Stream:	Internet Engineering Task Force (IETF)			
RFC:	9580			
Obsoletes:	4880, 5581, 6637			
Category:	Standards Track			
Published:	May 2024			
ISSN:	2070-1721			
Authors:	P. Wouters, Ed. *Aiven*	D. Huigens *Proton AG*	J. Winter *Sequoia-PGP*	Y. Niibe *FSIJ*

RFC 9580
OpenPGP

Abstract

This document specifies the message formats used in OpenPGP. OpenPGP provides encryption with public-key or symmetric cryptographic algorithms, digital signatures, compression, and key management.

This document is maintained in order to publish all necessary information needed to develop interoperable applications based on the OpenPGP format. It is not a step-by-step cookbook for writing an application. It describes only the format and methods needed to read, check, generate, and write conforming packets crossing any network. It does not deal with storage and implementation questions. It does, however, discuss implementation issues necessary to avoid security flaws.

This document obsoletes RFCs 4880 ("OpenPGP Message Format"), 5581 ("The Camellia Cipher in OpenPGP"), and 6637 ("Elliptic Curve Cryptography (ECC) in OpenPGP").

Status of This Memo

This is an Internet Standards Track document.

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

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at https://www.rfc-editor.org/info/rfc9580.

Copyright Notice

Table of Contents

1. Introduction

This document provides information on the message-exchange packet formats used by OpenPGP to provide encryption, decryption, signing, and key management functions. It is a revision of [RFC4880] ("OpenPGP Message Format"), which is a revision of [RFC2440], which itself replaces [RFC1991] ("PGP Message Exchange Formats").

This document obsoletes [RFC4880] (OpenPGP), [RFC5581] (Camellia in OpenPGP), and [RFC6637] (Elliptic Curves in OpenPGP). At the time of writing, this document incorporates all outstanding verified errata, which are listed in Appendix C.

Software that has already implemented those previous specifications may want to review Appendix B for pointers to what has changed.

1.1. Terms

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.

The key words "Private Use", "Specification Required", and "RFC Required" that appear in this document when used to describe namespace allocation are to be interpreted as described in [RFC8126].

Some terminology used in this document has been improved from previous versions of the OpenPGP specification. See Appendix B.1 for more details.

2. General Functions

OpenPGP provides data confidentiality and integrity for messages and data files by using public-key and/or symmetric encryption and digital signatures. It provides formats for encoding and transferring encrypted and/or signed messages. In addition, OpenPGP provides functionality for encoding and transferring keys and certificates, though key storage and management are beyond the scope of this document.

2.1. Confidentiality via Encryption

OpenPGP combines symmetric-key encryption and (optionally) public-key encryption to provide confidentiality. When using public keys, first the object is encrypted using a symmetric encryption algorithm. Each symmetric key is used only once, for a single object. A new "session key" is generated as a random number for each object (sometimes referred to as a "session"). Since it is used only once, the session key is bound to the message and transmitted with it. To protect the key, it is encrypted with the receiver's public key. The sequence is as follows:

1. The sender creates a message.
2. The sending OpenPGP implementation generates a random session key for this message.
3. The session key is encrypted using each recipient's public key. These "encrypted session keys" start the message.
4. The sending OpenPGP implementation optionally compresses the message and then encrypts it using a message key derived from the session key. The encrypted message forms the remainder of the OpenPGP message.
5. The receiving OpenPGP implementation decrypts the session key using the recipient's private key.
6. The receiving OpenPGP implementation decrypts the message using the message key derived from the session key. If the message was compressed, it will be decompressed.

When using symmetric-key encryption, a similar process as described above is used, but the session key is encrypted with a symmetric algorithm derived from a shared secret.

Both digital signature and confidentiality services may be applied to the same message. First, a signature is generated for the message and attached to the message. Then, the message plus signature is encrypted using a symmetric message key derived from the session key. Finally, the session key is encrypted using public-key encryption and prefixed to the encrypted block.

2.2. Authentication via Digital Signature

The digital signature uses a cryptographic hash function and a public-key signature algorithm. The sequence is as follows:

1. The sender creates a message.
2. The sending implementation generates a hash digest of the message.
3. The sending implementation generates a signature from the hash digest using the sender's private key.
4. The signature is attached to or transmitted alongside the message.
5. The receiving implementation obtains a copy of the message and the message signature.
6. The receiving implementation generates a new hash digest for the received message and verifies it using the message's signature. If the verification is successful, the message is accepted as authentic.

2.3. Compression

An OpenPGP implementation **MAY** support the compression of data. Many existing OpenPGP messages are compressed. Implementers, such as those working on constrained implementations that do not want to support compression, might want to consider at least implementing decompression.

2.4. Conversion to Base64

OpenPGP's underlying native representation for encrypted messages, signatures, keys, and certificates is a stream of arbitrary octets. Some systems only permit the use of blocks consisting of seven-bit, printable text. For transporting OpenPGP's native raw binary octets through channels that are not safe to transport raw binary data, a printable encoding of these binary octets is defined. The raw 8-bit binary octet stream can be converted to a stream of printable ASCII characters using base64 encoding in a format called "ASCII Armor" (see Section 6).

Implementations **SHOULD** support base64 conversions.

2.5. Signature-Only Applications

OpenPGP is designed for applications that use both encryption and signatures, but there are a number of use cases that only require a signature-only implementation. Although this specification requires both encryption and signatures, it is reasonable for there to be subset implementations that are non-conformant only in that they omit encryption support.

3. Data Element Formats

This section describes the data elements used by OpenPGP.

3.1. Scalar Numbers

Scalar numbers are unsigned and always stored in big-endian format. Using n[k] to refer to the kth octet being interpreted, the value of a two-octet scalar is ((n[0] << 8) + n[1]). The value of a four-octet scalar is ((n[0] << 24) + (n[1] << 16) + (n[2] << 8) + n[3]).

3.2. Multiprecision Integers

Multiprecision Integers (MPIs) are unsigned integers used to hold large integers such as the ones used in cryptographic calculations.

An MPI consists of two pieces: a two-octet scalar that is the length of the MPI in bits, followed by a string of octets that contain the actual integer.

These octets form a big-endian number; a big-endian number can be made into an MPI by prefixing it with the appropriate length.

Examples:

(Note that all numbers in the octet strings identified by square brackets are in hexadecimal.)

> The string of octets [00 00] forms an MPI with the value 0.
>
> The string of octets [00 01 01] forms an MPI with the value 1.
>
> The string [00 09 01 FF] forms an MPI with the value 511.

Additional rules:

- The size of an MPI is ((MPI.length + 7) / 8) + 2 octets.
- The length field of an MPI describes the length starting from its most significant non-zero bit. Thus, the MPI [00 02 01] is not formed correctly. It should be [00 01 01]. When parsing an MPI in a v6 Key, Signature, or Public-Key Encrypted Session Key (PKESK) packet, the implementation **MUST** check that the encoded length matches the length starting from the most significant non-zero bit; if it doesn't match, reject the packet as malformed.
- Unused bits of an MPI **MUST** be zero.

3.2.1. Using MPIs to Encode Other Data

Note that in some places, MPIs are used to encode non-integer data, such as an elliptic curve (EC) point (see Section 11.2) or an octet string of known, fixed length (see Section 11.3). The wire representation is the same: two octets of length in bits counted from the first non-zero bit, followed by the smallest series of octets that can represent the value while stripping off any leading zero octets.

3.3. Key IDs and Fingerprints

A Key ID is an eight-octet scalar that identifies a key. Implementations **SHOULD NOT** assume that Key IDs are unique. A fingerprint is more likely to be unique than a key ID. The fingerprint and key ID of a key are calculated differently according to the version of the key.

Section 5.5.4 describes how Key IDs and Fingerprints are formed.

3.4. Text

Unless otherwise specified, the character set for text is the UTF-8 [RFC3629] encoding of Unicode [ISO10646].

3.5. Time Fields

A time field is an unsigned four-octet number containing the number of seconds elapsed since midnight, 1 January 1970 UTC.

3.6. Keyrings

A keyring is a collection of one or more keys in a file or database. Traditionally, a keyring is simply a sequential list of keys, but it may be any suitable database. It is beyond the scope of this specification to discuss the details of keyrings or other databases.

3.7. String-to-Key (S2K) Specifier

A string-to-key (S2K) specifier type is used to convert a passphrase string into a symmetric-key encryption/decryption key. Passphrases requiring use of S2K conversion are currently used in two places: to encrypt the secret part of private keys and for symmetrically encrypted messages.

3.7.1. S2K Specifier Types

There are four types of S2K Specifier Types currently specified and some reserved values:

ID	S2K Type	S2K Field Size (Octets)	Generate?	Reference
0	Simple S2K	2	No	[RFC9580], Section 3.7.1.1
1	Salted S2K	10	Only when string is high entropy	[RFC9580], Section 3.7.1.2
2	Reserved value	-	No	[RFC9580]
3	Iterated and Salted S2K	11	Yes	[RFC9580], Section 3.7.1.3

ID	S2K Type	S2K Field Size (Octets)	Generate?	Reference
4	Argon2	20	Yes	[RFC9580], Section 3.7.1.4
100-110	Private/ Experimental S2K	-	As appropriate	[RFC9580]

Table 1: OpenPGP String-to-Key (S2K) Types Registry

The S2K Specifier Types are described in the subsections below. If "Yes" is not present in the "Generate?" column, the S2K entry is used only for reading in backward-compatibility mode and **SHOULD NOT** be used to generate new output.

3.7.1.1. Simple S2K

Simple S2K directly hashes the string to produce the key data. This hashing is done as shown below.

```
Octet 0:        0x00
Octet 1:        hash algorithm
```

Simple S2K hashes the passphrase to produce the session key. The manner in which this is done depends on the size of the session key (which depends on the cipher the session key will be used with) and the size of the hash algorithm's output. If the hash size is greater than the session key size, the high-order (leftmost) octets of the hash are used as the key.

If the hash size is less than the key size, multiple instances of the hash context are created -- enough to produce the required key data. These instances are preloaded with 0, 1, 2, ... octets of zeros (that is, the first instance has no preloading, the second gets preloaded with 1 octet of zero, the third is preloaded with two octets of zeros, and so forth).

As the data is hashed, it is given independently to each hash context. Since the contexts have been initialized differently, they will each produce a different hash output. Once the passphrase is hashed, the output data from the multiple hashes is concatenated, first hash leftmost, to produce the key data, and any excess octets on the right are discarded.

3.7.1.2. Salted S2K

Salted S2K includes a "salt" value in the S2K specifier -- some arbitrary data -- that gets hashed along with the passphrase string to help prevent dictionary attacks.

```
Octet 0:        0x01
Octet 1:        hash algorithm
Octets 2-9:     8-octet salt value
```

Salted S2K is exactly like Simple S2K, except that the input to the hash function(s) consists of the 8 octets of salt from the S2K specifier, followed by the passphrase.

3.7.1.3. Iterated and Salted S2K

Iterated and Salted S2K includes both a salt and an octet count. The salt is combined with the passphrase, and the resulting value is repeated and then hashed. This further increases the amount of work an attacker must do to try dictionary attacks.

```
Octet  0:        0x03
Octet  1:        hash algorithm
Octets 2-9:      8-octet salt value
Octet  10:       count; a one-octet coded value
```

The count is coded into a one-octet number using the following formula:

```
#define EXPBIAS 6
    count = ((Int32)16 + (c & 15)) << ((c >> 4) + EXPBIAS);
```

The above formula is described in [C99], where "Int32" is a type for a 32-bit integer, and the variable "c" is the coded count, Octet 10.

Iterated and Salted S2K hashes the passphrase and salt data multiple times. The total number of octets to be hashed is specified in the encoded count in the S2K specifier. Note that the resulting count value is an octet count of how many octets will be hashed, not an iteration count.

Initially, one or more hash contexts are set up as with the other S2K algorithms, depending on how many octets of key data are needed. Then the salt, followed by the passphrase data, is repeatedly processed as input to each hash context until the number of octets specified by the octet count has been hashed. The input is truncated to the octet count, except if the octet count is less than the initial isize of the salt plus passphrase. That is, at least one copy of the full salt plus passphrase will be provided as input to each hash context regardless of the octet count. After the hashing is done, the key data is produced from the hash digest(s) as with the other S2K algorithms.

3.7.1.4. Argon2

This S2K method hashes the passphrase using Argon2, as specified in [RFC9106]. This provides memory hardness, further protecting the passphrase against brute-force attacks.

```
Octet  0:        0x04
Octets 1-16:     16-octet salt value
Octet  17:       one-octet number of passes t
Octet  18:       one-octet degree of parallelism p
Octet  19:       one-octet encoded_m, specifying the exponent of
                    the memory size
```

The salt **SHOULD** be unique for each passphrase.

The number of passes t and the degree of parallelism p **MUST** be non-zero.

The memory size m is 2**encoded_m kibibytes (KiB) of RAM. The encoded memory size **MUST** be a value from 3+ceil(log_2(p)) to 31, such that the decoded memory size m is a value from 8*p to 2**31. Note that memory-hardness size is indicated in KiB, not octets.

Argon2 is invoked with the passphrase as P, the salt as S, the values of t, p, and, m as described above, the required key size as the tag length T, 0x13 as the version v, and Argon2id as the type.

For the recommended values of t, p, and m, see Section 4 of [RFC9106]. If the recommended value of m for a given application is not a power of 2, it is **RECOMMENDED** to round up to the next power of 2 if the resulting performance would be acceptable; otherwise, round down (keeping in mind that m must be at least 8*p).

As an example, with the first recommended option (t=1, p=4, m=2**21), the full S2K specifier would be:

```
04 XX XX XX XX XX XX XX XX XX XX XX XX XX XX XX
XX 01 04 15
```

where XX represents a random octet of salt.

3.7.2. S2K Usage

Simple S2K and Salted S2K specifiers can be brute-forced when used with a low-entropy string, such as those typically provided by users. In addition, the usage of Simple S2K can lead to key and initialization vector (IV) reuse (see Section 5.3). Therefore, when generating an S2K specifier, an implementation **MUST NOT** use Simple S2K. Furthermore, an implementation **SHOULD NOT** generate a Salted S2K unless the implementation knows that the input string is high entropy (for example, it generated the string itself using a known good source of randomness).

It is **RECOMMENDED** that implementations use Argon2. If Argon2 is not available, Iterated and Salted S2K **MAY** be used if care is taken to use a high octet count and a strong passphrase. However, this method does not provide memory hardness, unlike Argon2.

3.7.2.1. Secret-Key Encryption

The first octet following the public key material in a secret key packet (Section 5.5.3) indicates whether and how the secret key material is passphrase protected. This first octet is known as the "S2K usage octet".

If the S2K usage octet is zero, the secret key data is unprotected. If it is non-zero, it describes how to use a passphrase to unlock the secret key.

Implementations predating [RFC2440] indicated a protected key by storing a symmetric cipher algorithm ID (see Section 9.3) in the S2K usage octet. In this case, the MD5 hash function was always used to convert the passphrase to a key for the specified cipher algorithm.

Later implementations indicate a protected secret key by storing special value 253 (AEAD), 254 (CFB), or 255 (MalleableCFB) in the S2K usage octet. The S2K usage octet is then followed immediately by a set of fields that describe how to convert a passphrase to a symmetric key that can unlock the secret material, plus other parameters relevant to the type of encryption used.

The wire format fields also differ based on the version of the enclosing OpenPGP packet. The table below, indexed by the S2K usage octet, summarizes the specifics described in Section 5.5.3.

In the table below, `check(x)` means the "2-octet checksum", which is the sum of all octets in x mod 65536. The `info` and `packetprefix` parameters are described in detail in Section 5.5.3. Note that the "Generated?" column header has been shortened to "Gen?" here.

S2K Usage Octet	Shorthand	Encryption Parameter Fields	Encryption	Gen?
0	Unprotected	-	**v3 or v4 keys:** [cleartext secrets || check(secrets)] **v6 keys:** [cleartext secrets]	Yes
Known symmetric cipher algo ID (see Section 9.3)	LegacyCFB	IV	CFB(MD5(passphrase), secrets || check(secrets))	No
253	AEAD	params-length (**v6-only**), cipher-algo, AEAD-mode, S2K-specifier-length (**v6-only**),S2K-specifier, nonce	AEAD(HKDF(S2K(passphrase), info), secrets, packetprefix)	Yes
254	CFB	params-length (**v6-only**), cipher-algo, S2K-specifier-length (**v6-only**), S2K-specifier, IV	CFB(S2K(passphrase), secrets || SHA1(secrets))	Yes
255	MalleableCFB	cipher-algo, S2K-specifier, IV	CFB(S2K(passphrase), secrets || check(secrets))	No

Table 2: OpenPGP Secret Key Encryption (S2K Usage Octet) Registry

When emitting a secret key (with or without passphrase protection), an implementation **MUST** only produce data from a row with "Generate?" marked as "Yes". Each row with "Generate?" marked as "No" is described for backward compatibility (for reading v4 and earlier keys only) and **MUST NOT** be used to generate new output. Version 6 secret keys using these formats **MUST** be rejected.

Note that compared to a version 4 secret key, the parameters of a passphrase-protected version 6 secret key are stored with an additional pair of length counts, each of which is one octet wide.

Argon2 is only used with Authenticated Encryption with Associated Data (AEAD) (S2K usage octet 253). An implementation **MUST NOT** create and **MUST** reject as malformed any secret key packet where the S2K usage octet is not AEAD (253) and the S2K specifier type is Argon2.

3.7.2.2. Symmetric-Key Message Encryption

OpenPGP can create a Symmetric-key Encrypted Session Key (SKESK) packet at the front of a message. This is used to allow S2K specifiers to be used for the passphrase conversion or to create messages with a mix of SKESK packets and PKESK packets. This allows a message to be decrypted with either a passphrase or a public-key pair.

Implementations predating [RFC2440] always used the International Data Encryption Algorithm (IDEA) with Simple S2K conversion when encrypting a message with a symmetric algorithm; see Section 5.7. IDEA **MUST NOT** be generated but **MAY** be consumed for backward compatibility.

4. Packet Syntax

This section describes the packets used by OpenPGP.

4.1. Overview

An OpenPGP message is constructed from a number of records that are traditionally called packets. A packet is a chunk of data that has a type ID specifying its meaning. An OpenPGP message, keyring, certificate, detached signature, and so forth consists of a number of packets. Some of those packets may contain other OpenPGP packets (for example, a compressed data packet, when uncompressed, contains OpenPGP packets).

Each packet consists of a packet header, followed by the packet body. The packet header is of variable length.

When handling a stream of packets, the length information in each packet header is the canonical source of packet boundaries. An implementation handling a packet stream that wants to find the next packet **MUST** look for it at the precise offset indicated in the previous packet header.

Additionally, some packets contain internal length indicators (for example, a subfield within the packet). In the event that a subfield length indicator within a packet implies inclusion of octets outside the range indicated in the packet header, a parser **MUST** abort without writing outside the indicated range and **MUST** treat the packet as malformed and unusable.

An implementation **MUST NOT** interpret octets outside the range indicated in the packet header as part of the contents of the packet.

4.2. Packet Headers

The first octet of the packet denotes the format of the rest of the header, and it encodes the Packet Type ID, indicating the type of the packet (see Section 5). The remainder of the packet header is the length of the packet.

There are two packet formats: 1) the (current) OpenPGP packet format specified by this document and its predecessors [RFC4880] and [RFC2440] and 2) the Legacy packet format as used by implementations predating any IETF specification of the protocol.

Note that the most significant bit is the leftmost bit, called "bit 7". A mask for this bit is 0x80 in hexadecimal.

```
                        +---------------+
Encoded Packet Type ID: |7 6 5 4 3 2 1 0|
                        +---------------+
OpenPGP format:
  Bit 7 -- always one
  Bit 6 -- always one
  Bits 5 to 0 -- packet type ID

Legacy format:
  Bit 7 -- always one
  Bit 6 -- always zero
  Bits 5 to 2 -- packet type ID
  Bits 1 to 0 -- length-type
```

Bit 6 of the first octet of the packet header indicates whether the packet is encoded in the OpenPGP or Legacy packet format. The Legacy packet format **MAY** be used when consuming packets to facilitate interoperability and accessing archived data. The Legacy packet format **SHOULD NOT** be used to generate new data, unless the recipient is known to only support the Legacy packet format. This latter case is extremely unlikely, as the Legacy packet format was obsoleted by [RFC2440] in 1998.

An implementation that consumes and redistributes pre-existing OpenPGP data (such as Transferable Public Keys) may encounter packets framed with the Legacy packet format. Such an implementation **MAY** either redistribute these packets in their Legacy format or transform them to the current OpenPGP packet format before redistribution.

Note that Legacy format headers only have 4 bits for the packet type ID and hence can only encode packet type IDs less than 16, whereas the OpenPGP format headers can encode IDs as great as 63.

4.2.1. OpenPGP Format Packet Lengths

OpenPGP format packets have four possible ways of encoding length:

1. A one-octet Body Length header encodes packet lengths of up to 191 octets.
2. A two-octet Body Length header encodes packet lengths of 192 to 8383 octets.
3. A five-octet Body Length header encodes packet lengths of up to 4,294,967,295 (0xFFFFFFFF) octets in length. (This actually encodes a four-octet scalar number.)
4. When the length of the packet body is not known in advance by the issuer, Partial Body Length headers encode a packet of indeterminate length, effectively making it a stream.

4.2.1.1. One-Octet Lengths

A one-octet Body Length header encodes a length of 0 to 191 octets. This type of length header is recognized because the one-octet value is less than 192. The body length is equal to:

```
bodyLen = 1st_octet;
```

4.2.1.2. Two-Octet Lengths

A two-octet Body Length header encodes a length of 192 to 8383 octets. It is recognized because its first octet is in the range 192 to 223. The body length is equal to:

```
bodyLen = ((1st_octet - 192) << 8) + (2nd_octet) + 192
```

4.2.1.3. Five-Octet Lengths

A five-octet Body Length header consists of a single octet holding the value 255, followed by a four-octet scalar. The body length is equal to:

```
bodyLen = (2nd_octet << 24) | (3rd_octet << 16) |
          (4th_octet << 8)  | 5th_octet
```

This basic set of one, two, and five-octet lengths is also used internally to some packets.

4.2.1.4. Partial Body Lengths

A Partial Body Length header is one octet long and encodes the length of only part of the data packet. This length is a power of 2, from 1 to 1,073,741,824 (2 to the 30th power). It is recognized by its one-octet value that is greater than or equal to 224, and less than 255. The Partial Body Length is equal to:

```
partialBodyLen = 1 << (1st_octet & 0x1F);
```

Each Partial Body Length header is followed by a portion of the packet body data; the Partial Body Length header specifies this portion's length. Another length header (one octet, two octets, five octets, or partial) follows that portion. The last length header in the packet **MUST NOT** be a Partial Body Length header. Partial Body Length headers may only be used for the non-final parts of the packet.

Note also that the last Body Length header can be a zero-length header.

An implementation **MAY** use Partial Body Lengths for data packets, whether they are literal, compressed, or encrypted. The first partial length **MUST** be at least 512 octets long. Partial Body Lengths **MUST NOT** be used for any other packet types.

4.2.2. Legacy Format Packet Lengths

A zero in bit 6 of the first octet of the packet indicates a Legacy packet format. Bits 1 and 0 of the first octet of a Legacy packet are the "length-type" field. The meaning of length-type in Legacy format packets is as follows:

0 The packet has a one-octet length. The header is 2 octets long.

1 The packet has a two-octet length. The header is 3 octets long.

2 The packet has a four-octet length. The header is 5 octets long.

3 The packet is of indeterminate length. The header is 1 octet long, and the implementation must determine how long the packet is. If the packet is in a file, it means that the packet extends until the end of the file. The OpenPGP format headers have a mechanism for precisely encoding data of indeterminate length. An implementation **MUST NOT** generate a Legacy format packet with indeterminate length. An implementation **MAY** interpret an indeterminate length Legacy format packet in order to deal with historic data or data generated by a legacy system that predates support for [RFC2440].

4.2.3. Packet Length Examples

These examples show ways that OpenPGP format packets might encode the packet body lengths.

- A packet body with length 100 may have its length encoded in one octet: 0x64. This is followed by 100 octets of data.
- A packet body with length 1723 may have its length encoded in two octets: 0xC5, 0xFB. This header is followed by the 1723 octets of data.
- A packet body with length 100000 may have its length encoded in five octets: 0xFF, 0x00, 0x01, 0x86, 0xA0.

It might also be encoded in the following octet stream:

- 0xEF, first 32768 octets of data;
- 0xE1, next two octets of data;
- 0xE0, next one octet of data;

- 0xF0, next 65536 octets of data; and
- 0xC5, 0xDD, last 1693 octets of data.

This is just one possible encoding, and many variations are possible on the size of the Partial Body Length headers, as long as a regular Body Length header encodes the last portion of the data.

Please note that in all of these explanations, the total length of the packet is the length of the header(s) plus the length of the body.

4.3. Packet Criticality

The Packet Type ID space is partitioned into critical packets and non-critical packets. If an implementation encounters a critical packet where the packet type is unknown in a packet sequence, it **MUST** reject the whole packet sequence (see Section 10). On the other hand, an unknown non-critical packet **MUST** be ignored.

Packets with Type IDs from 0 to 39 are critical. Packets with Type IDs from 40 to 63 are non-critical.

5. Packet Types

The defined packet types are as follows:

ID	Critical	Packet Type Description	Shorthand	Reference
0	Yes	Reserved - a packet **MUST NOT** have this packet type ID		[RFC9580]
1	Yes	Public-Key Encrypted Session Key Packet	PKESK	[RFC9580], Section 5.1
2	Yes	Signature Packet	SIG	[RFC9580], Section 5.2
3	Yes	Symmetric-Key Encrypted Session Key Packet	SKESK	[RFC9580], Section 5.3
4	Yes	One-Pass Signature Packet	OPS	[RFC9580], Section 5.4
5	Yes	Secret-Key Packet	SECKEY	[RFC9580], Section 5.5.1.3
6	Yes	Public-Key Packet	PUBKEY	[RFC9580], Section 5.5.1.1

ID	Critical	Packet Type Description	Shorthand	Reference
7	Yes	Secret-Subkey Packet	SECSUBKEY	[RFC9580], Section 5.5.1.4
8	Yes	Compressed Data Packet	COMP	[RFC9580], Section 5.6
9	Yes	Symmetrically Encrypted Data Packet	SED	[RFC9580], Section 5.6
10	Yes	Marker Packet	MARKER	[RFC9580], Section 5.8
11	Yes	Literal Data Packet	LIT	[RFC9580], Section 5.8
12	Yes	Trust Packet	TRUST	[RFC9580], Section 5.10
13	Yes	User ID Packet	UID	[RFC9580], Section 5.11
14	Yes	Public-Subkey Packet	PUBSUBKEY	[RFC9580], Section 5.5.1.2
17	Yes	User Attribute Packet	UAT	[RFC9580], Section 5.12
18	Yes	Symmetrically Encrypted and Integrity Protected Data Packet	SEIPD	[RFC9580], Section 5.13
19	Yes	Reserved (formerly Modification Detection Code Packet)		[RFC9580], Section 5.13.1
20	Yes	Reserved		[RFC9580]
21	Yes	Padding Packet	PADDING	[RFC9580], Section 5.14
22-39	Yes	Unassigned Critical Packet		
40-59	No	Unassigned Non-Critical Packet		
60-63	No	Private or Experimental Values		[RFC9580]

Table 3: OpenPGP Packet Types Registry

The labels in the "Shorthand" column are used for compact reference elsewhere in this document, and they may also be used by implementations that provide debugging or inspection affordances for streams of OpenPGP packets.

5.1. Public-Key Encrypted Session Key Packet (Type ID 1)

Zero or more PKESK packets and/or SKESK packets (Section 5.3) precede an encryption container (that is, a Symmetrically Encrypted Integrity Protected Data (SEIPD) packet or -- for historic data -- a Symmetrically Encrypted Data (SED) packet), which holds an encrypted message. The message is encrypted with the session key, and the session key is itself encrypted and stored in the Encrypted Session Key packet(s). The encryption container is preceded by one Public-Key Encrypted Session Key packet for each OpenPGP key to which the message is encrypted. The recipient of the message finds a session key that is encrypted to their public key, decrypts the session key, and then uses the session key to decrypt the message.

The body of this packet starts with a one-octet number giving the version number of the packet type. The currently defined versions are 3 and 6. The remainder of the packet depends on the version.

The versions differ in how they identify the recipient key and in what they encode. The version of the PKESK packet must align with the version of the SEIPD packet (see Section 10.3.2.1). Any new version of the PKESK packet should be registered in the registry established in Section 10.3.2.1.

5.1.1. Version 3 Public-Key Encrypted Session Key Packet Format

A version 3 PKESK packet precedes a version 1 SEIPD packet (see Section 5.13.1). In historic data, it is sometimes found preceding a deprecated SED packet; see Section 5.7. A v3 PKESK packet **MUST NOT** precede a v2 SEIPD packet (see Section 10.3.2.1).

The v3 PKESK packet consists of:

- A one-octet version number with value 3.
- An eight-octet number that gives the Key ID of the public key to which the session key is encrypted. If the session key is encrypted to a subkey, then the Key ID of this subkey is used here instead of the Key ID of the primary key. The Key ID may also be all zeros, for an "anonymous recipient" (see Section 5.1.8).
- A one-octet number giving the public-key algorithm used.
- A series of values comprising the encrypted session key. This is algorithm specific and described below.

The public-key encryption algorithm (described in subsequent sections) is passed two values:

- The session key.
- The one-octet algorithm identifier that specifies the symmetric encryption algorithm used to encrypt the v1 SEIPD packet described in the following section.

5.1.2. Version 6 Public-Key Encrypted Session Key Packet Format

A version 6 PKESK packet precedes a version 2 SEIPD packet (see Section 5.13.2). A v6 PKESK packet **MUST NOT** precede a v1 SEIPD packet or a deprecated SED packet (see Section 10.3.2.1).

The v6 PKESK packet consists of the following fields:

- A one-octet version number with value 6.
- A one-octet size of the following two fields. This size may be zero, if the key version number field and the fingerprint field are omitted for an "anonymous recipient" (see Section 5.1.8).
- A one-octet key version number.
- The fingerprint of the public key or subkey to which the session key is encrypted. Note that the length N of the fingerprint for a version 4 key is 20 octets; for a version 6 key, N is 32.
- A one-octet number giving the public-key algorithm used.
- A series of values comprising the encrypted session key. This is algorithm specific and described below.

The session key is encrypted according to the public-key algorithm used, as described below. No symmetric encryption algorithm identifier is passed to the public-key algorithm for a v6 PKESK packet, as it is included in the v2 SEIPD packet.

5.1.3. Algorithm-Specific Fields for RSA Encryption

- MPI of RSA-encrypted value m**e mod n.

To produce the value "m" in the above formula, first concatenate the following values:

- The one-octet algorithm identifier, if it was passed (in the case of a v3 PKESK packet).
- The session key.
- A two-octet checksum of the session key, equal to the sum of the session key octets, modulo 65536.

Then, the above values are encoded using the PKCS#1 block encoding EME-PKCS1-v1_5, as described in Step 2 in Section 7.2.1 of [RFC8017] (see also Section 12.1.1). When decoding "m" during decryption, an implementation should follow Step 3 in Section 7.2.2 of [RFC8017] (see also Section 12.1.2).

Note that when an implementation forms several PKESKs with one session key, forming a message that can be decrypted by several keys, the implementation **MUST** make a new PKCS#1 encoding for each key. This defends against attacks such as those discussed in [HASTAD].

5.1.4. Algorithm-Specific Fields for Elgamal Encryption

- MPI of Elgamal (Diffie-Hellman) value g**k mod p.
- MPI of Elgamal (Diffie-Hellman) value m * y**k mod p.

To produce the value "m" in the above formula, first concatenate the following values:

- The one-octet algorithm identifier, if it was passed (in the case of a v3 PKESK packet).
- The session key.
- A two-octet checksum of the session key, equal to the sum of the session key octets, modulo 65536.

Then, the above values are encoded using the PKCS#1 block encoding EME-PKCS1-v1_5, as described in Step 2 in Section 7.2.1 of [RFC8017] (see also Section 12.1.1). When decoding "m" during decryption, an implementation should follow Step 3 in Section 7.2.2 of [RFC8017] (see also Section 12.1.2).

Note that when an implementation forms several PKESKs with one session key, forming a message that can be decrypted by several keys, the implementation **MUST** make a new PKCS#1 encoding for each key. This defends against attacks such as those discussed in [HASTAD].

An implementation **MUST NOT** generate ElGamal v6 PKESKs.

5.1.5. Algorithm-Specific Fields for ECDH Encryption

- MPI of an EC point representing an ephemeral public key in the point format associated with the curve as specified in Section 9.2.
- A one-octet size, followed by a symmetric key encoded using the method described in Section 11.5.

5.1.6. Algorithm-Specific Fields for X25519 Encryption

- 32 octets representing an ephemeral X25519 public key.
- A one-octet size of the following fields.
- The one-octet algorithm identifier, if it was passed (in the case of a v3 PKESK packet).
- The encrypted session key.

See Section 6.1 of [RFC7748] for more details on the computation of the ephemeral public key and the shared secret. The HMAC-based Key Derivation Function (HKDF) [RFC5869] is then used with SHA256 [RFC6234] and an info parameter of "OpenPGP X25519" and no salt. The input of HKDF is the concatenation of the following three values:

- 32 octets of the ephemeral X25519 public key from this packet.
- 32 octets of the recipient public key material.
- 32 octets of the shared secret.

The key produced from HKDF is used to encrypt the session key with AES-128 key wrap, as defined in [RFC3394].

Note that unlike Elliptic Curve Diffie-Hellman (ECDH), no checksum or padding are appended to the session key before key wrapping. Finally, note that unlike the other public-key algorithms, in the case of a v3 PKESK packet, the symmetric algorithm ID is not encrypted. Instead, it is prepended to the encrypted session key in plaintext. In this case, the symmetric algorithm used **MUST** be AES-128, AES-192, or AES-256 (algorithm IDs 7, 8, or 9, respectively).

5.1.7. Algorithm-Specific Fields for X448 Encryption

- 56 octets representing an ephemeral X448 public key.
- A one-octet size of the following fields.
- The one-octet algorithm identifier, if it was passed (in the case of a v3 PKESK packet).
- The encrypted session key.

See Section 6.2 of [RFC7748] for more details on the computation of the ephemeral public key and the shared secret. HKDF [RFC5869] is then used with SHA512 [RFC6234] and an info parameter of "OpenPGP X448" and no salt. The input of HKDF is the concatenation of the following three values:

- 56 octets of the ephemeral X448 public key from this packet.
- 56 octets of the recipient public key material.
- 56 octets of the shared secret.

The key produced from HKDF is used to encrypt the session key with AES-256 key wrap, as defined in [RFC3394].

Note that unlike ECDH, no checksum or padding are appended to the session key before key wrapping. Finally, note that unlike the other public-key algorithms, in the case of a v3 PKESK packet, the symmetric algorithm ID is not encrypted. Instead, it is prepended to the encrypted session key in plaintext. In this case, the symmetric algorithm used **MUST** be AES-128, AES-192, or AES-256 (algorithm ID 7, 8, or 9).

5.1.8. Notes on PKESK

An implementation **MAY** accept or use a Key ID of all zeros, or an omitted key fingerprint, to hide the intended decryption key. In this case, the receiving implementation would try all available private keys, checking for a valid decrypted session key. This format helps reduce traffic analysis of messages.

5.2. Signature Packet (Type ID 2)

A Signature packet describes a binding between some public key and some data. The most common signatures are a signature of a file or a block of text and a signature that is a certification of a User ID.

Three versions of Signature packets are defined. Version 3 provides basic signature information, while versions 4 and 6 provide an expandable format with subpackets that can specify more information about the signature.

For historical reasons, versions 1, 2, and 5 of the Signature packet are unspecified. Any new Signature packet version should be registered in the registry established in Section 10.3.2.2.

An implementation **MUST** generate a version 6 signature when signing with a version 6 key. An implementation **MUST** generate a version 4 signature when signing with a version 4 key. Implementations **MUST NOT** create version 3 signatures; they **MAY** accept version 3 signatures. See Section 10.3.2.2 for more details about packet version correspondence between keys and signatures.

5.2.1. Signature Types

There are a number of possible meanings for a signature, which are indicated by the signature type ID in any given signature. Please note that the vagueness of these meanings is not a flaw but rather a feature of the system. Because OpenPGP places final authority for validity upon the receiver of a signature, it may be that one signer's casual act might be more rigorous than some other authority's positive act. See Section 5.2.4 for detailed information on how to compute and verify signatures of each type.

ID	Name	Reference
0x00	Binary Signature	Section 5.2.1.1
0x01	Text Signature	Section 5.2.1.2
0x02	Standalone Signature	Section 5.2.1.3
0x10	Generic Certification	Section 5.2.1.4
0x11	Persona Certification	Section 5.2.1.5
0x12	Casual Certification	Section 5.2.1.6
0x13	Positive Certification	Section 5.2.1.7
0x18	Subkey Binding Signature	Section 5.2.1.8
0x19	Primary Key Binding Signature	Section 5.2.1.9
0x1F	Direct Key Signature	Section 5.2.1.10
0x20	Key Revocation	Section 5.2.1.11
0x28	Subkey Revocation	Section 5.2.1.12
0x30	Certification Revocation	Section 5.2.1.13
0x40	Timestamp Signature	Section 5.2.1.14
0x50	Third-Party Confirmation	Section 5.2.1.15

ID	Name	Reference
0xFF	Reserved	Section 5.2.1.16

Table 4: OpenPGP Signature Types Registry

The meanings of each signature type are described in the subsections below.

5.2.1.1. Signature of a Binary Document (type ID 0x00)

This means the signer owns it, created it, or certifies that it has not been modified.

5.2.1.2. Signature of a Canonical Text Document (type ID 0x01)

This means the signer owns it, created it, or certifies that it has not been modified. The signature is calculated over the text data with its line endings converted to <CR><LF>.

5.2.1.3. Standalone Signature (type ID 0x02)

This signature is a signature of only its own subpacket contents. It is calculated identically to a signature over a zero-length binary document. V3 standalone signatures **MUST NOT** be generated and **MUST** be ignored.

5.2.1.4. Generic Certification of a User ID and Public-Key Packet (type ID 0x10)

The issuer of this certification does not make any particular assertion as to how well the certifier has checked that the owner of the key is in fact the person described by the User ID.

5.2.1.5. Persona Certification of a User ID and Public-Key Packet (type ID 0x11)

The issuer of this certification has not done any verification of the claim that the owner of this key is the User ID specified.

5.2.1.6. Casual Certification of a User ID and Public-Key Packet (type ID 0x12)

The issuer of this certification has done some casual verification of the claim of identity.

5.2.1.7. Positive Certification of a User ID and Public-Key Packet (type ID 0x13)

The issuer of this certification has done substantial verification of the claim of identity.

Most OpenPGP implementations make their "key signatures" as generic (type ID 0x10) certifications. Some implementations can issue 0x11-0x13 certifications, but few differentiate between the types.

5.2.1.8. Subkey Binding Signature (type ID 0x18)

This signature is a statement by the top-level signing key, indicating that it owns the subkey. This signature is calculated directly on the primary key and subkey, and not on any User ID or other packets. A signature that binds a signing subkey **MUST** have an Embedded Signature subpacket in this binding signature that contains a 0x19 signature made by the signing subkey on the primary key and subkey.

5.2.1.9. Primary Key Binding Signature (type ID 0x19)

This signature is a statement by a signing subkey, indicating that it is owned by the primary key. This signature is calculated the same way as a subkey binding signature (0x18): directly on the primary key and subkey, and not on any User ID or other packets.

5.2.1.10. Direct Key Signature (type ID 0x1F)

This signature is calculated directly on a key. It binds the information in the Signature subpackets to the key and is appropriate to be used for subpackets that provide information about the key, such as the Key Flags subpacket or the (deprecated) Revocation Key subpacket. It is also appropriate for statements that non-self certifiers want to make about the key itself rather than the binding between a key and a name.

5.2.1.11. Key Revocation Signature (type ID 0x20)

This signature is calculated directly on the key being revoked. A revoked key is not to be used. Only revocation signatures by the key being revoked, or by a (deprecated) Revocation Key, should be considered valid revocation signatures.

5.2.1.12. Subkey Revocation Signature (type ID 0x28)

This signature is calculated directly on the primary key and the subkey being revoked. A revoked subkey is not to be used. Only revocation signatures by the top-level signature key that is bound to this subkey, or by a (deprecated) Revocation Key, should be considered valid revocation signatures.

5.2.1.13. Certification Revocation Signature (type ID 0x30)

This signature revokes an earlier User ID certification signature (signature class 0x10 through 0x13) or direct key signature (0x1F). It should be issued by the same key that issued the revoked signature or by a (deprecated) Revocation Key. The signature is computed over the same data as the certification that it revokes, and it should have a later creation date than that certification.

5.2.1.14. Timestamp Signature (type ID 0x40)

This signature is only meaningful for the timestamp contained in it.

5.2.1.15. Third-Party Confirmation Signature (type ID 0x50)

This signature is a signature over some other OpenPGP Signature packet(s). It is analogous to a notary seal on the signed data. A third-party signature **SHOULD** include a Signature Target subpacket(s) to give easy identification. Note that we really do mean **SHOULD**. There are plausible uses for this (such as a blind party that only sees the signature, not the key or source document) that cannot include a target subpacket.

5.2.1.16. Reserved (type ID 0xFF)

An implementation **MUST NOT** create any signature with this type and **MUST NOT** validate any signature made with this type. See Section 5.2.4.1 for more details.

5.2.2. Version 3 Signature Packet Format

The body of a version 3 Signature packet contains:

- A one-octet version number with value 3.
- A one-octet length of the following hashed material; it **MUST** be 5:
  - A one-octet signature type ID.
  - A four-octet creation time.
- An eight-octet Key ID of the signer.
- A one-octet public-key algorithm.
- A one-octet hash algorithm.
- A two-octet field holding left 16 bits of the signed hash value.
- One or more MPIs comprising the signature. This portion is algorithm specific, as described below.

The concatenation of the data to be signed, the signature type, and the creation time from the Signature packet (5 additional octets) is hashed. The resulting hash value is used in the signature algorithm. The high 16 bits (first two octets) of the hash are included in the Signature packet to provide a way to reject some invalid signatures without performing a signature verification.

Algorithm-specific fields for RSA signatures:

- MPI of RSA signature value m**d mod n.

Algorithm-specific fields for DSA signatures:

- MPI of DSA value r.
- MPI of DSA value s.

The signature calculation is based on a hash of the signed data, as described above. The details of the calculation are different for DSA signatures than for RSA signatures; see Sections 5.2.3.1 and 5.2.3.2.

5.2.3. Versions 4 and 6 Signature Packet Formats

The body of a v4 or v6 Signature packet contains:

- A one-octet version number. This is 4 for v4 signatures and 6 for v6 signatures.
- A one-octet signature type ID.
- A one-octet public-key algorithm.
- A one-octet hash algorithm.
- A Scalar octet count for the hashed subpacket data that follows this field. For a v4 signature, this is a two-octet field. For a v6 signature, this is a four-octet field. Note that this is the length in octets of all of the hashed subpackets; an implementation's pointer incremented by this number will skip over the hashed subpackets.

- A hashed subpacket data set (zero or more subpackets).
- A scalar octet count for the unhashed subpacket data that follows this field. For a v4 signature, this is a two-octet field. For a v6 signature, this is a four-octet field. Note that this is the length in octets of all of the unhashed subpackets; an implementation's pointer incremented by this number will skip over the unhashed subpackets.
- An unhashed subpacket data set (zero or more subpackets).
- A two-octet field holding the left 16 bits of the signed hash value.
- Only for v6 signatures, a variable-length field containing:
  - A one-octet salt size. The value **MUST** match the value defined for the hash algorithm as specified in Table 23.
  - The salt, which is a random value of the specified size.
- One or more MPIs comprising the signature. This portion is algorithm specific.

5.2.3.1. Algorithm-Specific Fields for RSA Signatures

- MPI of RSA signature value m**d mod n.

With RSA signatures, the hash value is encoded using PKCS#1 encoding type EMSA-PKCS1-v1_5, as described in Section 9.2 of [RFC8017] (see also Section 12.1.3). This requires inserting the hash value as an octet string into an ASN.1 structure. The object identifier (OID) for the hash algorithm itself is also included in the structure; see the OIDs in Table 24.

5.2.3.2. Algorithm-Specific Fields for DSA or ECDSA Signatures

- MPI of DSA or ECDSA value r.
- MPI of DSA or ECDSA value s.

A version 3 signature **MUST NOT** be created and **MUST NOT** be used with the Elliptic Curve Digital Signature Algorithm (ECDSA).

A DSA signature **MUST** use a hash algorithm with a digest size of at least the number of bits of q, the group generated by the DSA key's generator value.

If the output size of the chosen hash is larger than the number of bits of q, the hash result is truncated to fit by taking the number of leftmost bits equal to the number of bits of q. This (possibly truncated) hash function result is treated as a number and used directly in the DSA signature algorithm.

An ECDSA signature **MUST** use a hash algorithm with a digest size of at least the curve's "fsize" value (see Section 9.2), except in the case of NIST P-521, for which at least a 512-bit hash algorithm **MUST** be used.

5.2.3.3. Algorithm-Specific Fields for EdDSALegacy Signatures (Deprecated)

- Two MPI-encoded values, whose contents and formatting depend on the choice of curve used (see Section 9.2.1).

A version 3 signature **MUST NOT** be created and **MUST NOT** be used with EdDSALegacy.

An EdDSALegacy signature **MUST** use a hash algorithm with a digest size of at least the curve's "fsize" value (see Section 9.2). A verifying implementation **MUST** reject any EdDSALegacy signature that uses a hash algorithm with a smaller digest size.

5.2.3.3.1. Algorithm-Specific Fields for Ed25519Legacy Signatures (Deprecated)

The two MPIs for Ed25519Legacy use octet strings R and S as described in [RFC8032]. Ed25519Legacy **MUST NOT** be used in signature packets version 6 or above.

- MPI of an EC point R, represented as a (non-prefixed) native (little-endian) octet string up to 32 octets.
- MPI of the Edwards-curve Digital Signature Algorithm (EdDSA) value S, also in (non-prefixed) native (little-endian) format with a length up to 32 octets.

5.2.3.4. Algorithm-Specific Fields for Ed25519 Signatures

- 64 octets of the native signature.

For more details, see Section 12.7.

A version 3 signature **MUST NOT** be created and **MUST NOT** be used with Ed25519.

An Ed25519 signature **MUST** use a hash algorithm with a digest size of at least 256 bits. A verifying implementation **MUST** reject any Ed25519 signature that uses a hash algorithm with a smaller digest size.

5.2.3.5. Algorithm-Specific Fields for Ed448 Signatures

- 114 octets of the native signature.

For more details, see Section 12.7.

A version 3 signature **MUST NOT** be created and **MUST NOT** be used with Ed448.

An Ed448 signature **MUST** use a hash algorithm with a digest size of at least 512 bits. A verifying implementation **MUST** reject any Ed448 signature that uses a hash algorithm with a smaller digest size.

5.2.3.6. Notes on Signatures

The concatenation of the data being signed, the signature data from the version number through the hashed subpacket data (inclusive), and (for signature versions later than 3) a six-octet trailer (see Section 5.2.4) are hashed. The resulting hash value is what is signed. The high 16 bits (first two octets) of the hash are included in the Signature packet to provide a way to reject some invalid signatures without performing a signature verification. When verifying a v6 signature, an implementation **MUST** reject the signature if these octets do not match the first two octets of the computed hash.

There are two fields consisting of Signature subpackets. The first field is hashed with the rest of the signature data, while the second is not hashed into the signature. The second set of subpackets (the "unhashed section") is not cryptographically protected by the signature and should include only advisory information. See Section 13.13 for more information.

The differences between a v4 and v6 signature are two-fold: first, a v6 signature increases the width of the fields that indicate the size of the hashed and unhashed subpackets, making it possible to include significantly more data in subpackets. Second, the hash is salted with random data (see Section 13.2).

The algorithms for converting the hash function result to a signature are described in Section 5.2.4.

5.2.3.7. Signature Subpacket Specification

A subpacket data set consists of zero or more Signature subpackets. In Signature packets, the subpacket data set is preceded by a two-octet (for v4 signatures) or four-octet (for v6 signatures) scalar count of the length in octets of all the subpackets. A pointer incremented by this number will skip over the subpacket data set.

Each subpacket consists of a subpacket header and a body. The header consists of:

- the subpacket length (1, 2, or 5 octets)
- the encoded subpacket type ID (1 octet)

and is followed by the subpacket-specific data.

The length includes the encoded subpacket type ID octet but not this length. Its format is similar to the OpenPGP format packet header lengths, but it cannot have Partial Body Lengths. That is:

```
if the 1st octet <  192, then
    lengthOfLength = 1
    subpacketLen = 1st_octet

if the 1st octet >= 192 and < 255, then
    lengthOfLength = 2
    subpacketLen = ((1st_octet - 192) << 8) + (2nd_octet) + 192

if the 1st octet = 255, then
    lengthOfLength = 5
    subpacket length = [four-octet scalar starting at 2nd_octet]
```

Bit 7 of the encoded subpacket type ID is the "critical" bit. If set, it denotes that the subpacket is one that is critical for the evaluator of the signature to recognize. If a subpacket is encountered that is marked critical but is unknown to the evaluating implementation, the evaluator **SHOULD** consider the signature to be in error.

An implementation **SHOULD** ignore any non-critical subpacket of a type that it does not recognize.

An evaluator may "recognize" a subpacket but not implement it. The purpose of the critical bit is to allow the signer to tell an evaluator that it would prefer a new, unknown feature to generate an error rather than being ignored.

The other bits of the encoded subpacket type ID (i.e., bits 6-0) contain the subpacket type ID.

The following signature subpackets are defined:

ID	Description	Reference
0	Reserved	[RFC9580]
1	Reserved	[RFC9580]
2	Signature Creation Time	[RFC9580], Section 5.2.3.11
3	Signature Expiration Time	[RFC9580], Section 5.2.3.18
4	Exportable Certification	[RFC9580], Section 5.2.3.19
5	Trust Signature	[RFC9580], Section 5.2.3.21
6	Regular Expression	[RFC9580], Section 5.2.3.22
7	Revocable	[RFC9580], Section 5.2.3.20
8	Reserved	[RFC9580]
9	Key Expiration Time	[RFC9580], Section 5.2.3.13
10	Placeholder for backward compatibility	[RFC9580]
11	Preferred Symmetric Ciphers for v1 SEIPD	[RFC9580], Section 5.2.3.14
12	Revocation Key (deprecated)	[RFC9580], Section 5.2.3.23
13-15	Reserved	[RFC9580]
16	Issuer Key ID	[RFC9580], Section 5.2.3.12
17-19	Reserved	[RFC9580]
20	Notation Data	[RFC9580], Section 5.2.3.24
21	Preferred Hash Algorithms	[RFC9580], Section 5.2.3.16
22	Preferred Compression Algorithms	[RFC9580], Section 5.2.3.17
23	Key Server Preferences	[RFC9580], Section 5.2.3.25

ID	Description	Reference
24	Preferred Key Server	[RFC9580], Section 5.2.3.26
25	Primary User ID	[RFC9580], Section 5.2.3.27
26	Policy URI	[RFC9580], Section 5.2.3.28
27	Key Flags	[RFC9580], Section 5.2.3.29
28	Signer's User ID	[RFC9580], Section 5.2.3.30
29	Reason for Revocation	[RFC9580], Section 5.2.3.31
30	Features	[RFC9580], Section 5.2.3.32
31	Signature Target	[RFC9580], Section 5.2.3.33
32	Embedded Signature	[RFC9580], Section 5.2.3.34
33	Issuer Fingerprint	[RFC9580], Section 5.2.3.35
34	Reserved	[RFC9580]
35	Intended Recipient Fingerprint	[RFC9580], Section 5.2.3.36
37	Reserved (Attested Certifications)	[RFC9580]
38	Reserved (Key Block)	[RFC9580]
39	Preferred AEAD Ciphersuites	[RFC9580], Section 5.2.3.15
100-110	Private or Experimental	[RFC9580]

Table 5: OpenPGP Signature Subpacket Types Registry

Implementations **SHOULD** implement the four preferred algorithm subpackets (11, 21, 22, and 39), as well as the "Features" (30) and "Reason for Revocation" (29) subpackets. To avoid surreptitious forwarding (see Section 13.12), implementations **SHOULD** also implement the "Intended Recipients Fingerprint" (35) subpacket. Note that if an implementation chooses not to implement some of the preferences subpackets, it **MUST** default to the mandatory-to-implement algorithms to ensure interoperability. An encrypting implementation that does not implement the "Features" (30) subpacket **SHOULD** select the type of encrypted data format based on the versions of the recipient keys or external inference (see Section 13.7 for more details).

5.2.3.8. Signature Subpacket Types

A number of subpackets are currently defined for OpenPGP signatures. Some subpackets apply to the signature itself and some are attributes of the key. Subpackets that are found on a self-signature are placed on a certification made by the key itself. Note that a key may have more than one User ID and thus may have more than one self-signature and differing subpackets.

A subpacket may be found in either the hashed or the unhashed subpacket sections of a signature. If a subpacket is not hashed, then the information in it cannot be considered definitive because it is not part of the signature proper. See Section 13.13 for more discussion about hashed and unhashed subpackets.

5.2.3.9. Notes on Subpackets

It is certainly possible for a signature to contain conflicting information in subpackets. For example, a signature may contain multiple copies of a preference or multiple expiration times. In most cases, an implementation **SHOULD** use the last subpacket in the hashed section of the signature, but it **MAY** use any conflict resolution scheme that makes more sense. Please note that we are intentionally leaving conflict resolution to the implementer; most conflicts are simply syntax errors, and the wishy-washy language here allows a receiver to be generous in what they accept, while putting pressure on a creator to be stingy in what they generate.

Some apparent conflicts may actually make sense. For example, suppose a keyholder has a v3 key and a v4 key that share the same RSA key material. Either of these keys can verify a signature created by the other, and it may be reasonable for a signature to contain an Issuer Key ID subpacket (Section 5.2.3.12) for each key, as a way of explicitly tying those keys to the signature.

5.2.3.10. Notes on Self-Signatures

A self-signature is a binding signature made by the key to which the signature refers. There are three types of self-signatures: the certification signatures (type IDs 0x10-0x13), the direct key signature (type ID 0x1F), and the subkey binding signature (type ID 0x18). A cryptographically valid self-signature should be accepted from any primary key, regardless of what Key Flags (Section 5.2.3.29) apply to the primary key. In particular, a primary key does not need to have 0x01 set in the first octet of the Key Flags order to make a valid self-signature.

For certification self-signatures, each User ID **MAY** have a self-signature and thus different subpackets in those self-signatures. For subkey binding signatures, each subkey **MUST** have a self-signature. Subpackets that appear in a certification self-signature apply to the User ID, and subpackets that appear in the subkey self-signature apply to the subkey. Lastly, subpackets on the direct key signature apply to the entire key.

An implementation should interpret a self-signature's preference subpackets as narrowly as possible. For example, suppose a key has two user names, Alice and Bob. Suppose that Alice prefers the AEAD ciphersuite AES-256 with OCB, and Bob prefers Camellia-256 with Galois/Counter Mode (GCM). If the implementation locates this key via Alice's name, then the preferred

AEAD ciphersuite is AES-256 with OCB; if the implementation locates the key via Bob's name, then the preferred algorithm is Camellia-256 with GCM. If the key is located by Key ID, the algorithm of the primary User ID of the key provides the preferred AEAD ciphersuite.

Revoking a self-signature or allowing it to expire has a semantic meaning that varies with the signature type. Revoking the self-signature on a User ID effectively retires that user name. The self-signature is a statement, "My name X is tied to my signing key K", and it is corroborated by other users' certifications. If another user revokes their certification, they are effectively saying that they no longer believe that name and that key are tied together. Similarly, if the users themselves revoke their self-signature, then the users no longer go by that name, no longer have that email address, etc. Revoking a binding signature effectively retires that subkey. Revoking a direct key signature cancels that signature. Please see Section 5.2.3.31 for more relevant details.

Since a self-signature contains important information about the key's use, an implementation **SHOULD** allow the user to rewrite the self-signature and important information in it, such as preferences and key expiration.

When an implementation imports a secret key, it **SHOULD** verify that the key's internal self-signatures do not advertise features or algorithms that the implementation doesn't support. If an implementation observes such a mismatch, it **SHOULD** warn the user and offer to create new self-signatures that advertise the actual set of features and algorithms supported by the implementation.

An implementation that encounters multiple self-signatures on the same object **MUST** select the most recent valid self-signature and ignore all other self-signatures.

By convention, a version 4 key stores information about the primary Public-Key (key flags, key expiration, etc.) and the Transferable Public Key as a whole (features, algorithm preferences, etc.) in a User ID self-signature of type 0x10 or 0x13. To use a v4 key, some implementations require at least one User ID with a valid self-signature to be present. For this reason, it is **RECOMMENDED** to include at least one User ID with a self-signature in v4 keys.

For version 6 keys, it is **RECOMMENDED** to store information about the primary Public-Key as well as the Transferable Public Key as a whole (key flags, key expiration, features, algorithm preferences, etc.) in a direct key signature (type ID 0x1F) over the Public-Key, instead of placing that information in a User ID self-signature. An implementation **MUST** ensure that a valid direct key signature is present before using a v6 key. This prevents certain attacks where an adversary strips a self-signature specifying a key expiration time or certain preferences.

An implementation **SHOULD NOT** require a User ID self-signature to be present in order to consume or use a key, unless the particular use is contingent on the keyholder identifying themselves with the textual label in the User ID. For example, when refreshing a key to learn about changes in expiration, advertised features, algorithm preferences, revocation, subkey rotation, and so forth, there is no need to require a User ID self-signature. On the other hand, when verifying a signature over an email message, an implementation **MAY** choose to only accept a signature from a key that has a valid self-signature over a User ID that matches the message's From: header, as a way to avoid a signature transplant attack.

5.2.3.11. Signature Creation Time

(4-octet time field)

The time the signature was made.

This subpacket **MUST** be present in the hashed area.

When generating this subpacket, it **SHOULD** be marked as critical.

5.2.3.12. Issuer Key ID

(8-octet Key ID)

The OpenPGP Key ID of the key issuing the signature. If the version of that key is greater than 4, this subpacket **MUST NOT** be included in the signature. For these keys, consider the Issuer Fingerprint subpacket (Section 5.2.3.35) instead.

Note: in previous versions of this specification, this subpacket was simply known as the "Issuer" subpacket.

5.2.3.13. Key Expiration Time

(4-octet time field)

The validity period of the key. This is the number of seconds after the key creation time that the key expires. For a direct or certification self-signature, the key creation time is that of the primary key. For a subkey binding signature, the key creation time is that of the subkey. If this is not present or has a value of zero, the key never expires. This is found only on a self-signature.

When an implementation generates this subpacket, it **SHOULD** be marked as critical.

5.2.3.14. Preferred Symmetric Ciphers for v1 SEIPD

(array of one-octet values)

A series of symmetric cipher algorithm IDs indicating how the keyholder prefers to receive version 1 Symmetrically Encrypted Integrity Protected Data (Section 5.13.1). The subpacket body is an ordered list of octets with the most preferred listed first. It is assumed that only the algorithms listed are supported by the recipient's implementation. Algorithm IDs are defined in Section 9.3. This is only found on a self-signature.

When generating a v2 SEIPD packet, this preference list is not relevant. See Section 5.2.3.15 instead.

5.2.3.15. Preferred AEAD Ciphersuites

(array of pairs of octets indicating Symmetric Cipher and AEAD algorithms)

A series of paired algorithm IDs indicating how the keyholder prefers to receive version 2 Symmetrically Encrypted Integrity Protected Data (Section 5.13.2). Each pair of octets indicates a combination of a symmetric cipher and an AEAD mode that the keyholder prefers to use. The symmetric cipher algorithm ID precedes the AEAD algorithm ID in each pair. The subpacket body is an ordered list of pairs of octets with the most preferred algorithm combination listed first.

It is assumed that only the combinations of algorithms listed are supported by the recipient's implementation, with the exception of the mandatory-to-implement combination of AES-128 and OCB. If AES-128 and OCB are not found in the subpacket, it is implicitly listed at the end.

AEAD algorithm IDs are listed in Section 9.6. Symmetric cipher algorithm IDs are listed in Section 9.3.

For example, a subpacket with the content of these six octets

```
09 02 09 03 13 02
```

indicates that the keyholder prefers to receive v2 SEIPD using AES-256 with OCB, then AES-256 with GCM, then Camellia-256 with OCB, and finally the implicit AES-128 with OCB.

Note that support for version 2 of the Symmetrically Encrypted Integrity Protected Data packet (Section 5.13.2) in general is indicated by a Features Flag (Section 5.2.3.32).

This subpacket is only found on a self-signature.

When generating a v1 SEIPD packet, this preference list is not relevant. See Section 5.2.3.14 instead.

5.2.3.16. Preferred Hash Algorithms

(array of one-octet values)

Message digest algorithm IDs that indicate which algorithms the keyholder prefers to receive. Like the preferred AEAD ciphersuites, the list is ordered. Algorithm IDs are defined in Section 9.5. This is only found on a self-signature.

5.2.3.17. Preferred Compression Algorithms

(array of one-octet values)

Compression algorithm IDs that indicate which algorithms the keyholder prefers to use. Like the preferred AEAD ciphersuites, the list is ordered. Algorithm IDs are defined in Section 9.4. A zero, or the absence of this subpacket, denotes that uncompressed data is preferred; the keyholder's implementation might have no compression support available. This is only found on a self-signature.

5.2.3.18. Signature Expiration Time

(4-octet time field)

The validity period of the signature. This is the number of seconds after the signature creation time that the signature expires. If this is not present or has a value of zero, it never expires.

When an implementation generates this subpacket, it **SHOULD** be marked as critical.

5.2.3.19. Exportable Certification

(1 octet of exportability, 0 for not, 1 for exportable)

This subpacket denotes whether a certification signature is "exportable"; it is intended for use by users other than the signature's issuer. The packet body contains a Boolean flag indicating whether the signature is exportable. If this packet is not present, the certification is exportable; it is equivalent to a flag containing a 1.

Non-exportable, or "local", certifications are signatures made by a user to mark a key as valid within that user's implementation only.

Thus, when an implementation prepares a user's copy of a key for transport to another user (this is the process of "exporting" the key), any local certification signatures are deleted from the key.

The receiver of a transported key "imports" it and likewise trims any local certifications. In normal operation, there won't be any local certifications, assuming the import is performed on an exported key. However, there are instances where this can reasonably happen. For example, if an implementation allows keys to be imported from a key database in addition to an exported key, then this situation can arise.

Some implementations do not represent the interest of a single user (for example, a key server). Such implementations always trim local certifications from any key they handle.

When an implementation generates this subpacket and denotes the signature as non-exportable, the subpacket **MUST** be marked as critical.

5.2.3.20. Revocable

(1 octet of revocability, 0 for not, 1 for revocable)

A Signature's revocability status. The packet body contains a Boolean flag indicating whether the signature is revocable. Signatures that are not revocable ignore any later revocation signatures. They represent the signer's commitment that its signature cannot be revoked for the life of its key. If this packet is not present, the signature is revocable.

5.2.3.21. Trust Signature

(1 octet "level" (depth), 1 octet of trust amount)

A signer asserts that the key is not only valid but also trustworthy at the specified level. Level 0 has the same meaning as an ordinary validity signature. Level 1 means that the signed key is asserted to be a valid trusted introducer, with the 2nd octet of the body specifying the degree of trust. Level 2 means that the signed key is asserted to be trusted to issue level 1 trust signatures; that is, the signed key is a "meta introducer". Generally, a level n trust signature asserts that a key

is trusted to issue level n-1 trust signatures. The trust amount is in a range from 0-255, interpreted such that values less than 120 indicate partial trust and values of 120 or greater indicate complete trust. Implementations **SHOULD** emit values of 60 for partial trust and 120 for complete trust.

5.2.3.22. Regular Expression

(null-terminated UTF-8 encoded regular expression)

Used in conjunction with trust Signature packets (of level > 0) to limit the scope of trust that is extended. Only signatures by the target key on User IDs that match the regular expression in the body of this packet have trust extended by the trust Signature subpacket. The regular expression uses the same syntax as Henry Spencer's "almost public domain" regular expression [REGEX] package. A description of the syntax is found in Section 8. The regular expression matches (or does not match) a sequence of UTF-8-encoded Unicode characters from User IDs. The expression itself is also written with UTF-8 characters.

For historical reasons, this subpacket includes a null character (an octet with value zero) after the regular expression. When an implementation parses a regular expression subpacket, it **MUST** remove this octet; if it is not present, it **MUST** reject the subpacket (i.e., ignore the subpacket if it's non-critical and reject the signature if it's critical). When an implementation generates a regular expression subpacket, it **MUST** include the null terminator.

When generating this subpacket, it **SHOULD** be marked as critical.

5.2.3.23. Revocation Key (Deprecated)

(1 octet of class, 1 octet of public-key algorithm ID, 20 octets of v4 fingerprint)

This mechanism is deprecated. Applications **MUST NOT** generate such a subpacket.

An application that wants the functionality of delegating revocation can use an escrowed Revocation Signature. See Section 13.9 for more details.

The remainder of this section describes how some implementations attempt to interpret this deprecated subpacket.

This packet was intended to authorize the specified key to issue revocation signatures for this key. The class octet must have bit 0x80 set. If bit 0x40 is set, it means the revocation information is sensitive. Other bits are for future expansion to other kinds of authorizations. This is only found on a direct key self-signature (type ID 0x1F). The use on other types of self-signatures is unspecified.

If the "sensitive" flag is set, the keyholder feels this subpacket contains private trust information that describes a real-world sensitive relationship. If this flag is set, implementations **SHOULD NOT** export this signature to other users except in cases where the data needs to be available, i.e., when the signature is being sent to the designated revoker or when it is accompanied by a

revocation signature from that revoker. Note that it may be appropriate to isolate this subpacket within a separate signature so that it is not combined with other subpackets that need to be exported.

5.2.3.24. Notation Data

(4 octets of flags, 2 octets of name length (M), 2 octets of value length (N), M octets of name data, N octets of value data)

This subpacket describes a "notation" on the signature that the issuer wishes to make. The notation has a name and a value, each of which are strings of octets. There may be more than one notation in a signature. Notations can be used for any extension the issuer of the signature cares to make. The "flags" field holds four octets of flags.

All undefined flags **MUST** be zero. Defined flags are as follows:

Flag Position	Shorthand	Description	Reference
0x80000000 (first bit of the first octet)	human-readable	Notation value is UTF-8 text	[RFC9580]

Table 6: OpenPGP Signature Notation Data Subpacket Notation Flags Registry

Notation names are arbitrary strings encoded in UTF-8. They reside in two namespaces: the IETF namespace and the user namespace.

The IETF namespace is registered with IANA. These names **MUST NOT** contain the "@" character (0x40). This is a tag for the user namespace.

Notation Name	Data Type	Allowed Values	Reference
No registrations at this time.			

Table 7: OpenPGP Signature Notation Data Subpacket Types Registry

This registry is initially empty.

Names in the user namespace consist of a UTF-8 string tag followed by "@", followed by a DNS domain name. Note that the tag **MUST NOT** contain an "@" character. For example, the "sample" tag used by Example Corporation could be "sample@example.com".

Names in a user space are owned and controlled by the owners of that domain. Obviously, it's bad form to create a new name in a DNS space that you don't own.

Since the user namespace is in the form of an email address, implementers **MAY** wish to arrange for that address to reach a person who can be consulted about the use of the named tag. Note that due to UTF-8 encoding, not all valid user space name tags are valid email addresses.

If there is a critical notation, the criticality applies to that specific notation and not to notations in general.

5.2.3.25. Key Server Preferences

(N octets of flags)

This is a list of one-bit flags that indicates preferences that the keyholder has about how the key is handled on a key server. All undefined flags **MUST** be zero.

Flag	Shorthand	Definition
0x80...	No-modify	The keyholder requests that this key only be modified or updated by the keyholder or an administrator of the key server.

Table 8: OpenPGP Key Server Preference Flags Registry

This is found only on a self-signature.

5.2.3.26. Preferred Key Server

(String)

This is a URI of a key server that the keyholder prefers be used for updates. Note that keys with multiple User IDs can have a preferred key server for each User ID. Note also that since this is a URI, the key server can actually be a copy of the key retrieved by https, ftp, http, etc.

5.2.3.27. Primary User ID

(1 octet, Boolean)

This is a flag in a User ID's self-signature that states whether this User ID is the main User ID for this key. It is reasonable for an implementation to resolve ambiguities in preferences, for example, by referring to the primary User ID. If this flag is absent, its value is zero. If more than one User ID in a key is marked as primary, the implementation may resolve the ambiguity in any way it sees fit, but it is **RECOMMENDED** that priority be given to the User ID with the most recent self-signature.

When appearing on a self-signature on a User ID packet, this subpacket applies only to User ID packets. When appearing on a self-signature on a User Attribute packet, this subpacket applies only to User Attribute packets. That is, there are two different and independent "primaries" -- one for User IDs and one for User Attributes.

5.2.3.28. Policy URI

(String)

This subpacket contains a URI of a document that describes the policy under which the signature was issued.

5.2.3.29. Key Flags

(N octets of flags)

This subpacket contains a list of binary flags that hold information about a key. It is a string of octets, and an implementation **MUST NOT** assume a fixed size so that it can grow over time. If a list is shorter than an implementation expects, the unstated flags are considered to be zero. The defined flags are as follows:

Flag	Definition
0x01...	This key may be used to make User ID certifications (signature type IDs 0x10-0x13) or direct key signatures (signature type ID 0x1F) over other keys.
0x02...	This key may be used to sign data.
0x04...	This key may be used to encrypt communications.
0x08...	This key may be used to encrypt storage.
0x10...	The private component of this key may have been split by a secret-sharing mechanism.
0x20...	This key may be used for authentication.
0x80...	The private component of this key may be in the possession of more than one person.
0x0004...	Reserved (ADSK)
0x0008...	Reserved (timestamping)

Table 9: OpenPGP Key Flags Registry

Usage notes:

The flags in this packet may appear in self-signatures or in certification signatures. They mean different things depending on who is making the statement. For example, a certification signature that has the "sign data" flag is stating that the certification is for that use. On the other hand, the "communications encryption" flag in a self-signature is stating a preference that a given key be used for communications. However, note that determining what is "communications" and what is "storage" is a thorny issue. This decision is left wholly up to the implementation; the authors of this document do not claim any special wisdom on the issue and realize that accepted opinion may change.

The "split key" (0x10) and "group key" (0x80) flags are placed on a self-signature only; they are meaningless on a certification signature. They **SHOULD** be placed only on a direct key signature (type ID 0x1F) or a subkey signature (type ID 0x18), one that refers to the key the flag applies to.

When an implementation generates this subpacket, it **SHOULD** be marked as critical.

5.2.3.30. Signer's User ID

(String)

This subpacket allows a keyholder to state which User ID is responsible for the signing. Many keyholders use a single key for different purposes, such as business communications as well as personal communications. This subpacket allows such a keyholder to state which of their roles is making a signature.

This subpacket is not appropriate to use to refer to a User Attribute packet.

5.2.3.31. Reason for Revocation

(1 octet of revocation code, N octets of reason string)

This subpacket is used only in key revocation and certification revocation signatures. It describes the reason why the key or certification was revoked.

The first octet contains a machine-readable code that denotes the reason for the revocation:

Code	Reason
0	No reason specified (key revocations or cert revocations)
1	Key is superseded (key revocations)
2	Key material has been compromised (key revocations)
3	Key is retired and no longer used (key revocations)
32	User ID information is no longer valid (cert revocations)
100-110	Private Use

Table 10: OpenPGP Reason for Revocation Code Registry

Following the revocation code is a string of octets that gives information about the Reason for Revocation in human-readable form (UTF-8). The string may be null (of zero length). The length of the subpacket is the length of the reason string plus one. An implementation **SHOULD** implement this subpacket, include it in all revocation signatures, and interpret revocations appropriately. There are important semantic differences between the reasons, and there are thus important reasons for revoking signatures.

If a key has been revoked because of a compromise, all signatures created by that key are suspect. However, if it was merely superseded or retired, old signatures are still valid. If the revoked signature is the self-signature for certifying a User ID, a revocation denotes that that user name is no longer in use. Such a signature revocation **SHOULD** include a Reason for Revocation subpacket containing code 32.

Note that any certification may be revoked, including a certification on some other person's key. There are many good reasons for revoking a certification signature, such as the case where the keyholder leaves the employ of a business with an email address. A revoked certification is no longer a part of validity calculations.

5.2.3.32. Features

(N octets of flags)

The Features subpacket denotes which advanced OpenPGP features a user's implementation supports. This is so that as features are added to OpenPGP that cannot be backward compatible, a user can state that they can use that feature. The flags are single bits that indicate that a given feature is supported.

This subpacket is similar to a preferences subpacket and only appears in a self-signature.

An implementation **SHOULD NOT** use a feature listed when sending to a user who does not state that they can use it, unless the implementation can infer support for the feature from another implementation-dependent mechanism.

Defined features are as follows:

First octet:

Feature	Definition	Reference
0x01...	Symmetrically Encrypted Integrity Protected Data packet version 1	[RFC9580], Section 5.13.1
0x02...	Reserved	[RFC9580]
0x04...	Reserved	[RFC9580]
0x08...	Symmetrically Encrypted Integrity Protected Data packet version 2	[RFC9580], Section 5.13.2

Table 11: OpenPGP Features Flags Registry

If an implementation implements any of the defined features, it **SHOULD** implement the Features subpacket, too.

See Section 13.7 for details about how to use the Features subpacket when generating encryption data.

5.2.3.33. Signature Target

(1 octet public-key algorithm, 1 octet hash algorithm, N octets hash)

This subpacket identifies a specific target signature to which a signature refers. For revocation signatures, this subpacket provides explicit designation of which signature is being revoked. For a third-party or timestamp signature, this designates what signature is signed. All arguments are an identifier of that target signature.

The N octets of hash data **MUST** be the size of the signature's hash. For example, a target signature with a SHA-1 hash **MUST** have 20 octets of hash data.

5.2.3.34. Embedded Signature

(1 signature packet body)

This subpacket contains a complete Signature packet body as specified in Section 5.2. It is useful when one signature needs to refer to, or be incorporated in, another signature.

5.2.3.35. Issuer Fingerprint

(1 octet key version number, N octets of fingerprint)

The OpenPGP Key fingerprint of the key issuing the signature. This subpacket **SHOULD** be included in all signatures. If the version of the issuing key is 4 and an Issuer Key ID subpacket (Section 5.2.3.12) is also included in the signature, the key ID of the Issuer Key ID subpacket **MUST** match the low 64 bits of the fingerprint.

Note that the length N of the fingerprint for a version 4 key is 20 octets; for a version 6 key, N is 32. Since the version of the signature is bound to the version of the key, the version octet here **MUST** match the version of the signature. If the version octet does not match the signature version, the receiving implementation **MUST** treat it as a malformed signature (see Section 5.2.5).

5.2.3.36. Intended Recipient Fingerprint

(1 octet key version number, N octets of fingerprint)

The OpenPGP Key fingerprint of the intended recipient primary key. If one or more subpackets of this type are included in a signature, it **SHOULD** be considered valid only in an encrypted context, where the key it was encrypted to is one of the indicated primary keys or one of their subkeys. This can be used to prevent forwarding a signature outside of its intended, encrypted context (see Section 13.12).

Note that the length N of the fingerprint for a version 4 key is 20 octets; for a version 6 key, N is 32.

An implementation **SHOULD** generate this subpacket when creating a signed and encrypted message.

When generating this subpacket in a v6 signature, it **SHOULD** be marked as critical.

5.2.4. Computing Signatures

All signatures are formed by producing a hash over the signature data and then using the resulting hash in the signature algorithm.

When creating or verifying a v6 signature, the salt is fed into the hash context before any other data.

For binary document signatures (type ID 0x00), the document data is hashed directly. For text document signatures (type ID 0x01), the implementation **MUST** first canonicalize the document by converting line endings to <CR><LF> and encoding it in UTF-8 (see [RFC3629]). The resulting UTF-8 byte stream is hashed.

When a v4 signature is made over a key, the hash data starts with the octet 0x99, followed by a two-octet length of the key, followed by the body of the key packet. When a v6 signature is made over a key, the hash data starts with the salt and then octet 0x9B, followed by a four-octet length of the key, followed by the body of the key packet.

A subkey binding signature (type ID 0x18) or primary key binding signature (type ID 0x19) then hashes the subkey using the same format as the main key (also using 0x99 or 0x9B as the first octet). Primary key revocation signatures (type ID 0x20) hash only the key being revoked. Subkey revocation signatures (type ID 0x28) first hash the primary key and then the subkey being revoked.

A certification signature (type ID 0x10 through 0x13) hashes the User ID that is bound to the key into the hash context after the above data. A v3 certification hashes the contents of the User ID or User Attribute packet without the packet header. A v4 or v6 certification hashes the constant 0xB4 for User ID certifications or the constant 0xD1 for User Attribute certifications, followed by a four-octet number giving the length of the User ID or User Attribute data, followed by the User ID or User Attribute data.

When a signature is made over a Signature packet (type ID 0x50, "Third-Party Confirmation signature"), the hash data starts with the salt (v6 signatures only), followed by the octet 0x88, followed by the four-octet length of the signature, and then the body of the Signature packet. (Note that this is a Legacy packet header for a Signature packet with the length-of-length field set to zero.) The unhashed subpacket data of the Signature packet being hashed is not included in the hash, and the unhashed subpacket data length value is set to zero.

Once the data body is hashed, then a trailer is hashed. This trailer depends on the version of the signature.

- A v3 signature hashes five octets of the packet body, starting from the signature type field. This data is the signature type, followed by the four-octet signature creation time.
- A v4 or v6 signature hashes the packet body starting from its first field, the version number, through the end of the hashed subpacket data and a final extra trailer. Thus, the hashed fields are:
  - an octet indicating the signature version (0x04 for v4, and 0x06 for v6),
  - the signature type,
  - the public-key algorithm,
  - the hash algorithm,
  - the hashed subpacket length,

- the hashed subpacket body,
- a second version octet (0x04 for v4, and 0x06 for v6),
- a single octet 0xFF, and
- a number representing the length (in octets) of the hashed data from the Signature packet through the hashed subpacket body. This a four-octet big-endian unsigned integer of the length modulo 2**32.

After all this has been hashed in a single hash context, the resulting hash field is used in the signature algorithm, and its first two octets are placed in the Signature packet, as described in Section 5.2.3.

For worked examples of the data hashed during a signature, see Appendix A.3.1.

5.2.4.1. Notes about Signature Computation

The data actually hashed by OpenPGP varies depending on the signature version, in order to ensure that a signature made using one version cannot be repurposed as a signature with a different version over subtly different data. The hashed data streams differ based on their trailer, most critically in the fifth and sixth octets from the end of the stream. In particular:

- A v3 signature uses the fifth octet from the end to store its signature type ID. This **MUST NOT** be signature type ID `0xFF`.
- All signature versions later than v3 always use a literal `0xFF` in the fifth octet from the end. For these later signature versions, the sixth octet from the end (the octet before the `0xFF`) stores the signature version number.

5.2.5. Malformed and Unknown Signatures

In some cases, a signature packet (or its corresponding One-Pass Signature packet; see Section 5.4) may be malformed or unknown. For example, it might encounter any of the following problems (this is not an exhaustive list):

- An unknown signature type
- An unknown signature version
- An unsupported signature version
- An unknown "critical" subpacket (see Section 5.2.3.7) in the hashed area
- A subpacket with a length that diverges from the expected length
- A hashed subpacket area with length that exceeds the length of the signature packet itself
- A hash algorithm known to be weak (e.g., MD5)
- A mismatch between the expected salt length and the actual salt length of the hash algorithm
- A mismatch between the One-Pass Signature version and the Signature version (see Section 10.3.2.2)
- A signature with a version other than 6, made by a v6 key

When an implementation encounters such a malformed or unknown signature, it **MUST** ignore the signature for validation purposes. It **MUST NOT** indicate a successful signature validation for such a signature. At the same time, it **MUST NOT** halt processing on the packet stream or reject other signatures in the same packet stream just because an unknown or invalid signature exists.

This requirement is necessary for forward compatibility. Producing an output that indicates that no successful signatures were found is preferable to aborting processing entirely.

5.3. Symmetric-Key Encrypted Session Key Packet (Type ID 3)

The Symmetric-Key Encrypted Session Key (SKESK) packet holds the symmetric-key encryption of a session key used to encrypt a message. Zero or more Public-Key Encrypted Session Key packets (Section 5.1) and/or Symmetric-Key Encrypted Session Key packets precede an encryption container (that is, a Symmetrically Encrypted Integrity Protected Data packet or -- for historic data -- a Symmetrically Encrypted Data packet) that holds an encrypted message. The message is encrypted with a session key, and the session key is itself encrypted and stored in the Encrypted Session Key packet(s).

If the encryption container is preceded by one or more Symmetric-Key Encrypted Session Key packets, each specifies a passphrase that may be used to decrypt the message. This allows a message to be encrypted to a number of public keys, and also to one or more passphrases.

The body of this packet starts with a one-octet number giving the version number of the packet type. The currently defined versions are 4 and 6. The remainder of the packet depends on the version.

The versions differ in how they encrypt the session key with the passphrase and in what they encode. The version of the SKESK packet must align with the version of the SEIPD packet (see Section 10.3.2.1). Any new version of the SKESK packet should be registered in the registry established in Section 10.3.2.1.

5.3.1. Version 4 Symmetric-Key Encrypted Session Key Packet Format

A version 4 SKESK packet precedes a v1 SEIPD (see Section 5.13.1). In historic data, it is sometimes found preceding a deprecated SED packet (see Section 5.7). A v4 SKESK packet **MUST NOT** precede a v2 SEIPD packet (see Section 10.3.2.1).

A version 4 Symmetric-Key Encrypted Session Key packet consists of:

- A one-octet version number with value 4.
- A one-octet number describing the symmetric algorithm used.
- An S2K specifier. The length of the S2K specifier depends on its type (see Section 3.7.1).
- Optionally, the encrypted session key itself, which is decrypted with the S2K object.

If the encrypted session key is not present (which can be detected on the basis of packet length and S2K specifier size), then the S2K algorithm applied to the passphrase produces the session key for decrypting the message, using the symmetric cipher algorithm from the Symmetric-Key Encrypted Session Key packet.

If the encrypted session key is present, the result of applying the S2K algorithm to the passphrase is used to decrypt just that encrypted session key field, using CFB mode with an IV of all zeros. The decryption result consists of a one-octet algorithm identifier that specifies the symmetric-key encryption algorithm used to encrypt the following encryption container, followed by the session key octets themselves.

Note: because an all-zero IV is used for this decryption, the S2K specifier **MUST** use a salt value, a Salted S2K, an Iterated and Salted S2K, or Argon2. The salt value will ensure that the decryption key is not repeated even if the passphrase is reused.

5.3.2. Version 6 Symmetric-Key Encrypted Session Key Packet Format

A version 6 SKESK packet precedes a version 2 SEIPD packet (see Section 5.13.2). A v6 SKESK packet **MUST NOT** precede a v1 SEIPD packet or a deprecated Symmetrically Encrypted Data packet (see Section 10.3.2.1).

A version 6 Symmetric-Key Encrypted Session Key packet consists of:

- A one-octet version number with value 6.
- A one-octet scalar octet count for the 5 fields following this octet.
- A one-octet symmetric cipher algorithm ID (from Table 21).
- A one-octet AEAD algorithm identifier (from Table 25).
- A one-octet scalar octet count of the following field.
- An S2K specifier. The length of the S2K specifier depends on its type (see Section 3.7.1).
- A starting initialization vector of a size specified by the AEAD algorithm.
- The encrypted session key itself.
- An authentication tag for the AEAD mode.

A key-encryption key (KEK) is derived using HKDF [RFC5869] with SHA256 [RFC6234] as the hash algorithm. The Initial Keying Material (IKM) for HKDF is the key derived from S2K. No salt is used. The info parameter is comprised of the Packet Type ID in OpenPGP format encoding (bits 7 and 6 are set, and bits 5-0 carry the packet type ID), the packet version, and the cipher-algo and AEAD-mode used to encrypt the key material.

Then, the session key is encrypted using the resulting key, with the AEAD algorithm specified for version 2 of the Symmetrically Encrypted Integrity Protected Data packet. Note that no chunks are used and that there is only one authentication tag. The Packet Type ID encoded in OpenPGP format (bits 7 and 6 are set, and bits 5-0 carry the packet type ID), the packet version number, the cipher algorithm ID, and the AEAD algorithm ID are given as additional data. For example, the additional data used with AES-128 with OCB consists of the octets 0xC3, 0x06, 0x07, and 0x02.

5.4. One-Pass Signature Packet (Type ID 4)

The One-Pass Signature packet precedes the signed data and contains enough information to allow the receiver to begin calculating any hashes needed to verify the signature. It allows the Signature packet to be placed at the end of the message so that the signer can compute the entire signed message in one pass.

The body of this packet consists of:

- A one-octet version number. The currently defined versions are 3 and 6. Any new One-Pass Signature packet version should be registered in the registry established in Section 10.3.2.2.
- A one-octet signature type ID. Signature types are described in Section 5.2.1.
- A one-octet number describing the hash algorithm used.
- A one-octet number describing the public-key algorithm used.
- Only for v6 packets, a variable-length field containing:
  - A one-octet salt size. The value **MUST** match the value defined for the hash algorithm as specified in Table 23.
  - The salt; a random value of the specified size. The value **MUST** match the salt field of the corresponding Signature packet.
- Only for v3 packets, an eight-octet number holding the Key ID of the signing key.
- Only for v6 packets, 32 octets of the fingerprint of the signing key. Since a v6 signature can only be made by a v6 key, the length of the fingerprint is fixed.
- A one-octet number holding a flag showing whether the signature is nested. A zero value indicates that the next packet is another One-Pass Signature packet that describes another signature to be applied to the same message data.

When generating a one-pass signature, the OPS packet version **MUST** correspond to the version of the associated signature packet, except for the historical accident that v4 keys use a v3 one-pass signature packet (there is no v4 OPS). See Section 10.3.2.2 for the full correspondence of versions between Keys, Signatures, and One-Pass Signatures.

Note that if a message contains more than one one-pass signature, then the Signature packets bracket the message; that is, the first Signature packet after the message corresponds to the last one-pass packet and the final Signature packet corresponds to the first one-pass packet.

5.5. Key Material Packets

A key material packet contains all the information about a public or private key. There are four variants of this packet type: two major versions (versions 4 and 6) and two strongly deprecated versions (versions 2 and 3). Consequently, this section is complex.

For historical reasons, versions 1 and 5 of the key packets are unspecified.

5.5.1. Key Packet Variants

5.5.1.1. Public-Key Packet (Type ID 6)

A Public-Key packet starts a series of packets that forms an OpenPGP key (sometimes called an OpenPGP certificate).

5.5.1.2. Public-Subkey Packet (Type ID 14)

A Public-Subkey packet (type ID 14) has exactly the same format as a Public-Key packet, but it denotes a subkey. One or more subkeys may be associated with a top-level key. By convention, the top-level key offers certification capability, but it does not provide encryption services, while a dedicated subkey provides encryption (see Section 10.1.5).

5.5.1.3. Secret-Key Packet (Type ID 5)

A Secret-Key packet contains all the information that is found in a Public-Key packet, including the public-key material, but it also includes the secret-key material after all the public-key fields.

5.5.1.4. Secret-Subkey Packet (Type ID 7)

A Secret-Subkey packet (type ID 7) is the subkey analog of the Secret-Key packet and has exactly the same format.

5.5.2. Public-Key Packet Formats

There are four versions of key-material packets. The V2 and V3 versions have been deprecated since 1998. The V4 version has been deprecated by this document.

OpenPGP implementations **SHOULD** create keys with version 6 format. V4 keys are deprecated; an implementation **SHOULD NOT** generate a v4 key but **SHOULD** accept it. V3 keys are deprecated; an implementation **MUST NOT** generate a v3 key but **MAY** accept it. V2 keys are deprecated; an implementation **MUST NOT** generate a v2 key but **MAY** accept it.

Any new Key version must be registered in the registry established in Section 10.3.2.2.

5.5.2.1. Version 3 Public Keys

V2 keys are identical to v3 keys except for the version number. A version 3 public key or public-subkey packet contains:

- A one-octet version number (3).
- A four-octet number denoting the time that the key was created.
- A two-octet number denoting the time in days that the key is valid. If this number is zero, then it does not expire.
- A one-octet number denoting the public-key algorithm of the key.
- A series of multiprecision integers comprising the key material:
  - MPI of RSA public modulus n.
  - MPI of RSA public encryption exponent e.

V3 keys are deprecated. They contain three weaknesses. First, it is relatively easy to construct a v3 key that has the same Key ID as any other key because the Key ID is simply the low 64 bits of the public modulus. Second, because the fingerprint of a v3 key hashes the key material, but not its length, there is an increased opportunity for fingerprint collisions. Third, there are weaknesses in the MD5 hash algorithm that cause developers to prefer other algorithms. See Section 5.5.4 for a fuller discussion of Key IDs and fingerprints.

5.5.2.2. Version 4 Public Keys

The version 4 format is similar to the version 3 format except for the absence of a validity period. This has been moved to the Signature packet. In addition, fingerprints of version 4 keys are calculated differently from version 3 keys, as described in Section 5.5.4.

A version 4 packet contains:

- A one-octet version number (4).
- A four-octet number denoting the time that the key was created.
- A one-octet number denoting the public-key algorithm of the key.
- A series of values comprising the key material. This is algorithm specific and described in Section 5.5.5.

5.5.2.3. Version 6 Public Keys

The version 6 format is similar to the version 4 format except for the addition of a count for the key material. This count helps parsing secret key packets (which are an extension of the public key packet format) in the case of an unknown algorithm. In addition, fingerprints of version 6 keys are calculated differently from version 4 keys, as described in Section 5.5.4.

A version 6 packet contains:

- A one-octet version number (6).
- A four-octet number denoting the time that the key was created.
- A one-octet number denoting the public-key algorithm of the key.
- A four-octet scalar octet count for the following public key material.
- A series of values comprising the public key material. This is algorithm specific and described in Section 5.5.5.

5.5.3. Secret-Key Packet Formats

The Secret-Key and Secret-Subkey packets contain all the data of the Public-Key and Public-Subkey packets, with additional algorithm-specific secret-key data appended, usually in encrypted form.

The packet contains:

- The fields of a Public-Key or Public-Subkey packet, as described above.

- One octet (the "S2K usage octet") indicating whether and how the secret key material is protected by a passphrase. Zero indicates that the secret-key data is not encrypted. 255 (MalleableCFB), 254 (CFB), or 253 (AEAD) indicates that a string-to-key specifier and other parameters will follow. Any other value is a symmetric-key encryption algorithm identifier. A version 6 packet **MUST NOT** use the value 255 (MalleableCFB).
- Only for a version 6 packet where the secret key material is encrypted (that is, where the previous octet is not zero), a one-octet scalar octet count of the cumulative length of all the following conditionally included string-to-key parameter fields.
- Conditionally included string-to-key parameter fields:
  - If the string-to-key usage octet was 255, 254, or 253, a one-octet symmetric encryption algorithm.
  - If the string-to-key usage octet was 253 (AEAD), a one-octet AEAD algorithm.
  - Only for a version 6 packet, and if the string-to-key usage octet was 254 or 253, a one-octet count of the size of the one field following this octet.
  - If the string-to-key usage octet was 255, 254, or 253, a string-to-key specifier. The length of the string-to-key specifier depends on its type (see Section 3.7.1).
  - If the string-to-key usage octet was 253 (AEAD), an IV of a size specified by the AEAD algorithm (see Section 5.13.2), which is used as the nonce for the AEAD algorithm.
  - If the string-to-key usage octet was 255, 254, or a cipher algorithm ID (that is, the secret data uses some form of CFB encryption), an IV of the same length as the cipher's block size.
- Plain or encrypted multiprecision integers comprising the secret key data. This is algorithm specific and described in Section 5.5.5. If the string-to-key usage octet is 253 (AEAD), then an AEAD authentication tag is at the end of that data. If the string-to-key usage octet is 254 (CFB), a 20-octet SHA-1 hash of the plaintext of the algorithm-specific portion is appended to plaintext and encrypted with it. If the string-to-key usage octet is 255 (MalleableCFB) or another non-zero value (that is, a symmetric-key encryption algorithm identifier), a two-octet checksum of the plaintext of the algorithm-specific portion (sum of all octets, mod 65536) is appended to plaintext and encrypted with it. (This is deprecated and **SHOULD NOT** be used; see below.)
- Only for a version 3 or 4 packet where the string-to-key usage octet is zero, a two-octet checksum of the algorithm-specific portion (sum of all octets, mod 65536).

The details about storing algorithm-specific secrets above are summarized in Table 2.

Note that the version 6 packet format adds two count values to help parsing packets with unknown S2K or public key algorithms.

Secret MPI values can be encrypted using a passphrase. If a string-to-key specifier is given, it describes the algorithm for converting the passphrase to a key; otherwise, a simple MD5 hash of the passphrase is used. An implementation producing a passphrase-protected secret key packet **MUST** use a string-to-key specifier; the simple hash is for read-only backward compatibility, though implementations **MAY** continue to use existing private keys in the old format. The cipher for encrypting the MPIs is specified in the Secret-Key packet.

Encryption/decryption of the secret data is done using the key created from the passphrase and the initialization vector from the packet. If the string-to-key usage octet is not 253, CFB mode is used. A different mode is used with v3 keys (which are only RSA) than with other key formats. With v3 keys, the MPI bit count prefix (that is, the first two octets) is not encrypted. Only the MPI non-prefix data is encrypted. Furthermore, the CFB state is resynchronized at the beginning of each new MPI value so that the CFB block boundary is aligned with the start of the MPI data.

With v4 and v6 keys, a simpler method is used. All secret MPI values are encrypted, including the MPI bit count prefix.

If the string-to-key usage octet is 253, the KEK is derived using HKDF [RFC5869] to provide key separation. SHA256 [RFC6234] is used as the hash algorithm for HKDF. IKM for HKDF is the key derived from S2K. No salt is used. The info parameter is comprised of the Packet Type ID encoded in OpenPGP format (bits 7 and 6 are set, and bits 5-0 carry the packet type ID), the packet version, and the cipher-algo and AEAD-mode used to encrypt the key material.

Then, the encrypted MPI values are encrypted as one combined plaintext using one of the AEAD algorithms specified for version 2 of the Symmetrically Encrypted Integrity Protected Data packet. Note that no chunks are used and that there is only one authentication tag. As additional data, the Packet Type ID in OpenPGP format encoding (bits 7 and 6 are set, and bits 5-0 carry the packet type ID), followed by the public key packet fields, starting with the packet version number, are passed to the AEAD algorithm. For example, the additional data used with a Secret-Key packet of version 4 consists of the octets 0xC5, 0x04, followed by four octets of creation time, one octet denoting the public-key algorithm, and the algorithm-specific public-key parameters. For a Secret-Subkey packet, the first octet would be 0xC7. For a version 6 key packet, the second octet would be 0x06, and the four-octet octet count of the public key material would be included as well (see Section 5.5.2).

The two-octet checksum that follows the algorithm-specific portion is the algebraic sum, mod 65536, of the plaintext of all the algorithm-specific octets (including the MPI prefix and data). With v3 keys, the checksum is stored in the clear. With v4 keys, the checksum is encrypted like the algorithm-specific data. This value is used to check that the passphrase was correct. However, this checksum is deprecated, and an implementation **SHOULD NOT** use it; instead, an implementation should use the SHA-1 hash denoted with a usage octet of 254. The reason for this is that there are some attacks that involve modifying the secret key undetected. If the string-to-key usage octet is 253, no checksum or SHA-1 hash is used, but the authentication tag of the AEAD algorithm follows.

When decrypting the secret key material using any of these schemes (that is, where the usage octet is non-zero), the resulting cleartext octet stream must be well formed. In particular, an implementation **MUST NOT** interpret octets beyond the unwrapped cleartext octet stream as part of any of the unwrapped MPI objects. Furthermore, an implementation **MUST** reject any secret key material whose cleartext length does not align with the lengths of the unwrapped MPI objects as unusable.

5.5.4. Key IDs and Fingerprints

Every OpenPGP key has a fingerprint and a key ID. The computation of these values differs based on the key version. The fingerprint length varies with the key version, but the key ID (which is only used in v3 PKESK packets; see Section 5.1.1) is always 64 bits. The following registry represents the subsections below:

Key Version	Fingerprint	Fingerprint Length (Bits)	Key ID	Reference
3	MD5(MPIs without length octets)	128	low 64 bits of RSA modulus	Section 5.5.4.1
4	SHA1(normalized pubkey packet)	160	last 64 bits of fingerprint	Section 5.5.4.2
6	SHA256(normalized pubkey packet)	256	first 64 bits of fingerprint	Section 5.5.4.3

Table 12: OpenPGP Key ID and Fingerprint Registry

5.5.4.1. Version 3 Key ID and Fingerprint

For a v3 key, the eight-octet Key ID consists of the low 64 bits of the public modulus of the RSA key.

The fingerprint of a v3 key is formed by hashing the body (but not the two-octet length) of the MPIs that form the key material (public modulus n, followed by exponent e) with MD5. Note that both v3 keys and MD5 are deprecated.

5.5.4.2. Version 4 Key ID and Fingerprint

A v4 fingerprint is the 160-bit SHA-1 hash of the octet 0x99, followed by the two-octet packet length, followed by the entire Public-Key packet starting with the version field. The Key ID is the low-order 64 bits of the fingerprint. Here are the fields of the hash material, including an example of an Ed25519 key:

a.1) 0x99 (1 octet)
a.2) two-octet, big-endian scalar octet count of (b)-(e)

b) version number = 4 (1 octet)
c) timestamp of key creation (4 octets)
d) algorithm (1 octet): 27 = Ed25519 (example)
e) algorithm-specific fields

Algorithm-specific fields for Ed25519 keys (example):

e.1) 32 octets representing the public key

5.5.4.3. Version 6 Key ID and Fingerprint

A v6 fingerprint is the 256-bit SHA2-256 hash of the octet 0x9B, followed by the four-octet packet length, followed by the entire Public-Key packet starting with the version field. The Key ID is the high-order 64 bits of the fingerprint. Here are the fields of the hash material, including an example of an Ed25519 key:

a.1) 0x9B (1 octet)
a.2) four-octet scalar octet count of (b)-(f)

b) version number = 6 (1 octet)
c) timestamp of key creation (4 octets)
d) algorithm (1 octet): 27 = Ed25519 (example)
e) four-octet scalar octet count for the following key material
f) algorithm-specific fields

Algorithm-specific fields for Ed25519 keys (example):

e.1) 32 octets representing the public key

Note that it is possible for there to be collisions of Key IDs -- that is, two different keys with the same Key ID. Note that there is a much smaller, but still non-zero, probability that two different keys have the same fingerprint.

Also note that if v3, v4, and v6 format keys share the same RSA key material, they will have different Key IDs as well as different fingerprints.

Finally, the Key ID and fingerprint of a subkey are calculated in the same way as for a primary key, including the 0x99 (v4 key) or 0x9B (v6 key) as the first octet (even though this is not a valid packet type ID for a public subkey).

5.5.5. Algorithm-Specific Parts of Keys

The public and secret key formats specify algorithm-specific parts of a key. The following sections describe them in detail.

5.5.5.1. Algorithm-Specific Part for RSA Keys

For RSA keys, the public key is this series of multiprecision integers:

- MPI of RSA public modulus n,
- MPI of RSA public encryption exponent e.

The secret key is this series of multiprecision integers:

- MPI of RSA secret exponent d;
- MPI of RSA secret prime value p;
- MPI of RSA secret prime value q (p < q); and
- MPI of u, the multiplicative inverse of p, mod q.

5.5.5.2. Algorithm-Specific Part for DSA Keys

For DSA keys, the public key is this series of multiprecision integers:

- MPI of DSA prime p;
- MPI of DSA group order q (q is a prime divisor of p-1);
- MPI of DSA group generator g; and
- MPI of DSA public-key value y (= g**x mod p where x is secret).

The secret key is this single multiprecision integer:

- MPI of DSA secret exponent x.

5.5.5.3. Algorithm-Specific Part for Elgamal Keys

For Elgamal keys, the public key is this series of multiprecision integers:

- MPI of Elgamal prime p;
- MPI of Elgamal group generator g; and
- MPI of Elgamal public key value y (= g**x mod p where x is secret).

The secret key is this single multiprecision integer:

- MPI of Elgamal secret exponent x.

5.5.5.4. Algorithm-Specific Part for ECDSA Keys

For ECDSA keys, the public key is this series of values:

- A variable-length field containing a curve OID, which is formatted as follows:
  - A one-octet size of the following field; values 0 and 0xFF are reserved for future extensions.
  - The octets representing a curve OID, as defined in Section 9.2.
- An MPI of an EC point representing a public key.

The secret key is this single multiprecision integer:

- An MPI of an integer representing the secret key, which is a scalar of the public EC point.

5.5.5.5. Algorithm-Specific Part for EdDSALegacy Keys (Deprecated)

For EdDSALegal keys (deprecated), the public key is this series of values:

- A variable-length field containing a curve OID, formatted as follows:
  - A one-octet size of the following field; values 0 and 0xFF are reserved for future extensions.
  - The octets representing a curve OID, as defined in Section 9.2.
- An MPI of an EC point representing a public key Q in prefixed native form (see Section 11.2.2).

The secret key is this single multiprecision integer:

- An MPI-encoded octet string representing the native form of the secret key in the curve-specific format, as described in Section 9.2.1.

Note that the native form for an EdDSA secret key is a fixed-width sequence of unstructured random octets, with size corresponding to the specific curve. That sequence of random octets is used with a cryptographic digest to produce both a curve-specific secret scalar and a prefix used when making a signature. See Section 5.1.5 of [RFC8032] for more details about how to use the native octet strings for Ed25519Legacy. The value stored in an OpenPGP EdDSALegacy secret key packet is the original sequence of random octets.

Note that the only curve defined for use with EdDSALegacy is the Ed25519Legacy OID.

5.5.5.6. Algorithm-Specific Part for ECDH Keys

For ECDH keys, the public key is this series of values:

- A variable-length field containing a curve OID, which is formatted as follows:
  - A one-octet size of the following field; values 0 and 0xFF are reserved for future extensions.
  - The octets representing a curve OID, as defined in Section 9.2.
- An MPI of an EC point representing a public key, in the point format associated with the curve, as specified in Section 9.2.1.
- A variable-length field containing key derivation function (KDF) parameters, which is formatted as follows:
  - A one-octet size of the following fields; values 0 and 0xFF are reserved for future extensions.
  - A one-octet value 1, reserved for future extensions.
  - A one-octet hash function ID used with a KDF.
  - A one-octet algorithm ID for the symmetric algorithm that is used to wrap the symmetric key for message encryption; see Section 11.5 for details.

The secret key is this single multiprecision integer:

- An MPI representing the secret key, in the curve-specific format described in Section 9.2.1.

5.5.5.6.1. ECDH Secret Key Material

When curve NIST P-256, NIST P-384, NIST P-521, brainpoolP256r1, brainpoolP384r1, or brainpoolP512r1 are used in ECDH, their secret keys are represented as a simple integer in standard MPI form. Other curves are presented on the wire differently (though still as a single MPI), as described below and in Section 9.2.1.

5.5.5.6.1.1. Curve25519Legacy ECDH Secret Key Material (Deprecated)

A Curve25519Legacy secret key is stored as a standard integer in big-endian MPI form. Curve25519Legacy **MUST NOT** be used in key packets version 6 or above. Note that this form is in reverse octet order from the little-endian "native" form found in [RFC7748].

Note also that the integer for a Curve25519Legacy secret key for OpenPGP **MUST** have the appropriate form; that is, it **MUST** be divisible by 8, **MUST** be at least 2**254, and **MUST** be less than 2**255. The length of this MPI in bits is by definition always 255, so the two leading octets of the MPI will always be `00 FF`, and reversing the following 32 octets from the wire will produce the "native" form.

When generating a new Curve25519Legacy secret key from 32 fully random octets, the following pseudocode produces the MPI wire format (note the similarity to `decodeScalar25519` as described in [RFC7748]):

```
def curve25519Legacy_MPI_from_random(octet_list):
    octet_list[0] &= 248
    octet_list[31] &= 127
    octet_list[31] |= 64
    mpi_header = [ 0x00, 0xFF ]
    return mpi_header || reversed(octet_list)
```

5.5.5.7. Algorithm-Specific Part for X25519 Keys

For X25519 keys, the public key is this single value:

- 32 octets of the native public key.

The secret key is this single value:

- 32 octets of the native secret key.

See Section 6.1 of [RFC7748] for more details about how to use the native octet strings. The value stored in an OpenPGP X25519 secret key packet is the original sequence of random octets. The value stored in an OpenPGP X25519 public key packet is the value X25519(secretKey, 9).

5.5.5.8. Algorithm-Specific Part for X448 Keys

For X448 keys, the public key is this single value:

- 56 octets of the native public key.

The secret key is this single value:

- 56 octets of the native secret key.

See Section 6.2 of [RFC7748] for more details about how to use the native octet strings. The value stored in an OpenPGP X448 secret key packet is the original sequence of random octets. The value stored in an OpenPGP X448 public key packet is the value X448(secretKey, 5).

5.5.5.9. Algorithm-Specific Part for Ed25519 Keys

For Ed25519 keys, the public key is this single value:

- 32 octets of the native public key.

The secret key is this single value:

- 32 octets of the native secret key.

See Section 5.1.5 of [RFC8032] for more details about how to use the native octet strings. The value stored in an OpenPGP Ed25519 secret key packet is the original sequence of random octets.

5.5.5.10. Algorithm-Specific Part for Ed448 Keys

For Ed448 keys, the public key is this single value:

- 57 octets of the native public key.

The secret key is this single value:

- 57 octets of the native secret key.

See Section 5.2.5 of [RFC8032] for more details about how to use the native octet strings. The value stored in an OpenPGP Ed448 secret key packet is the original sequence of random octets.

5.6. Compressed Data Packet (Type ID 8)

The Compressed Data packet contains compressed data. Typically, this packet is found as the contents of an encrypted packet, or following a Signature or One-Pass Signature packet, and contains a literal data packet.

The body of this packet consists of:

- One octet that gives the algorithm used to compress the packet.
- Compressed data, which makes up the remainder of the packet.

A Compressed Data packet's body contains data that is a compression of a series of OpenPGP packets. See Section 10 for details on how messages are formed.

A ZIP-compressed series of packets is compressed into raw DEFLATE blocks [RFC1951].

A ZLIB-compressed series of packets is compressed with raw ZLIB-style blocks [RFC1950].

A BZip2-compressed series of packets is compressed using the BZip2 [BZ2] algorithm.

An implementation that generates a Compressed Data packet **MUST** use the non-Legacy format for packet framing (see Section 4.2.1). It **MUST NOT** generate a Compressed Data packet with Legacy format (Section 4.2.2)

An implementation that deals with either historic data or data generated by legacy implementations predating support for [RFC2440] **MAY** interpret Compressed Data packets that use the Legacy format for packet framing.

5.7. Symmetrically Encrypted Data Packet (Type ID 9)

The Symmetrically Encrypted Data packet contains data encrypted with a symmetric-key algorithm. When it has been decrypted, it contains other packets (usually a literal data packet or compressed data packet, but in theory, it could be other Symmetrically Encrypted Data packets or sequences of packets that form whole OpenPGP messages).

This packet is obsolete. An implementation **MUST NOT** create this packet. An implementation **SHOULD** reject such a packet and stop processing the message. If an implementation chooses to process the packet anyway, it **MUST** return a clear warning that a non-integrity-protected packet has been processed.

This packet format is impossible to handle safely in general because the ciphertext it provides is malleable. See Section 13.7 about selecting a better OpenPGP encryption container that does not have this flaw.

The body of this packet consists of:

- A random prefix, containing block-size random octets (for example, 16 octets for a 128-bit block length) followed by a copy of the last two octets, encrypted together using Cipher Feedback (CFB) mode, with an IV of all zeros.
- Data encrypted using CFB mode, with the last block-size octets of the first ciphertext as the IV.

The symmetric cipher used may be specified in a Public-Key or Symmetric-Key Encrypted Session Key packet that precedes the Symmetrically Encrypted Data packet. In that case, the cipher algorithm ID is prefixed to the session key before it is encrypted. If no packets of these types precede the encrypted data, the IDEA algorithm is used with the session key calculated as the MD5 hash of the passphrase, though this use is deprecated.

The data is encrypted in CFB mode (see Section 12.9). For the random prefix, the IV is specified as all zeros. Instead of achieving randomized encryption through an IV, a string of length equal to the block size of the cipher plus two is encrypted for this purpose. The first block-size octets (for example, 16 octets for a 128-bit block length) are random, and the following two octets are copies of the last two octets of the first block-size random octets. For example, for a 16-octet block length, octet 17 is a copy of octet 15, and octet 18 is a copy of octet 16. For a cipher of block length 8, octet 9 is a copy of octet 7, and octet 10 is a copy of octet 8. (In both of these examples, we consider the first octet to be numbered 1.)

After encrypting these block-size-plus-two octets, a new CFB context is created for the encryption of the data, with the last block-size octets of the first ciphertext as the IV. (Alternatively and equivalently, the CFB state is resynchronized: the last block-size octets of ciphertext are passed through the cipher, and the block boundary is reset.)

The repetition of two octets in the random prefix allows the receiver to immediately check whether the session key is incorrect. See Section 13.4 for hints on the proper use of this "quick check".

5.8. Marker Packet (Type ID 10)

The body of the Marker packet consists of:

- The three octets 0x50, 0x47, 0x50 (which spell "PGP" in UTF-8).

Such a packet **MUST** be ignored when received.

5.9. Literal Data Packet (Type ID 11)

A Literal Data packet contains the body of a message; that is, data that is not to be further interpreted.

The body of this packet consists of:

- A one-octet field that describes how the data is formatted.

  If it is a `b` (0x62), then the Literal packet contains binary data. If it is a `u` (0x75), then the Literal packet contains UTF-8-encoded text data and thus may need line ends converted to local form or other text mode changes.

  Previous versions of the OpenPGP specification used `t` (0x74) to indicate textual data but did not specify the character encoding. Implementations **SHOULD NOT** emit this value. An implementation that receives a literal data packet with this value in the format field **SHOULD** interpret the packet data as UTF-8 encoded text, unless reliable (not attacker-controlled) context indicates a specific alternate text encoding. This mode is deprecated due to its ambiguity.

  Some implementations predating [RFC2440] also defined a value of `l` as a "local" mode for machine-local conversions. [RFC1991] incorrectly states that this local mode flag is `1` (ASCII numeral one). Both of these local modes are deprecated.

- The file name as a string (one-octet length, followed by a file name). This may be a zero-length string. Commonly, if the source of the encrypted data is a file, it will be the name of the encrypted file. An implementation **MAY** consider the file name in the Literal packet to be a more authoritative name than the actual file name.
- A four-octet number that indicates a date associated with the literal data. Commonly, the date might be the modification date of a file, or the time the packet was created, or a zero that indicates no specific time.
- The remainder of the packet is literal data.

  Text data **MUST** be encoded with UTF-8 (see [RFC3629]) and stored with <CR><LF> text endings (that is, network-normal line endings). These should be converted to native line endings by the receiving implementation.

Note that OpenPGP signatures do not include the formatting octet, the file name, and the date field of the literal packet in a signature hash; therefore, those fields are not protected against tampering in a signed document. A receiving implementation **MUST NOT** treat those fields as though they were cryptographically secured by the surrounding signature when either representing them to the user or acting on them.

Due to their inherent malleability, an implementation that generates a literal data packet **SHOULD** avoid storing any significant data in these fields. If the implementation is certain that the data is textual and is encoded with UTF-8 (for example, if it will follow this literal data packet with a signature packet of type 0x01 (see Section 5.2.1), it **MAY** set the format octet to `u`. Otherwise, it **MUST** set the format octet to `b`. It **SHOULD** set the filename to the empty string (encoded as a single zero octet) and the timestamp to zero (encoded as four zero octets).

An application that wishes to include such filesystem metadata within a signature is advised to sign an encapsulated archive (for example, [PAX]).

An implementation that generates a Literal Data packet **MUST** use the OpenPGP format for packet framing (see Section 4.2.1). It **MUST NOT** generate a Literal Data packet with Legacy format (Section 4.2.2).

An implementation that deals with either historic data or data generated by an implementation that predates support for [RFC2440] **MAY** interpret Literal Data packets that use the Legacy format for packet framing.

5.9.1. Special Filename _CONSOLE (Deprecated)

The Literal Data packet's filename field has a historical special case for the special name `_CONSOLE`. When the filename field is `_CONSOLE`, the message is considered to be "for your eyes only". This advises that the message data is unusually sensitive, and the receiving program should process it more carefully, perhaps avoiding storing the received data to disk, for example.

An OpenPGP deployment that generates literal data packets **MUST NOT** depend on this indicator being honored in any particular way. It cannot be enforced, and the field itself is not covered by any cryptographic signature.

It is **NOT RECOMMENDED** to use this special filename in a newly generated literal data packet.

5.10. Trust Packet (Type ID 12)

The Trust packet is used only within keyrings and is not normally exported. Trust packets contain data that record the user's specifications of which keyholders are trustworthy introducers, along with other information that implementation uses for trust information. The format of Trust packets is defined by a given implementation.

Trust packets **SHOULD NOT** be emitted to output streams that are transferred to other users, and they **SHOULD** be ignored on any input other than local keyring files.

5.11. User ID Packet (Type ID 13)

A User ID packet consists of UTF-8 text that is intended to represent the name and email address of the keyholder. By convention, it includes a mail name-addr as described in [RFC2822], but there are no restrictions on its content. The packet length in the header specifies the length of the User ID.

5.12. User Attribute Packet (Type ID 17)

The User Attribute packet is a variation of the User ID packet. It is capable of storing more types of data than the User ID packet, which is limited to text. Like the User ID packet, a User Attribute packet may be certified by the key owner ("self-signed") or any other key owner who cares to certify it. Except as noted, a User Attribute packet may be used anywhere that a User ID packet may be used.

While User Attribute packets are not a required part of the OpenPGP specification, implementations **SHOULD** provide at least enough compatibility to properly handle a certification signature on the User Attribute packet. A simple way to do this is by treating the User Attribute packet as a User ID packet with opaque contents, but an implementation may use any method desired.

The User Attribute packet is made up of one or more attribute subpackets. Each subpacket consists of a subpacket header and a body. The header consists of:

- the subpacket length (1, 2, or 5 octets)
- the subpacket type ID (1 octet)

and is followed by the subpacket specific data.

The following table lists the currently known subpackets:

ID	Attribute Subpacket	Reference
1	Image Attribute Subpacket	[RFC9580], Section 5.12.1

ID	Attribute Subpacket	Reference
100-110	Private/Experimental Use	[RFC9580]

Table 13: OpenPGP User Attribute Subpacket Types Registry

An implementation **SHOULD** ignore any subpacket of a type that it does not recognize.

5.12.1. Image Attribute Subpacket

The Image Attribute subpacket is used to encode an image, presumably (but not required to be) that of the key owner.

The Image Attribute subpacket begins with an image header. The first two octets of the image header contain the length of the image header. Note that unlike other multi-octet numerical values in this document, due to a historical accident, this value is encoded as a little-endian number. The image header length is followed by a single octet for the image header version. The only currently defined version of the image header is 1, which is a 16-octet image header. The first three octets of a version 1 image header are thus 0x10, 0x00, 0x01.

Version	Reference
1	Section 5.12.1

Table 14: OpenPGP Image Attribute Version Registry

The fourth octet of a version 1 image header designates the encoding format of the image. The only currently defined encoding format is the value 1 to indicate JPEG. Image format IDs 100 through 110 are reserved for Private or Experimental Use. The rest of the version 1 image header is made up of 12 reserved octets, all of which **MUST** be set to 0.

ID	Encoding	Reference
1	JPEG	JPEG File Interchange Format [JFIF]
100-110	Private or Experimental Use	[RFC9580]

Table 15: OpenPGP Image Attribute Encoding Format Registry

The rest of the image subpacket contains the image itself. As the only currently defined image type is JPEG, the image is encoded in the JPEG File Interchange Format (JFIF), a standard file format for JPEG images [JFIF].

An implementation **MAY** try to determine the type of an image by examination of the image data if it is unable to handle a particular version of the image header or if a specified encoding format value is not recognized.

5.13. Symmetrically Encrypted Integrity Protected Data Packet (Type ID 18)

The SEIPD packet contains integrity-protected and encrypted data. When it has been decrypted, it will contain other packets forming an OpenPGP Message (see Section 10.3).

The first octet of this packet is always used to indicate the version number, but different versions contain ciphertext that is structured differently. Version 1 of this packet contains data encrypted with a symmetric-key algorithm and is thus protected against modification by the SHA-1 hash algorithm. This mechanism was introduced in [RFC4880] and offers some protections against ciphertext malleability.

Version 2 of this packet contains data encrypted with an AEAD construction. This offers a more cryptographically rigorous defense against ciphertext malleability. See Section 13.7 for more details on choosing between these formats.

Any new version of the SEIPD packet should be registered in the registry established in Section 10.3.2.1.

5.13.1. Version 1 Symmetrically Encrypted Integrity Protected Data Packet Format

A version 1 Symmetrically Encrypted Integrity Protected Data packet consists of:

- A one-octet version number with value 1.
- Encrypted data -- the output of the selected symmetric-key cipher operating in CFB mode.

The symmetric cipher used **MUST** be specified in a Public-Key or Symmetric-Key Encrypted Session Key packet that precedes the Symmetrically Encrypted Integrity Protected Data packet. In either case, the cipher algorithm ID is prefixed to the session key before it is encrypted.

The data is encrypted in CFB mode (see Section 12.9). The IV is specified as all zeros. Instead of achieving randomized encryption through an IV, OpenPGP prefixes an octet string to the data before it is encrypted for this purpose. The length of the octet string equals the block size of the cipher in octets, plus two. The first octets in the group, of length equal to the block size of the cipher, are random; the last two octets are each copies of their 2nd preceding octet. For example, with a cipher whose block size is 128 bits or 16 octets, the prefix data will contain 16 random octets, then two more octets, which are copies of the 15th and 16th octets, respectively. Unlike the deprecated Symmetrically Encrypted Data packet (Section 5.7), this prefix data is encrypted in the same CFB context, and no special CFB resynchronization is done.

The repetition of 16 bits in the random data prefixed to the message allows the receiver to immediately check whether the session key is incorrect. See Section 13.4 for hints on the proper use of this "quick check".

Two constant octets with the values 0xD3 and 0x14 are appended to the plaintext. Then, the plaintext of the data to be encrypted is passed through the SHA-1 hash function. The input to the hash function includes the prefix data described above; it includes all of the plaintext, including

the trailing constant octets 0xD3, 0x14. The 20 octets of the SHA-1 hash are then appended to the plaintext (after the constant octets 0xD3, 0x14) and encrypted along with the plaintext using the same CFB context. This trailing checksum is known as the Modification Detection Code (MDC).

During decryption, the plaintext data should be hashed with SHA-1, including the prefix data as well as the trailing constant octets 0xD3, 0x14, but excluding the last 20 octets containing the SHA-1 hash. The computed SHA-1 hash is then compared with the last 20 octets of plaintext. A mismatch of the hash indicates that the message has been modified and **MUST** be treated as a security problem. Any failure **SHOULD** be reported to the user.

NON-NORMATIVE EXPLANATION

The MDC system, as the integrity protection mechanism of version 1 of the Symmetrically Encrypted Integrity Protected Data packet is called, was created to provide an integrity mechanism that is less strong than a signature, yet stronger than bare CFB encryption.

CFB encryption has a limitation as damage to the ciphertext will corrupt the affected cipher blocks and the block following. Additionally, if data is removed from the end of a CFB-encrypted block, that removal is undetectable. (Note also that CBC mode has a similar limitation, but data removed from the front of the block is undetectable.)

The obvious way to protect or authenticate an encrypted block is to digitally sign it. However, many people do not wish to habitually sign data for a large number of reasons that are beyond the scope of this document. Suffice it to say that many people consider properties such as deniability to be as valuable as integrity.

OpenPGP addresses this desire to have more security than raw encryption and yet preserve deniability with the MDC system. An MDC is intentionally not a Message Authentication Code (MAC). Its name was not selected by accident. It is analogous to a checksum.

Despite the fact that it is a relatively modest system, it has proved itself in the real world. It is an effective defense to several attacks that have surfaced since it has been created. It has met its modest goals admirably.

Consequently, because it is a modest security system, it has modest requirements on the hash function(s) it employs. It does not rely on a hash function being collision-free; it relies on a hash function being one-way. If a forger, Frank, wishes to send Alice a (digitally) unsigned message that says, "I've always secretly loved you, signed Bob", it is far easier for him to construct a new message than it is to modify anything intercepted from Bob. (Note also that if Bob wishes to communicate secretly with Alice, but without authentication or identification and with a threat model that includes forgers, he has a problem that transcends mere cryptography.)

Note also that unlike nearly every other OpenPGP subsystem, there are no parameters in the MDC system. It hard-defines SHA-1 as its hash function. This is not an accident. It is an intentional choice to avoid downgrade and cross-grade attacks while making a simple, fast

system. (A downgrade attack is an attack that would replace SHA2-256 with SHA-1, for example. A cross-grade attack would replace SHA-1 with another 160-bit hash, such as RIPEMD-160, for example.)

However, no update will be needed because the MDC has been replaced by the AEAD encryption described in this document.

5.13.2. Version 2 Symmetrically Encrypted Integrity Protected Data Packet Format

A version 2 Symmetrically Encrypted Integrity Protected Data packet consists of:

- A one-octet version number with value 2.
- A one-octet cipher algorithm ID.
- A one-octet AEAD algorithm identifier.
- A one-octet chunk size.
- 32 octets of salt. The salt is used to derive the message key and **MUST** be securely generated (see Section 13.10).
- Encrypted data; that is, the output of the selected symmetric-key cipher operating in the given AEAD mode.
- A final summary authentication tag for the AEAD mode.

The decrypted session key and the salt are used to derive an M-bit message key and N-64 bits used as the initialization vector, where M is the key size of the symmetric algorithm and N is the nonce size of the AEAD algorithm. M + N - 64 bits are derived using HKDF (see [RFC5869]). The leftmost M bits are used as a symmetric algorithm key, and the remaining N - 64 bits are used as an initialization vector. HKDF is used with SHA256 [RFC6234] as hash algorithm, the session key as IKM, the salt as salt, and the Packet Type ID in OpenPGP format encoding (bits 7 and 6 are set, and bits 5-0 carry the packet type ID), version number, cipher algorithm ID, AEAD algorithm ID, and chunk size octet as info parameter.

The KDF mechanism provides key separation between cipher and AEAD algorithms. Furthermore, an implementation can securely reply to a message even if a recipient's certificate is unknown by reusing the encrypted session key packets and replying with a different salt that yields a new, unique message key. See Section 13.8 for guidance on how applications can securely implement this feature.

A v2 SEIPD packet consists of one or more chunks of data. The plaintext of each chunk is of a size specified by the chunk size octet using the method specified below.

The encrypted data consists of the encryption of each chunk of plaintext, followed immediately by the relevant authentication tag. If the last chunk of plaintext is smaller than the chunk size, the ciphertext for that data may be shorter; nevertheless, it is followed by a full authentication tag.

For each chunk, the AEAD construction is given the Packet Type ID encoded in OpenPGP format (bits 7 and 6 are set, and bits 5-0 carry the packet type ID), version number, cipher algorithm ID, AEAD algorithm ID, and chunk size octet as additional data. For example, the additional data of the first chunk using EAX and AES-128 with a chunk size of 2**22 octets consists of the octets 0xD2, 0x02, 0x07, 0x01, and 0x10.

After the final chunk, the AEAD algorithm is used to produce a final authentication tag encrypting the empty string. This AEAD instance is given the additional data specified above, plus an eight-octet, big-endian value specifying the total number of plaintext octets encrypted. This allows detection of a truncated ciphertext.

The chunk size octet specifies the size of chunks using the following formula (see [C99]), where c is the chunk size octet:

```
chunk_size = (uint32_t) 1 << (c + 6)
```

An implementation **MUST** accept chunk size octets with values from 0 to 16. An implementation **MUST NOT** create data with a chunk size octet value larger than 16 (4 MiB chunks).

The nonce for AEAD mode consists of two parts. Let N be the size of the nonce. The leftmost N - 64 bits are the initialization vector derived using HKDF. The rightmost 64 bits are the chunk index as a big-endian value. The index of the first chunk is zero.

5.13.3. EAX Mode

The EAX AEAD algorithm used in this document is defined in [EAX].

The EAX algorithm can only use block ciphers with 16-octet blocks. The nonce is 16 octets long. EAX authentication tags are 16 octets long.

5.13.4. OCB Mode

The OCB AEAD algorithm used in this document is defined in [RFC7253].

The OCB algorithm can only use block ciphers with 16-octet blocks. The nonce is 15 octets long. OCB authentication tags are 16 octets long.

5.13.5. GCM Mode

The GCM AEAD algorithm used in this document is defined in [SP800-38D].

The GCM algorithm can only use block ciphers with 16-octet blocks. The nonce is 12 octets long. GCM authentication tags are 16 octets long.

5.14. Padding Packet (Type ID 21)

The Padding packet contains random data and can be used to defend against traffic analysis (see Section 13.11) on version 2 SEIPD messages (see Section 5.13.2) and Transferable Public Keys (see Section 10.1).

Such a packet **MUST** be ignored when received.

Its contents **SHOULD** be random octets to make the length obfuscation it provides more robust even when compressed.

An implementation adding padding to an OpenPGP stream **SHOULD** place such a packet:

- At the end of a v6 Transferable Public Key that is transferred over an encrypted channel (see Section 10.1).
- As the last packet of an Optionally Padded Message within a version 2 Symmetrically Encrypted Integrity Protected Data packet (see Section 10.3.1).

An implementation **MUST** be able to process padding packets anywhere else in an OpenPGP stream so that future revisions of this document may specify further locations for padding.

Policy about how large to make such a packet to defend against traffic analysis is beyond the scope of this document.

6. Base64 Conversions

As stated in the introduction, OpenPGP's underlying native representation for objects is a stream of arbitrary octets, and some systems desire these objects to be immune to damage caused by character set translation, data conversions, etc.

In principle, any printable encoding scheme that met the requirements of the unsafe channel would suffice, since it would not change the underlying binary bit streams of the native OpenPGP data structures. The OpenPGP specification specifies one such printable encoding scheme to ensure interoperability; see Section 6.2.

The encoding is composed of two parts: a base64 encoding of the binary data and an optional checksum. The base64 encoding used is described in Section 4 of [RFC4648], and it is wrapped into lines of no more than 76 characters each.

When decoding base64, an OpenPGP implementation **MUST** ignore all whitespace.

6.1. Optional Checksum

The optional checksum is a 24-bit Cyclic Redundancy Check (CRC) converted to four characters of base64 encoding by the same MIME base64 transformation, preceded by an equal sign (=). The CRC is computed by using the generator 0x864CFB and an initialization of 0xB704CE. The accumulation is done on the data before it is converted to base64 rather than on the converted data. A sample implementation of this algorithm is in Section 6.1.1.

If present, the checksum with its leading equal sign **MUST** appear on the next line after the base64-encoded data.

An implementation **MUST NOT** reject an OpenPGP object when the CRC24 footer is present, missing, malformed, or disagrees with the computed CRC24 sum. When forming ASCII Armor, the CRC24 footer **SHOULD NOT** be generated, unless interoperability with implementations that require the CRC24 footer to be present is a concern.

The CRC24 footer **MUST NOT** be generated if it can be determined by the context or by the OpenPGP object being encoded that the consuming implementation accepts base64-encoded blocks without a CRC24 footer. Notably:

- An ASCII-armored Encrypted Message packet sequence that ends in a v2 SEIPD packet **MUST NOT** contain a CRC24 footer.
- An ASCII-armored sequence of Signature packets that only includes v6 Signature packets **MUST NOT** contain a CRC24 footer.
- An ASCII-armored Transferable Public Key packet sequence of a v6 key **MUST NOT** contain a CRC24 footer.
- An ASCII-armored keyring consisting of only v6 keys **MUST NOT** contain a CRC24 footer.

Rationale: Previous draft versions of this document stated that the CRC24 footer is optional, but the text was ambiguous. In practice, very few implementations require the CRC24 footer to be present. Computing the CRC24 incurs a significant cost, while providing no meaningful integrity protection. Therefore, generating it is now discouraged.

6.1.1. An Implementation of the CRC24 in "C"

The following code is written in [C99].

```
#define CRC24_INIT 0xB704CEL
#define CRC24_GENERATOR 0x864CFBL

typedef unsigned long crc24;
crc24 crc_octets(unsigned char *octets, size_t len)
{
    crc24 crc = CRC24_INIT;
    int i;
    while (len--) {
        crc ^= (*octets++) << 16;
        for (i = 0; i < 8; i++) {
            crc <<= 1;
            if (crc & 0x1000000) {
                crc &= 0XFFFFFF; /* Clear bit 25 to avoid overflow */
                crc ^= CRC24_GENERATOR;
            }
        }
    }
    return crc & 0xFFFFFFL;
}
```

6.2. Forming ASCII Armor

When OpenPGP encodes data into ASCII Armor, it puts specific headers around the base64-encoded data, so OpenPGP can reconstruct the data later. An OpenPGP implementation **MAY** use ASCII armor to protect raw binary data. OpenPGP informs the user what kind of data is encoded in the ASCII armor through the use of the headers.

Concatenating the following data creates ASCII Armor:

- An Armor Header Line, appropriate for the type of data
- Armor Headers
- A blank (zero length or containing only whitespace) line
- The ASCII-Armored data
- An optional Armor Checksum (discouraged; see Section 6.1)
- The Armor Tail, which depends on the Armor Header Line

6.2.1. Armor Header Line

An Armor Header Line consists of the appropriate header line text surrounded by five (5) dashes (-, 0x2D) on either side of the header line text. The header line text is chosen based on the type of data being encoded in Armor and how it is being encoded. Header line texts include the following strings:

Armor Header	Use
`BEGIN PGP MESSAGE`	Used for signed, encrypted, or compressed files.
`BEGIN PGP PUBLIC KEY BLOCK`	Used for armoring public keys.
`BEGIN PGP PRIVATE KEY BLOCK`	Used for armoring private keys.
`BEGIN PGP SIGNATURE`	Used for detached signatures, OpenPGP/MIME signatures, and cleartext signatures.

Table 16: OpenPGP Armor Header Line Registry

Note that all of these Armor Header Lines are to consist of a complete line. Therefore, the header lines **MUST** start at the beginning of a line and **MUST NOT** have text other than whitespace following them on the same line.

6.2.2. Armor Headers

The Armor Headers are pairs of strings that can give the user or the receiving OpenPGP implementation some information about how to decode or use the message. The Armor Headers are a part of the armor, not the message, and hence are not protected by any signatures applied to the message.

The format of an Armor Header is that of a key-value pair. A colon (`:` 0x3A) and a single space (0x20) separate the key and value. An OpenPGP implementation may consider improperly formatted Armor Headers to be a corruption of the ASCII Armor, but it **SHOULD** make an effort to recover. Unknown keys should be silently ignored, and an OpenPGP implementation **SHOULD** continue to process the message.

Note that some transport methods are sensitive to line length. For example, the SMTP protocol that transports email messages has a line length limit of 998 characters (see Section 2.1.1 of [RFC5322]).

While there is a limit of 76 characters for the base64 data (Section 6), there is no limit for the length of Armor Headers. Care should be taken to ensure that the Armor Headers are short enough to survive transport. One way to do this is to repeat an Armor Header Key multiple times with different values for each so that no one line is overly long.

Currently defined Armor Header Keys are as follows:

Key	Summary	Reference
`Version`	Implementation information	Section 6.2.2.1
`Comment`	Arbitrary text	Section 6.2.2.2
`Hash`	Hash algorithms used in some v4 cleartext signed messages	Section 6.2.2.3
`Charset`	Character set	Section 6.2.2.4

Table 17: OpenPGP Armor Header Key Registry

6.2.2.1. "Version" Armor Header

The armor header key `Version` describes the OpenPGP implementation and version used to encode the message. To minimize metadata, implementations **SHOULD NOT** emit this key and its corresponding value except for debugging purposes with explicit user consent.

6.2.2.2. "Comment" Armor Header

The armor header key `Comment` supplies a user-defined comment. OpenPGP defines all text to be in UTF-8. A comment may be any UTF-8 string. However, the whole point of armoring is to provide seven-bit clean data. Consequently, if a comment has characters that are outside the US-ASCII range of UTF, they may very well not survive transport.

6.2.2.3. "Hash" Armor Header

The armor header key `Hash` is deprecated, but some older implementations expect it in messages using the Cleartext Signature Framework (Section 7). When present, this armor header key contains a comma-separated list of hash algorithms used in the signatures on message, with digest names as specified in the "Text Name" column in Table 23. These headers **SHOULD NOT** be emitted unless:

- the cleartext signed message contains a v4 signature made using a SHA2-based digest (`SHA224`, `SHA256`, `SHA384`, or `SHA512`), and
- the cleartext signed message might be verified by a legacy OpenPGP implementation that requires this header.

A verifying application **MUST** decline to validate any signature in a message with a non-conformant `Hash` header (that is, a `Hash` header that contains anything other than a comma-separated list of hash algorithms). When a conformant `Hash` header is present, a verifying application **MUST** ignore its contents, deferring to the hash algorithm indicated in the embedded signature.

6.2.2.4. "Charset" Armor Header

The armor header key `Charset` contains a description of the character set that the plaintext is in (see [RFC2978]). Please note that OpenPGP defines text to be in UTF-8. An implementation will get the best results by translating into and out of UTF-8. However, there are many instances where this is easier said than done. Also, there are communities of users who have no need for UTF-8 because they are all happy with a character set like ISO Latin-5 or a Japanese character set. In such instances, an implementation **MAY** override the UTF-8 default by using this header key. An implementation **MAY** implement this key and any translations it cares to; an implementation **MAY** ignore it and assume all text is UTF-8.

6.2.3. Armor Tail Line

The Armor Tail Line is composed in the same manner as the Armor Header Line, except the string "BEGIN" is replaced by the string "END".

7. Cleartext Signature Framework

It is desirable to be able to sign a textual octet stream without ASCII armoring the stream itself, so the signed text is still readable with any tool capable of rendering text. In order to bind a signature to such a cleartext, the Cleartext Signature Framework is used, which follows the same basic format and restrictions as the ASCII armoring described in Section 6.2. (Note that this framework is not intended to be reversible. [RFC3156] defines another way to sign cleartext messages for environments that support MIME.)

7.1. Cleartext Signed Message Structure

An OpenPGP cleartext signed message consists of:

- The cleartext header `-----BEGIN PGP SIGNED MESSAGE-----` on a single line.
- One or more legacy `Hash` Armor Headers that **MAY** be included by some implementations and **MUST** be ignored when well formed (see Section 6.2.2.3).
- An empty line (not included in the message digest).
- The dash-escaped cleartext.
- A line ending separating the cleartext and following armored signature (not included in the message digest).
- The ASCII-armored signature(s), including the `-----BEGIN PGP SIGNATURE-----` Armor Header and Armor Tail Lines.

As with any other text signature (Section 5.2.1.2), a cleartext signature is calculated on the text using canonical <CR><LF> line endings. As described above, the line ending before the `-----BEGIN PGP SIGNATURE-----` Armor Header Line of the armored signature is not considered part of the signed text.

Also, any trailing whitespace -- spaces (0x20) and tabs (0x09) -- at the end of any line is removed before signing or verifying a cleartext signed message.

Between the `-----BEGIN PGP SIGNED MESSAGE-----` line and the first empty line, the only Armor Header permitted is a well-formed `Hash` Armor Header (see Section 6.2.2.3). To reduce the risk of confusion about what has been signed, a verifying implementation **MUST** decline to validate any signature in a cleartext message if that message has any other Armor Header present in this location.

7.2. Dash-Escaped Text

The cleartext content of the message must also be dash-escaped.

Dash-escaped cleartext is the ordinary cleartext where every line starting with a "-" (HYPHEN-MINUS, U+002D) is prefixed by the sequence "-" (HYPHEN-MINUS, U+002D) and " " (SPACE, U+0020). This prevents the parser from recognizing armor headers of the cleartext itself. An implementation **MAY** dash-escape any line, **SHOULD** dash-escape lines commencing in "From" followed by a space, and **MUST** dash-escape any line commencing in a dash. The message digest is computed using the cleartext itself, not the dash-escaped form.

When reversing dash-escaping, an implementation **MUST** strip the string `- ` if it occurs at the beginning of a line, and it **SHOULD** provide a warning for `-` and any character other than a space at the beginning of a line.

7.3. Issues with the Cleartext Signature Framework

Since creating a cleartext signed message involves trimming trailing whitespace on every line, the Cleartext Signature Framework will fail to safely round-trip any textual stream that may include semantically meaningful whitespace.

For example, the Unified Diff format [UNIFIED-DIFF] contains semantically meaningful whitespace: an empty line of context will consist of a line with a single " " (SPACE, U+0020) character, and any line that has trailing whitespace added or removed will represent such a change with semantically meaningful whitespace.

Furthermore, a Cleartext Signature Framework message that is very large is unlikely to work well. In particular, it will be difficult for any human reading the message to know which part is covered by the signature because they can't understand the whole message at once, especially in the case where an Armor Header line is placed somewhere in the body. And, very large Cleartext Signature Framework messages cannot be processed in a single pass, since the signature salt and digest algorithms are only discovered at the end.

An implementation that knows it is working with a textual stream with any of the above characteristics **SHOULD NOT** use the Cleartext Signature Framework. Safe alternatives for a semantically meaningful OpenPGP signature over such a file format are:

- A Signed Message, as described in Section 10.3.
- A detached signature, as described in Section 10.4.

Either of these alternatives may be ASCII-armored (see Section 6.2) if they need to be transmitted across a text-only (or 7-bit clean) channel.

Finally, when a Cleartext Signature Framework message is presented to the user as is, an attacker can include additional text in the `Hash` header, which may mislead the user into thinking it is part of the signed text. The signature validation constraints described in Sections 6.2.2.3 and 7.1 help to mitigate the risk of arbitrary or misleading text in the Armor Headers.

8. Regular Expressions

A regular expression is zero or more branches, separated by `|`. It matches anything that matches one of the branches.

A branch is zero or more pieces, concatenated. It matches a match for the first, followed by a match for the second, etc.

A piece is an atom possibly followed by `*`, `+`, or `?`. An atom followed by `*` matches a sequence of 0 or more matches of the atom. An atom followed by `+` matches a sequence of 1 or more matches of the atom. An atom followed by `?` matches a match of the atom or the null string.

An atom is a regular expression in parentheses (matching a match for the regular expression), a range (see below), `.` (matching any single Unicode character), `^` (matching the null string at the beginning of the input string), `$` (matching the null string at the end of the input string), a `\` followed by a single Unicode character (matching that character), or a single Unicode character with no other significance (matching that character).

A range is a sequence of characters enclosed in `[ ]`. It normally matches any single character from the sequence. If the sequence begins with `^`, it matches any single Unicode character not from the rest of the sequence. If two characters in the sequence are separated by `-`, this is shorthand for the full list of Unicode characters between them in codepoint order (for example, `[0-9]` matches any decimal digit). To include a literal `]` in the sequence, make it the first character (following a possible `^`). To include a literal `-`, make it the first or last character.

9. Constants

This section describes the constants used in OpenPGP.

Note that these tables are not exhaustive lists; an implementation **MAY** implement an algorithm that is not on these lists, as long as the algorithm IDs are chosen from the Private or Experimental Use algorithm range.

See Section 12 for more discussion of the algorithms.

9.1. Public-Key Algorithms

ID	Algorithm	Public Key Format	Secret Key Format	Signature Format	PKESK Format
0	Reserved				
1	RSA (Encrypt or Sign) [FIPS186]	MPI(n), MPI(e) [Section 5.5.5.1]	MPI(d), MPI(p), MPI(q), MPI(u)	MPI(m**d mod n) [Section 5.2.3.1]	MPI(m**e mod n) [Section 5.1.3]
2	RSA Encrypt-Only [FIPS186]	MPI(n), MPI(e) [Section 5.5.5.1]	MPI(d), MPI(p), MPI(q), MPI(u)	N/A	MPI(m**e mod n) [Section 5.1.3]
3	RSA Sign-Only [FIPS186]	MPI(n), MPI(e) [Section 5.5.5.1]	MPI(d), MPI(p), MPI(q), MPI(u)	MPI(m**d mod n) [Section 5.2.3.1]	N/A

ID	Algorithm	Public Key Format	Secret Key Format	Signature Format	PKESK Format
16	Elgamal (Encrypt-Only) [ELGAMAL]	MPI(p), MPI(g), MPI(y) [Section 5.5.5.3]	MPI(x)	N/A	MPI(g**k mod p), MPI(m * y**k mod p) [Section 5.1.4]
17	DSA (Digital Signature Algorithm) [FIPS186]	MPI(p), MPI(q), MPI(g), MPI(y) [Section 5.5.5.2]	MPI(x)	MPI(r), MPI(s) [Section 5.2.3.2]	N/A
18	ECDH public key algorithm	OID, MPI(point in curve-specific point format), KDFParams [Sections 9.2.1 and 5.5.5.6]	MPI(value in curve-specific format) [Section 9.2.1]	N/A	MPI(point in curve-specific point format), size octet, encoded key [Sections 9.2.1, 5.1.5, and 11.5]
19	ECDSA public key algorithm [FIPS186]	OID, MPI(point in SEC1 format) [Section 5.5.5.4]	MPI(value)	MPI(r), MPI(s) [Section 5.2.3.2]	N/A
20	Reserved (formerly Elgamal Encrypt or Sign)				
21	Reserved for Diffie-Hellman (X9.42, as defined for IETF-S/MIME)				
22	EdDSALegacy (deprecated)	OID, MPI(point in prefixed native format) [Sections 11.2.2 and 5.5.5.5]	MPI(value in curve-specific format) [Section 9.2.1]	MPI, MPI [Sections 9.2.1 and 5.2.3.3]	N/A

ID	Algorithm	Public Key Format	Secret Key Format	Signature Format	PKESK Format
23	Reserved (AEDH)				
24	Reserved (AEDSA)				
25	X25519	32 octets [Section 5.5.5.7]	32 octets	N/A	32 octets, size octet, encoded key [Section 5.1.6]
26	X448	56 octets [Section 5.5.5.8]	56 octets	N/A	56 octets, size octet, encoded key [Section 5.1.7]
27	Ed25519	32 octets [Section 5.5.5.9]	32 octets	64 octets [Section 5.2.3.4]	
28	Ed448	57 octets [Section 5.5.5.10]	57 octets	114 octets [Section 5.2.3.5]	
100 to 110	Private/ Experimental algorithm				

Table 18: OpenPGP Public Key Algorithms Registry

Implementations **MUST** implement Ed25519 (27) for signatures and X25519 (25) for encryption. Implementations **SHOULD** implement Ed448 (28) and X448 (26).

RSA (1) keys are deprecated and **SHOULD NOT** be generated but may be interpreted. RSA Encrypt-Only (2) and RSA Sign-Only (3) are deprecated and **MUST NOT** be generated (see Section 12.4). Elgamal (16) keys are deprecated and **MUST NOT** be generated (see Section 12.6). DSA (17) keys are deprecated and **MUST NOT** be generated (see Section 12.5). For notes on Elgamal Encrypt or Sign (20) and X9.42 (21), see Section 12.8. Implementations **MAY** implement any other algorithm.

Note that an implementation conforming to the previous version of this specification [RFC4880] has only DSA (17) and Elgamal (16) as the algorithms that **MUST** be implemented.

A compatible specification of ECDSA is given in [RFC6090] (as "KT-I Signatures") and in [SEC1]; ECDH is defined in Section 11.5 of this document.

9.2. ECC Curves for OpenPGP

The parameter curve OID is an array of octets that defines a named curve.

The table below specifies the exact sequence of octets for each named curve referenced in this document. It also specifies which public key algorithms the curve can be used with, as well as the size of expected elements in octets:

ASN.1 Object Identifier	OID Len	Curve OID Octets	Curve Name	Usage (fsize)
1.2.840.10045.3.1.7	8	2A 86 48 CE 3D 03 01 07	NIST P-256	ECDSA, ECDH (32)
1.3.132.0.34	5	2B 81 04 00 22	NIST P-384	ECDSA, ECDH (48)
1.3.132.0.35	5	2B 81 04 00 23	NIST P-521	ECDSA, ECDH (66)
1.3.36.3.3.2.8.1.1.7	9	2B 24 03 03 02 08 01 01 07	brainpoolP256r1	ECDSA, ECDH (32)
1.3.36.3.3.2.8.1.1.11	9	2B 24 03 03 02 08 01 01 0B	brainpoolP384r1	ECDSA, ECDH (48)
1.3.36.3.3.2.8.1.1.13	9	2B 24 03 03 02 08 01 01 0D	brainpoolP512r1	ECDSA, ECDH (64)
1.3.6.1.4.1.11591.15.1	9	2B 06 01 04 01 DA 47 0F 01	Ed25519Legacy	EdDSALegacy (32)
1.3.6.1.4.1.3029.1.5.1	10	2B 06 01 04 01 97 55 01 05 01	Curve25519Legacy	ECDH (32)

Table 19: OpenPGP ECC Curve OID and Usage Registry

The "Field Size (fsize)" column represents the field size of the group in number of octets, rounded up, such that x or y coordinates for a point on the curve or native point representations for the curve can be represented in that many octets. The curves specified here, and scalars such as the base point order and the private key, can be represented in fsize octets. However, note that curves exist outside this specification where the representation of scalars requires an additional octet.

The sequence of octets in the third column is the result of applying the Distinguished Encoding Rules (DER) to the ASN.1 Object Identifier with subsequent truncation. The truncation removes the two fields of encoded Object Identifier. The first omitted field is one octet representing the Object Identifier tag, and the second omitted field is the length of the Object Identifier body. For example, the complete ASN.1 DER encoding for the NIST P-256 curve OID is "06 08 2A 86 48 CE 3D 03 01 07", from which the first entry in the table above is constructed by omitting the first two octets. Only the truncated sequence of octets is the valid representation of a curve OID.

The deprecated OIDs for Ed25519Legacy and Curve25519Legacy are used only in version 4 keys and signatures. Implementations **MAY** implement these variants for compatibility with existing v4 key material and signatures. Implementations **MUST NOT** accept or generate v6 key material using the deprecated OIDs.

9.2.1. Curve-Specific Wire Formats

Some elliptic curve public key algorithms use different conventions for specific fields depending on the curve in use. Each field is always formatted as an MPI, but with a curve-specific framing. This table summarizes those distinctions.

Curve	ECDH Point Format	ECDH Secret Key MPI	EdDSA Secret Key MPI	EdDSA Signature first MPI	EdDSA Signature second MPI
NIST P-256	SEC1	integer	N/A	N/A	N/A
NIST P-384	SEC1	integer	N/A	N/A	N/A
NIST P-521	SEC1	integer	N/A	N/A	N/A
brainpoolP256r1	SEC1	integer	N/A	N/A	N/A
brainpoolP384r1	SEC1	integer	N/A	N/A	N/A
brainpoolP512r1	SEC1	integer	N/A	N/A	N/A
Ed25519Legacy	N/A	N/A	32 octets of secret	32 octets of R	32 octets of S
Curve25519Legacy	prefixed native	integer (see Section 5.5.5.6.1.1)	N/A	N/A	N/A

Table 20: OpenPGP ECC Curve-Specific Wire Formats Registry

For the native octet-string forms of Ed25519Legacy values, see [RFC8032]. For the native octet-string forms of Curve25519Legacy secret scalars and points, see [RFC7748].

9.3. Symmetric-Key Algorithms

ID	Algorithm
0	Plaintext or unencrypted data
1	IDEA [IDEA]
2	TripleDES (DES-EDE [SP800-67] -- 168-bit key derived from 192)

ID	Algorithm
3	CAST5 (128-bit key, as per [RFC2144])
4	Blowfish (128-bit key, 16 rounds) [BLOWFISH]
5	Reserved
6	Reserved
7	AES with 128-bit key [AES]
8	AES with 192-bit key
9	AES with 256-bit key
10	Twofish with 256-bit key [TWOFISH]
11	Camellia with 128-bit key [RFC3713]
12	Camellia with 192-bit key
13	Camellia with 256-bit key
100-110	Private/Experimental algorithm
253-255	Reserved to avoid collision with Secret Key Encryption (see Table 2 and Section 5.5.3)

Table 21: OpenPGP Symmetric Key Algorithms Registry

Implementations **MUST** implement AES-128. Implementations **SHOULD** implement AES-256. Implementations **MUST NOT** encrypt data with IDEA, TripleDES, or CAST5. Implementations **MAY** decrypt data that uses IDEA, TripleDES, or CAST5 for the sake of reading older messages or new messages from implementations predating support for [RFC2440]. An Implementation that decrypts data using IDEA, TripleDES, or CAST5 **SHOULD** generate a deprecation warning about the symmetric algorithm, indicating that message confidentiality is suspect. Implementations **MAY** implement any other algorithm.

9.4. Compression Algorithms

ID	Algorithm
0	Uncompressed [RFC9580]
1	ZIP [RFC1951]
2	ZLIB [RFC1950]

ID	Algorithm
3	BZip2 [BZ2]
100-110	Private/Experimental algorithm

Table 22: OpenPGP Compression Algorithms Registry

Implementations **MUST** implement uncompressed data. Implementations **SHOULD** implement ZLIB. For interoperability reasons, implementations **SHOULD** be able to decompress using ZIP. Implementations **MAY** implement any other algorithm.

9.5. Hash Algorithms

ID	Algorithm	Text Name	V6 Signature Salt Size
1	MD5 [RFC1321]	"MD5"	N/A
2	SHA-1 [FIPS180] Section 13.1	"SHA1"	N/A
3	RIPEMD-160 [RIPEMD-160]	"RIPEMD160"	N/A
4	Reserved [RFC9580]		
5	Reserved [RFC9580]		
6	Reserved [RFC9580]		
7	Reserved [RFC9580]		
8	SHA2-256 [FIPS180]	"SHA256"	16
9	SHA2-384 [FIPS180]	"SHA384"	24
10	SHA2-512 [FIPS180]	"SHA512"	32
11	SHA2-224 [FIPS180]	"SHA224"	16
12	SHA3-256 [FIPS202]	"SHA3-256"	16
13	Reserved [RFC9580]		
14	SHA3-512 [FIPS202]	"SHA3-512"	32
100-110	Private/Experimental algorithm		

Table 23: OpenPGP Hash Algorithms Registry

Hash Algorithm	OID	Full Hash Prefix
MD5	1.2.840.113549.2.5	0x30, 0x20, 0x30, 0x0C, 0x06, 0x08, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x02, 0x05, 0x05, 0x00, 0x04, 0x10
SHA-1	1.3.14.3.2.26	0x30, 0x21, 0x30, 0x09, 0x06, 0x05, 0x2B, 0x0E, 0x03, 0x02, 0x1A, 0x05, 0x00, 0x04, 0x14
RIPEMD-160	1.3.36.3.2.1	0x30, 0x21, 0x30, 0x09, 0x06, 0x05, 0x2B, 0x24, 0x03, 0x02, 0x01, 0x05, 0x00, 0x04, 0x14
SHA2-256	2.16.840.1.101.3.4.2.1	0x30, 0x31, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01, 0x05, 0x00, 0x04, 0x20
SHA2-384	2.16.840.1.101.3.4.2.2	0x30, 0x41, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x02, 0x05, 0x00, 0x04, 0x30
SHA2-512	2.16.840.1.101.3.4.2.3	0x30, 0x51, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x03, 0x05, 0x00, 0x04, 0x40
SHA2-224	2.16.840.1.101.3.4.2.4	0x30, 0x2D, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x04, 0x05, 0x00, 0x04, 0x1C
SHA3-256	2.16.840.1.101.3.4.2.8	0x30, 0x31, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x08, 0x05, 0x00, 0x04, 0x20
SHA3-512	2.16.840.1.101.3.4.2.10	0x30, 0x51, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x0a, 0x05, 0x00, 0x04, 0x40

Table 24: OpenPGP Hash Algorithm Identifiers for RSA Signatures' Use of EMSA-PKCS1-v1_5 Padding Registry

Implementations **MUST** implement SHA2-256. Implementations **SHOULD** implement SHA2-384 and SHA2-512. Implementations **MAY** implement other algorithms. Implementations **SHOULD NOT** create messages that require the use of SHA-1, with the exception of computing version 4 key fingerprints for purposes of the MDC in version 1 Symmetrically Encrypted Integrity Protected Data packets. Implementations **MUST NOT** generate signatures with MD5, SHA-1, or RIPEMD-160. Implementations **MUST NOT** use MD5, SHA-1, or RIPEMD-160 as a hash function in an ECDH KDF. Implementations **MUST NOT** generate packets using MD5, SHA-1, or RIPEMD-160 as a hash function in an S2K KDF. Implementations **MUST NOT** decrypt a secret using MD5, SHA-1, or

RIPEMD-160 as a hash function in an S2K KDF in a version 6 (or later) packet. Implementations **MUST NOT** validate any recent signature that depends on MD5, SHA-1, or RIPEMD-160. Implementations **SHOULD NOT** validate any old signature that depends on MD5, SHA-1, or RIPEMD-160 unless the signature's creation date predates known weakness of the algorithm used, and the implementation is confident that the message has been in the secure custody of the user the whole time.

9.6. AEAD Algorithms

ID	Name	Nonce Length (Octets)	Authentication Tag Length (Octets)	Reference
1	EAX	16	16	[EAX]
2	OCB	15	16	[RFC7253]
3	GCM	12	16	[SP800-38D]
100-110	Private/Experimental algorithm			[RFC9580]

Table 25: OpenPGP AEAD Algorithms Registry

Implementations **MUST** implement OCB. Implementations **MAY** implement EAX, GCM, and other algorithms.

10. Packet Sequence Composition

OpenPGP packets are assembled into sequences in order to create messages and to transfer keys. Not all possible packet sequences are meaningful and correct. This section describes the rules for how packets should be placed into sequences.

There are three distinct sequences of packets:

- Transferable Public Keys (Section 10.1) and their close counterpart, Transferable Secret Keys (Section 10.2)
- OpenPGP Messages (Section 10.3)
- Detached Signatures (Section 10.4)

Each sequence has an explicit grammar of what packet types (Table 3) can appear in what place. The presence of an unknown critical packet, or a known but unexpected packet, is a critical error, invalidating the entire sequence (see Section 4.3). On the other hand, unknown non-critical packets can appear anywhere within any sequence. This provides a structured way to introduce new packets into the protocol, while making sure that certain packets will be handled strictly.

An implementation may "recognize" a packet but not implement it. The purpose of Packet Criticality is to allow the producer to tell the consumer whether it would prefer a new, unknown packet to generate an error or be ignored.

Note that previous versions of this document did not have a concept of Packet Criticality and did not give clear guidance on what to do when unknown packets are encountered. Therefore, implementations of the previous versions may reject unknown non-critical packets or accept unknown critical packets.

When generating a sequence of OpenPGP packets according to one of the three grammars, an implementation **MUST NOT** inject a critical packet of a type that does not adhere to the grammar.

According to one of the three grammars, when consuming a sequence of OpenPGP packets, an implementation **MUST** reject the sequence with an error if it encounters a critical packet of an inappropriate type.

10.1. Transferable Public Keys

OpenPGP users may transfer public keys. This section describes the structure of public keys in transit to ensure interoperability. An OpenPGP Transferable Public Key is also known as an OpenPGP certificate, in order to distinguish it from both its constituent Public-Key packets (Sections 5.5.1.1 and 5.5.1.2) and the underlying cryptographic key material.

10.1.1. OpenPGP v6 Certificate Structure

The format of an OpenPGP v6 certificate is as follows. Entries in square brackets are optional and ellipses indicate repetition.

```
Primary Key
   [Revocation Signature...]
    Direct Key Signature...
   [User ID or User Attribute
           [Certification Revocation Signature...]
           [Certification Signature...]]...
   [Subkey [Subkey Revocation Signature...]
           Subkey Binding Signature...]...
   [Padding]
```

In addition to these rules, a marker packet (Section 5.8) can appear anywhere in the sequence.

Note that a v6 key uses a self-signed direct key signature to store algorithm preferences.

Every subkey for a v6 primary key **MUST** be a v6 subkey. Every subkey **MUST** have at least one subkey binding signature. Every subkey binding signature **MUST** be a self-signature (that is, made by the v6 primary key). Like all other signatures, every self-signature made by a v6 key **MUST** be a v6 signature.

10.1.2. OpenPGP v6 Revocation Certificate

When a primary v6 Public Key is revoked, it is sometimes distributed with only the revocation signature:

```
Primary Key
    Revocation Signature
```

In this case, the direct key signature is no longer necessary, since the primary key itself has been marked as unusable.

10.1.3. OpenPGP v4 Certificate Structure

The format of an OpenPGP v4 key is as follows.

```
Primary Key
   [Revocation Signature]
   [Direct Key Signature...]
   [User ID or User Attribute [Signature...]]...
   [Subkey [Subkey Revocation Signature...]
           Subkey Binding Signature...]...
```

In addition to these rules, a marker packet (Section 5.8) can appear anywhere in the sequence.

A subkey always has at least one subkey binding signature after it that is issued using the primary key to tie the two keys together. These binding signatures may be in either v3 or v4 format, but they **SHOULD** be in v4 format. Subkeys that can issue signatures **MUST** have a v4 binding signature due to the **REQUIRED** embedded primary key binding signature.

Every subkey for a v4 primary key **MUST** be a v4 subkey.

When a primary v4 Public Key is revoked, the revocation signature is sometimes distributed by itself, without the primary key packet it applies to. This is referred to as a "revocation certificate". Instead, a v6 revocation certificate **MUST** include the primary key packet, as described in Section 10.1.2.

10.1.4. OpenPGP v3 Key Structure

The format of an OpenPGP v3 key is as follows.

```
RSA Public Key
   [Revocation Signature]
    User ID [Signature...]
   [User ID [Signature...]]...
```

In addition to these rules, a marker packet (Section 5.8) can appear anywhere in the sequence.

Each signature certifies the RSA public key and the preceding User ID. The RSA public key can have many User IDs, and each User ID can have many signatures. V3 keys are deprecated. Implementations **MUST NOT** generate new v3 keys but **MAY** continue to use existing ones.

V3 keys **MUST NOT** have subkeys.

10.1.5. Common Requirements

The Public-Key packet occurs first.

The primary key **MUST** be an algorithm capable of making signatures (that is, not an encryption-only algorithm). This is because the primary key needs to be able to create self-signatures (see Section 5.2.3.10). The subkeys may be keys of any type. For example, there may be a single-key RSA key, an Ed25519 primary key with an RSA encryption subkey, an Ed25519 primary key with an X25519 subkey, etc.

Each of the following User ID packets provides the identity of the owner of this public key. If there are multiple User ID packets, this corresponds to multiple means of identifying the same unique individual user; for example, a user may have more than one email address and construct a User ID for each one. A transferable public key **SHOULD** include at least one User ID packet unless storage requirements prohibit this.

Immediately following each User ID packet, there are zero or more Signature packets. Each Signature packet is calculated on the immediately preceding User ID packet and the initial Public-Key packet. The signature serves to certify the corresponding public key and User ID. In effect, the signer is testifying to the belief that this public key belongs to the user identified by this User ID.

Within the same section as the User ID packets, there are zero or more User Attribute packets. Like the User ID packets, a User Attribute packet is followed by zero or more Signature packets calculated on the immediately preceding User Attribute packet and the initial Public-Key packet.

User Attribute packets and User ID packets may be freely intermixed in this section, as long as the signatures that follow them are maintained on the proper User Attribute or User ID packet.

After the sequence of User ID packets and Attribute packets and their associated signatures, zero or more Subkey packets follow, each with their own signatures. In general, subkeys are provided in cases where the top-level public key is a certification-only key. However, any v4 or v6 key may have subkeys, and the subkeys may be encryption keys, signing keys, authentication keys, etc. It is good practice to use separate subkeys for every operation (i.e., signature-only, encryption-only, authentication-only keys, etc.).

Each Subkey packet **MUST** be followed by one Signature packet, which should be a subkey binding signature issued by the top-level key. For subkeys that can issue signatures, the subkey binding signature **MUST** contain an Embedded Signature subpacket with a primary key binding signature (0x19) issued by the subkey on the top-level key.

Subkey and Key packets may each be followed by a revocation Signature packet to indicate that the key is revoked. Revocation signatures are only accepted if they are issued by the key itself or by a key that is authorized to issue revocations via a Revocation Key subpacket in a self-signature by the top-level key.

The optional trailing Padding packet is a mechanism to defend against traffic analysis (see Section 13.11). For maximum interoperability, if the Public-Key packet is a v4 key, the optional Padding packet **SHOULD NOT** be present unless the recipient has indicated that they are capable of ignoring it successfully. An implementation that is capable of receiving a transferable public key with a v6 Public-Key primary key **MUST** be able to accept (and ignore) the trailing optional Padding packet.

Transferable public-key packet sequences may be concatenated to allow transferring multiple public keys in one operation (see Section 3.6).

10.2. Transferable Secret Keys

OpenPGP users may transfer secret keys. The format of a transferable secret key is the same as a transferable public key except that Secret-Key and Secret-Subkey packets can be used in addition to the Public-Key and Public-Subkey packets. If a single Secret-Key or Secret-Subkey packet is included in a packet sequence, it is a transferable secret key and should be handled and marked as such (see Section 6.2). An implementation **SHOULD** include self-signatures on any User IDs and subkeys, as this allows for a complete public key to be automatically extracted from the transferable secret key. An implementation **MAY** choose to omit the self-signatures, especially if a transferable public key accompanies the transferable secret key.

10.3. OpenPGP Messages

An OpenPGP message is a packet or sequence of packets that correspond to the following grammatical rules (a comma (,) represents sequential composition, and a vertical bar (|) separates alternatives):

OpenPGP Message :- Encrypted Message | Signed Message | Compressed Message | Literal Message.

Compressed Message :- Compressed Data Packet.

Literal Message :- Literal Data Packet.

ESK :- Public-Key Encrypted Session Key Packet | Symmetric-Key Encrypted Session Key Packet.

ESK Sequence :- ESK | ESK Sequence, ESK.

Encrypted Data :- Symmetrically Encrypted Data Packet | Symmetrically Encrypted Integrity Protected Data Packet.

Encrypted Message :- Encrypted Data | ESK Sequence, Encrypted Data.

One-Pass Signed Message :- One-Pass Signature Packet, OpenPGP Message, Corresponding Signature Packet.

Signed Message :- Signature Packet, OpenPGP Message | One-Pass Signed Message.

Optionally Padded Message :- OpenPGP Message | OpenPGP Message, Padding Packet.

In addition to these rules, a marker packet (Section 5.8) can appear anywhere in the sequence.

10.3.1. Unwrapping Encrypted and Compressed Messages

In addition to the above grammar, certain messages can be "unwrapped" to yield new messages. In particular:

- Decrypting a version 2 Symmetrically Encrypted and Integrity Protected Data packet **MUST** yield a valid Optionally Padded Message.
- Decrypting a version 1 Symmetrically Encrypted and Integrity Protected Data packet or -- for historic data -- a Symmetrically Encrypted Data packet **MUST** yield a valid OpenPGP Message.
- Decompressing a Compressed Data packet **MUST** also yield a valid OpenPGP Message.

When any unwrapping is performed, the resulting stream of octets is parsed into a series of OpenPGP packets like any other stream of octets. The packet boundaries found in the series of octets are expected to align with the length of the unwrapped octet stream. An implementation **MUST NOT** interpret octets beyond the boundaries of the unwrapped octet stream as part of any OpenPGP packet. If an implementation encounters a packet whose header length indicates that it would extend beyond the boundaries of the unwrapped octet stream, the implementation **MUST** reject that packet as malformed and unusable.

10.3.2. Additional Constraints on Packet Sequences

Note that some subtle combinations that are formally acceptable by this grammar are nonetheless unacceptable.

10.3.2.1. Packet Versions in Encrypted Messages

As noted above, an Encrypted Message is a sequence of zero or more PKESKs (Section 5.1) and SKESKs (Section 5.3), followed by an SEIPD (Section 5.13) payload. In some historic data, the payload may be a deprecated SED packet (Section 5.7) instead of SEIPD, though implementations **MUST NOT** generate SED packets (see Section 13.7). The versions of the preceding ESK packets within an Encrypted Message **MUST** align with the version of the payload SEIPD packet, as described in this section.

v3 PKESK and v4 SKESK packets both contain the symmetric cipher algorithm ID and the session key for the subsequent SEIPD packet in their cleartext. Since a v1 SEIPD does not contain a symmetric algorithm ID, all ESK packets preceding a v1 SEIPD payload **MUST** be either v3 PKESK or v4 SKESK.

On the other hand, the cleartext of the v6 ESK packets (either PKESK or SKESK) do not contain a symmetric cipher algorithm ID, so they cannot be used in combination with a v1 SEIPD payload. The payload following any v6 PKESK or v6 SKESK packet **MUST** be a v2 SEIPD.

Additionally, to avoid potentially conflicting cipher algorithm IDs, and for simplicity, implementations **MUST NOT** precede a v2 SEIPD payload with either v3 PKESK or v4 SKESK packets.

The versions of packets found in an Encrypted Message are summarized in the following table. An implementation **MUST** only generate an Encrypted Message using packet versions that match a row with "Yes" in the "Generate?" column. Other rows are provided for the purpose of historic interoperability. A conforming implementation **MUST** only generate an Encrypted Message using packets whose versions correspond to a single row.

Version of Encrypted Data Payload	Version of Preceding Symmetric-Key ESK (If Any)	Version of Preceding Public-Key ESK (If Any)	Generate?
SED (Section 5.7)	-	v2 PKESK [RFC2440]	No
SED (Section 5.7)	v4 SKESK (Section 5.3.1)	v3 PKESK (Section 5.1.1)	No
v1 SEIPD (Section 5.13.1)	v4 SKESK (Section 5.3.1)	v3 PKESK (Section 5.1.1)	Yes
v2 SEIPD (Section 5.13.2)	v6 SKESK (Section 5.3.2)	v6 PKESK (Section 5.1.2)	Yes

Table 26: OpenPGP Encrypted Message Packet Versions Registry

An implementation processing an Encrypted Message **MUST** discard any preceding ESK packet with a version that does not align with the version of the payload.

10.3.2.2. Packet Versions in Signatures

OpenPGP key packets and signature packets are also versioned. The version of a Signature typically matches the version of the signing key. When a v6 key produces a signature packet, it **MUST** produce a version 6 signature packet, regardless of the signature packet type. When a message is signed or verified using the one-pass construction, the version of the One-Pass Signature packet (Section 5.4) should also be aligned to the other versions.

Some legacy implementations have produced unaligned signature versions for older key material, which are also described in the table below for the purpose of historic interoperability. A conforming implementation **MUST** only generate signature packets with version numbers matching rows with "Yes" in the "Generate?" column.

Signing Key Version	Signature Packet Version	OPS Packet Version	Generate?
3 (Section 5.5.2.1)	3 (Section 5.2.2)	3 (Section 5.4)	No
4 (Section 5.5.2.2)	3 (Section 5.2.2)	3 (Section 5.4)	No
4 (Section 5.5.2.2)	4 (Section 5.2.3)	3 (Section 5.4)	Yes
6 (Section 5.5.2.3)	6 (Section 5.2.3)	6 (Section 5.4)	Yes

Table 27: OpenPGP Key and Signature Versions Registry

Note, however, that a version mismatch between these packets does not invalidate the packet sequence as a whole; it merely invalidates the signature, as a signature with an unknown version **SHOULD** be discarded (see Section 5.2.5).

10.4. Detached Signatures

Some OpenPGP applications use so-called "detached signatures". For example, a program bundle may contain a file, and with it a second file that is a detached signature of the first file. These detached signatures are simply one or more Signature packets stored separately from the data for which they are a signature.

In addition, a marker packet (Section 5.8) and a padding packet (Section 5.14) can appear anywhere in the sequence.

11. Elliptic Curve Cryptography

This section describes algorithms and parameters used with Elliptic Curve Cryptography (ECC) keys. A thorough introduction to ECC can be found in [KOBLITZ].

None of the ECC methods described in this document are allowed with deprecated v3 keys. Refer to [FIPS186], Appendix B.4.1, for the method to generate a uniformly distributed ECC private key.

11.1. ECC Curves

This document references three named prime field curves defined in [FIPS186] as "Curve P-256", "Curve P-384", and "Curve P-521" and three named prime field curves defined in [RFC5639] as "brainpoolP256r1", "brainpoolP384r1", and "brainpoolP512r1". All six curves can be used with ECDSA and ECDH public key algorithms. They are referenced using a sequence of octets, referred to as the curve OID. Section 9.2 describes in detail how this sequence of octets is formed.

Separate algorithms are also defined for the use of X25519 and X448 [RFC7748] and Ed25519 and Ed448 [RFC8032]. Additionally, legacy OIDs are defined for "Curve25519Legacy" (for encryption using the ECDH algorithm) and "Ed25519Legacy" (for signing using the EdDSALegacy algorithm).

11.2. EC Point Wire Formats

A point on an elliptic curve will always be represented on the wire as an MPI. Each curve uses a specific point format for the data within the MPI itself. Each format uses a designated prefix octet to ensure that the high octet has at least one bit set to make the MPI a constant size.

Name	Wire Format	Reference
SEC1	0x04 || x || y	Section 11.2.1
Prefixed native	0x40 || native	Section 11.2.2

Table 28: OpenPGP Elliptic Curve Point Wire Formats Registry

11.2.1. SEC1 EC Point Wire Format

For a SEC1-encoded (uncompressed) point, the content of the MPI is:

```
B = 04 || x || y
```

where x and y are coordinates of the point P = (x, y), and each is encoded in the big-endian format and zero-padded to the adjusted underlying field size. The adjusted underlying field size is the underlying field size rounded up to the nearest 8-bit boundary, as noted in the "fsize" column in Section 9.2. This encoding is compatible with the definition given in [SEC1].

11.2.2. Prefixed Native EC Point Wire Format

For a custom compressed point, the content of the MPI is:

```
B = 40 || p
```

where p is the public key of the point encoded using the rules defined for the specified curve. This format is used for ECDH keys based on curves expressed in Montgomery form and for points when using EdDSA.

11.2.3. Notes on EC Point Wire Formats

Given the above definitions, the exact size of the MPI payload for an encoded point is 515 bits for both NIST P-256 and brainpoolP256r1, 771 for both NIST P-384 and brainpoolP384r1, 1059 for NIST P-521, 1027 for brainpoolP512r1, and 263 for both Curve25519Legacy and Ed25519Legacy. For example, the length of an EdDSALegacy public key for the curve Ed25519Legacy is 263 bits: 7 bits to represent the 0x40 prefix octet and 32 octets for the native value of the public key.

Even though the zero point (also called the "point at infinity") may occur as a result of arithmetic operations on points of an elliptic curve, it **SHALL NOT** appear in data structures defined in this document.

Each particular curve uses a designated wire format for the point found in its public key or ECDH data structure. An implementation **MUST NOT** use a different wire format for a point other than the wire format associated with the curve.

11.3. EC Scalar Wire Formats

Some non-curve values in elliptic curve cryptography (for example, secret keys and signature components) are not points on a curve, but they are also encoded on the wire in OpenPGP as an MPI.

Because of different patterns of deployment, some curves treat these values as opaque bit strings with the high bit set, while others are treated as actual integers, encoded in the standard OpenPGP big-endian form. The choice of encoding is specific to the public key algorithm in use.

Type	Description	Reference
integer	An integer encoded in big-endian format as a standard OpenPGP MPI	Section 3.2
octet string	An octet string of fixed length that may be shorter on the wire due to leading zeros being stripped by the MPI encoding and may need to be zero-padded before use	Section 11.3.1
prefixed N octets	An octet string of fixed length N, prefixed with octet 0x40 to ensure no leading zero octet	Section 11.3.2

Table 29: OpenPGP Elliptic Curve Scalar Encodings Registry

11.3.1. EC Octet String Wire Format

Some opaque strings of octets are represented on the wire as an MPI by simply stripping the leading zeros and counting the remaining bits. These strings are of known, fixed length. They are represented in this document as `MPI(N octets of X)`, where `N` is the expected length in octets of the octet string.

For example, a five-octet opaque string (`MPI(5 octets of X)`) where `X` has the value `00 02 EE 19 00` would be represented on the wire as an MPI like so: `00 1A 02 EE 19 00`.

To encode `X` to the wire format, set the MPI's two-octet bit counter to the value of the highest set bit (bit 26, or 0x001A), and do not transfer the leading all-zero octet to the wire.

To reverse the process, an implementation that knows this value has an expected length of 5 octets and can take the following steps:

- Ensure that the MPI's two-octet bit count is less than or equal to 40 (5 octets of 8 bits)
- Allocate 5 octets, setting all to zero initially
- Copy the MPI data octets (without the two count octets) into the lower octets of the allocated space

11.3.2. EC Prefixed Octet String Wire Format

Another way to ensure that a fixed-length bytes string is encoded simply to the wire while remaining in MPI format is to prefix the byte string with a dedicated non-zero octet. This specification uses 0x40 as the prefix octet. This is represented in this specification as `MPI(prefixed N octets of X)`, where `N` is the known byte string length.

For example, a five-octet opaque string using `MPI(prefixed 5 octets of X)` where `X` has the value `00 02 EE 19 00` would be written to the wire form as: `00 2F 40 00 02 EE 19 00`.

To encode the string, prefix it with the octet 0x40 (whose 7th bit is set), and then set the MPI's two-octet bit counter to 47 (0x002F -- 7 bits for the prefix octet and 40 bits for the string).

To decode the string from the wire, an implementation that knows that the variable is formed in this way can:

- ensure that the first three octets of the MPI (the two-bit count octets plus the prefix octet) are `00 2F 40`, and
- use the remainder of the MPI directly off the wire.

Note that this is a similar approach to that used in the EC point encodings found in Section 11.2.2.

11.4. Key Derivation Function

A key derivation function (KDF) is necessary to implement EC encryption. The Concatenation Key Derivation Function (Approved Alternative 1) [SP800-56A] with the KDF hash function that is SHA2-256 [FIPS180] or stronger is **REQUIRED**.

For convenience, the synopsis of the encoding method is given below with significant simplifications attributable to the restricted choice of hash functions in this document. However, [SP800-56A] is the normative source of the definition.

```
//   Implements KDF( X, oBits, Param );
//   Input: point X = (x,y)
//   oBits - the desired size of output
//   hBits - the size of output of hash function Hash
//   Param - octets representing the parameters
//   Assumes that oBits <= hBits
// Convert the point X to the octet string:
//   ZB' = 04 || x || y
// and extract the x portion from ZB'
ZB = x;
MB = Hash ( 00 || 00 || 00 || 01 || ZB || Param );
return oBits leftmost bits of MB.
```

Note that ZB in the KDF description above is the compact representation of X as defined in Section 4.2 of [RFC6090].

11.5. EC DH Algorithm (ECDH)

The method is a combination of an ECC Diffie-Hellman method to establish a shared secret, a key derivation method to process the shared secret into a derived key, and a key wrapping method that uses the derived key to protect a session key used to encrypt a message.

The One-Pass Diffie-Hellman method C(1, 1, ECC CDH) [SP800-56A] **MUST** be implemented with the following restrictions: the ECC Cofactor Diffie-Hellman (CDH) primitive employed by this method is modified to always assume the cofactor is 1, the KDF specified in Section 11.4 is used, and the KDF parameters specified below are used.

The KDF parameters are encoded as a concatenation of the following 5 variable-length and fixed-length fields, which are compatible with the definition of the OtherInfo bit string [SP800-56A]:

- A variable-length field containing a curve OID, which is formatted as follows:
  - A one-octet size of the following field.
  - The octets representing a curve OID, as defined in Section 9.2.
- A one-octet public key algorithm ID, as defined in Section 9.1.
- A variable-length field containing KDF parameters, which are identical to the corresponding field in the ECDH public key and formatted as follows:
  - A one-octet size of the following fields; values 0 and 0xFF are reserved for future extensions.
  - A one-octet value 0x01, reserved for future extensions.
  - A one-octet hash function ID used with the KDF.
  - A one-octet algorithm ID for the symmetric algorithm that is used to wrap the symmetric key for message encryption; see Section 11.5 for details.
- 20 octets representing the UTF-8 encoding of the string `Anonymous Sender`, which is the octet sequence 41 6E 6F 6E 79 6D 6F 75 73 20 53 65 6E 64 65 72 20 20 20 20.
- A variable-length field containing the fingerprint of the recipient encryption subkey identifying the key material that is needed for decryption. For version 4 keys, this field is 20 octets. For version 6 keys, this field is 32 octets.

The size in octets of the KDF parameters sequence, as defined above, for encrypting to a v4 key is 54 for curve NIST P-256; 51 for curves NIST P-384 and NIST P-521; 55 for curves brainpoolP256r1, brainpoolP384r1, and brainpoolP512r1; or 56 for Curve25519Legacy. For encrypting to a v6 key, the size of the sequence is 66 for curve NIST P-256; 63 for curves NIST P-384 and NIST P-521; or 67 for curves brainpoolP256r1, brainpoolP384r1, and brainpoolP512r1.

The key wrapping method is described in [RFC3394]. The KDF produces a symmetric key that is used as a KEK as specified in [RFC3394]. Refer to Section 11.5.1 for the details regarding the choice of the KEK algorithm, which **SHOULD** be one of the three AES algorithms. Key wrapping and unwrapping is performed with the default initial value of [RFC3394].

To produce the input to the key wrapping method, first concatenate the following values:

- The one-octet algorithm identifier, if it was passed (in the case of a v3 PKESK packet).
- The session key.
- A two-octet checksum of the session key, equal to the sum of the session key octets, modulo 65536.

Then, the above values are padded to an 8-octet granularity using the method described in [RFC2898].

For example, in a v3 Public-Key Encrypted Session Key packet, an AES-256 session key is encoded as follows, forming a 40-octet sequence:

```
09 k0 k1 ... k31 s0 s1 05 05 05 05 05
```

The octets k0 to k31 above denote the session key, and the octets s0 and s1 denote the checksum of the session key octets. This encoding allows the sender to obfuscate the size of the symmetric encryption key used to encrypt the data. For example, assuming that an AES algorithm is used for the session key, the sender **MAY** use 21, 13, and 5 octets of padding for AES-128, AES-192, and AES-256, respectively, to provide the same number of octets, 40 total, as an input to the key wrapping method.

In a v6 Public-Key Encrypted Session Key packet, the symmetric algorithm is not included, as described in Section 5.1. For example, an AES-256 session key would be composed as follows:

```
k0 k1 ... k31 s0 s1 06 06 06 06 06 06
```

The octets k0 to k31 above again denote the session key, and the octets s0 and s1 denote the checksum. In this case, assuming that an AES algorithm is used for the session key, the sender **MAY** use 22, 14, and 6 octets of padding for AES-128, AES-192, and AES-256, respectively, to provide the same number of octets, 40 total, as an input to the key wrapping method.

The output of the method consists of two fields. The first field is the MPI containing the ephemeral key used to establish the shared secret. The second field is composed of the following two subfields:

- One octet encoding the size in octets of the result of the key wrapping method; the value 255 is reserved for future extensions.
- Up to 254 octets representing the result of the key wrapping method, applied to the 8-octet padded session key, as described above.

Note that for session key sizes 128, 192, and 256 bits, the size of the result of the key wrapping method is, respectively, 32, 40, and 48 octets, unless size obfuscation is used.

For convenience, the synopsis of the encoding method is given below; however, this section, [SP800-56A], and [RFC3394] are the normative sources of the definition.

- Obtain the authenticated recipient public key R
- Generate an ephemeral, single-use key pair {v, V=vG}
- Compute the shared point S = vR;
- m = symm_alg_ID || session key || checksum || pkcs5_padding;
- curve_OID_len = (octet)len(curve_OID);
- Param = curve_OID_len || curve_OID || public_key_alg_ID || 03 || 01 || KDF_hash_ID || KEK_alg_ID for AESKeyWrap || `Anonymous Sender` || recipient_fingerprint;
- Z_len = the key size for the KEK_alg_ID used with AESKeyWrap
- Compute Z = KDF(S, Z_len, Param);
- Compute C = AESKeyWrap(Z, m); (per [RFC3394])

- Wipe the memory that contained S, v, and Z to avoid leaking ephemeral secrets
- VB = convert point V to the octet string
- Output (MPI(VB) || len(C) || C).

The decryption is the inverse of the method given. Note that the recipient with key pair (r,R) obtains the shared secret by calculating:

```
S = rV = rvG
```

11.5.1. ECDH Parameters

ECDH keys have a hash algorithm parameter for key derivation and a symmetric algorithm for key encapsulation.

For v6 keys, the following algorithms **MUST** be used depending on the curve. An implementation **MUST NOT** generate a v6 ECDH key over any listed curve that uses different KDF or KEK parameters. An implementation **MUST NOT** encrypt any message to a v6 ECDH key over a listed curve that announces a different KDF or KEK parameter.

For v4 keys, the following algorithms **SHOULD** be used depending on the curve. An implementation **SHOULD** only use an AES algorithm as a KEK algorithm.

Curve	Hash Algorithm	Symmetric Algorithm
NIST P-256	SHA2-256	AES-128
NIST P-384	SHA2-384	AES-192
NIST P-521	SHA2-512	AES-256
brainpoolP256r1	SHA2-256	AES-128
brainpoolP384r1	SHA2-384	AES-192
brainpoolP512r1	SHA2-512	AES-256
Curve25519Legacy	SHA2-256	AES-128

Table 30: OpenPGP ECDH KDF and KEK Parameters Registry

12. Notes on Algorithms

12.1. PKCS#1 Encoding in OpenPGP

This specification makes use of the PKCS#1 functions EME-PKCS1-v1_5 and EMSA-PKCS1-v1_5. However, the calling conventions of these functions have changed in the past. To avoid potential confusion and interoperability problems, we are including local copies in this document, adapted

from those in PKCS#1 v2.1 [RFC8017]. [RFC8017] should be treated as the ultimate authority on PKCS#1 for OpenPGP. Nonetheless, we believe that there is value in having a self-contained document that avoids problems in the future with needed changes in the conventions.

12.1.1. EME-PKCS1-v1_5-ENCODE

Input:

k = key modulus length in octets.

M = message to be encoded; an octet string of length mLen, where mLen <= k - 11.

Output:

EM = encoded message; an octet string of length k.

Error: "message too long".

1. Length checking: If mLen > k - 11, output "message too long" and stop.
2. Generate an octet string PS of length k - mLen - 3 consisting of pseudorandomly generated non-zero octets. The length of PS will be at least eight octets.
3. Concatenate PS, the message M, and other padding to form an encoded message EM of length k octets as

```
EM = 0x00 || 0x02 || PS || 0x00 || M.
```

4. Output EM.

12.1.2. EME-PKCS1-v1_5-DECODE

Input:

EM = encoded message; an octet string.

Output:

M = message; an octet string.

Error: "decryption error".

To decode an EME-PKCS1_v1_5 message, separate the encoded message EM into an octet string PS consisting of non-zero octets and a message M as follows

```
EM = 0x00 || 0x02 || PS || 0x00 || M.
```

If the first octet of EM does not have hexadecimal value 0x00, the second octet of EM does not have hexadecimal value 0x02, there is no octet with hexadecimal value 0x00 to separate PS from M, or the length of PS is less than 8 octets, output "decryption error" and stop. See also Section 13.5 regarding differences in reporting between a decryption error and a padding error.

12.1.3. EMSA-PKCS1-v1_5

This encoding method is deterministic and only has an encoding operation.

Option:

Hash - hash function in which hLen denotes the length in octets of the hash function output.

Input:

M = message to be encoded.

emLen = intended length of the encoded message in octets, at least tLen + 11, where tLen is the octet length of the DER encoding T of a certain value computed during the encoding operation.

Output:

EM = encoded message; an octet string of length emLen.

Errors: "message too long"; "intended encoded message length too short".

Steps:

1. Apply the hash function to the message M to produce hash value H:

   H = Hash(M).

   If the hash function outputs "message too long," output "message too long" and stop.
2. Using the list in Section 9.5, produce an ASN.1 DER value for the hash function used. Let T be the full hash prefix from the list, and let tLen be the length in octets of T.
3. If emLen < tLen + 11, output "intended encoded message length too short" and stop.
4. Generate an octet string PS consisting of emLen - tLen - 3 octets with hexadecimal value 0xFF. The length of PS will be at least 8 octets.
5. Concatenate PS, the hash prefix T, and other padding to form the encoded message EM as

   ```
   EM = 0x00 || 0x01 || PS || 0x00 || T.
   ```

6. Output EM.

12.2. Symmetric Algorithm Preferences

The symmetric algorithm preference is an ordered list of algorithms that the keyholder accepts. Since it is found on a self-signature, it is possible that a keyholder may have multiple, different preferences. For example, Alice may have AES-128 only specified for "alice@work.com" but Camellia-256, Twofish, and AES-128 specified for "alice@home.org". Note that it is also possible for preferences to be in a subkey's binding signature.

Since AES-128 is the algorithm that **MUST** be implemented, if it is not explicitly in the list, it is tacitly at the end. However, it is good form to place it there explicitly. Note also that if an implementation does not implement the preference, then it is implicitly an AES-128-only implementation. Furthermore, note that implementations conforming to previous versions of this specification [RFC4880] have TripleDES as the only algorithm that **MUST** be implemented.

An implementation **MUST NOT** use a symmetric algorithm that is not in the recipient's preference list. When encrypting to more than one recipient, the implementation finds a suitable algorithm by taking the intersection of the preferences of the recipients. Note that the AES-128 algorithm **MUST** be implemented to ensure that the intersection is non-empty. The implementation may use any mechanism to pick an algorithm in the intersection.

If an implementation can decrypt a message that a keyholder doesn't have in their preferences, the implementation **SHOULD** decrypt the message anyway, but it **MUST** warn the keyholder that the protocol has been violated. For example, suppose that Alice (above) has an implementation that implements all algorithms in this specification. Nonetheless, she prefers subsets for work or home. If she is sent a message encrypted with IDEA, which is not in her preferences, the implementation warns her that someone sent an IDEA-encrypted message, but it would ideally decrypt it anyway.

12.2.1. Plaintext

Algorithm 0, "plaintext", may only be used to denote secret keys that are stored in the clear. An implementation **MUST NOT** use algorithm 0 as the indicated symmetric cipher for an encrypted data packet (Sections 5.7 or 5.13); it can use a Literal Data packet (Section 5.9) to encode unencrypted literal data.

12.3. Other Algorithm Preferences

Other algorithm preferences work similarly to the symmetric algorithm preference in that they specify which algorithms the keyholder accepts. There are two interesting cases in which further comments are needed: the compression preferences and the hash preferences.

12.3.1. Compression Preferences

Like the algorithm preferences, an implementation **MUST NOT** use an algorithm that is not in the preference vector. If Uncompressed (0) is not explicitly in the list, it is tacitly at the end. That is, uncompressed messages may always be sent.

Note that earlier implementations may assume that the absence of compression preferences means that [ZIP(1), Uncompressed(0)] are preferred, and default to ZIP compression. Therefore, an implementation that prefers uncompressed data **SHOULD** explicitly state this in the preferred compression algorithms.

12.3.1.1. Uncompressed

Algorithm 0, "uncompressed", may only be used to denote a preference for uncompressed data. An implementation **MUST NOT** use algorithm 0 as the indicated compression algorithm in a Compressed Data packet (Section 5.6); it can use a Literal Data packet (Section 5.9) to encode uncompressed literal data.

12.3.2. Hash Algorithm Preferences

Typically, the signer chooses what hash algorithm to use, rather than the verifier, because a signer rarely knows who is going to be verifying the signature. This preference allows a protocol based upon digital signatures ease in negotiation.

Thus, if Alice is authenticating herself to Bob with a signature, it makes sense for her to use a hash algorithm that Bob's implementation uses. This preference allows Bob to state which algorithms Alice may use in his key.

Since SHA2-256 is the hash algorithm that **MUST** be implemented, if it is not explicitly in the list, it is tacitly at the end. However, it is good form to place it there explicitly.

12.4. RSA

The PKCS1-v1_5 padding scheme, used by the RSA algorithms defined in this document, is no longer recommended, and its use is deprecated by [SP800-131A]. Therefore, an implementation **SHOULD NOT** generate RSA keys.

There are algorithm types for RSA Sign-Only and RSA Encrypt-Only keys. These types are deprecated. The "key flags" subpacket in a signature is a much better way to express the same idea and generalize it to all algorithms. An implementation **MUST NOT** create such a key, but it **MAY** interpret it.

An implementation **MUST NOT** generate RSA keys of a size less than 3072 bits. An implementation **SHOULD NOT** encrypt, sign, or verify using RSA keys of a size less than 3072 bits. An implementation **MUST NOT** encrypt, sign, or verify using RSA keys of a size less than 2048 bits. An implementation that decrypts a message using an RSA secret key of a size less than 3072 bits **SHOULD** generate a deprecation warning that the key is too weak for modern use.

12.5. DSA

DSA is no longer recommended. It has also been deprecated in [FIPS186]. Therefore, an implementation **MUST NOT** generate DSA keys.

An implementation **MUST NOT** sign or verify using DSA keys.

12.6. Elgamal

The PKCS1-v1_5 padding scheme, used by the Elgamal algorithm defined in this document, is no longer recommended, and its use is deprecated by [SP800-131A]. Therefore, an implementation **MUST NOT** generate Elgamal keys.

An implementation **MUST NOT** encrypt using Elgamal keys. An implementation that decrypts a message using an Elgamal secret key **SHOULD** generate a deprecation warning that the key is too weak for modern use.

12.7. EdDSA

Although the EdDSA algorithm allows arbitrary data as input, its use with OpenPGP requires that a digest of the message be used as input (pre-hashed). See Section 5.2.4 for details. Truncation of the resulting digest is never applied; the resulting digest value is used verbatim as input to the EdDSA algorithm.

For clarity: while [RFC8032] describes different variants of EdDSA, OpenPGP uses the "pure" variant (PureEdDSA). The hashing that happens with OpenPGP is done as part of the standard OpenPGP signature process, and that hash itself is fed as the input message to the PureEdDSA algorithm.

As specified in [RFC8032], Ed448 also expects a "context string". In OpenPGP, Ed448 is used with the empty string as a context string.

12.8. Reserved Algorithm IDs

A number of algorithm IDs have been reserved for algorithms that would be useful to use in an OpenPGP implementation, yet there are issues that prevent an implementer from actually implementing the algorithm. These are marked as reserved in Section 9.1.

The reserved public-key algorithm X9.42 (21) does not have the necessary parameters, parameter order, or semantics defined. The same is currently true for reserved public-key algorithms AEDH (23) and AEDSA (24).

Previous versions of the OpenPGP specification permitted Elgamal [ELGAMAL] signatures with a public-key algorithm ID of 20. These are no longer permitted. An implementation **MUST NOT** generate such keys. An implementation **MUST NOT** generate Elgamal signatures; see [BLEICHENBACHER].

12.9. CFB Mode

The Cipher Feedback (CFB) mode used in this document is defined in Section 6.3 of [SP800-38A].

The CFB segment size `s` is equal to the block size of the cipher (i.e., n-bit CFB mode, where n is the block size used).

12.10. Private or Experimental Parameters

S2K specifiers, Signature subpacket type IDs, User Attribute subpacket type IDs, image format IDs, and the various algorithm IDs described in Section 9 all reserve the range 100 to 110 for Private and Experimental Use. Packet type IDs reserve the range 60 to 63 for Private and Experimental Use. These are intentionally managed by the Private Use method, as described in [RFC8126].

However, implementations need to be careful with these and promote them to full IANA-managed parameters when they grow beyond the original, limited system.

12.11. Meta Considerations for Expansion

If OpenPGP is extended in a way that is not backward compatible, meaning that old implementations will not gracefully handle their absence of a new feature, the extension proposal can be declared in the keyholder's self-signature as part of the Features signature subpacket.

We cannot state definitively what extensions will not be upward compatible, but typically new algorithms are upward compatible, whereas new packets are not.

If an extension proposal does not update the Features system, it **SHOULD** include an explanation of why this is unnecessary. If the proposal contains neither an extension to the Features system nor an explanation of why such an extension is unnecessary, the proposal **SHOULD** be rejected.

13. Security Considerations

- As with any technology involving cryptography, implementers should check the current literature to determine if any algorithms used here have been found to be vulnerable to an attack. If so, implementers should consider disallowing such algorithms for new data and warning the end user, or preventing use, when they are trying to consume data protected by such algorithms that are now vulnerable.
- This specification uses Public-Key Cryptography technologies. It is assumed that the private key portion of a public-private key pair is controlled and secured by the proper party or parties.
- The MD5 and SHA-1 hash algorithms have been found to have weaknesses, with collisions found in a number of cases. MD5 and SHA-1 are deprecated for use in OpenPGP (see Section 9.5).
- Many security protocol designers think that it is a bad idea to use a single key for both privacy (encryption) and integrity (signatures). In fact, this was one of the motivating forces behind the v4 key format with separate signature and encryption keys. Using a single key for encrypting and signing is discouraged.

- The DSA algorithm will work with any hash, but it is sensitive to the quality of the hash algorithm. Verifiers should be aware that even if the signer used a strong hash, an attacker could have modified the signature to use a weak one. Only signatures using acceptably strong hash algorithms should be accepted as valid.
- As OpenPGP combines many different asymmetric, symmetric, and hash algorithms, each with different measures of strength, care should be taken to ensure that the weakest element of an OpenPGP message is still sufficiently strong for the purpose at hand. While consensus about the strength of a given algorithm may evolve, NIST Special Publication 800-57 [SP800-57] contains recommendations (current at the time of this publication) about equivalent security levels of different algorithms.
- There is a somewhat-related potential security problem in signatures. If an attacker can find a message that hashes to the same hash with a different algorithm, a bogus signature structure can be constructed that evaluates correctly.

  For example, suppose Alice DSA-signs message M using hash algorithm H. Suppose that Mallet finds a message M' that has the same hash value as M with H'. Mallet can then construct a signature block that verifies as Alice's signature of M' with H'. However, this would also constitute a weakness in either H or H', or both. Should this ever occur, a revision will have to be made to this document to revise the allowed hash algorithms.
- If you are building an authentication system, the recipient may specify a preferred signing algorithm. However, the signer would be foolish to use a weak algorithm simply because the recipient requests it.
- Some of the encryption algorithms mentioned in this document have been analyzed less than others. For example, although TWOFISH is presently considered reasonably strong, it has been analyzed much less than AES. Other algorithms may have other concerns surrounding them.
- In late summer 2002, Jallad, Katz, and Schneier published an interesting attack on previous versions of the OpenPGP protocol and some of its implementations [JKS02]. In this attack, the attacker modifies a message and sends it to a user who then returns the erroneously decrypted message to the attacker. The attacker is thus using the user as a decryption oracle and can often decrypt the message. This attack is a particular form of ciphertext malleability. See Section 13.7 for information on how to defend against such an attack using more recent versions of OpenPGP.

13.1. SHA-1 Collision Detection

As described in [SHAMBLES], the SHA-1 digest algorithm is not collision resistant. However, an OpenPGP implementation cannot completely discard the SHA-1 algorithm, because it is required for implementing v4 public keys. In particular, the v4 fingerprint derivation uses SHA-1. So as long as an OpenPGP implementation supports v4 public keys, it will need to implement SHA-1 in at least some scenarios.

To avoid the risk of uncertain breakage from a maliciously introduced SHA-1 collision, an OpenPGP implementation **MAY** attempt to detect when a hash input is likely from a known collision attack and then either reject the hash input deliberately or modify the hash output. This should convert an uncertain breakage (where it is unclear what the effect of a collision will be) to an explicit breakage, which is more desirable for a robust implementation.

[STEVENS2013] describes a method for detecting indicators of well-known SHA-1 collision attacks. Some example C code implementing this technique can be found at [SHA1CD].

13.2. Advantages of Salted Signatures

V6 signatures include a salt that is hashed first, and it's size depends on the hashing algorithm. This makes v6 OpenPGP signatures non-deterministic and protects against a broad class of attacks that depend on creating a signature over a predictable message. By selecting a new random salt for each signature made, the signed hashes and the signatures are not predictable.

While the material to be signed could be attacker controlled, hashing the salt first means that there is no attacker-controlled hashed prefix. An example of this kind of attack is described in the paper "SHA-1 is a Shambles" [SHAMBLES], which leverages a chosen prefix collision attack against SHA-1. This means that an adversary carrying out a chosen-message attack will not be able to control the hash that is being signed and will need to break second-preimage resistance instead of the simpler collision resistance to create two messages having the same signature. The size of the salt is bound to the hash function to match the expected collision-resistance level and is at least 16 octets.

In some cases, an attacker may be able to induce a signature to be made, even if they do not control the content of the message. In some scenarios, a repeated signature over the exact same message may risk leakage of part or all of the signing key; for example, see discussion of hardware faults over EdDSA and deterministic ECDSA in [PSSLR17]. Choosing a new random salt for each signature ensures that no repeated signatures are produced, which mitigates this risk.

13.3. Elliptic Curve Side Channels

Side-channel attacks are a concern when a compliant application's use of the OpenPGP format can be modeled by a decryption or signing oracle, for example, when an application is a network service performing decryption to unauthenticated remote users. ECC scalar multiplication operations used in ECDSA and ECDH are vulnerable to side-channel attacks. Countermeasures can often be taken at the higher protocol level, such as limiting the number of allowed failures or time-blinding the operations associated with each network interface. Mitigations at the scalar multiplication level seek to eliminate any measurable distinction between the ECC point addition and doubling operations.

13.4. Risks of a Quick Check Oracle

In winter 2005, Serge Mister and Robert Zuccherato from Entrust released a paper describing a way that the "quick check" in v1 SEIPD and SED packets can be used as an oracle to decrypt two octets of every cipher block [MZ05]. This check was intended for early detection of session key decryption errors, particularly to detect a wrong passphrase, since v4 SKESK packets do not include an integrity check.

There is a danger when using the quick check if timing or error information about the check can be exposed to an attacker, particularly via an automated service that allows rapidly repeated queries.

Disabling the quick check prevents the attack.

For very large encrypted data whose session key is protected by a passphrase using a version 4 SKESK, the quick check may be convenient to the user by informing them early that they typed the wrong passphrase. But the implementation should use the quick check with care. The recommended approach for secure and early detection of decryption failure is to encrypt data using v2 SEIPD. If the session key is public-key encrypted, the quick check is not useful as the public-key encryption of the session key should guarantee that it is the right session key.

The quick check oracle attack is a particular type of attack that exploits ciphertext malleability. For information about other similar attacks, see Section 13.7.

13.5. Avoiding Leaks from PKCS#1 Errors

The PKCS#1 padding (used in RSA-encrypted and ElGamal-encrypted PKESK) has been found to be vulnerable to attacks in which a system that allows distinguishing padding errors from other decryption errors can act as a decryption and/or signing oracle that can leak the session key or allow signing arbitrary data, respectively [BLEICHENBACHER-PKCS1]. The number of queries required to carry out an attack can range from thousands to millions, depending on how strict and careful an implementation is in processing the padding.

To make the attack more difficult, an implementation **SHOULD** implement strict, robust, and constant time padding checks.

To prevent the attack, in settings where the attacker does not have access to timing information concerning message decryption, the simplest solution is to report a single error code for all variants of PKESK processing errors as well as SEIPD integrity errors (this also includes session key parsing errors, such as on an invalid cipher algorithm for v3 PKESK, or a session key size mismatch for v6 PKESK). If the attacker may have access to timing information, then a constant time solution is also needed. This requires careful design, especially for v3 PKESK, where session key size and cipher information is typically not known in advance, as it is part of the PKESK encrypted payload.

13.6. Fingerprint Usability

This specification uses fingerprints in several places on the wire (e.g., Sections 5.2.3.23, 5.2.3.35, and 5.2.3.36) and in processing (e.g., in ECDH KDF Section 11.5). An implementation may also use the fingerprint internally, for example, as an index to a keystore.

Additionally, some OpenPGP users have historically used manual fingerprint comparison to verify the public key of a peer. For a version 4 fingerprint, this has typically been done with the fingerprint represented as 40 hexadecimal digits, often broken into groups of four digits with whitespace between each group.

When a human is actively involved, the result of such a verification is dubious. There is little evidence that most humans are good at precise comparison of high-entropy data, particularly when that data is represented in compact textual form like a hexadecimal [USENIX-STUDY].

The version 6 fingerprint makes the challenge for a human verifier even worse. At 256 bits (compared to v4's 160-bit fingerprint), a v6 fingerprint is even harder for a human to successfully compare.

An OpenPGP implementation should prioritize mechanical fingerprint transfer and comparison where possible and **SHOULD NOT** promote manual transfer or comparison of full fingerprints by a human unless there is no other way to achieve the desired result.

While this subsection acknowledges existing practice for human-representable v4 fingerprints, this document does not attempt to standardize any specific human-readable form of v6 fingerprint for this discouraged use case.

NOTE: the topic of interoperable human-in-the-loop key verification needs more work, which will be done in a separate document.

13.7. Avoiding Ciphertext Malleability

If ciphertext can be modified by an attacker but still subsequently decrypted to some new plaintext, it is considered "malleable". A number of attacks can arise in any cryptosystem that uses malleable encryption, so [RFC4880] and later versions of OpenPGP offer mechanisms to defend against it. However, OpenPGP data may have been created before these defense mechanisms were available. Because OpenPGP implementations deal with historic stored data, they may encounter malleable ciphertexts.

When an OpenPGP implementation discovers that it is decrypting data that appears to be malleable, it **MUST** generate a clear error message that indicates the integrity of the message is suspect, it **SHOULD NOT** attempt to parse nor release decrypted data to the user, and it **SHOULD** halt with an error. Parsing or releasing decrypted data before having confirmed its integrity can leak the decrypted data [EFAIL] [MRLG15].

In the case of AEAD encrypted data, if the authentication tag fails to verify, the implementation **MUST NOT** attempt to parse nor release decrypted data to the user, and it **MUST** halt with an error.

An implementation that encounters malleable ciphertext **MAY** choose to release cleartext to the user if it is not encrypted using AEAD, it is known to be dealing with historic archived legacy data, and the user is aware of the risks.

In the case of AEAD encrypted messages, if the message is truncated, i.e., the final zero-octet chunk and possibly (part of) some chunks before it are missing, the implementation **MAY** choose to release cleartext from the fully authenticated chunks before it to the user if it is operating in a streaming fashion, but it **MUST** indicate a clear error message as soon as the truncation is detected.

Any of the following OpenPGP data elements indicate that malleable ciphertext is present:

- All Symmetrically Encrypted Data packets (Section 5.7).
- Within any encrypted container, any Compressed Data packet (Section 5.6) where there is a decompression failure.
- Any version 1 Symmetrically Encrypted Integrity Protected Data packet (Section 5.13.1) where the internal Modification Detection Code does not validate.
- Any version 2 Symmetrically Encrypted Integrity Protected Data packet (Section 5.13.2) where the authentication tag of any chunk fails or where there is no final zero-octet chunk.
- Any Secret-Key packet with encrypted secret key material (Section 3.7.2.1) where there is an integrity failure, based on the value of the secret key protection octet:
  - Value 255 (MalleableCFB) or raw cipher algorithm: where the trailing 2-octet checksum does not match.
  - Value 254 (CFB): where the SHA1 checksum is mismatched.
  - Value 253 (AEAD): where the AEAD authentication tag is invalid.

To avoid these circumstances, an implementation that generates OpenPGP encrypted data **SHOULD** select the encrypted container format with the most robust protections that can be handled by the intended recipients. In particular:

- The SED packet is deprecated and **MUST NOT** be generated.
- When encrypting to one or more public keys:
  - If all recipient keys indicate support for version 2 of the Symmetrically Encrypted Integrity Protected Data packet in their Features subpacket (Section 5.2.3.32), if all recipient keys are v6 keys without a Features subpacket, or the implementation can otherwise infer that all recipients support v2 SEIPD packets, the implementation **SHOULD** encrypt using a v2 SEIPD packet.
  - If one of the recipients does not support v2 SEIPD packets, then the message generator **MAY** use a v1 SEIPD packet instead.

- Passphrase-protected secret key material in a v6 Secret Key or v6 Secret Subkey packet **SHOULD** be protected with AEAD encryption (S2K usage octet 253) unless it will be transferred to an implementation that is known to not support AEAD. An implementation should be aware that, in scenarios where an attacker has write access to encrypted private keys, CFB-encrypted keys (S2K usage octet 254 or 255) are vulnerable to corruption attacks that can cause leakage of secret data when the secret key is used [KOPENPGP] [KR02].

Implementers should implement AEAD (v2 SEIPD and S2K usage octet 253) promptly and encourage its spread.

Users are **RECOMMENDED** to migrate to AEAD.

13.8. Secure Use of the v2 SEIPD Session-Key-Reuse Feature

The salted key derivation of v2 SEIPD packets (Section 5.13.2) allows the recipient of an encrypted message to reply to the sender and all other recipients without needing their public keys but by using the same v6 PKESK packets it received and a different random salt value. This ensures a secure mechanism on the cryptographic level that enables the use of message encryption in cases where a sender does not have a copy of an encryption-capable certificate for one or more participants in the conversation and thus can enhance the overall security of an application. However, care must be taken when using this mechanism not to create security vulnerabilities, such as the following:

- Replying to only a subset of the original recipients and the original sender by use of the session-key-reuse feature would mean that the remaining recipients (including the sender) of the original message could read the encrypted reply message, too.
- Adding a further recipient to the reply that is encrypted using the session-key-reuse feature gives that further recipient also cryptographic access to the original message that is being replied to (and potentially to a longer history of previous messages).
- A modification of the list of recipients addressed in the above points also needs to be safeguarded when a message is initially composed as a reply with session-key reuse but then is stored (e.g., as a draft) and later reopened for further editing and to be finally sent.
- There is the potential threat that an attacker with network or mailbox access, who is at the same time a recipient of the original message, silently removes themselves from the message before the victim's client receives it. The victim's client that then uses the mechanism for replying with session-key reuse would unknowingly compose an encrypted message that could be read by the attacker. Implementations are encouraged to use the Intended Recipient Fingerprint subpacket (Section 5.2.3.36) when composing messages and checking the consistency of the set of recipients of a message before replying to it with session-key reuse.
- When using the session-key-reuse feature in any higher-layer protocol, care should be taken to ensure that there is no other potentially interfering practice of session-key reuse established in that protocol. Such interfering session-key reuse could be given, for instance, if an initial message is already composed by reusing the session key of an existing encrypted file the access to which may be shared among a group of users already. Using the session-key-reuse feature to compose an encrypted reply to such a message would unknowingly give this whole group of users cryptographic access to the encrypted message.

- Generally, the use of the session-key-reuse feature should be under the control of the user. Specifically, care should be taken so that this feature is not silently used when the user assumes that proper public-key encryption is used. This can be the case, for instance, when the public key of one of the recipients of the reply is known but has expired. Special care should be taken to ensure that users do not get caught in continued use of the session-key reuse unknowingly but instead receive the chance to switch to proper fresh public-key encryption as soon as possible.
- Whenever possible, a client should prefer a fresh public key encryption over the session-key reuse.

Even though this is not necessarily a security aspect, note that initially composing an encrypted reply using the session-key-reuse feature on one client and then storing it (e.g., as a draft) and later reopening the stored unfinished reply with another client that does not support the session-key-reuse feature may lead to interoperability problems.

Avoiding the pitfalls described above requires context-specific expertise. An implementation should only make use of the session-key-reuse feature in any particular application layer when it can follow reasonable documentation about how to deploy the feature safely in the specific application. At the time of this writing, there is no known documentation about safe reuse of OpenPGP session keys for any specific context. An implementer that intends to make use of this feature should publish their own proposed guidance for others to review.

13.9. Escrowed Revocation Signatures

A keyholder, Alice, may wish to designate a third party to be able to revoke her own key.

The preferred way for Alice to do this is to produce a specific Revocation Signature (signature type IDs 0x20, 0x28, or 0x30) and distribute it securely to a preferred revoker who can hold it in escrow. The preferred revoker can then publish the escrowed Revocation Signature at whatever time is deemed appropriate rather than generating the revocation signature themselves.

There are multiple advantages of using an escrowed Revocation Signature over the deprecated Revocation Key subpacket (Section 5.2.3.23):

- The keyholder can constrain what types of revocation the preferred revoker can issue, by only escrowing those specific signatures.
- There is no public/visible linkage between the keyholder and the preferred revoker.
- Third parties can verify the revocation without needing to find the key of the preferred revoker.
- The preferred revoker doesn't even need to have a public OpenPGP key if some other secure transport is possible between them and the keyholder.
- Implementation support for enforcing a revocation from an authorized Revocation Key subpacket is uneven and unreliable.
- If the fingerprint mechanism suffers a cryptanalytic flaw, the escrowed Revocation Signature is not affected.

A Revocation Signature may also be split up into shares and distributed among multiple parties, requiring some subset of those parties to collaborate before the escrowed Revocation Signature is recreated.

13.10. Random Number Generation and Seeding

OpenPGP requires a cryptographically secure pseudorandom number generator (CSPRNG). In most cases, the operating system provides an appropriate facility such as a `getrandom()` syscall on Linux or BSD, which should be used absent other (for example, performance) concerns. It is **RECOMMENDED** to use an existing CSPRNG implementation as opposed to crafting a new one. Many adequate cryptographic libraries are already available under favorable license terms. Should those prove unsatisfactory, [RFC4086] provides guidance on the generation of random values.

OpenPGP uses random data with three different levels of visibility:

- In publicly visible fields such as nonces, IVs, public padding material, or salts.
- In shared-secret values, such as session keys for encrypted data or padding material within an encrypted packet.
- In entirely private data, such as asymmetric key generation.

With a properly functioning CSPRNG, this range of visibility does not present a security problem, as it is not feasible to determine the CSPRNG state from its output. However, with a broken CSPRNG, it may be possible for an attacker to use visible output to determine the CSPRNG internal state and thereby predict less-visible data like keying material, as documented in [CHECKOWAY].

An implementation can provide extra security against this form of attack by using separate CSPRNGs to generate random data with different levels of visibility.

13.11. Traffic Analysis

When sending OpenPGP data through the network, the size of the data may leak information to an attacker. There are circumstances where such a leak could be unacceptable from a security perspective.

For example, if possible cleartext messages for a given protocol are known to be either `yes` (three octets) or `no` (two octets) and the messages are sent within a Symmetrically Encrypted Integrity Protected Data packet, the length of the encrypted message will reveal the contents of the cleartext.

In another example, sending an OpenPGP Transferable Public Key over an encrypted network connection might reveal the length of the certificate. Since the length of an OpenPGP certificate varies based on the content, an external observer interested in metadata (who is trying to contact whom) may be able to guess the identity of the certificate sent, if its length is unique.

In both cases, an implementation can adjust the size of the compound structure by including a Padding packet (see Section 5.14).

13.12. Surreptitious Forwarding

When an attacker obtains a signature for some text, e.g., by receiving a signed message, they may be able to use that signature maliciously by sending a message purporting to come from the original sender, with the same body and signature, to a different recipient. To prevent this, an implementation **SHOULD** implement the Intended Recipient Fingerprint signature subpacket (Section 5.2.3.36).

13.13. Hashed vs. Unhashed Subpackets

Each OpenPGP signature can have subpackets in two different sections. The first set of subpackets (the "hashed section") is covered by the signature itself. The second set has no cryptographic protections and is used for advisory material only, including locally stored annotations about the signature.

For example, consider an implementation working with a specific signature that happens to know that the signature was made by a certain key, even though the signature contains no Issuer Fingerprint subpacket (Section 5.2.3.35) in the hashed section. That implementation **MAY** synthesize an Issuer Fingerprint subpacket and store it in the unhashed section so that it will be able to recall which key issued the signature in the future.

Some subpackets are only useful when they are in the hashed section, and an implementation **SHOULD** ignore them when they are found with unknown provenance in the unhashed section. For example, a Preferred AEAD Ciphersuites subpacket (Section 5.2.3.15) in a direct key self-signature indicates the preferences of the keyholder when encrypting SEIPD v2 data to the key. An implementation that observes such a subpacket found in the unhashed section would open itself to an attack where the recipient's certificate is tampered with to encourage the use of a specific cipher or mode of operation.

13.14. Malicious Compressed Data

It is possible to form a compression quine that produces itself upon decompression, leading to infinite regress in any implementation willing to parse arbitrary numbers of layers of compression. This could cause resource exhaustion, which itself could lead to termination by the operating system. If the operating system would create a "crash report", that report could contain confidential information.

An OpenPGP implementation **SHOULD** limit the number of layers of compression it is willing to decompress in a single message.

14. Implementation Considerations

This section is a collection of comments to help an implementer who has a particular interest in backward compatibility. Often the differences are small, but small differences are frequently more vexing than large differences. Thus, this is a non-comprehensive list of potential problems and gotchas for a developer who is trying to achieve backward compatibility.

- There are many possible ways for two keys to have the same key material but different fingerprints (and thus different Key IDs). For example, since a v4 fingerprint is constructed by hashing the key creation time along with other things, two v4 keys created at different times yet with the same key material will have different fingerprints.
- OpenPGP does not put limits on the size of public keys. However, larger keys are not necessarily better keys. Larger keys take more computation time to use, and this can quickly become impractical. Different OpenPGP implementations may also use different upper bounds for public key sizes, so care should be taken when choosing sizes to maintain interoperability.
- ASCII armor is an optional feature of OpenPGP. The OpenPGP Working Group strives for a minimal set of mandatory-to-implement features, and since there could be useful implementations that only use binary object formats, this is not a "**MUST**" feature for an implementation. For example, an implementation that is using OpenPGP as a mechanism for file signatures may find ASCII armor unnecessary. OpenPGP permits an implementation to declare what features it does and does not support, but ASCII armor is not one of these. Since most implementations allow binary and armored objects to be used indiscriminately, an implementation that does not implement ASCII armor may find itself with compatibility issues with general-purpose implementations. Moreover, implementations of OpenPGP-MIME [RFC3156] already have a requirement for ASCII armor, so those implementations will necessarily have support.
- What this document calls the "Legacy packet format" (Section 4.2.2) is what older documents called the "old packet format". It is the packet format used by implementations predating [RFC2440]. The current "OpenPGP packet format" (Section 4.2.1) was called the "new packet format" by older RFCs. This is the format introduced in [RFC2440] and maintained through [RFC4880] to this document.

14.1. Constrained Legacy Fingerprint Storage for v6 Keys

Some OpenPGP implementations have fixed length constraints for key fingerprint storage that will not fit all 32 octets of a v6 fingerprint. For example, [OPENPGPCARD] reserves 20 octets for each stored fingerprint.

An OpenPGP implementation **MUST NOT** attempt to map any part of a v6 fingerprint to such a constrained field unless the relevant specification for the constrained environment has explicit guidance for storing a v6 fingerprint that distinguishes it from a v4 fingerprint. An

implementation interacting with such a constrained field **SHOULD** directly calculate the v6 fingerprint from public key material and associated metadata instead of relying on the constrained field.

15. IANA Considerations

This document obsoletes [RFC4880]. IANA has updated all registration information that references [RFC4880] to reference this RFC instead.

15.1. Renamed Protocol Group

IANA bundles a set of registries associated with a particular protocol into a "protocol group". IANA has updated the name of the "Pretty Good Privacy (PGP)" protocol group (i.e., the group of registries described at <https://www.iana.org/assignments/pgp-parameters>) to "OpenPGP". IANA has arranged a permanent redirect from the existing URL to the new URL for the registries in this protocol group. All further updates specified below are for registries within this same OpenPGP protocol group.

15.2. Renamed and Updated Registries

IANA has renamed the "PGP String-to-Key (S2K)" registry to "OpenPGP String-to-Key (S2K) Types" and updated its content as shown in Table 1.

IANA has renamed the "PGP Packet Types/Tags" registry to "OpenPGP Packet Types" and updated its content as shown in Table 3.

IANA has renamed the "PGP User Attribute Types" registry to "OpenPGP User Attribute Subpacket Types" and updated its content as shown in Table 13.

IANA has renamed the "Image Format Subpacket Types" registry to "OpenPGP Image Attribute Encoding Format" and updated its content as shown in Table 15.

IANA has renamed the "Key Server Preference Extensions" registry to "OpenPGP Key Server Preference Flags" and updated its contents as shown in Table 8.

IANA has renamed the "Reason for Revocation Extensions" registry to "OpenPGP Reason for Revocation Code" and updated its contents as shown in Table 10.

IANA has renamed the "Key Flags Extensions" registry to "OpenPGP Key Flags" and updated its contents as shown in Table 9.

IANA has renamed the "Implementation Features" registry to "OpenPGP Features Flags" and updated its contents as shown in Table 11.

IANA has renamed the "Public Key Algorithms" registry to "OpenPGP Public Key Algorithms" and updated its contents as shown in Table 18.

IANA has renamed the "Symmetric Key Algorithms" registry to "OpenPGP Symmetric Key Algorithms" and updated its contents as shown in Table 21.

IANA has renamed the "Compression Algorithms" registry to "OpenPGP Compression Algorithms" and updated its contents as shown in Table 22.

IANA has renamed the "Hash Algorithms" registry to "OpenPGP Hash Algorithms" and updated its contents as shown in Table 23.

IANA has renamed the "Signature Subpacket Types" registry to "OpenPGP Signature Subpacket Types" and updated its contents as shown in Table 5.

15.3. Removed Registry

IANA has marked the empty "New Packet Versions" registry as OBSOLETE.

A tombstone note has been added to the OpenPGP protocol group with the following content:

> Those wishing to use the removed "New Packet Versions" registry should instead register new versions of the relevant packets in the "OpenPGP Key and Signature Versions", "OpenPGP Key ID and Fingerprint", and "OpenPGP Encrypted Message Packet Versions" registries.

15.4. Added Registries

IANA has added the following registries in the OpenPGP protocol group. Note that the initial contents of each registry is shown in the corresponding table.

- "OpenPGP Secret Key Encryption (S2K Usage Octet)" (Table 2).
- "OpenPGP Signature Types" (Table 4).
- "OpenPGP Signature Notation Data Subpacket Notation Flags" (Table 6).
- "OpenPGP Signature Notation Data Subpacket Types" (Table 7).
- "OpenPGP Key ID and Fingerprint" (Table 12).
- "OpenPGP Image Attribute Version" (Table 14).
- "OpenPGP Armor Header Line" (Table 16).
- "OpenPGP Armor Header Key" (Table 17).
- "OpenPGP ECC Curve OID and Usage" (Table 19).
- "OpenPGP ECC Curve-Specific Wire Formats" (Table 20).
- "OpenPGP Hash Algorithm Identifiers for RSA Signatures' Use of EMSA-PKCS1-v1_5 Padding" (Table 24).
- "OpenPGP AEAD Algorithms" (Table 25).
- "OpenPGP Encrypted Message Packet Versions" (Table 26).
- "OpenPGP Key and Signature Versions" (Table 27).

- "OpenPGP Elliptic Curve Point Wire Formats" (Table 28).
- "OpenPGP Elliptic Curve Scalar Encodings" (Table 29).
- "OpenPGP ECDH KDF and KEK Parameters" (Table 30).

15.5. Registration Policies

All registries within the OpenPGP protocol group, with the exception of the registries listed in Section 15.5.1, use the Specification Required registration policy; see Section 4.6 of [RFC8126]. This policy means that review and approval by a designated expert is required and that the IDs and their meanings must be documented in a permanent and readily available public specification, in sufficient detail, so that interoperability between independent implementations is possible.

15.5.1. Registries That Use RFC Required

The following registries use the RFC Required registration policy, as described in Section 4.7 of [RFC8126]:

- "OpenPGP Packet Types" (Table 3).
- "OpenPGP Key and Signature Versions" (Table 27).
- "OpenPGP Key ID and Fingerprint" (Table 12).
- "OpenPGP Encrypted Message Packet Versions" (Table 26).

15.6. Designated Experts

The designated experts will determine whether the new registrations retain the security properties that are expected by the base implementation and whether these new registrations do not cause interoperability issues with existing implementations, other than not producing or consuming the IDs associated with these new registrations. Registration proposals that fail to meet these criteria could instead be proposed as new work items for the OpenPGP Working Group or its successor.

The subsections below describe specific guidance for classes of registry updates that a designated expert will consider.

The designated experts should also consider Section 12.11 when reviewing proposed additions to the OpenPGP protocol group.

15.6.1. Key and Signature Versions

When defining a new version of OpenPGP keys or signatures, the "OpenPGP Key and Signature Versions" registry (Table 27) should be updated. When a new version of OpenPGP key is defined, the "OpenPGP Key ID and Fingerprint" registry (Table 12) should also be updated.

15.6.2. Encryption Versions

When defining a new version of the Symmetrically Encrypted Integrity Protected Data Packet (Section 5.13), Public Key Encrypted Session Key Packet (Section 5.1), and/or Symmetric Key Encrypted Session Key Packet (Section 5.3), the "OpenPGP Encrypted Message Packet Versions" registry (Table 26) should be updated. When the SEIPD is updated, consider also adding a corresponding flag to the "OpenPGP Features Flags" registry (Table 11).

15.6.3. Algorithms

Section 9 lists the cryptographic and compression algorithms that OpenPGP uses. Adding new algorithms is usually simple; in some cases, allocating an ID and pointing to a reference is only needed. But some algorithm registries require some subtle additional details when a new algorithm is introduced.

15.6.3.1. Elliptic Curve Algorithms

To register a new elliptic curve for use with OpenPGP, its OID needs to be registered in the "OpenPGP ECC Curve OID and Usage" registry (Table 19), its wire format needs to be documented in the "OpenPGP ECC Curve-Specific Wire Formats" registry (Table 20), and if used for ECDH, its KDF and KEK parameters must be populated in the "OpenPGP ECDH KDF and KEK Parameters" registry (Table 30). If the wire format(s) used is not already defined in the "OpenPGP Elliptic Curve Point Wire Formats" (Table 28) or "OpenPGP Elliptic Curve Scalar Encodings" (Table 29) registries, they should be defined there as well.

15.6.3.2. Symmetric-Key Algorithms

When registering a new symmetric cipher with a block size of 64 or 128 bits and a key size that is a multiple of 64 bits, no new considerations are needed.

If the new cipher has a different block size, there needs to be additional documentation describing how to use the cipher in CFB mode.

If the new cipher has an unusual key size, then padding needs to be considered for X25519 and X448 key wrapping, which currently needs no padding.

15.6.3.3. Hash Algorithms

When registering a new hash algorithm in the "OpenPGP Hash Algorithms" registry (Table 23), if the algorithm is also to be used with RSA signing schemes, it must also have an entry in the "OpenPGP Hash Algorithm Identifiers for RSA Signatures' Use of EMSA-PKCS1-v1_5 Padding" registry (Table 24).

16. References

16.1. Normative References

[AES]

NIST, "Advanced Encryption Standard (AES)", Updated May 2023, FIPS PUB 197, DOI 10.6028/NIST.FIPS.197-upd1, November 2001, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.197-upd1.pdf>.

[BLOWFISH] Schneier, B., "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)", Fast Software Encryption, Cambridge Security Workshop Proceedings, pp. 191-204, December 1993, <https://www.schneier.com/academic/archives/1994/09/description_of_a_new.html>.

[BZ2] bzip2, "bzip2", 2010, <http://www.bzip.org/>.

[EAX] Bellare, M., Rogaway, P., and D. Wagner, "A Conventional Authenticated-Encryption Mode", April 2003, <https://seclab.cs.ucdavis.edu/papers/eax.pdf>.

[ELGAMAL] Elgamal, T., "A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms", IEEE Transactions on Information Theory, Vol. 31, Issue 4, pp. 469-472, DOI 10.1109/TIT.1985.1057074, July 1985, <https://doi.org/10.1109/TIT.1985.1057074>.

[FIPS180] NIST, "Secure Hash Standard (SHS)", FIPS PUB 180-4, DOI 10.6028/NIST.FIPS.180-4, August 2015, <https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.180-4.pdf>.

[FIPS186] NIST, "Digital Signature Standard (DSS)", FIPS PUB 186-5, DOI 10.6028/NIST.FIPS.186-5, February 2023, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-5.pdf>.

[FIPS202] NIST, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions", FIPS PUB 202, DOI 10.6028/NIST.FIPS.202, August 2015, <https://nvlpubs.nist.gov/nistpubs/fips/nist.fips.202.pdf>.

[IDEA] Lai, X., "On the Design and Security of Block Ciphers", ETH Series in Information Processing, Vol. 1, Hartung-Gorre Verlag Konstanz, Technische Hochschule (Zurich), Dissertation, January 1992.

[ISO10646] ISO, "Information technology - Universal coded character set (UCS)", ISO/IEC 10646:2020, December 2020, <https://www.iso.org/standard/76835.html>.

[JFIF] ITU-T, "Information technology - Digital compression and coding of continuous-tone still images: JPEG File Interchange Format (JFIF)", Recommendation ITU-T T.871, May 2011, <https://www.itu.int/rec/T-REC-T.871-201105-I>.

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, DOI 10.17487/RFC1321, April 1992, <https://www.rfc-editor.org/info/rfc1321>.

[RFC1950] Deutsch, P. and J. Gailly, "ZLIB Compressed Data Format Specification version 3.3", RFC 1950, DOI 10.17487/RFC1950, May 1996, <https://www.rfc-editor.org/info/rfc1950>.

[RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, DOI 10.17487/RFC1951, May 1996, <https://www.rfc-editor.org/info/rfc1951>.

[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>.

[RFC2144] Adams, C., "The CAST-128 Encryption Algorithm", RFC 2144, DOI 10.17487/RFC2144, May 1997, <https://www.rfc-editor.org/info/rfc2144>.

[RFC2822] Resnick, P., Ed., "Internet Message Format", RFC 2822, DOI 10.17487/RFC2822, April 2001, <https://www.rfc-editor.org/info/rfc2822>.

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

[RFC3156] Elkins, M., Del Torto, D., Levien, R., and T. Roessler, "MIME Security with OpenPGP", RFC 3156, DOI 10.17487/RFC3156, August 2001, <https://www.rfc-editor.org/info/rfc3156>.

[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>.

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <https://www.rfc-editor.org/info/rfc3629>.

[RFC3713] Matsui, M., Nakajima, J., and S. Moriai, "A Description of the Camellia Encryption Algorithm", RFC 3713, DOI 10.17487/RFC3713, April 2004, <https://www.rfc-editor.org/info/rfc3713>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC4648] Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, DOI 10.17487/RFC4648, October 2006, <https://www.rfc-editor.org/info/rfc4648>.

[RFC5322] Resnick, P., Ed., "Internet Message Format", RFC 5322, DOI 10.17487/RFC5322, October 2008, <https://www.rfc-editor.org/info/rfc5322>.

[RFC6234] Eastlake 3rd, D. and T. Hansen, "US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF)", RFC 6234, DOI 10.17487/RFC6234, May 2011, <https://www.rfc-editor.org/info/rfc6234>.

[RFC7253] Krovetz, T. and P. Rogaway, "The OCB Authenticated-Encryption Algorithm", RFC 7253, DOI 10.17487/RFC7253, May 2014, <https://www.rfc-editor.org/info/rfc7253>.

[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>.

[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>.

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

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

[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>.

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

[RIPEMD-160] ISO, "Information technology - Security techniques - Hash-functions - Part 3: Dedicated hash-functions", ISO/IEC 10118-3, 1998.

[SP800-38A] NIST, "Recommendation for Block Cipher Modes of Operation: Methods and Techniques", NIST Special Publication 800-38A, DOI 10.6028/NIST.SP.800-38A, December 2001, <https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38a.pdf>.

[SP800-38D] NIST, "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST Special Publication 800-38D, DOI 10.6028/NIST.SP.800-38D, November 2007, <https://nvlpubs.nist.gov/nistpubs/legacy/sp/nistspecialpublication800-38d.pdf>.

[SP800-56A] NIST, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", NIST Special Publication 800-56A Revision 1, DOI 10.6028/NIST.SP.800-56Ar, March 2007, <https://doi.org/10.6028/NIST.SP.800-56Ar>.

[SP800-67] NIST, "Recommendation for the Triple Data Encryption Algorithm (TDEA) Block Cipher", NIST Special Publication 800-67 Rev. 2, DOI 10.6028/NIST.SP.800-67r2, November 2017, <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-67r2.pdf>.

[TWOFISH] Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., and N. Ferguson, "Twofish: A 128-Bit Block Cipher", June 1998, <https://www.schneier.com/wp-content/uploads/2016/02/paper-twofish-paper.pdf>.

16.2. Informative References

[BLEICHENBACHER]

Bleichenbacher, D., "Generating ElGamal Signatures Without Knowing the Secret Key", EUROCRYPT'96: International Conference on the Theory and Applications of Cryptographic Techniques Proceedings, Vol. 1070, pp. 10-18, May 1996.

[BLEICHENBACHER-PKCS1] Bleichenbacher, D., "Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1", CRYPTO '98: International Cryptology Conference Proceedings, Vol. 1462, pp. 1-12, August 1998, <http://archiv.infsec.ethz.ch/education/fs08/secsem/Bleichenbacher98.pdf>.

[C99] ISO, "Programming languages - C: Technical Corrigendum 3", ISO/IEC 9899:1999/Cor3:2007, November 2007, <https://www.iso.org/standard/50510.html>.

[CHECKOWAY]
Checkoway, S., Maskiewicz, J., Garman, C., Fried, J., Cohney, S., Green, M., Heninger, N., Weinmann, RP., Rescorla, E., and H. Shacham, "A Systematic Analysis of the Juniper Dual EC Incident", Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, DOI 10.1145/2976749.2978395, October 2016, <https://doi.org/10.1145/2976749.2978395>.

[EFAIL] Poddebniak, D., Dresen, C., Müller, J., Ising, F., Schinzel, S., Friedberger, S., Somorovsky, J., and J. Schwenk, "Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels", Proceedings of the 27th USENIX Security Symposium, August 2018, <https://www.usenix.org/system/files/conference/usenixsecurity18/sec18-poddebniak.pdf>.

[Errata-2199] RFC Errata, Erratum ID 2199, RFC 4880, <https://www.rfc-editor.org/errata/eid2199>.

[Errata-2200] RFC Errata, Erratum ID 2200, RFC 4880, <https://www.rfc-editor.org/errata/eid2200>.

[Errata-2206] RFC Errata, Erratum ID 2206, RFC 4880, <https://www.rfc-editor.org/errata/eid2206>.

[Errata-2208] RFC Errata, Erratum ID 2208, RFC 4880, <https://www.rfc-editor.org/errata/eid2208>.

[Errata-2214] RFC Errata, Erratum ID 2214, RFC 4880, <https://www.rfc-editor.org/errata/eid2214>.

[Errata-2216] RFC Errata, Erratum ID 2216, RFC 4880, <https://www.rfc-editor.org/errata/eid2216>.

[Errata-2219] RFC Errata, Erratum ID 2219, RFC 4880, <https://www.rfc-editor.org/errata/eid2219>.

[Errata-2222] RFC Errata, Erratum ID 2222, RFC 4880, <https://www.rfc-editor.org/errata/eid2222>.

[Errata-2226] RFC Errata, Erratum ID 2226, RFC 4880, <https://www.rfc-editor.org/errata/eid2226>.

[Errata-2234] RFC Errata, Erratum ID 2234, RFC 4880, <https://www.rfc-editor.org/errata/eid2234>.

[Errata-2235] RFC Errata, Erratum ID 2235, RFC 4880, <https://www.rfc-editor.org/errata/eid2235>.

[Errata-2236] RFC Errata, Erratum ID 2236, RFC 4880, <https://www.rfc-editor.org/errata/eid2236>.

[Errata-2238] RFC Errata, Erratum ID 2238, RFC 4880, <https://www.rfc-editor.org/errata/eid2238>.

[Errata-2240] RFC Errata, Erratum ID 2240, RFC 4880, <https://www.rfc-editor.org/errata/eid2240>.

[Errata-2242] RFC Errata, Erratum ID 2242, RFC 4880, <https://www.rfc-editor.org/errata/eid2242>.

[Errata-2243] RFC Errata, Erratum ID 2243, RFC 4880, <https://www.rfc-editor.org/errata/eid2243>.

[Errata-2270] RFC Errata, Erratum ID 2270, RFC 4880, <https://www.rfc-editor.org/errata/eid2270>.

[Errata-2271] RFC Errata, Erratum ID 2271, RFC 4880, <https://www.rfc-editor.org/errata/eid2271>.

[Errata-3298] RFC Errata, Erratum ID 3298, RFC 4880, <https://www.rfc-editor.org/errata/eid3298>.

[Errata-5491] RFC Errata, Erratum ID 5491, RFC 4880, <https://www.rfc-editor.org/errata/eid5491>.

[Errata-7545] RFC Errata, Erratum ID 7545, RFC 4880, <https://www.rfc-editor.org/errata/eid7545>.

[Errata-7889] RFC Errata, Erratum ID 7889, RFC 4880, <https://www.rfc-editor.org/errata/eid7889>.

[HASTAD] Hastad, J., "Solving Simultaneous Modular Equations of Low Degree", DOI 10.1137/0217019, April 1988, <https://doi.org/10.1137/0217019>.

[JKS02] Jallad, K., Katz, J., and B. Schneier, "Implementation of Chosen-Ciphertext Attacks against PGP and GnuPG", DOI 0.1007/3-540-45811-5_7, September 2002, <https://www.schneier.com/academic/archives/2002/01/implementation_of_ch.html>.

[KOBLITZ] Koblitz, N., "A course in number theory and cryptography", Chaper VI: Elliptic Curves, DOI 10.2307/3618498, 1997, <https://doi.org/10.2307/3618498>.

[KOPENPGP] Bruseghini, L., Paterson, K. G., and D. Huigens, "Victory by KO: Attacking OpenPGP Using Key Overwriting", Proceedings of the ACM SIGSAC Conference on Computer and Communications Security, pp. 411-423, DOI 10.1145/3548606.3559363, November 2022, <https://dl.acm.org/doi/10.1145/3548606.3559363>.

[KR02] Klíma, V. and T. Rosa, "Attack on Private Signature Keys of the OpenPGP Format, PGP(TM) Programs and Other Applications Compatible with OpenPGP", Cryptology ePrint Archive, Paper 2002/076, March 2001, <https://eprint.iacr.org/2002/076>.

[MRLG15] Maury, F., Reinhard, JR., Levillain, O., and H. Gilbert, "Format Oracles on OpenPGP", Topics in Cryptology -- CT-RSA 2015, Vol. 9048, pp. 220-236, DOI 10.1007/978-3-319-16715-2_12, January 2015, <https://doi.org/10.1007/978-3-319-16715-2_12>.

[MZ05] Mister, S. and R. Zuccherato, "An Attack on CFB Mode Encryption As Used By OpenPGP", Cryptology ePrint Archive, Paper 2005/033, February 2005, <http://eprint.iacr.org/2005/033>.

[OPENPGPCARD] Pietig, A., "Functional Specification of the OpenPGP application on ISO Smart Card Operating Systems", Version 3.4.1, March 2020, <https://gnupg.org/ftp/specs/OpenPGP-smart-card-application-3.4.1.pdf>.

[PAX] The Open Group, "IEEE Standard for Information Technology-Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7: pax - portable archive interchange", IEEE Standard 1003.1-2017, DOI 10.1109/IEEESTD.2018.8277153, 2018, <https://pubs.opengroup.org/onlinepubs/9699919799/utilities/pax.html>.

[PSSLR17] Poddebniak, D., Somorovsky, J., Schinzel, S., Lochter, M., and P. Rösler, "Attacking Deterministic Signature Schemes using Fault Attacks", Cryptology ePrint Archive, Paper 2017/1014 , October 2017, <https://eprint.iacr.org/2017/1014>.

[REGEX] Friedl, J. E., "Mastering Regular Expressions", ISBN 0-596-00289-0, July 2002.

[RFC1991] Atkins, D., Stallings, W., and P. Zimmermann, "PGP Message Exchange Formats", RFC 1991, DOI 10.17487/RFC1991, August 1996, <https://www.rfc-editor.org/info/rfc1991>.

[RFC2440] Callas, J., Donnerhacke, L., Finney, H., and R. Thayer, "OpenPGP Message Format", RFC 2440, DOI 10.17487/RFC2440, November 1998, <https://www.rfc-editor.org/info/rfc2440>.

[RFC2978] Freed, N. and J. Postel, "IANA Charset Registration Procedures", BCP 19, RFC 2978, DOI 10.17487/RFC2978, October 2000, <https://www.rfc-editor.org/info/rfc2978>.

[RFC4880] Callas, J., Donnerhacke, L., Finney, H., Shaw, D., and R. Thayer, "OpenPGP Message Format", RFC 4880, DOI 10.17487/RFC4880, November 2007, <https://www.rfc-editor.org/info/rfc4880>.

[RFC5581] Shaw, D., "The Camellia Cipher in OpenPGP", RFC 5581, DOI 10.17487/RFC5581, June 2009, <https://www.rfc-editor.org/info/rfc5581>.

[RFC5639] Lochter, M. and J. Merkle, "Elliptic Curve Cryptography (ECC) Brainpool Standard Curves and Curve Generation", RFC 5639, DOI 10.17487/RFC5639, March 2010, <https://www.rfc-editor.org/info/rfc5639>.

[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>.

[RFC6637] Jivsov, A., "Elliptic Curve Cryptography (ECC) in OpenPGP", RFC 6637, DOI 10.17487/RFC6637, June 2012, <https://www.rfc-editor.org/info/rfc6637>.

[SEC1] Standards for Efficient Cryptography Group, "SEC 1: Elliptic Curve Cryptography", , May 2009, <https://www.secg.org/sec1-v2.pdf>.

[SHA1CD] "sha1collisiondetection", commit b4a7b0b, December 2020, <https://github.com/cr-marcstevens/sha1collisiondetection>.

[SHAMBLES] Leurent, G. and T. Peyrin, "Sha-1 is a shambles: first chosen-prefix collision on sha-1 and application to the PGP web of trust", August 2020, <https://shambles.github.io/>.

[SP800-131A] NIST, "Transitioning the Use of Cryptographic Algorithms and Key Lengths", NIST Special Publication 800-131A, Revision 2, DOI 10.6028/NIST.SP.800-131Ar2, March 2019, <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-131Ar2.pdf>.

[SP800-57] NIST, "Recommendation for Key Management: Part 1 - General", NIST Special Publication 800-57 Part 1, Revision 5, DOI 10.6028/NIST.SP.800-57pt1r5, May 2020, <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-57pt1r5.pdf>.

[STEVENS2013] Stevens, M., "Counter-cryptanalysis", Cryptology ePrint Archive, Paper 2013/358, June 2013, <https://eprint.iacr.org/2013/358>.

[UNIFIED-DIFF] Free Software Foundation, "Comparing and Merging Files", 'Detailed Description of Unified Format', Section 2.2.2.2, January 2021, <https://www.gnu.org/software/diffutils/manual/html_node/Detailed-Unified.html>.

[USENIX-STUDY] Dechand, S., Schürmann, D., Busse, K., Acar, Y., Fahl, S., and M. Smith, "An Empirical Study of Textual Key-Fingerprint Representations", ISBN 978-1-931971-32-4, August 2016, <https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_dechand.pdf>.

Appendix A. Test Vectors

To help with the implementation of this specification, a set of non-normative examples follow.

A.1. Sample v4 Ed25519Legacy Key

The secret key used for this example is:

```
D: 1a8b1ff05ded48e18bf50166c664ab023ea70003d78d9e41f5758a91d850f8d2
```

Note that this is the raw secret key used as input to the EdDSA signing operation. The key was created on 2014-08-19 14:28:27 and thus the fingerprint of the OpenPGP key is:

```
   C959 BDBA FA32 A2F8 9A15  3B67 8CFD E121 9796 5A9A
```

The algorithm-specific input parameters without the MPI length headers are:

```
oid: 2b06010401da470f01

q: 403f098994bdd916ed4053197934e4a87c80733a1280d62f8010992e43ee3b2406
```

The entire public key packet is thus:

```
   98 33 04 53 f3 5f 0b 16  09 2b 06 01 04 01 da 47
   0f 01 01 07 40 3f 09 89  94 bd d9 16 ed 40 53 19
   79 34 e4 a8 7c 80 73 3a  12 80 d6 2f 80 10 99 2e
   43 ee 3b 24 06
```

The same packet represented in ASCII-armored form is:

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

xjMEU/NfCxYJKwYBBAHaRw8BAQdAPwmJlL3ZFu1AUxl5NOSofIBzOhKA1i+AEJku
Q+47JAY=
-----END PGP PUBLIC KEY BLOCK-----
```

A.2. Sample v4 Ed25519Legacy Signature

The signature is created using the sample key over the input data "OpenPGP" on 2015-09-16 12:24:53 UTC and thus the input to the hash function is:

```
m: 4f70656e50475004001608000605025 5f95f9504ff0000000c
```

Using the SHA2-256 hash algorithm yields the digest:

```
d: f6220a3f757814f4c2176ffbb68b00249cd4ccdc059c4b34ad871f30b1740280
```

which is fed into the EdDSA signature function and yields the following signature:

```
r: 56f90cca98e2102637bd983fdb16c131dfd27ed82bf4dde5606e0d756aed3366

s: d09c4fa11527f038e0f57f2201d82f2ea2c9033265fa6ceb489e854bae61b404
```

The entire signature packet is thus:

```
   88 5e 04 00 16 08 00 06  05 02 55 f9 5f 95 00 0a
   09 10 8c fd e1 21 97 96  5a 9a f6 22 00 ff 56 f9
   0c ca 98 e2 10 26 37 bd  98 3f db 16 c1 31 df d2
   7e d8 2b f4 dd e5 60 6e  0d 75 6a ed 33 66 01 00
   d0 9c 4f a1 15 27 f0 38  e0 f5 7f 22 01 d8 2f 2e
   a2 c9 03 32 65 fa 6c eb  48 9e 85 4b ae 61 b4 04
```

The same packet represented in ASCII-armored form is:

```
-----BEGIN PGP SIGNATURE-----

iF4EABYIAAYFAlX5X5UACgkQjP3hIZeWWpr2IgD/VvkMypjiECY3vZg/2xbBMd/S
ftgr9N3lYG4NdWrtM2YBANCcT6EVJ/A44PV/IgHYLy6iyQMyZfps60iehUuuYbQE
-----END PGP SIGNATURE-----
```

A.3. Sample v6 Certificate (Transferable Public Key)

Here is a Transferable Public Key consisting of:

- A v6 Ed25519 Public-Key packet
- A v6 direct key self-signature
- A v6 X25519 Public-Subkey packet
- A v6 subkey binding signature

The primary key has the following fingerprint:

```
CB186C4F0609A697E4D52DFA6C722B0C1F1E27C18A56708F6525EC27BAD9ACC9
```

The subkey has the following fingerprint:

```
12C83F1E706F6308FE151A417743A1F033790E93E9978488D1DB378DA9930885
```

```
-----BEGIN PGP PUBLIC KEY BLOCK-----

xioGY4d/4xsAAAAg+U2nu0jWCmHlZ3BqZYfQMxmZu52JGggkLq2EVD34laPCsQYf
GwoAAABCBYJjh3/jAwsJBwUVCg4IDAIWAAKbAwIeCSIhBssYbE8GCaaX5NUt+mxy
KwwfHifBilZwj2Ul7Ce62azJBScJAgcCAAAAAK0oIBA+LX0ifsDm185Ecds2v8lw
gyU2kCcUmKfvBXbAf6rhRYWzuQOwEn7E/aLwIwRaLsdry0+VcallHhSu4RN6HWaE
QsiPlR4zxP/TP7mhfVEe7XWPxtnMUMtf15OyA51YBM4qBmOHf+MZAAAAIIaTJINn
+eUBXbki+PSAld2nhJh/LVmFsS+60WyvXkQ1wpsGGBsKAAAALAWCY4d/4wKbDCIh
BssYbE8GCaaX5NUt+mxyKwwfHifBilZwj2Ul7Ce62azJAAAAAAQBIKbpGG2dWTX8
j+VjFM21J0hqWlEg+bdiojWnKfA5AQpWUWtnNwDEM0g12vYxoWM8Y81W+bHBw805
I8kWVkXU6vFOi+HWvv/ira7ofJu16NnoUkhclkUrk0mXubZvyl4GBg==
-----END PGP PUBLIC KEY BLOCK-----
```

The corresponding Transferable Secret Key can be found in Appendix A.4.

A.3.1. Hashed Data Stream for Signature Verification

The direct key self-signature in the certificate in Appendix A.3 is made over the following sequence of data:

```
0x0000  10 3e 2d 7d 22 7e c0 e6
0x0008  d7 ce 44 71 db 36 bf c9
0x0010  70 83 25 36 90 27 14 98
0x0018  a7 ef 05 76 c0 7f aa e1
0x0020  9b 00 00 00 2a 06 63 87
0x0028  7f e3 1b 00 00 00 20 f9
0x0030  4d a7 bb 48 d6 0a 61 e5
0x0038  67 70 6a 65 87 d0 33 19
0x0040  99 bb 9d 89 1a 08 24 2e
0x0048  ad 84 54 3d f8 95 a3 06
0x0050  1f 1b 0a 00 00 00 42 05
0x0058  82 63 87 7f e3 03 0b 09
0x0060  07 05 15 0a 0e 08 0c 02
0x0068  16 00 02 9b 03 02 1e 09
0x0070  22 21 06 cb 18 6c 4f 06
0x0078  09 a6 97 e4 d5 2d fa 6c
0x0080  72 2b 0c 1f 1e 27 c1 8a
0x0088  56 70 8f 65 25 ec 27 ba
0x0090  d9 ac c9 05 27 09 02 07
0x0098  02 06 ff 00 00 00 4a
```

The same data, broken out by octet and semantics, is:

```
0x0000  10 3e 2d 7d 22 7e c0 e6  salt
0x0008  d7 ce 44 71 db 36 bf c9
0x0010  70 83 25 36 90 27 14 98
0x0018  a7 ef 05 76 c0 7f aa e1
        [ pubkey begins ]
0x0020  9b                       v6 pubkey
0x0021     00 00 00 2a           pubkey length
0x0025                 06        pubkey version
0x0026                    63 87  creation time
0x0028  7f e3                      (2022-11-30T16:08:03Z)
0x002a        1b                 key algo: Ed25519
0x002b           00 00 00 20     key length
0x002f                       f9  Ed25519 public key
0x0030  4d a7 bb 48 d6 0a 61 e5
0x0038  67 70 6a 65 87 d0 33 19
0x0040  99 bb 9d 89 1a 08 24 2e
0x0048  ad 84 54 3d f8 95 a3
         [ trailer begins ]
0x004f                       06  sig version
0x0050  1f                       sig type: direct key signature
0x0051     1b                    sig algo: Ed25519
0x0052        0a                 hash ago: SHA2-512
0x0053           00 00 00 42     hashed subpackets length
0x0057                       05  subpkt length
0x0058  82                       critical subpkt: Sig Creation Time
0x0059     63 87 7f e3           Signature Creation Time
0x005d                 03        subpkt length
0x005e                    0b     subpkt type: Pref. v1 SEIPD Ciphers
0x005f                       09  Ciphers: [AES256 AES128]
0x0060  07
0x0061     05                    subpkt length
0x0062        15                 subpkt type: Pref. Hash Algorithms
0x0063           0a 0e           Hashes: [SHA2-512 SHA3-512
0x0065                 08 0c               SHA2-256 SHA3-256]
0x0067                       02  subpkt length
0x0068  16                       subpkt type: Pref. Compression
0x0069     00                    Compression: [none]
0x006a        02                 subpkt length
0x006b           9b              critical subpkt: Key Flags
0x006c              03           Key Flags: {certify, sign}
0x006d                 02        subpkt length
0x006e                    1e     subpkt type: Features
0x006f                       09  Features: {SEIPDv1, SEIPDv2}
0x0070  22                       subpkt length
0x0071     21                    subpkt type: Issuer Fingerprint
0x0072        06                 Fingerprint version 6
0x0073           cb 18 6c 4f 06  Issuer Fingerprint
0x0078  09 a6 97 e4 d5 2d fa 6c
0x0080  72 2b 0c 1f 1e 27 c1 8a
0x0088  56 70 8f 65 25 ec 27 ba
0x0090  d9 ac c9
0x0093           05              subpkt length
0x0094              27           subpkt type: Pref. AEAD Ciphersuites
0x0095                 09 02 07  Ciphersuites:
0x0098  02                         [ AES256-OCB, AES128-OCB ]
0x0099     06                    sig version
```

```
0x009a        ff                  sentinel octet
0x009b           00 00 00 4a      trailer length
```

The subkey binding signature in Appendix A.3 is made over the following sequence of data:

```
0x0000  a6 e9 18 6d 9d 59 35 fc
0x0008  8f e5 63 14 cd b5 27 48
0x0010  6a 5a 51 20 f9 b7 62 a2
0x0018  35 a7 29 f0 39 01 0a 56
0x0020  9b 00 00 00 2a 06 63 87
0x0028  7f e3 1b 00 00 00 20 f9
0x0030  4d a7 bb 48 d6 0a 61 e5
0x0038  67 70 6a 65 87 d0 33 19
0x0040  99 bb 9d 89 1a 08 24 2e
0x0048  ad 84 54 3d f8 95 a3 9b
0x0050  00 00 00 2a 06 63 87 7f
0x0058  e3 19 00 00 00 20 86 93
0x0060  24 83 67 f9 e5 01 5d b9
0x0068  22 f8 f4 80 95 dd a7 84
0x0070  98 7f 2d 59 85 b1 2f ba
0x0078  d1 6c af 5e 44 35 06 18
0x0080  1b 0a 00 00 00 2c 05 82
0x0088  63 87 7f e3 02 9b 0c 22
0x0090  21 06 cb 18 6c 4f 06 09
0x0098  a6 97 e4 d5 2d fa 6c 72
0x00a0  2b 0c 1f 1e 27 c1 8a 56
0x00a8  70 8f 65 25 ec 27 ba d9
0x00b0  ac c9 06 ff 00 00 00 34
```

The same data, broken out by octet and semantics, is:

```
0x0000  a6 e9 18 6d 9d 59 35 fc  salt
0x0008  8f e5 63 14 cd b5 27 48
0x0010  6a 5a 51 20 f9 b7 62 a2
0x0018  35 a7 29 f0 39 01 0a 56
      [ primary pubkey begins ]
0x0020  9b                       v6 pubkey
0x0021     00 00 00 2a           pubkey length
0x0025                 06        pubkey version
0x0026                    63 87  creation time
0x0028  7f e3                      (2022-11-30T16:08:03Z)
0x002a        1b                 key algo: Ed25519
0x002b           00 00 00 20     key length
0x002f                       f9  Ed25519 public key
0x0030  4d a7 bb 48 d6 0a 61 e5
0x0038  67 70 6a 65 87 d0 33 19
0x0040  99 bb 9d 89 1a 08 24 2e
0x0048  ad 84 54 3d f8 95 a3
      [ subkey pubkey begins ]
0x004f                       9b  v6 key
0x0050  00 00 00 2a              pubkey length
0x0054              06           pubkey version
0x0055                 63 87 7f  creation time (2022-11-30T16:08:03Z)
0x0058  e3
0x0059     19                    key algo: X25519
0x005a        00 00 00 20        key length
0x005e                    86 93  X25519 public key
0x0060  24 83 67 f9 e5 01 5d b9
0x0068  22 f8 f4 80 95 dd a7 84
0x0070  98 7f 2d 59 85 b1 2f ba
0x0078  d1 6c af 5e 44 35
       [ trailer begins ]
0x007e                    06     sig version
0x007f                       18  sig type: Subkey Binding sig
0x0080  1b                       sig algo Ed25519
0x0081     0a                    hash algo: SHA2-512
0x0082        00 00 00 2c        hashed subpackets length
0x0086                    05     subpkt length
0x0087                       82  critical subpkt: Sig Creation Time
0x0088  63 87 7f e3              Signature Creation Time
0x008c              02           subpkt length
0x008d                 9b        critical subpkt: Key Flags
0x008e                    0c     Key Flags: {EncComms, EncStorage}
0x008f                       22  subpkt length
0x0090  21                       subpkt type: Issuer Fingerprint
0x0091     06                    Fingerprint version 6
0x0092        cb 18 6c 4f 06 09  Fingerprint
0x0098  a6 97 e4 d5 2d fa 6c 72
0x00a0  2b 0c 1f 1e 27 c1 8a 56
0x00a8  70 8f 65 25 ec 27 ba d9
0x00b0  ac c9
0x00b2        06                 sig version
0x00b3           ff              sentinel octet
0x00b4              00 00 00 34  trailer length
```

A.4. Sample v6 Secret Key (Transferable Secret Key)

Here is a Transferable Secret Key consisting of:

- A v6 Ed25519 Secret-Key packet
- A v6 direct key self-signature
- A v6 X25519 Secret-Subkey packet
- A v6 subkey binding signature

```
-----BEGIN PGP PRIVATE KEY BLOCK-----

xUsGY4d/4xsAAAAg+U2nu0jWCmHlZ3BqZYfQMxmZu52JGggkLq2EVD34laMAGXKB
exK+cH6NX1hs5hNhIB00TrJmosgv3mg1ditlsLfCsQYfGwoAAABCBYJjh3/jAwsJ
BwUVCg4IDAIWAAKbAwIeCSIhBssYbE8GCaaX5NUt+mxyKwwfHifBilZwj2Ul7Ce6
2azJBScJAgcCAAAAK0oIBA+LX0ifsDm185Ecds2v8lwgyU2kCcUmKfvBXbAf6rh
RYWzuQOwEn7E/aLwIwRaLsdry0+VcallHhSu4RN6HWaEQsiPlR4zxP/TP7mhfVEe
7XWPxtnMUMtf15OyA51YBMdLBmOHf+MZAAAAIIaTJINn+eUBXbki+PSAld2nhJh/
LVmFsS+60WyvXkQ1AE1gCk95TUR3XFeibg/u/tVY6a//1q0NWC1X+yui3O24wpsG
GBsKAAAALAWCY4d/4wKbDCIhBssYbE8GCaaX5NUt+mxyKwwfHifBilZwj2Ul7Ce6
2azJAAAAAAQBIKbpGG2dWTX8j+VjFM21J0hqWlEg+bdiojWnKfA5AQpWUWtnNwDE
M0g12vYxoWM8Y81W+bHBw805I8kWVkXU6vFOi+HWvv/ira7ofJu16NnoUkhclkUr
k0mXubZvyl4GBg==
-----END PGP PRIVATE KEY BLOCK-----
```

The corresponding Transferable Public Key can be found in Appendix A.3.

A.5. Sample Locked v6 Secret Key (Transferable Secret Key)

Here is the same secret key as in Appendix A.4, but the secret key material is locked with a passphrase using AEAD and Argon2.

The passphrase is the ASCII string:

```
correct horse battery staple
```

```
-----BEGIN PGP PRIVATE KEY BLOCK-----

xYIGY4d/4xsAAAAg+U2nu0jWCmHlZ3BqZYfQMxmZu52JGggkLq2EVD34laP9JgkC
FARdb9ccngltHraRe25uHuyuAQQVtKipJ0+r5jL4dacGWSAheCWPpITYiyfyIOPS
3gIDyg8f7strd1OB4+LZsUhcIjOMpVHgmiY/IutJkulneoBYwrEGHxsKAAAAQgWC
Y4d/4wMLCQcFFQoOCAwCFgACmwMCHgkiIQbLGGxPBgmml+TVLfpscisMHx4nwYpW
cI9lJewnutmsyQUnCQIHAgAAAACtKCAQPi19In7A5tfORHHbNr/JcIMlNpAnFJin
7wV2wH+q4UWFs7kDsBJ+xP2i8CMEWi7Ha8tPlXGpZR4UruETeh1mhELIj5UeM8T/
0z+5oX1RHu11j8bZzFDLX9eTsgOdWATHggZjh3/jGQAAACCGkySDZ/nlAV25Ivj0
gJXdp4SYfy1ZhbEvutFsr15ENf0mCQIUBA5hhGgp2oaavg6mFUXcFMwBBBUuE8qf
9Ock+xwusd+GAglBr5LVyr/lup3xxQvHXFSjjA2haXfoN6xUGRdDEHI6+uevKjVR
v5oAxgu7eJpaXNjCmwYYGwoAAAAsBYJjh3/jApsMIiEGyxhsTwYJppfk1S36bHIr
DB8eJ8GKVnCPZSXsJ7rZrMkAAAAABAEgpukYbZ1ZNfyP5WMUzbUnSGpaUSD5t2Ki
Nacp8DkBClZRa2c3AMQzSDXa9jGhYzxjzVb5scHDzTkjyRZWRdTq8U6L4da+/+Kt
ruh8m7Xo2ehSSFyWRSuTSZe5tm/KXgYG
-----END PGP PRIVATE KEY BLOCK-----
```

A.5.1. Intermediate Data for Locked Primary Key

The S2K-derived material for the primary key is:

```
832bd2662a5c2b251ee3fc82aec349a766ca539015880133002e5a21960b3bcf
```

After HKDF, the symmetric key used for AEAD encryption of the primary key is:

```
9e37cb26787f37e18db172795c4c297550d39ac82511d9af4c8706db6a77fd51
```

The additional data for AEAD for the primary key is:

```
c50663877fe31b00000020f94da7bb48d60a61e567706a6587d0331999bb9d89
1a08242ead84543df895a3
```

A.5.2. Intermediate Data for Locked Subkey

The S2K-derived key material for the subkey is:

```
f74a6ce873a089ef13a3da9ac059777bb22340d15eaa6c9dc0f8ef09035c67cd
```

After HKDF, the symmetric key used for AEAD encryption of the subkey is:

```
3c60cb63285f62f4c3de49835786f011cf6f4c069f61232cd7013ff5fd31e603
```

The additional data for AEAD for the subkey is:

```
c70663877fe319000000208693248367f9e5015db922f8f48095dda784987f2d
5985b12fbad16caf5e4435
```

A.6. Sample Cleartext Signed Message

Here is a signed message that uses the Cleartext Signature Framework (Section 7). It can be verified with the certificate from Appendix A.3.

Note that this message makes use of dash-escaping (Section 7.2) due to its contents.

```
-----BEGIN PGP SIGNED MESSAGE-----

What we need from the grocery store:

- - tofu
- - vegetables
- - noodles

-----BEGIN PGP SIGNATURE-----

wpgGARsKAAAAKQWCY5ijYyIhBssYbE8GCaaX5NUt+mxyKwwfHifBilZwj2Ul7Ce6
2azJAAAAAGk2IHZJX1AhiJD39eLuPBgiUU9wUA9VHYblySHkBONKU/usJ9BvuAqo
/FvLFuGWMbKAdA+epq7V4HOtAPlBWmU8QOd6aud+aSunHQaaEJ+iTFjP2OMW0KBr
NK2ay45cX1IVAQ==
-----END PGP SIGNATURE-----
```

The signature packet here is:

```
0x0000  c2                        packet type: Signature
0x0001     98                     packet length
0x0002        06                  sig version 6
0x0003           01               sig type: canonical text
0x0004              1b            pubkey algorithm: Ed25519
0x0005                 0a         hash algorithm used: SHA2-512
0x0006                    00 00   hashed subpackets length: 41
0x0008  00 29
0x000a        05                  subpkt length
0x000b           82               critical subpkt: Sig Creation Time
0x000c              63 98 a3 63    (2022-12-13T16:08:03Z)
0x0010  22                        subpkt length
0x0011     21                     subpkt type: issuer fingerprint
0x0012        06                  key version
0x0013           cb 18 6c 4f 06   v6 fingerprint
0x001a  09 a6 97 e4 d5 2d fa 6c
0x0020  72 2b 0c 1f 1e 27 c1 8a
0x0028  56 70 8f 65 25 ec 27 ba
0x0030  d9 ac c9
0x0033           00 00 00 00      unhashed subpackets length: 0
0x0037                       69   left 16 bits of signed hash
0x0038  36
0x0039     20                     salt length
0x003a        76 49 5f 50 21 88   salt
0x0040  90 f7 f5 e2 ee 3c 18 22
0x0048  51 4f 70 50 0f 55 1d 86
0x0050  e5 c9 21 e4 04 e3 4a 53
0x0058  fb ac
0x005a        27 d0 6f b8 0a a8   Ed25519 signature
0x0060  fc 5b cb 16 e1 96 31 b2
0x0068  80 74 0f 9e a6 ae d5 e0
0x0070  73 ad 00 f9 41 5a 65 3c
0x0078  40 e7 7a 6a e7 7e 69 2b
0x0080  a7 1d 06 9a 10 9f a2 4c
0x0088  58 cf d8 e3 16 d0 a0 6b
0x0090  34 ad 9a cb 8e 5c 5f 52
0x0098  15 01
```

The signature is made over the following data:

```
0x0000  76 49 5f 50 21 88 90 f7
0x0008  f5 e2 ee 3c 18 22 51 4f
0x0010  70 50 0f 55 1d 86 e5 c9
0x0018  21 e4 04 e3 4a 53 fb ac
0x0020  57 68 61 74 20 77 65 20
0x0028  6e 65 65 64 20 66 72 6f
0x0030  6d 20 74 68 65 20 67 72
0x0038  6f 63 65 72 79 20 73 74
0x0040  6f 72 65 3a 0d 0a 0d 0a
0x0048  2d 20 74 6f 66 75 0d 0a
0x0050  2d 20 76 65 67 65 74 61
0x0058  62 6c 65 73 0d 0a 2d 20
0x0060  6e 6f 6f 64 6c 65 73 0d
0x0068  0a 06 01 1b 0a 00 00 00
0x0070  29 05 82 63 98 a3 63 22
0x0078  21 06 cb 18 6c 4f 06 09
0x0080  a6 97 e4 d5 2d fa 6c 72
0x0088  2b 0c 1f 1e 27 c1 8a 56
0x0090  70 8f 65 25 ec 27 ba d9
0x0098  ac c9 06 ff 00 00 00 31
```

The same data, broken out by octet and semantics, is:

```
0x0000  76 49 5f 50 21 88 90 f7  salt
0x0008  f5 e2 ee 3c 18 22 51 4f
0x0010  70 50 0f 55 1d 86 e5 c9
0x0018  21 e4 04 e3 4a 53 fb ac
      [ message begins ]
0x0020  57 68 61 74 20 77 65 20  canonicalized message
0x0028  6e 65 65 64 20 66 72 6f
0x0030  6d 20 74 68 65 20 67 72
0x0038  6f 63 65 72 79 20 73 74
0x0040  6f 72 65 3a 0d 0a 0d 0a
0x0048  2d 20 74 6f 66 75 0d 0a
0x0050  2d 20 76 65 67 65 74 61
0x0058  62 6c 65 73 0d 0a 2d 20
0x0060  6e 6f 6f 64 6c 65 73 0d
0x0068  0a
      [ trailer begins ]
0x0069     06                    sig version
0x006a        01                 sig type: canonical text
0x006b           1b              pubkey algorithm: Ed25519
0x006c              0a           hash algorithm: SHA2-512
0x006d                 00 00 00  hashed subpackets length
0x0070  29
0x0071     05                    subpacket length
0x0072        82                 critical subpkt: Sig Creation Time
0x0073           63 98 a3 63       (2022-12-13T16:08:03Z)
0x0077                       22  subpkt length
0x0078  21                       subpkt type: issuer fingerprint
0x0079     06                    key version
0x007a        cb 18 6c 4f 06 09  v6 fingerprint
0x0080  a6 97 e4 d5 2d fa 6c 72
0x0088  2b 0c 1f 1e 27 c1 8a 56
0x0090  70 8f 65 25 ec 27 ba d9
0x0098  ac c9
0x009a        06                 sig version
0x009b           ff              sentinel octet
0x009c              00 00 00 31  trailer length
```

The calculated SHA2-512 hash digest over this data is:

```
69365bf44a97af1f0844f1f6ab83fdf6b36f26692efaa621a8aac91c4e29ea07
e894cabc6e2f20eedfce6c03b89141a2cc7cbe245e6e7a5654addbec5000b89b
```

A.7. Sample Inline-Signed Message

This is the same message and signature as in Appendix A.6 but as an inline-signed message. The hashed data is exactly the same, and all intermediate values and annotated hex dumps are also applicable.

```
-----BEGIN PGP MESSAGE-----

xEYGAQobIHZJX1AhiJD39eLuPBgiUU9wUA9VHYblySHkBONKU/usyxhsTwYJppfk
1S36bHIrDB8eJ8GKVnCPZSXsJ7rZrMkBy0p1AAAAAABXaGF0IHdlIG5lZWQgZnJv
bSB0aGUgZ3JvY2VyeSBzdG9yZToKCi0gdG9mdQotIHZlZ2V0YWJsZXMKLSBub29k
bGVzCsKYBgEbCgAAACkFgmOYo2MiIQbLGGxPBgmml+TVLfpscisMHx4nwYpWcI9l
JewnutmsyQAAAABpNiB2SV9QIYiQ9/Xi7jwYIlFPcFAPVR2G5ckh5ATjSlP7rCfQ
b7gKqPxbyxbhljGygHQPnqau1eBzrQD5QVplPEDnemrnfmkrpx0GmhCfokxYz9jj
FtCgazStmsuOXF9SFQE=
-----END PGP MESSAGE-----
```

A.8. Sample X25519-AEAD-OCB Encryption and Decryption

This example encrypts the cleartext string `Hello, world!` for the sample cert (see Appendix A.3), using AES-128 with AEAD-OCB encryption.

A.8.1. Sample Public-Key Encrypted Session Key Packet (v6)

This packet contains the following series of octets:

```
0x0000  c1 5d 06 21 06 12 c8 3f
0x0008  1e 70 6f 63 08 fe 15 1a
0x0010  41 77 43 a1 f0 33 79 0e
0x0018  93 e9 97 84 88 d1 db 37
0x0020  8d a9 93 08 85 19 87 cf
0x0028  18 d5 f1 b5 3f 81 7c ce
0x0030  5a 00 4c f3 93 cc 89 58
0x0038  bd dc 06 5f 25 f8 4a f5
0x0040  09 b1 7d d3 67 64 18 de
0x0048  a3 55 43 79 56 61 79 01
0x0050  e0 69 57 fb ca 8a 6a 47
0x0058  a5 b5 15 3e 8d 3a b7
```

The same data, broken out by octet and semantics, is:

```
0x0000  c1                      packet type: PKESK
0x0001     5d                   packet length
0x0002        06                PKESK version 6
0x0003           21             length of fingerprint
0x0004              06          Key version 6
0x0005                 12 c8 3f Key fingerprint
0x0008  1e 70 6f 63 08 fe 15 1a
0x0010  41 77 43 a1 f0 33 79 0e
0x0018  93 e9 97 84 88 d1 db 37
0x0020  8d a9 93 08 85
0x0025                 19       algorithm: X25519
0x0026                    87 cf Ephemeral key
0x0028  18 d5 f1 b5 3f 81 7c ce
0x0030  5a 00 4c f3 93 cc 89 58
0x0038  bd dc 06 5f 25 f8 4a f5
0x0040  09 b1 7d d3 67 64
0x0046                    18    ESK length
0x0047                       de ESK
0x0048  a3 55 43 79 56 61 79 01
0x0050  e0 69 57 fb ca 8a 6a 47
0x0058  a5 b5 15 3e 8d 3a b7
```

A.8.2. X25519 Encryption/Decryption of the Session Key

Ephemeral key:

```
87 cf 18 d5 f1 b5 3f 81 7c ce 5a 00 4c f3 93 cc
89 58 bd dc 06 5f 25 f8 4a f5 09 b1 7d d3 67 64
```

This ephemeral key is derived from the following ephemeral secret key material, which is never placed on the wire:

```
af 1e 43 c0 d1 23 ef e8 93 a7 d4 d3 90 f3 a7 61
e3 fa c3 3d fc 7f 3e da a8 30 c9 01 13 52 c7 79
```

Public key from the target certificate (see Appendix A.3):

```
86 93 24 83 67 f9 e5 01 5d b9 22 f8 f4 80 95 dd
a7 84 98 7f 2d 59 85 b1 2f ba d1 6c af 5e 44 35
```

The corresponding long-lived X25519 private key material (see Appendix A.4):

```
4d 60 0a 4f 79 4d 44 77 5c 57 a2 6e 0f ee fe d5
58 e9 af ff d6 ad 0d 58 2d 57 fb 2b a2 dc ed b8
```

Shared point:

```
67 e3 0e 69 cd c7 ba b2 a2 68 0d 78 ac a4 6a 2f
8b 6e 2a e4 4d 39 8b dc 6f 92 c5 ad 4a 49 25 14
```

HKDF output:

```
f6 6d ad cf f6 45 92 23 9b 25 45 39 b6 4f f6 07
```

Decrypted session key:

```
dd 70 8f 6f a1 ed 65 11 4d 68 d2 34 3e 7c 2f 1d
```

A.8.3. Sample v2 SEIPD Packet

This packet contains the following series of octets:

```
0x0000  d2 69 02 07 02 06 61 64
0x0008  16 53 5b e0 b0 71 6d 60
0x0010  e0 52 a5 6c 4c 40 7f 9e
0x0018  b3 6b 0e fa fe 9a d0 a0
0x0020  df 9b 03 3c 69 a2 1b a9
0x0028  eb d2 c0 ec 95 bf 56 9d
0x0030  25 c9 99 ee 4a 3d e1 70
0x0038  58 f4 0d fa 8b 4c 68 2b
0x0040  e3 fb bb d7 b2 7e b0 f5
0x0048  9b b5 00 5f 80 c7 c6 f4
0x0050  03 88 c3 0a d4 06 ab 05
0x0058  13 dc d6 f9 fd 73 76 56
0x0060  28 6e 11 77 d0 0f 88 8a
0x0068  db 31 c4
```

The same data, broken out by octet and semantics, is:

```
0x0000  d2                      packet type: SEIPD
0x0001     69                   packet length
0x0002        02                SEIPD version 2
0x0003           07             cipher: AES128
0x0004              02          AEAD mode: OCB
0x0005                 06       chunk size (2**12 octets)
0x0006                    61 64  salt
0x0008  16 53 5b e0 b0 71 6d 60
0x0010  e0 52 a5 6c 4c 40 7f 9e
0x0018  b3 6b 0e fa fe 9a d0 a0
0x0020  df 9b 03 3c 69 a2
0x0026                    1b a9  chunk #0 encrypted data
0x0028  eb d2 c0 ec 95 bf 56 9d
0x0030  25 c9 99 ee 4a 3d e1 70
0x0038  58 f4 0d fa 8b 4c 68 2b
0x0040  e3 fb bb d7 b2 7e b0 f5
0x0048  9b b5 00
0x004b           5f 80 c7 c6 f4  chunk #0 AEAD tag
0x0050  03 88 c3 0a d4 06 ab 05
0x0058  13 dc d6
0x005b           f9 fd 73 76 56  final AEAD tag (#1)
S0x0060  28 6e 11 77 d0 0f 88 8a
0x0068  db 31 c4
```

A.8.4. Decryption of Data

Starting AEAD-OCB decryption of data, using the session key.

HKDF info:

```
d2 02 07 02 06
```

HKDF output:

```
45 12 f7 14 9d 86 33 41 52 7c 65 67 d5 bf fc 42
5f af 32 50 21 2f f9
```

Message key:

```
45 12 f7 14 9d 86 33 41 52 7c 65 67 d5 bf fc 42
```

Initialization vector:

```
5f af 32 50 21 2f f9
```

Chunk #0:

Nonce:

```
5f af 32 50 21 2f f9 00 00 00 00 00 00 00 00
```

Additional authenticated data:

```
d2 02 07 02 06
```

Encrypted data chunk:

```
1b a9 eb d2 c0 ec 95 bf 56 9d 25 c9 99 ee 4a 3d
e1 70 58 f4 0d fa 8b 4c 68 2b e3 fb bb d7 b2 7e
b0 f5 9b b5 00 5f 80 c7 c6 f4 03 88 c3 0a d4 06
ab 05 13 dc d6
```

Decrypted chunk #0.

Literal data packet with the string contents `Hello, world!`:

```
cb 13 62 00 00 00 00 00 48 65 6c 6c 6f 2c 20 77
6f 72 6c 64 21
```

Padding packet:

```
d5 0e c5 a2 93 07 29 91 62 81 47 d7 2c 8f 86 b7
```

Authenticating final tag:

Final nonce:

```
5f af 32 50 21 2f f9 00 00 00 00 00 00 00 01
```

Final additional authenticated data:

```
d2 02 07 02 06 00 00 00 00 00 00 00 25
```

A.8.5. Complete X25519-AEAD-OCB Encrypted Packet Sequence

```
-----BEGIN PGP MESSAGE-----

wV0GIQYSyD8ecG9jCP4VGkF3Q6HwM3kOk+mXhIjR2zeNqZMIhRmHzxjV8bU/gXzO
WgBM85PMiVi93AZfJfhK9QmxfdNnZBjeo1VDeVZheQHgaVf7yopqR6W1FT6NOrfS
aQIHAgZhZBZTW+CwcW1g4FKlbExAf56zaw76/prQoN+bAzxpohup69LA7JW/Vp0l
yZnuSj3hcFj0DfqLTGgr4/u717J+sPWbtQBfgMfG9AOIwwrUBqsFE9zW+f1zdlYo
bhF30A+IitsxxA==
-----END PGP MESSAGE-----
```

A.9. Sample AEAD-EAX Encryption and Decryption

This example encrypts the cleartext string `Hello, world!` with the passphrase `password`, using AES-128 with AEAD-EAX encryption.

A.9.1. Sample Symmetric-Key Encrypted Session Key Packet (v6)

This packet contains the following series of octets:

```
0x0000  c3 40 06 1e 07 01 0b 03
0x0008  08 a5 ae 57 9d 1f c5 d8
0x0010  2b ff 69 22 4f 91 99 93
0x0018  b3 50 6f a3 b5 9a 6a 73
0x0020  cf f8 c5 ef c5 f4 1c 57
0x0028  fb 54 e1 c2 26 81 5d 78
0x0030  28 f5 f9 2c 45 4e b6 5e
0x0038  be 00 ab 59 86 c6 8e 6e
0x0040  7c 55
```

The same data, broken out by octet and semantics, is:

```
0x0000  c3                        packet type: SKESK
0x0001     40                     packet length
0x0002        06                  SKESK version 6
0x0003           1e               length through end of AEAD nonce
0x0004              07            cipher: AES128
0x0005                 01         AEAD mode: EAX
0x0006                    0b      length of S2K
0x0007                       03   S2K type: iterated+salted
0x0008  08                        S2K hash: SHA2-256
0x0009     a5 ae 57 9d 1f c5 d8   S2K salt
0x0010  2b
0x0011     ff                     S2K iterations (65011712 octets)
0x0012        69 22 4f 91 99 93   AEAD nonce
0x0018  b3 50 6f a3 b5 9a 6a 73
0x0020  cf f8
0x0022        c5 ef c5 f4 1c 57   encrypted session key
0x0028  fb 54 e1 c2 26 81 5d 78
0x0030  28 f5
0x0032        f9 2c 45 4e b6 5e   AEAD tag
0x0038  be 00 ab 59 86 c6 8e 6e
0x0040  7c 55
```

A.9.2. Starting AEAD-EAX Decryption of the Session Key

The derived key is:

```
15 49 67 e5 90 aa 1f 92 3e 1c 0a c6 4c 88 f2 3d
```

HKDF info:

```
c3 06 07 01
```

HKDF output:

```
2f ce 33 1f 39 dd 95 5c c4 1e 95 d8 70 c7 21 39
```

Authenticated Data:

```
c3 06 07 01
```

Nonce:

```
69 22 4f 91 99 93 b3 50 6f a3 b5 9a 6a 73 cf f8
```

Decrypted session key:

```
  38 81 ba fe 98 54 12 45 9b 86 c3 6f 98 cb 9a 5e
```

A.9.3. Sample v2 SEIPD Packet

This packet contains the following series of octets:

```
0x0000  d2 69 02 07 01 06 9f f9
0x0008  0e 3b 32 19 64 f3 a4 29
0x0010  13 c8 dc c6 61 93 25 01
0x0018  52 27 ef b7 ea ea a4 9f
0x0020  04 c2 e6 74 17 5d 4a 3d
0x0028  22 6e d6 af cb 9c a9 ac
0x0030  12 2c 14 70 e1 1c 63 d4
0x0038  c0 ab 24 1c 6a 93 8a d4
0x0040  8b f9 9a 5a 99 b9 0b ba
0x0048  83 25 de 61 04 75 40 25
0x0050  8a b7 95 9a 95 ad 05 1d
0x0058  da 96 eb 15 43 1d fe f5
0x0060  f5 e2 25 5c a7 82 61 54
0x0068  6e 33 9a
```

The same data, broken out by octet and semantics, is:

```
0x0000  d2                      packet type: SEIPD
0x0001     69                   packet length
0x0002        02                SEIPD version 2
0x0003           07             cipher: AES128
0x0004              01          AEAD mode: EAX
0x0005                 06       chunk size (2**12 octets)
0x0005                    9f f9 salt
0x0008  0e 3b 32 19 64 f3 a4 29
0x0010  13 c8 dc c6 61 93 25 01
0x0018  52 27 ef b7 ea ea a4 9f
0x0020  04 c2 e6 74 17 5d
0x0026                    4a 3d chunk #0 encrypted data
0x0028  22 6e d6 af cb 9c a9 ac
0x0030  12 2c 14 70 e1 1c 63 d4
0x0038  c0 ab 24 1c 6a 93 8a d4
0x0040  8b f9 9a 5a 99 b9 0b ba
0x0048  83 25 de
0x004b           61 04 75 40 25 chunk #0 AEAD tag
0x0050  8a b7 95 9a 95 ad 05 1d
0x0058  da 96 eb
0x005b           15 43 1d fe f5 final AEAD tag (#1)
0x0060  f5 e2 25 5c a7 82 61 54
0x0068  6e 33 9a
```

A.9.4. Decryption of Data

Starting AEAD-EAX decryption of data, using the session key.

HKDF info:

```
d2 02 07 01 06
```

HKDF output:

```
b5 04 22 ac 1c 26 be 9d dd 83 1d 5b bb 36 b6 4f
78 b8 33 f2 e9 4a 60 c0
```

Message key:

```
b5 04 22 ac 1c 26 be 9d dd 83 1d 5b bb 36 b6 4f
```

Initialization vector:

```
78 b8 33 f2 e9 4a 60 c0
```

Chunk #0:

Nonce:

```
78 b8 33 f2 e9 4a 60 c0 00 00 00 00 00 00 00 00
```

Additional authenticated data:

```
d2 02 07 01 06
```

Decrypted chunk #0.

Literal data packet with the string contents `Hello, world!`:

```
cb 13 62 00 00 00 00 00 48 65 6c 6c 6f 2c 20 77
6f 72 6c 64 21
```

Padding packet:

```
d5 0e ae 5b f0 cd 67 05 50 03 55 81 6c b0 c8 ff
```

Authenticating final tag:

Final nonce:

```
  78 b8 33 f2 e9 4a 60 c0 00 00 00 00 00 00 00 01
```

Final additional authenticated data:

```
  d2 02 07 01 06 00 00 00 00 00 00 00 00 25
```

A.9.5. Complete AEAD-EAX Encrypted Packet Sequence

```
-----BEGIN PGP MESSAGE-----

w0AGHgcBCwMIpa5XnR/F2Cv/aSJPkZmTs1Bvo7WaanPP+MXvxfQcV/tU4cImgV14
KPX5LEVOtl6+AKtZhsaObnxV0mkCBwEGn/kOOzIZZPOkKRPI3MZhkyUBUifvt+rq
pJ8EwuZ0F11KPSJu1q/LnKmsEiwUcOEcY9TAqyQcapOK1Iv5mlqZuQu6gyXeYQR1
QCWKt5Wala0FHdqW6xVDHf719eIlXKeCYVRuM5o=
-----END PGP MESSAGE-----
```

A.10. Sample AEAD-OCB Encryption and Decryption

This example encrypts the cleartext string `Hello, world!` with the passphrase `password`, using AES-128 with AEAD-OCB encryption.

A.10.1. Sample Symmetric-Key Encrypted Session Key Packet (v6)

This packet contains the following series of octets:

```
0x0000  c3 3f 06 1d 07 02 0b 03
0x0008  08 56 a2 98 d2 f5 e3 64
0x0010  53 ff cf cc 5c 11 66 4e
0x0018  db 9d b4 25 90 d7 dc 46
0x0020  b0 72 41 b6 12 c3 81 2c
0x0028  ff fb ea 00 f2 34 7b 25
0x0030  64 11 23 f8 87 ae 60 d4
0x0038  fd 61 4e 08 37 d8 19 d3
0x0040  6c
```

The same data, broken out by octet and semantics, is:

```
0x0000  c3                        packet type: SKESK
0x0001     3f                     packet length
0x0002        06                  SKESK version 6
0x0003           1d               length through end of AEAD nonce
0x0004              07            cipher: AES128
0x0005                 02         AEAD mode: OCB
0x0006                    0b      length of S2K
0x0007                       03   S2K type: iterated+salted
0x0008  08                        S2K hash: SHA2-256
0x0009     56 a2 98 d2 f5 e3 64   S2K salt
0x0010  53
0x0011    ff                      S2K iterations (65011712 octets)
0x0012        cf cc 5c 11 66 4e   AEAD nonce
0x0018  db 9d b4 25 90 d7 dc 46
0x0020  b0
0x0021     72 41 b6 12 c3 81 2c   encrypted session key
0x0028  ff fb ea 00 f2 34 7b 25
0x0030  64
0x0031     11 23 f8 87 ae 60 d4   AEAD tag
0x0038  fd 61 4e 08 37 d8 19 d3
0x0040  6c
```

A.10.2. Starting AEAD-OCB Decryption of the Session Key

The derived key is:

```
e8 0d e2 43 a3 62 d9 3b 9d c6 07 ed e9 6a 73 56
```

HKDF info:

```
c3 06 07 02
```

HKDF output:

```
38 a9 b3 45 b5 68 0b b6 1b b6 5d 73 ee c7 ec d9
```

Authenticated Data:

```
c3 06 07 02
```

Nonce:

```
cf cc 5c 11 66 4e db 9d b4 25 90 d7 dc 46 b0
```

Decrypted session key:

```
  28 e7 9a b8 23 97 d3 c6 3d e2 4a c2 17 d7 b7 91
```

A.10.3. Sample v2 SEIPD Packet

This packet contains the following series of octets:

```
0x0000  d2 69 02 07 02 06 20 a6
0x0008  61 f7 31 fc 9a 30 32 b5
0x0010  62 33 26 02 7e 3a 5d 8d
0x0018  b5 74 8e be ff 0b 0c 59
0x0020  10 d0 9e cd d6 41 ff 9f
0x0028  d3 85 62 75 80 35 bc 49
0x0030  75 4c e1 bf 3f ff a7 da
0x0038  d0 a3 b8 10 4f 51 33 cf
0x0040  42 a4 10 0a 83 ee f4 ca
0x0048  1b 48 01 a8 84 6b f4 2b
0x0050  cd a7 c8 ce 9d 65 e2 12
0x0058  f3 01 cb cd 98 fd ca de
0x0060  69 4a 87 7a d4 24 73 23
0x0068  f6 e8 57
```

The same data, broken out by octet and semantics, is:

```
0x0000  d2                      packet type: SEIPD
0x0001     69                   packet length
0x0002        02                SEIPD version 2
0x0003           07             cipher: AES128
0x0004              02          AEAD mode: OCB
0x0005                 06       chunk size (2**21 octets)
0x0006                    20 a6 salt
0x0008  61 f7 31 fc 9a 30 32 b5
0x0010  62 33 26 02 7e 3a 5d 8d
0x0018  b5 74 8e be ff 0b 0c 59
0x0020  10 d0 9e cd d6 41
0x0026                    ff 9f chunk #0 encrypted data
0x0028  d3 85 62 75 80 35 bc 49
0x0030  75 4c e1 bf 3f ff a7 da
0x0038  d0 a3 b8 10 4f 51 33 cf
0x0040  42 a4 10 0a 83 ee f4 ca
0x0048  1b 48 01
0x004b           a8 84 6b f4 2b chunk #0 authentication tag
0x0050  cd a7 c8 ce 9d 65 e2 12
0x0058  f3 01 cb
0x005b           cd 98 fd ca de final AEAD tag (#1)
0x0060  69 4a 87 7a d4 24 73 23
0x0068  f6 e8 57
```

A.10.4. Decryption of Data

Starting AEAD-OCB decryption of data, using the session key.

HKDF info:

```
d2 02 07 02 06
```

HKDF output:

```
71 66 2a 11 ee 5b 4e 08 14 4e 6d e8 83 a0 09 99
eb de 12 bb 57 0d cf
```

Message key:

```
71 66 2a 11 ee 5b 4e 08 14 4e 6d e8 83 a0 09 99
```

Initialization vector:

```
eb de 12 bb 57 0d cf
```

Chunk #0:

Nonce:

```
eb de 12 bb 57 0d cf 00 00 00 00 00 00 00 00
```

Additional authenticated data:

```
d2 02 07 02 06
```

Decrypted chunk #0.

Literal data packet with the string contents `Hello, world!`:

```
cb 13 62 00 00 00 00 00 48 65 6c 6c 6f 2c 20 77
6f 72 6c 64 21
```

Padding packet:

```
d5 0e ae 6a a1 64 9b 56 aa 83 5b 26 13 90 2b d2
```

Authenticating final tag:

Final nonce:

```
  eb de 12 bb 57 0d cf 00 00 00 00 00 00 00 01
```

Final additional authenticated data:

```
  d2 02 07 02 06 00 00 00 00 00 00 00 25
```

A.10.5. Complete AEAD-OCB Encrypted Packet Sequence

```
-----BEGIN PGP MESSAGE-----

wz8GHQcCCwMIVqKY0vXjZFP/z8xcEWZO2520JZDX3EawckG2EsOBLP/76gDyNHsl
ZBEj+IeuYNT9YU4IN9gZ02zSaQIHAgYgpmH3MfyaMDK1YjMmAn46XY21dI6+/wsM
WRDQns3WQf+f04VidYA1vEl1TOG/P/+n2tCjuBBPUTPPQqQQCoPu9MobSAGohGv0
K82nyM6dZeIS8wHLzZj9yt5pSod61CRzI/boVw==
-----END PGP MESSAGE-----
```

A.11. Sample AEAD-GCM Encryption and Decryption

This example encrypts the cleartext string `Hello, world!` with the passphrase `password`, using AES-128 with AEAD-GCM encryption.

A.11.1. Sample Symmetric-Key Encrypted Session Key Packet (v6)

This packet contains the following series of octets:

```
0x0000  c3 3c 06 1a 07 03 0b 03
0x0008  08 e9 d3 97 85 b2 07 00
0x0010  08 ff b4 2e 7c 48 3e f4
0x0018  88 44 57 cb 37 26 b9 b3
0x0020  db 9f f7 76 e5 f4 d9 a4
0x0028  09 52 e2 44 72 98 85 1a
0x0030  bf ff 75 26 df 2d d5 54
0x0038  41 75 79 a7 79 9f
```

The same data, broken out by octet and semantics, is:

```
0x0000  c3                        packet type: SKESK
0x0001     3c                     packet length
0x0002        06                  SKESK version 6
0x0003           1a               length through end of AEAD nonce
0x0004              07            cipher: AES128
0x0005                 03         AEAD mode: GCM
0x0006                    0b      length of S2K
0x0007                       03   S2K type: iterated+salted
0x0008  08                        S2K hash: SHA2-256
0x0009     e9 d3 97 85 b2 07 00   S2K salt
0x0010  08
0x0011     ff                     S2K iterations (65011712 octets)
0x0012        b4 2e 7c 48 3e f4   AEAD nonce
0x0018  88 44 57 cb 37 26
0x001e                    b9 b3  encrypted session key
0x0020  db 9f f7 76 e5 f4 d9 a4
0x0028  09 52 e2 44 72 98
0x002e                     85 1a  AEAD tag
0x0030  bf ff 75 26 df 2d d5 54
0x0038  41 75 79 a7 79 9f
```

A.11.2. Starting AEAD-GCM Decryption of the Session Key

The derived key is:

```
25 02 81 71 5b ba 78 28 ef 71 ef 64 c4 78 47 53
```

HKDF info:

```
c3 06 07 03
```

HKDF output:

```
7a 6f 9a b7 f9 9f 7e f8 db ef 84 1c 65 08 00 f5
```

Authenticated Data:

```
c3 06 07 03
```

Nonce:

```
b4 2e 7c 48 3e f4 88 44 57 cb 37 26
```

Decrypted session key:

```
  19 36 fc 85 68 98 02 74 bb 90 0d 83 19 36 0c 77
```

A.11.3. Sample v2 SEIPD Packet

This packet contains the following series of octets, is:

```
0x0000  d2 69 02 07 03 06 fc b9
0x0008  44 90 bc b9 8b bd c9 d1
0x0010  06 c6 09 02 66 94 0f 72
0x0018  e8 9e dc 21 b5 59 6b 15
0x0020  76 b1 01 ed 0f 9f fc 6f
0x0028  c6 d6 5b bf d2 4d cd 07
0x0030  90 96 6e 6d 1e 85 a3 00
0x0038  53 78 4c b1 d8 b6 a0 69
0x0040  9e f1 21 55 a7 b2 ad 62
0x0048  58 53 1b 57 65 1f d7 77
0x0050  79 12 fa 95 e3 5d 9b 40
0x0058  21 6f 69 a4 c2 48 db 28
0x0060  ff 43 31 f1 63 29 07 39
0x0068  9e 6f f9
```

The same data, broken out by octet and semantics, is:

```
0x0000  d2                      packet type: SEIPD
0x0001     69                   packet length
0x0002        02                SEIPD version 2
0x0003           07             cipher: AES128
0x0004              03          AEAD mode: GCM
0x0005                 06       chunk size (2**21 octets)
0x0006                    fc b9 salt
0x0008  44 90 bc b9 8b bd c9 d1
0x0010  06 c6 09 02 66 94 0f 72
0x0018  e8 9e dc 21 b5 59 6b 15
0x0020  76 b1 01 ed 0f 9f
0x0026                    fc 6f chunk #0 encrypted data
0x0028  c6 d6 5b bf d2 4d cd 07
0x0030  90 96 6e 6d 1e 85 a3 00
0x0038  53 78 4c b1 d8 b6 a0 69
0x0040  9e f1 21 55 a7 b2 ad 62
0x0048  58 53 1b
0x004b           57 65 1f d7 77 chunk #0 authentication tag
0x0050  79 12 fa 95 e3 5d 9b 40
0x0058  21 6f 69
0x005b           a4 c2 48 db 28 final AEAD tag (#1)
0x0060  ff 43 31 f1 63 29 07 39
0x0068  9e 6f f9
```

A.11.4. Decryption of Data

Starting AEAD-GCM decryption of data, using the session key.

HKDF info:

```
d2 02 07 03 06
```

HKDF output:

```
ea 14 38 80 3c b8 a4 77 40 ce 9b 54 c3 38 77 8d
4d 2b dc 2b
```

Message key:

```
ea 14 38 80 3c b8 a4 77 40 ce 9b 54 c3 38 77 8d
```

Initialization vector:

```
4d 2b dc 2b
```

Chunk #0:

Nonce:

```
4d 2b dc 2b 00 00 00 00 00 00 00 00
```

Additional authenticated data:

```
d2 02 07 03 06
```

Decrypted chunk #0.

Literal data packet with the string contents `Hello, world!`:

```
cb 13 62 00 00 00 00 00 48 65 6c 6c 6f 2c 20 77
6f 72 6c 64 21
```

Padding packet:

```
d5 0e 1c e2 26 9a 9e dd ef 81 03 21 72 b7 ed 7c
```

Authenticating final tag:

Final nonce:

```
  4d 2b dc 2b 00 00 00 00 00 00 00 01
```

Final additional authenticated data:

```
  d2 02 07 03 06 00 00 00 00 00 00 00 25
```

A.11.5. Complete AEAD-GCM Encrypted Packet Sequence

```
-----BEGIN PGP MESSAGE-----

wzwGGgcDCwMI6dOXhbIHAAj/tC58SD70iERXyzcmubPbn/d25fTZpAlS4kRymIUa
v/91Jt8t1VRBdXmneZ/SaQIHAwb8uUSQvLmLvcnRBsYJAmaUD3LontwhtVlrFXax
Ae0Pn/xvxtZbv9JNzQeQlm5tHoWjAFN4TLHYtqBpnvEhVaeyrWJYUxtXZR/Xd3kS
+pXjXZtAIW9ppMJI2yj/QzHxYykHOZ5v+Q==
-----END PGP MESSAGE-----
```

A.12. Sample Messages Encrypted Using Argon2

These messages are the literal data "Hello, world!" encrypted using v1 SEIPD, with Argon2 and the passphrase "password", using different session key sizes. In each example, the choice of symmetric cipher is the same in both the v4 SKESK packet and v1 SEIPD packet. In all cases, the Argon2 parameters are t = 1, p = 4, and m = 21.

A.12.1. Version 4 SKESK Using Argon2 with AES-128

```
-----BEGIN PGP MESSAGE-----
Comment: Encrypted using AES with 128-bit key
Comment: Session key: 01FE16BBACFD1E7B78EF3B865187374F

wycEBwScUvg8J/leUNU1RA7N/zE2AQQVnlL8rSLPP5VlQsunlO+ECxHSPgGYGKY+
YJz4u6F+DDlDBOr5NRQXt/KJIf4m4mOlKyC/uqLbpnLJZMnTq3o79GxBTdIdOzhH
XfA3pqV4mTzF
-----END PGP MESSAGE-----
```

A.12.2. Version 4 SKESK Using Argon2 with AES-192

```
-----BEGIN PGP MESSAGE-----
Comment: Encrypted using AES with 192-bit key
Comment: Session key: 27006DAE68E509022CE45A14E569E91001C2955...
Comment: Session key: ...AF8DFE194

wy8ECAThTKxHFTRZGKli3KNH4UP4AQQVhzLJ2va3FG8/pmpIPd/H/mdoVS5VBLLw
F9I+AdJ1Sw56PRYiKZjCvHg+2bnq02s33AJJoyBexBI4QKATFRkyez2gldJldRys
LVg77Mwwfgl2n/d572WciAM=
-----END PGP MESSAGE-----
```

A.12.3. Version 4 SKESK Using Argon2 with AES-256

```
-----BEGIN PGP MESSAGE-----
Comment: Encrypted using AES with 256-bit key
Comment: Session key: BBEDA55B9AAE63DAC45D4F49D89DACF4AF37FEF...
Comment: Session key: ...C13BAB2F1F8E18FB74580D8B0

wzcECQS4eJUgIG/3mcaILEJFpmJ8AQQVnZ9l7KtagdClm9UaQ/Z6M/5roklSGpGu
623YmaXezGj80j4B+Ku1sgTdJo87X1Wrup7l0wJypZls21Uwd67m9koF60eefH/K
95D1usliXOEm8ayQJQmZrjf6K6v9PWwqMQ==
-----END PGP MESSAGE-----
```

Appendix B. Upgrade Guidance (Adapting Implementations from RFCs 4880 and 6637)

This subsection offers a concise, non-normative summary of the substantial additions to and departures from [RFC4880] and [RFC6637]. It is intended to help implementers who are augmenting an existing implementation from those specifications to comply with this specification. Cryptographic algorithms marked with "MTI" are mandatory to implement.

- Public Key Signing Algorithms:
  - Ed25519 (Sections 5.5.5.9 and 5.2.3.4) -- MTI
  - Ed448 (Sections 5.5.5.10 and 5.2.3.5)
  - EdDSALegacy with Ed25519Legacy (Sections 5.5.5.5 and 5.2.3.3)
  - ECDSA with Brainpool curves (Section 9.2)
- Public Key Encryption Algorithms:
  - X25519 (Sections 5.5.5.7 and 5.1.6) -- MTI
  - X448 (Sections 5.5.5.8 and 5.1.7)
  - ECDH with Curve25519Legacy (Section 9.2)
  - ECDH with Brainpool curves (Section 9.2)
- AEAD Encryption:
  - Version 2 SEIPD (Section 5.13.2)
  - AEAD modes:
    - OCB mode (Section 5.13.4) -- MTI
    - EAX mode (Section 5.13.3)
    - GCM mode (Section 5.13.5)
  - Version 6 PKESK (Section 5.1.2)
  - Version 6 SKESK (Section 5.3.2)
  - Features subpacket: add flag for SEIPDv2 (Section 5.2.3.32)

- Subpacket: Preferred AEAD Ciphersuites (Section 5.2.3.15)
- Secret key encryption: AEAD "S2K usage octet" (Sections 3.7.2 and 5.5.3)

- Version 6 Keys and Signatures:
  - Version 6 Public keys (Section 5.5.2.3)
  - Version 6 Fingerprint and Key ID (Section 5.5.4.3)
  - Version 6 Secret keys (Section 5.5.3)
  - Version 6 Signatures (Section 5.2.3)
  - Version 6 One-Pass Signatures (Section 5.4)
- Certificate (Transferable Public Key) Structure:
  - Preferences subpackets in Direct Key Signatures (Section 5.2.3.10)
  - Self-verifying revocation certificate (Section 10.1.2)
  - User ID is explicitly optional (Section 10.1.1)
- S2K: Argon2 (Section 3.7.1.4)
- Subpacket: Intended Recipient Fingerprint (Section 5.2.3.36)
- Digest Algorithms: SHA3-256 and SHA3-512 (Section 9.5)
- Packet: Padding (Section 5.14)
- Message Structure: Packet Criticality (Section 4.3)
- Deprecations:
  - Public Key Algorithms:
    - Avoid RSA weak keys (Section 12.4)
    - Avoid DSA (Section 12.5)
    - Avoid ElGamal (Sections 12.6 and 5.1.4)
    - For Version 6 Keys: Avoid EdDSA25519Legacy and Curve25519Legacy (Section 9.2)
  - Digest Algorithms:
    - Avoid MD5, SHA1, and RIPEMD160 (Section 9.5)
  - Symmetric Key Algorithms:
    - Avoid IDEA, TripleDES, and CAST5 (Section 9.3)
  - S2K Specifier:
    - Avoid Simple S2K (Section 3.7.1.1)
  - Secret Key Protections (a.k.a. S2K Usage):
    - Avoid MalleableCFB (Section 3.7.2.1)
  - Packet Types:
    - Avoid Symmetrically Encrypted Data (Sections 5.7 and 13.7)

- Literal Data Packet Metadata:
  - Avoid Filename and Date fields (Section 5.9)
  - Avoid Special `_CONSOLE` "filename" (Section 5.9.1)
- Packet Versions:
  - Avoid Version 3 Public Keys (Section 5.5.2.1)
  - Avoid Version 3 Signatures (Section 5.2)
- Signature Types:
  - Avoid Reserved Signature type ID 0xFF (Sections 5.2.1.16 and 5.2.4.1)
- Signature Subpackets:
  - For Version 6 Signatures: Avoid Issuer Key ID (Section 5.2.3.12)
  - Avoid Revocation Key (Section 5.2.3.23)
- ASCII Armor:
  - Ignore; do not emit CRC (Section 6.1)
  - Do not emit "Version" armor header (Section 6.2.2.1)
- Cleartext Signature Framework:
  - Ignore; avoid emitting unnecessary Hash: headers (Section 6.2.2.3)
  - Reject CSF signatures with invalid Hash: headers (Section 6.2.2.3) or any other Armor Header (Section 7.1)

B.1. Terminology Changes

Note that some of the words used in previous versions of the OpenPGP specification have been improved in this document.

In previous versions, the following terms were used:

- "Radix-64" was used to refer to OpenPGP's ASCII Armor base64 encoding (Section 6).
- "Old packet format" was used to refer to the Legacy packet format (Section 4.2.2) predating [RFC2440].
- "New packet format" was used to refer to the OpenPGP packet format (Section 4.2.1) introduced in [RFC2440].
- "Certificate" was used ambiguously to mean multiple things. In this document, it means "Transferable Public Key" exclusively.
- "Preferred Symmetric Algorithms" was the old name for the "Preferred Symmetric Ciphers for v1 SEIPD" subpacket (Section 5.2.3.14).

- "Modification Detection Code" or "MDC" was originally described as a distinct packet (packet type ID 19), and its corresponding flag in the Features subpacket (Section 5.2.3.32) was known as "Modification Detection". It is now described as an intrinsic part of v1 SEIPD (Section 5.13.1), and the same corresponding flag is known as "Symmetrically Encrypted Integrity Protected Data packet version 1".
- "Packet Tag" was used to refer to the Packet Type ID (Section 5) or sometimes to the encoded Packet Type ID (Section 4.2).

Appendix C. Errata Addressed by This Document

The following verified errata have been incorporated or are otherwise resolved by this document:

- [Errata-2199] - S2K hash/cipher octet correction
- [Errata-2200] - No implicit use of IDEA correction
- [Errata-2206] - PKESK acronym expansion
- [Errata-2208] - Signature key owner clarification
- [Errata-2214] - Signature hashing clarification
- [Errata-2216] - Self-signature applies to user ID correction
- [Errata-2219] - Session key encryption storage clarification
- [Errata-2222] - Simple hash **MUST**/**MAY** clarification
- [Errata-2226] - Native line endings **SHOULD** clarification
- [Errata-2234] - Radix-64/base64 clarification
- [Errata-2235] - ASCII/UTF-8 collation sequence clarification
- [Errata-2236] - Packet Composition is a sequence clarification
- [Errata-2238] - Subkey packets come after all User ID packets clarification
- [Errata-2240] - Subkey removal clarification
- [Errata-2242] - mL/emLen variable correction
- [Errata-2243] - CFB mode initialization vector (IV) clarification
- [Errata-2270] - SHA-224 octet sequence correction
- [Errata-2271] - Radix-64 correction
- [Errata-3298] - Key revocation signatures correction
- [Errata-5491] - C code fix for CRC24_POLY define
- [Errata-7545] - Armor Header colon hex fix
- [Errata-7889] - Signature/certification correction

Acknowledgements

Thanks to the OpenPGP Design Team for working on this document and preparing it for working group consumption: Stephen Farrell, Daniel Kahn Gillmor, Daniel Huigens, Jeffrey Lau, Yutaka Niibe, Justus Winter, and Paul Wouters.

Thanks to Werner Koch for the work on earlier draft versions of this document and Andrey Jivsov for the work on [RFC6637].

This document also draws on much previous work from a number of other authors including Derek Atkins, Charles Breed, Dave Del Torto, Marc Dyksterhouse, Gail Haspert, Gene Hoffman, Paul Hoffman, Ben Laurie, Raph Levien, Colin Plumb, Will Price, Daphne Shaw, William Stallings, Mark Weaver, and Philip R. Zimmermann.

Authors' Addresses

Paul Wouters (EDITOR)
Aiven
Email: paul.wouters@aiven.io

Daniel Huigens
Proton AG
Email: d.huigens@protonmail.com

Justus Winter
Sequoia-PGP
Email: justus@sequoia-pgp.org

Yutaka Niibe
FSIJ
Email: gniibe@fsij.org