key for all data that needs encryption. The server can managed multiple DKs by managing multiple IDs. The SCD generates a single MK and single HK, encrypts the HK using the MK, and sends the HK cryptogram to the server over a secured channel. The server stores the HK cryptogram and at some previous or subsequent point in time the server generates multiple unique identifiers (IDs), one for each DK. Meanwhile, the SCD destroys the HK but retains the MK.

To obtain a seed s1 for the first DK1 the server sends a request to the SCD containing the HK cryptogram and the first unique ID1 over a secured channel. The SCD decrypts the HK using the MK, generates seed s1 using the HMAC algorithm with the HK and the unique ID1, and sends the seed s1 to the server over a secured channel. The server generates the first DK1 using a KDF with the seed s1 and installs the DK1 into memory for data encryption and decryption. Meanwhile, the SCD destroys the HK and the seed s1.

To obtain a seed s2 for the second DK2 the server sends a request to the SCD containing the HK cryptogram and the second unique ID2 over a secured channel. The SCD decrypts the HK using the MK, generates seed s2 using the HMAC algorithm with the HK and the unique ID2, and sends the seed s2 to the server over a secured channel. The server generates the second DK2 using a KDF with the seed s2 and installs the DK2 into memory for data encryption and decryption. Meanwhile, the SCD destroys the HK and the seed s2. When the server is restarted and the DKs are erased from memory, the server can regenerate the DKs by reacquiring the seeds at any time by resending the same HK cryptogram and unique IDs to the SCD.

9.2.4 Multiple data encryption keys with multiple servers

Multiple servers collocated within the same data-center might need a unique encryption key per server instead of sharing the same key on all servers. A secure connection is first established between the

SCD and each of the storage of each server. The SCD generates a single MK and single HK, encrypts the HK using the MK, and sends the HK cryptogram to both servers over a secured channel. The servers store the HK cryptogram and at some previous or subsequent point in time each server generates a unique identifier, ID1 for server S1 and ID2 for server S2. Meanwhile, the SCD destroys the HK but retains the MK.

To obtain a seed s1 for DK1 the server S1 sends a request to the SCD containing the HK cryptogram and the unique ID1 over a secured channel. The SCD decrypts the HK using the MK, generates seed s1 using the HMAC algorithm with the HK and the unique ID1, and sends the seed s1 to server S1 over a secured channel. The server S1 generates DK1 using a KDF with the seed s1 and installs the DK1 into memory for data encryption and decryption. Meanwhile, the SCD destroys the HK and the seed s1.

To obtain a seed s2 for DK2 the server S2 sends a request to the SCD containing the HK cryptogram and the second unique ID2 over a secured channel. The SCD decrypts the HK using the MK, generates seed s2 using the HMAC algorithm with the HK and the unique ID2, and sends the seed s2 to server S2 over a secured channel. The server generates DK2 using a KDF with the seed s2 and installs the DK2 into memory for data encryption and decryption. Meanwhile, the SCD destroys the HK and the seed s2. When any of the servers are restarted its DK is erased from memory, but the servers can regenerate their DK by reacquiring the seed at any time, by resending the same HK cryptogram and unique ID to the SCD.

9.2.5 Multiple HMAC keys with multiple servers

Multiple servers deployed regionally in different data-centers might need a unique HMAC key per server instead of sharing the same key on all servers. The SCD generates a single MK and multiple HK, one per each server, encrypts each HK using the MK, and sends each HK cryptogram to the corresponding server over a secured channel. The servers store the HK cryptogram and at some previous or subsequent point in time each server generates a unique identifier, ID1 for server S1 and ID2 for server S2. Meanwhile, the SCD destroys the HKs but retains the MK.

To obtain a seed s1 for DK1 the server S1 sends a request to the SCD containing the HK1 cryptogram and the unique ID1 over a secured channel. The SCD decrypts HK1 using the MK, generates seed s1 using the HMAC algorithm with HK1 and the unique ID1, and sends the seed s1 to server S1 over a secured channel. The server S1 generates DK1 using a KDF with the seed s1 and installs the DK1 into memory for data encryption and decryption. Meanwhile, the SCD destroys HK1 and the seed s1.

To obtain a seed s2 for DK2 the server S2 sends a request to the SCD containing the HK2 cryptogram and the second unique ID2 over a secured channel. The SCD decrypts HK2 using the MK, generates seed s2 using the HMAC algorithm with HK2 and the unique ID2, and sends the seed s2 to server S2 over a secured channel. The server generates DK2 using a KDF with the seed s2 and installs the DK2 into memory for data encryption and decryption. Meanwhile, the SCD destroys the HK and the seed s2. When any of the servers are restarted its DK is erased from memory, but the servers can regenerate the DK by reacquiring the seed at any time by resending the same HK cryptogram and unique ID to the SCD.

Annex A

ASN.1 modules

(This annex forms an integral part of this Recommendation | International Standard.)

1 Main CMS module (From RFC 6268)

```
CryptographicMessageSyntax-2010
    { iso(1) member-body(2) us(840) rsadsi(113549)
      pkcs(1) pkcs-9(9) smime(16) modules(0) id-mod-cms-2009(58) }

DEFINITIONS IMPLICIT TAGS ::=

BEGIN

IMPORTS

ParamOptions, DIGEST-ALGORITHM, SIGNATURE-ALGORITHM,
PUBLIC-KEY, KEY-DERIVATION, KEY-WRAP, MAC-ALGORITHM,
KEY-AGREE, KEY-TRANSPORT, CONTENT-ENCRYPTION, ALGORITHM,
AlgorithmIdentifier{}
FROM AlgorithmInformation-2009
{iso(1) identified-organization(3) dod(6) internet(1) security(5)
mechanisms(5) pkix(7) id-mod(0) id-mod-algorithmInformation-02(58)}

SignatureAlgs, MessageDigestAlgs, KeyAgreementAlgs,
MessageAuthAlgs, KeyWrapAlgs, ContentEncryptionAlgs,
KeyTransportAlgs, KeyDerivationAlgs, KeyAgreePublicKeys
FROM CryptographicMessageSyntaxAlgorithms-2009
{ iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-9(9)
smime(16) modules(0) id-mod-cmsalg-2001-02(37) }

Certificate, CertificateList, CertificateSerialNumber, Name, ATTRIBUTE
FROM PKIX1Explicit-2009
{ iso(1) identified-organization(3) dod(6) internet(1)
security(5) mechanisms(5) pkix(7) id-mod(0) id-mod-pkix1-explicit-02(51) }

AttributeCertificate
FROM PKIXAttributeCertificate-2009
{ iso(1) identified-organization(3) dod(6) internet(1)
security(5) mechanisms(5) pkix(7) id-mod(0) id-mod-attribute-cert-02(47) }

AttributeCertificateV1
FROM AttributeCertificateVersion1-2009
{iso(1) member-body(2) us(840) rsadsi(113549) pkcs(1) pkcs-9(9)
smime(16) modules(0) id-mod-v1AttrCert-02(49)}

CMSProfileAttributes
FROM CMSProfileAttributes
{itu-t recommendation(0) x(24) cms-profile(894) module(0)
CMSProfileAttributes(3) version1(1)}

tokenizedParts
FROM TokenizationManifest
{iso(1) identified-organization(3) tc68(133) country(16)
x9(840) x9Standards(9) x9-73(73) module(0) tokenMan(7)};

-- Cryptographic Message Syntax
-- The following are used for version numbers using the ASN.1
-- NOTE: The document reference represents where the versioned
-- feature was introduced to the module.
--
-- idiom "[n:"
-- Version 1 = PKCS #7
-- Version 2 = S/MIME V2
-- Version 3 = RFC 2630
-- Version 4 = RFC 3369
-- Version 5 = RFC 3852
```

```

CONTENT-TYPE ::= CLASS {
    &id          OBJECT IDENTIFIER UNIQUE,
    &Type         OPTIONAL
} WITH SYNTAX {
    [TYPE &Type] IDENTIFIED BY &id
}

ContentType ::= CONTENT-TYPE.&id

ContentInfo ::= SEQUENCE {
    contentType CONTENT-TYPE.&id({ContentSet}),
    content      [0] EXPLICIT CONTENT-TYPE.&Type({ContentSet}{@contentType})
}

ContentSet CONTENT-TYPE ::= {
    -- Define the set of content types to be recognized.
    ct-Data | ct-SignedData | ct-EncryptedData | ct-EnvelopedData |
    ct-AuthenticatedData | ct-DigestedData, ... }

SignedData ::= SEQUENCE {
    version          CMSVersion,
    digestAlgorithms SET OF DigestAlgorithmIdentifier,
    encapContentInfo EncapsulatedContentInfo,
    certificates      [0] IMPLICIT CertificateSet OPTIONAL,
    crls              [1] IMPLICIT RevocationInfoChoices OPTIONAL,
    signerInfos       SignerInfos }

SignerInfos ::= SET OF SignerInfo

EncapsulatedContentInfo ::= SEQUENCE {
    eContentType     CONTENT-TYPE.&id({ContentSet}),
    eContent          [0] EXPLICIT OCTET STRING
    (CONTAINING CONTENT-TYPE.&Type({ContentSet}{@eContentType})) OPTIONAL }

SignerInfo ::= SEQUENCE {
    version          CMSVersion,
    sid              SignerIdentifier,
    digestAlgorithm   DigestAlgorithmIdentifier,
    signedAttrs       [0] IMPLICIT SignedAttributes OPTIONAL,
    signatureAlgorithm SignatureAlgorithmIdentifier,
    signature         SignatureValue,
    unsignedAttrs     [1] IMPLICIT Attributes({UnsignedAttributes}) OPTIONAL
}

SignedAttributes ::= Attributes {{ SignedAttributesSet }}

SignerIdentifier ::= CHOICE {
    issuerAndSerialNumber IssuerAndSerialNumber,
    ...,
    [[3: subjectKeyIdentifier [0] SubjectKeyIdentifier ]] }

SignedAttributesSet ATTRIBUTE ::=
    { aa-signingTime | aa-messageDigest | aa-contentType, |
      CMSProfileAttributes, tokenizedParts ... }

UnsignedAttributes ATTRIBUTE ::= { aa-countersignature, ... }

SignatureValue ::= OCTET STRING

EnvelopedData ::= SEQUENCE {
    version          CMSVersion,
    originatorInfo    [0] IMPLICIT OriginatorInfo OPTIONAL,
    recipientInfos     RecipientInfos,
    encryptedContentInfo EncryptedContentInfo,
    ...,
    [[2: unprotectedAttrs [1] IMPLICIT Attributes
        {{ UnprotectedEnvAttributes }} OPTIONAL ]] }

OriginatorInfo ::= SEQUENCE {
    certs            [0] IMPLICIT CertificateSet OPTIONAL,
    crls             [1] IMPLICIT RevocationInfoChoices OPTIONAL }

RecipientInfos ::= SET SIZE (1..MAX) OF RecipientInfo

```

```

EncryptedContentInfo ::=
    EncryptedContentInfoType { ContentEncryptionAlgorithmIdentifier }

EncryptedContentInfoType { AlgorithmIdentifierType } ::= SEQUENCE {
    contentType                CONTENT-TYPE.&id({ContentSet}),
    contentEncryptionAlgorithm  AlgorithmIdentifierType,
    encryptedContent            [0] IMPLICIT OCTET STRING OPTIONAL }

-- If you want to do constraints, you might use:
-- EncryptedContentInfo ::= SEQUENCE {
--     contentType                CONTENT-TYPE.&id({ContentSet}),
--     contentEncryptionAlgorithm  ContentEncryptionAlgorithmIdentifier,
--     encryptedContent            [0] IMPLICIT ENCRYPTED {CONTENT-TYPE.
--     &Type({ContentSet}){@contentType}} OPTIONAL }

-- ENCRYPTED {ToBeEncrypted} ::= OCTET STRING ( CONSTRAINED BY { ToBeEncrypted } )

UnprotectedEnvAttributes ATTRIBUTE ::= { ... }
UnprotectedEncAttributes ATTRIBUTE ::= { ... }

RecipientInfo ::= CHOICE {
    ktri                KeyTransRecipientInfo,
    ...,
    [[3: kari            [1] KeyAgreeRecipientInfo ]],
    [[4: kekri            [2] KEKRecipientInfo]],
    [[5: pwri            [3] PasswordRecipientInfo,
    ori                [4] OtherRecipientInfo ]] }

EncryptedKey ::= OCTET STRING

KeyTransRecipientInfo ::= SEQUENCE {
    version                CMSVersion, -- always set to 0 or 2
    rid                    RecipientIdentifier,
    keyEncryptionAlgorithm AlgorithmIdentifier
        {KEY-TRANSPORT, {KeyTransportAlgorithmSet}},
    encryptedKey            EncryptedKey }

KeyTransportAlgorithmSet KEY-TRANSPORT ::= { KeyTransportAlgs, ... }

RecipientIdentifier ::= CHOICE {
    issuerAndSerialNumber IssuerAndSerialNumber,
    ...,
    [[2: subjectKeyIdentifier [0] SubjectKeyIdentifier ]] }

KeyAgreeRecipientInfo ::= SEQUENCE {
    version                CMSVersion, -- always set to 3
    originator              [0] EXPLICIT OriginatorIdentifierOrKey,
    ukm                    [1] EXPLICIT UserKeyingMaterial OPTIONAL,
    keyEncryptionAlgorithm AlgorithmIdentifier
        {KEY-AGREE, {KeyAgreementAlgorithmSet}},
    recipientEncryptedKeys  RecipientEncryptedKeys }

KeyAgreementAlgorithmSet KEY-AGREE ::= { KeyAgreementAlgs, ... }

OriginatorIdentifierOrKey ::= CHOICE {
    issuerAndSerialNumber IssuerAndSerialNumber,
    subjectKeyIdentifier  [0] SubjectKeyIdentifier,
    originatorKey          [1] OriginatorPublicKey }

OriginatorPublicKey ::= SEQUENCE {
    algorithm  AlgorithmIdentifier {PUBLIC-KEY, {OriginatorKeySet}},
    publicKey  BIT STRING }

OriginatorKeySet PUBLIC-KEY ::= { KeyAgreePublicKeys, ... }

RecipientEncryptedKeys ::= SEQUENCE OF RecipientEncryptedKey

RecipientEncryptedKey ::= SEQUENCE {
    rid            KeyAgreeRecipientIdentifier,
    encryptedKey  EncryptedKey }

KeyAgreeRecipientIdentifier ::= CHOICE {
    issuerAndSerialNumber IssuerAndSerialNumber,

```

```

        rKeyId                [0] IMPLICIT RecipientKeyIdentifier }

RecipientKeyIdentifier ::= SEQUENCE {
    subjectKeyIdentifier      SubjectKeyIdentifier,
    date                      GeneralizedTime OPTIONAL,
    other                     OtherKeyAttribute OPTIONAL }

SubjectKeyIdentifier ::= OCTET STRING

KEKRecipientInfo ::= SEQUENCE {
    version                  CMSVersion, -- always set to 4
    kekid                    KEKIdentifier,
    keyEncryptionAlgorithm   KeyEncryptionAlgorithmIdentifier,
    encryptedKey              EncryptedKey }

KEKIdentifier ::= SEQUENCE {
    keyIdentifier             OCTET STRING,
    date                      GeneralizedTime OPTIONAL,
    other                     OtherKeyAttribute OPTIONAL }

PasswordRecipientInfo ::= SEQUENCE {
    version                  CMSVersion, -- always set to 0
    keyDerivationAlgorithm   [0] KeyDerivationAlgorithmIdentifier
        OPTIONAL,
    keyEncryptionAlgorithm   KeyEncryptionAlgorithmIdentifier,
    encryptedKey              EncryptedKey }

OTHER-RECIPIENT ::= TYPE-IDENTIFIER

OtherRecipientInfo ::= SEQUENCE {
    oriType      OTHER-RECIPIENT.&id({SupportedOtherRecipInfo}),
    oriValue     OTHER-RECIPIENT.&Type({SupportedOtherRecipInfo}{@oriType})}

SupportedOtherRecipInfo OTHER-RECIPIENT ::= { ... }

DigestedData ::= SEQUENCE {
    version                  CMSVersion,
    digestAlgorithm          DigestAlgorithmIdentifier,
    encapContentInfo         EncapsulatedContentInfo,
    digest                   Digest, ... }

Digest ::= OCTET STRING

EncryptedData ::= SEQUENCE {
    version                  CMSVersion,
    encryptedContentInfo     EncryptedContentInfo,
    .../
    [[2: unprotectedAttrs    [1] IMPLICIT Attributes
        {{UnprotectedEncAttributes}} OPTIONAL ]] }

AuthenticatedData ::= SEQUENCE {
    version                  CMSVersion,
    originatorInfo           [0] IMPLICIT OriginatorInfo OPTIONAL,
    recipientInfos           RecipientInfos,
    macAlgorithm             MessageAuthenticationCodeAlgorithm,
    digestAlgorithm          [1] DigestAlgorithmIdentifier OPTIONAL,
    encapContentInfo         EncapsulatedContentInfo,
    authAttrs                [2] IMPLICIT AuthAttributes OPTIONAL,
    mac                      MessageAuthenticationCode,
    unauthAttrs              [3] IMPLICIT UnauthAttributes OPTIONAL }

AuthAttributes ::= SET SIZE (1..MAX) OF Attribute {{AuthAttributeSet}}

AuthAttributeSet ATTRIBUTE ::= { aa-contentType | aa-messageDigest | aa-signingTime, ...}

MessageAuthenticationCode ::= OCTET STRING

UnauthAttributes ::= SET SIZE (1..MAX) OF Attribute {{UnauthAttributeSet}}

UnauthAttributeSet ATTRIBUTE ::= {...}

--
-- General algorithm definitions
--

```