OpenPGP Message Format
draft-ietf-openpgp-crypto-refresh-06

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: RFC 4880 (OpenPGP), RFC 5581 (Camellia in
   OpenPGP) and RFC 6637 (Elliptic Curves in OpenPGP).

This Internet-Draft will expire on 8 December 2022.

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 RFC 4880, "OpenPGP Message Format", which is a revision of RFC 2440, which itself replaces RFC 1991, "PGP Message Exchange Formats" [RFC1991] [RFC2440] [RFC4880].

This document obsoletes: RFC 4880 (OpenPGP), RFC 5581 (Camellia in OpenPGP) and RFC 6637 (Elliptic Curves in OpenPGP).

1.1. Terms

* OpenPGP – This is a term for security software that uses PGP 5 as a basis, formalized in this document.

* PGP – Pretty Good Privacy. PGP is a family of software systems developed by Philip R. Zimmermann from which OpenPGP is based.

* PGP 2 – This version of PGP has many variants; where necessary a more detailed version number is used here. PGP 2 uses only RSA, MD5, and IDEA for its cryptographic transforms. An informational RFC, RFC 1991, was written describing this version of PGP.

* PGP 5 – This version of PGP is formerly known as "PGP 3" in the community. It has new formats and corrects a number of problems in the PGP 2 design. It is referred to here as PGP 5 because that software was the first release of the "PGP 3" code base.

* GnuPG – GNU Privacy Guard, also called GPG. GnuPG is an OpenPGP implementation that avoids all encumbered algorithms. Consequently, early versions of GnuPG did not include RSA public keys.

"PGP", "Pretty Good", and "Pretty Good Privacy" are trademarks of PGP Corporation and are used with permission. The term "OpenPGP" refers to the protocol described in this and related documents.

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].

[2](#). General functions

OpenPGP provides data integrity services for messages and data files
by using these core technologies:

* digital signatures

* encryption

* compression

* Radix-64 conversion

In addition, OpenPGP provides key management and certificate
services, but many of these are beyond the scope of this document.

[2.1](#). Confidentiality via Encryption

OpenPGP combines symmetric-key encryption and public-key encryption
to provide confidentiality. When made confidential, 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 generates a random number to be used as a
    session key for this message only.

3. The session key is encrypted using each recipient's public key.
    These "encrypted session keys" start the message.

4. The sending OpenPGP encrypts the message using the session key,
    which forms the remainder of the message.

5. The receiving OpenPGP decrypts the session key using the
    recipient's private key.

6. The receiving OpenPGP decrypts the message using the session key.
    If the message was compressed, it will be decompressed.

   With symmetric-key encryption, an object may be encrypted with a
   symmetric key derived from a passphrase (or other shared secret), or
   a two-stage mechanism similar to the public-key method described
   above in which a session key is itself encrypted with a symmetric
   algorithm keyed 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 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 hash code or message digest algorithm,
   and a public-key signature algorithm. The sequence is as follows:

   1. The sender creates a message.

   2. The sending software generates a hash code of the message.

   3. The sending software generates a signature from the hash code
       using the sender's private key.

   4. The binary signature is attached to the message.

   5. The receiving software keeps a copy of the message signature.

   6. The receiving software generates a new hash code 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

   If an implementation does not implement compression, its authors
   should be aware that most OpenPGP messages in the world are
   compressed. Thus, it may even be wise for a space-constrained
   implementation to implement decompression, but not compression.

2.4. Conversion to Radix-64

   OpenPGP's underlying native representation for encrypted messages,
   signature certificates, and keys 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 raw binary data, a printable
   encoding of these binary octets is needed. OpenPGP provides the

   service of converting the raw 8-bit binary octet stream to a stream
   of printable ASCII characters, called Radix-64 encoding or ASCII
   Armor.

   Implementations SHOULD provide Radix-64 conversions.

2.5. Signature-Only Applications

   OpenPGP is designed for applications that use both encryption and
   signatures, but there are a number of problems that are solved by 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.

3. Data Element Formats

   This section describes the data elements used by OpenPGP.

3.1. Scalar Numbers

   Scalar numbers are unsigned and are 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 (also called 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:

(all numbers 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 of 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].

Unused bits of an MPI MUST be zero.

Also note that when an MPI is encrypted, the length refers to the plaintext MPI. It may be ill-formed in its ciphertext.

3.2.1. Using MPIs to encode other data

Note that MPIs are used in some places used to encode non-integer data, such as an elliptic curve point (see Section 12.2, or an octet string of known, fixed length (see Section 12.3). 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

A Key ID is an eight-octet scalar that identifies a key.
Implementations SHOULD NOT assume that Key IDs are unique.
Section 5.5.4 describes how Key IDs 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 may
be any suitable database. It is beyond the scope of this standard to
discuss the details of keyrings or other databases.

3.7. String-to-Key (S2K) Specifiers

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

3.7.1. String-to-Key (S2K) Specifier Types

There are four types of S2K specifiers currently supported, and some
reserved values:

ID	S2K Type	Generate?	S2K field size (octets)	Reference
=====	=============	=================	================	===========
0	Simple S2K	N	2	Section

| | | | | | 3.7.1.1 |
+----+-----------+----------------+--------------+----------+
1	Salted S2K	Only when	10	Section
		string is		3.7.1.2
		high entropy		
+----+-----------+----------------+--------------+----------+
| 2 | Reserved | N | | |
| | value | | | |
+----+-----------+----------------+--------------+----------+
| 3 | Iterated and | Y | 11 | Section |
| | Salted S2K | | | 3.7.1.3 |
+----+-----------+----------------+--------------+----------+
| 4 | Argon2 | Y | 20 | Section |
| | | | | 3.7.1.4 |
+----+-----------+----------------+--------------+----------+
100	Private/	As		
to	Experimental	appropriate		
110	S2K			
+----+-----------+----------------+--------------+----------+

Table 1: S2K type registry

These are described in the subsections below.

3.7.1.1. Simple S2K

   This directly hashes the string to produce the key data. See below
   for how this hashing is done.

      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 will depend on the cipher used) 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 to say, 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 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, with any excess octets
on the right discarded.

3.7.1.2. Salted S2K

This 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

This includes both a salt and an octet count. The salt is combined
with the passphrase and the resulting value is hashed repeatedly.
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 in C, where "Int32" is a type for a 32-bit integer, and the variable "c" is the coded count, Octet 10.

Iterated-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 hashed until the number of octets specified by the octet count has been hashed. The one exception is that if the octet count is less than the size of the salt plus passphrase, the full salt plus passphrase will be hashed even though that is greater than the octet count. After the hashing is done, the data is unloaded from the hash context(s) as with the other S2K algorithms.

3.7.1.4. Argon2

This S2K method hashes the passphrase using Argon2, 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 exponent indicating the memory size m
```

The salt SHOULD be unique for each password.

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

The memory size m is 2**encoded_m kibibytes of RAM, where "encoded_m"
is the encoded memory size in Octet 19. 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 kibibytes (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, and round down otherwise
(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. String-to-Key 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 IV reuse
(see Section 5.3). Therefore, when generating S2K specifiers,
implementations MUST NOT use Simple S2K, and SHOULD NOT use Salted
S2K unless the implementation knows that the 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.

3.7.2.1. Secret-Key Encryption

An S2K specifier can be stored in the secret keyring to specify how
to convert the passphrase to a key that unlocks the secret data.
Older versions of PGP just stored a symmetric cipher algorithm octet
preceding the secret data or a zero to indicate that the secret data
was unencrypted. The MD5 hash function was always used to convert
the passphrase to a key for the specified cipher algorithm.

   For compatibility, when an S2K specifier is used, the special value
   253, 254, or 255 is stored in the position where the cipher algorithm
   octet would have been in the old data structure. This is then
   followed immediately by a one-octet algorithm identifier, and other
   fields relevant to the type of encryption used.

   Therefore, the first octet of the secret key material describes how
   the secret key data is presented. The structures differ based on the
   version of the enclosing OpenPGP packet. The tables below summarize
   the details described in Section 5.5.3.

   In the tables below, check(x) means the "2-octet checksum" meaning
   the sum of all octets in x mod 65536.

First octet	Encryption parameter fields	Encryption	Generate?
0	-	cleartext secrets || check(secrets)	Yes
Known symmetric cipher algo ID (see Section 9.3)	IV	CFB(MD5(password), secrets || check(secrets))	No
253	cipher-algo, AEAD-mode, S2K-specifier, nonce	AEAD(S2K(password), secrets, pubkey)	Yes
254	cipher-algo, S2K-specifier, IV	CFB(S2K(password), secrets || SHA1(secrets))	Yes
255	cipher-algo, S2K-specifier,	CFB(S2K(password), secrets ||	No

```
|             | IV             | check(secrets))    |          |
+-------------+---------------+--------------------+----------+
```

               Table 2: Version 4 Secret Key protection details

   Each row with "Generate?" marked as "No" is described for backward
   compatibility, and MUST NOT be generated.

   A version 5 secret key that is cryptographically protected is stored
   with an additional pair of length counts, each of which is one octet
   wide:

```
+=======+==============================+======================+
| First | Encryption parameter fields  | Encryption           |
| octet |                              |                      |
+=======+==============================+======================+
| 0     | -                            | cleartext secrets || |
|       |                              | check(secrets)       |
+-------+------------------------------+----------------------+
| 253   | params-length, cipher-algo,  | AEAD(S2K(password),  |
|       | AEAD-mode, S2K-specifier-    | secrets, pubkey)     |
|       | length, S2K-specifier, nonce |                      |
+-------+------------------------------+----------------------+
| 254   | params-length, cipher-algo,  | CFB(S2K(password),   |
|       | S2K-specifier-length, S2K-   | secrets ||           |
|       | specifier, IV                | SHA1(secrets))       |
+-------+------------------------------+----------------------+
```

               Table 3: Version 5 Secret Key protection details

   An implementation MUST NOT create and MUST reject as malformed a
   secret key packet where the S2K usage octet is anything but 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 (ESK) 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 symmetric-key ESKs and public-key ESKs. This allows a message
   to be decrypted either with a passphrase or a public-key pair.

PGP 2 always used IDEA with Simple string-to-key conversion when
encrypting a message with a symmetric algorithm. See [Section 5.7](#).
This 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
tag specifying its meaning. An OpenPGP message, keyring,
certificate, 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 truncate the
subfield at the octet boundary indicated in the packet header. Such
a truncation renders the packet 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.

Packet Headers

   The first octet of the packet header is called the "Packet Tag". It
   determines the format of the header and denotes the packet contents.
   The remainder of the packet header is the length of the packet.

   There are two packet formats, the (current) OpenPGP packet format
   specified by this document and its predecessors and the Legacy packet
   format as used by PGP 2.x implementations.

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

```
      PTag | 7 6 5 4 3 2 1 0 |

      Bit 7 -- Always one
      Bit 6 -- Always one (except for Legacy packet format)
```

   The Legacy packet format MAY be used when consuming packets to
   facilitate interoperability with legacy implementations 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.

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

   The current OpenPGP packet format packets contain:

      Bits 5 to 0 -- packet tag

   Legacy packet format packets contain:

Bits 5 to 2 -- packet tag
         Bits 1 to 0 -- length-type

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-octet, five-octet, 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, be they 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

The meaning of the length-type in Legacy format packets is:

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, this 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.

4.2.3. Packet Length Examples

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

   A packet with length 100 may have its length encoded in one octet:
   0x64. This is followed by 100 octets of data.

   A packet 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 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; 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 Tags

The packet tag denotes what type of packet the body holds. Note that
Legacy format headers can only have tags less than 16, whereas
OpenPGP format headers can have tags as great as 63. The defined
tags (in decimal) are as follows:

Tag	Critical	Packet Type
0	yes	Reserved - a packet tag MUST NOT have this value
1	yes	Public-Key Encrypted Session Key Packet
2	yes	Signature Packet
3	yes	Symmetric-Key Encrypted Session Key Packet
4	yes	One-Pass Signature Packet
5	yes	Secret-Key Packet
6	yes	Public-Key Packet
7	yes	Secret-Subkey Packet
8	yes	Compressed Data Packet
9	yes	Symmetrically Encrypted Data Packet
10	yes	Marker Packet
11	yes	Literal Data Packet
12	yes	Trust Packet
13	yes	User ID Packet
14	yes	Public-Subkey Packet
17	yes	User Attribute Packet
18	yes	Sym. Encrypted and Integrity Protected Data Packet

19	yes	Reserved (formerly Modification Detection Code Packet)
20	yes	Reserved (formerly AEAD Encrypted Data Packet)
21	yes	Padding Packet
22 to 39	yes	Unassigned Critical Packet
40 to 59	no	Unassigned Non-Critical Packet
60 to 63	no	Private or Experimental Values

Table 4: Packet type registry

4.3.1. Packet Criticality

The Packet Tag 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 11). On the other hand, an
unknown non-critical packet MUST be ignored.

Packet Tags from 0 to 39 are critical. Packet Tags from 40 to 63 are
non-critical.

5. Packet Types

5.1. Public-Key Encrypted Session Key Packets (Tag 1)

Zero or more Public-Key Encrypted Session Key (PKESK) packets and/or
Symmetric-Key Encrypted Session Key packets (Section 5.3) may precede
an encryption container (that is, a Symmetrically Encrypted Integrity
Protected Data packet or --- for historic data --- a Symmetrically
Encrypted Data 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 5. 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 11.3.2.1).

5.1.1. v3 PKESK

A version 3 Public-Key Encrypted Session Key (PKESK) packet precedes
a version 1 Symmetrically Encrypted Integrity Protected Data (v1
SEIPD, see Section 5.13.1) packet. In historic data, it is sometimes
found preceding a deprecated Symmetrically Encrypted Data packet
(SED, see Section 5.7). A v3 PKESK packet MUST NOT precede a v2
SEIPD packet (see Section 11.3.2.1).

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 5.1.6).

* 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.

When creating a v3 PKESK packet, the session key is first prefixed
with a one-octet algorithm identifier that specifies the symmetric
encryption algorithm used to encrypt the following encryption

container. Then a two-octet checksum is appended, which is equal to
the sum of the preceding session key octets, not including the
algorithm identifier, modulo 65536.

The resulting octet string (algorithm identifier, session key, and
checksum) is encrypted according to the public-key algorithm used, as
described below.

5.1.2. v5 PKESK

   A version 5 Public-Key Encrypted Session Key (PKESK) packet precedes
   a version 2 Symmetrically Encrypted Integrity Protected Data (v2
   SEIPD, see Section 5.13.2) packet. A v5 PKESK packet MUST NOT
   precede a v1 SEIPD packet or a deprecated Symmetrically Encrypted
   Data packet (see Section 11.3.2.1).

   The v5 PKESK packet consists of:

   * A one-octet version number with value 5.

   * A one octet key version number and N octets of 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 5 key N is 32. The key version number
      may also be zero, and the fingerprint omitted (that is, the length
      N is zero in this case), for an "anonymous recipient" (see
      Section 5.1.6).

   * 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.

   When creating a v5 PKESK packet, the symmetric encryption algorithm
   identifier is not included. Before encrypting, a two-octet checksum
   is appended, which is equal to the sum of the preceding session key

octets, modulo 65536.

   The resulting octet string (session key and checksum) is encrypted
   according to the public-key algorithm used, as described below.

5.1.3. Algorithm-Specific Fields for RSA encryption

   * Multiprecision integer (MPI) of RSA-encrypted value m**e mod n.

   The value "m" in the above formula is the plaintext value described
   above, encoded in the PKCS#1 block encoding EME-PKCS1-v1_5 described
   in Section 7.2.1 of [RFC8017] (see also Section 13.1). 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.

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.

   The value "m" in the above formula is the plaintext value described
   above, encoded in the PKCS#1 block encoding EME-PKCS1-v1_5 described
   in Section 7.2.1 of [RFC8017] (see also Section 13.1). 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.

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 12.5.

5.1.6. Notes on PKESK

   An implementation MAY accept or use a Key ID of all zeros, or a key

version of zero and no 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 (Tag 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 5 provide an
expandable format with subpackets that can specify more information
about the signature.

An implementation MUST generate a version 5 signature when signing
with a version 5 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.

5.2.1. Signature Types

There are a number of possible meanings for a signature, which are
indicated in a signature type octet in any given signature. Please
note that the vagueness of these meanings is not a flaw, but 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.

These meanings are as follows:

0x00: Signature of a binary document.

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

0x01: Signature of a canonical text document.
   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>.

0x02: Standalone signature.
   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.

0x10: Generic certification of a User ID and Public-Key packet.
   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.

0x11: Persona certification of a User ID and Public-Key packet.
   The issuer of this certification has not done any verification of
   the claim that the owner of this key is the User ID specified.

0x12: Casual certification of a User ID and Public-Key packet.
   The issuer of this certification has done some casual verification
   of the claim of identity.

0x13: Positive certification of a User ID and Public-Key packet.
   The issuer of this certification has done substantial verification
   of the claim of identity.

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

0x18: Subkey Binding Signature.
   This signature is a statement by the top-level signing key that
   indicates 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.

0x19: Primary Key Binding Signature.
   This signature is a statement by a signing subkey, indicating that
   it is owned by the primary key and subkey. This signature is
   calculated the same way as a 0x18 signature: directly on the
   primary key and subkey, and not on any User ID or other packets.

0x1F: Signature directly on a key.
   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 (deprecated)
   Revocation Key. 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.

0x20: Key revocation signature.
   The 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.

0x28: Subkey revocation signature.
   The signature is calculated directly on 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.

0x30: Certification revocation signature.
   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 certificate that
   it revokes, and should have a later creation date than that
   certificate.

0x40: Timestamp signature.

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

   0x50: Third-Party Confirmation signature.
      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 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.

5.2.2. Version 3 Signature Packet Format

   The body of a version 3 Signature Packet contains:

   * One-octet version number (3).

   * One-octet length of following hashed material. MUST be 5.

      - One-octet signature type.

      - Four-octet creation time.

   * Eight-octet Key ID of signer.

   * One-octet public-key algorithm.

   * One-octet hash algorithm.

   * Two-octet field holding left 16 bits of signed hash value.

   * One or more multiprecision integers comprising the signature.
      This portion is algorithm-specific, as described below.

   The concatenation of the data to be signed, the signature type, and
   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:

   * Multiprecision integer (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.

   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]. This
   requires inserting the hash value as an octet string into an ASN.1
   structure. The object identifier for the type of hash being used is
   included in the structure. The hexadecimal representations for the
   currently defined hash algorithms are as follows:

algorithm	hexadecimal representation
=============	==
MD5	0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x02, 0x05
RIPEMD-160	0x2B, 0x24, 0x03, 0x02, 0x01
SHA-1	0x2B, 0x0E, 0x03, 0x02, 0x1A
SHA224	0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x04
SHA256	0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01
SHA384	0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x02
SHA512	0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x03

                 Table 5: Hash hexadecimal representations

   The ASN.1 Object Identifiers (OIDs) are as follows:

```
+============+========================+
| algorithm  | OID                    |
+============+========================+
| MD5        | 1.2.840.113549.2.5     |
+------------+------------------------+
| RIPEMD-160 | 1.3.36.3.2.1           |
+------------+------------------------+
| SHA-1      | 1.3.14.3.2.26          |
+------------+------------------------+
| SHA224     | 2.16.840.1.101.3.4.2.4 |
+------------+------------------------+
| SHA256     | 2.16.840.1.101.3.4.2.1 |
+------------+------------------------+
| SHA384     | 2.16.840.1.101.3.4.2.2 |
+------------+------------------------+
| SHA512     | 2.16.840.1.101.3.4.2.3 |
+------------+------------------------+
```

Table 6: Hash OIDs

The full hash prefixes for these are as follows:

```
+============+==========================================+
| algorithm  | full hash prefix                         |
+============+==========================================+
| MD5        | 0x30, 0x20, 0x30, 0x0C, 0x06, 0x08,      |
|            | 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D,      |
|            | 0x02, 0x05, 0x05, 0x00, 0x04, 0x10       |
+------------+------------------------------------------+
| RIPEMD-160 | 0x30, 0x21, 0x30, 0x09, 0x06, 0x05,      |
|            | 0x2B, 0x24, 0x03, 0x02, 0x01, 0x05,      |
|            | 0x00, 0x04, 0x14                         |
+------------+------------------------------------------+
| SHA-1      | 0x30, 0x21, 0x30, 0x09, 0x06, 0x05,      |
|            | 0x2B, 0x0E, 0x03, 0x02, 0x1A, 0x05,      |
|            | 0x00, 0x04, 0x14                         |
+------------+------------------------------------------+
| SHA224     | 0x30, 0x2D, 0x30, 0x0D, 0x06, 0x09,      |
|            | 0x60, 0x86, 0x48, 0x01, 0x65, 0x03,      |
|            | 0x04, 0x02, 0x04, 0x05, 0x00, 0x04, 0x1C |
+------------+------------------------------------------+
| SHA256     | 0x30, 0x31, 0x30, 0x0D, 0x06, 0x09,      |
|            | 0x60, 0x86, 0x48, 0x01, 0x65, 0x03,      |
|            | 0x04, 0x02, 0x01, 0x05, 0x00, 0x04, 0x20 |
+------------+------------------------------------------+
| SHA384     | 0x30, 0x41, 0x30, 0x0D, 0x06, 0x09,      |
|            | 0x60, 0x86, 0x48, 0x01, 0x65, 0x03,      |
|            | 0x04, 0x02, 0x02, 0x05, 0x00, 0x04, 0x30 |
+------------+------------------------------------------+
| SHA512     | 0x30, 0x51, 0x30, 0x0D, 0x06, 0x09,      |
|            | 0x60, 0x86, 0x48, 0x01, 0x65, 0x03,      |
```

```
     |              | 0x04, 0x02, 0x03, 0x05, 0x00, 0x04, 0x40 |
     +-------------+--------------------------------------------+
```

                   Table 7: Hash hexadecimal prefixes

   DSA signatures MUST use hashes that are equal in size to 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.

5.2.3. Version 4 and 5 Signature Packet Formats

   The body of a v4 or v5 Signature packet contains:

   * One-octet version number. This is 4 for v4 signatures and 5 for
      v5 signatures.

   * One-octet signature type.

   * One-octet public-key algorithm.

   * One-octet hash algorithm.

   * A scalar octet count for following hashed subpacket data. For a
      v4 signature, this is a two-octet field. For a v5 signature, this
      is a four-octet field. Note that this is the length in octets of
      all of the hashed subpackets; a pointer incremented by this number
      will skip over the hashed subpackets.

   * Hashed subpacket data set (zero or more subpackets).

   * A scalar octet count for the following unhashed subpacket data.
      For a v4 signature, this is a two-octet field. For a v5
      signature, this is a four-octet field. Note that this is the
      length in octets of all of the unhashed subpackets; a pointer
      incremented by this number will skip over the unhashed subpackets.

* Unhashed subpacket data set (zero or more subpackets).

   * Two-octet field holding the left 16 bits of the signed hash value.

   * Only for v5 signatures, a 16 octet field containing random values
      used as salt.

   * One or more multiprecision integers comprising the signature.
      This portion is algorithm-specific:

5.2.3.1. Algorithm-Specific Fields for RSA signatures

   * Multiprecision integer (MPI) of RSA signature value m**d mod n.

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
   ECDSA.

5.2.3.3. Algorithm-Specific Fields for EdDSA signatures

   * 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
   EdDSA.

5.2.3.3.1. Algorithm-Specific Fields for Ed25519 signatures

   The two MPIs for Ed25519 use octet strings R and S as described in
   [RFC8032].

   * MPI of an EC point R, represented as a (non-prefixed) native
      (little-endian) octet string up to 32 octets.

   * MPI of EdDSA value S, also in (non-prefixed) native little-endian

format with a length up to 32 octets.

5.2.3.3.2. Algorithm-Specific Fields for Ed448 signatures

   For Ed448 signatures, the native signature format is used as
   described in [RFC8032]. The two MPIs are composed as follows:

   * The first MPI has a body of 115 octets: a prefix 0x40 octet,
      followed by 114 octets of the native signature.

   * The second MPI is set to 0 (this is a placeholder, and is unused).
      Note that an MPI with a value of 0 is encoded on the wire as a
      pair of zero octets: 00 00.

5.2.3.4. Notes on Signatures

   The concatenation of the data being signed and the signature data
   from the version number through the hashed subpacket data (inclusive)
   is 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.

   There are two fields consisting of Signature subpackets. The first
   field is hashed with the rest of the signature data, while the second
   is unhashed. The second set of subpackets is not cryptographically
   protected by the signature and should include only advisory
   information.

   The differences between a v4 and v5 signature are two-fold: first, a
   v5 signature increases the width of the size indicators for the
   signed data, making it more capable when signing large keys or
   messages. Second, the hash is salted with 128 bit of random data
   (see Section 14.2.

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

. 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 v5 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 subpacket type (1 octet),

   and is followed by the subpacket-specific data.

   The length includes the type octet but not this length. Its format
   is similar to the "new" format packet header lengths, but 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]

   The value of the subpacket type octet may be:

        +===========+==+

Type	Description
========	===
0	Reserved
1	Reserved
2	Signature Creation Time
3	Signature Expiration Time
4	Exportable Certification
5	Trust Signature
6	Regular Expression
7	Revocable
8	Reserved
9	Key Expiration Time
10	Placeholder for backward compatibility
11	Preferred Symmetric Ciphers for v1 SEIPD
12	Revocation Key (deprecated)
13 to 15	Reserved
16	Issuer Key ID
17 to 19	Reserved
20	Notation Data
21	Preferred Hash Algorithms
22	Preferred Compression Algorithms
23	Key Server Preferences
24	Preferred Key Server
25	Primary User ID
26	Policy URI

```
+------------+------------------------------------+
|         27 | Key Flags                          |
+------------+------------------------------------+
|         28 | Signer's User ID                   |
+------------+------------------------------------+
|         29 | Reason for Revocation              |
+------------+------------------------------------+
|         30 | Features                           |
+------------+------------------------------------+
|         31 | Signature Target                   |
+------------+------------------------------------+
|         32 | Embedded Signature                 |
+------------+------------------------------------+
|         33 | Issuer Fingerprint                 |
+------------+------------------------------------+
|         34 | Reserved                           |
+------------+------------------------------------+
|         35 | Intended Recipient Fingerprint     |
+------------+------------------------------------+
|         37 | Reserved (Attested Certifications) |
+------------+------------------------------------+
|         38 | Reserved (Key Block)               |
+------------+------------------------------------+
|         39 | Preferred AEAD Ciphersuites        |
+------------+------------------------------------+
| 100 to 110 | Private or experimental            |
+------------+------------------------------------+
```

                   Table 8: Subpacket type registry

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

   Bit 7 of the subpacket type 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 software, the
   evaluator SHOULD consider the signature to be in error.

   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 than be ignored.

Implementations SHOULD implement the four preferred algorithm
subpackets (11, 21, 22, and 34), as well as the "Reason for
Revocation" subpacket. Note, however, that if an implementation
chooses not to implement some of the preferences, it is required to
behave in a polite manner to respect the wishes of those users who do
implement these preferences.

5.2.3.6. Signature Subpacket Types

A number of subpackets are currently defined. 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 either in the hashed or 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.

5.2.3.7. 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 (types 0x10-0x13), the direct-key signature
(type 0x1F), and the subkey binding signature (type 0x18). A
cryptographically-valid self-signature should be accepted from any
primary key, regardless of what Key Flags (Section 5.2.3.26) apply to
the primary key. In particular, a primary key does not need to have
0x01 set in the first octet of 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 in fact has a self-
signature. Subpackets that appear in a certification self-signature
apply to the user name, 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.

Implementing software 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 GCM.
If the software locates this key via Alice's name, then the preferred
AEAD ciphersuite is AES-256 with OCB; if software 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
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 5.2.3.28 for more relevant detail.

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.

It is good practice to verify that a self-signature imported into an
implementation doesn't advertise features that the implementation
doesn't support, rewriting the signature as appropriate.

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. Some implementations
require at least one User ID with a valid self-signature to be
present to use a v4 key. For this reason, it is RECOMMENDED to
include at least one User ID with a self-signature in v4 keys.

For version 5 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 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 v5 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 e-mail 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.

5.2.3.8. Signature Creation Time

(4-octet time field)

The time the signature was made.

MUST be present in the hashed area.

[5.2.3.9](#). 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 5.2.3.32](#)) instead.

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

[5.2.3.10](#). 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.

[5.2.3.11](#). Preferred Symmetric Ciphers for v1 SEIPD

   (array of one-octet values)

   A series of symmetric cipher algorithm identifiers 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 algorithms listed are supported by the recipient's
   software. Algorithm numbers are 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 5.2.3.12](#) instead.

[5.2.3.12](#). Preferred AEAD Ciphersuites

   (array of pairs of octets indicating Symmetric Cipher and AEAD

algorithms)

A series of paired algorithm identifiers 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 key
holder prefers to use. The symmetric cipher identifier precedes the
AEAD identifier 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 software, 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 numbers are listed in Section 9.6. Symmetric cipher
algorithm numbers are listed in Section 9.3.

For example, a subpacket with 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 Feature Flag (Section 5.2.3.29).

This subpacket is only found on a self-signature.

When generating a v1 SEIPD packet, this preference list is not
relevant. See Section 5.2.3.11 instead.

5.2.3.13. Preferred Hash Algorithms

(array of one-octet values)

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

5.2.3.14. Preferred Compression Algorithms

    (array of one-octet values)

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

5.2.3.15. 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.

5.2.3.16. Exportable Certification

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

    This subpacket denotes whether a certification signature is
    "exportable", to be used by other users 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, 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.

5.2.3.17. Revocable

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

Signature's revocability status. The packet body contains a Boolean flag indicating whether the signature is revocable. Signatures that are not revocable have any later revocation signatures ignored. They represent a commitment by the signer that he cannot revoke his signature for the life of his key. If this packet is not present, the signature is revocable.

5.2.3.18. Trust Signature

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

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.

5.2.3.19. Regular Expression

   (null-terminated 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 the Henry Spencer's
   "almost public domain" regular expression [REGEX] package. A
   description of the syntax is found in Section 8.

5.2.3.20. Revocation Key

   (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
   SHOULD instead use an escrowed Revocation Signature. See
   Section 14.8 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. Class octet must have bit 0x80
   set. If the bit 0x40 is set, then this means that 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 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: 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.

5.2.3.21. 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	Shorthand	Description	Security Recommended	Interoperability Recommended	Reference
0x80 0x00 0x00 0x00	human-readable	Notation value is text.	No	Yes	This document

   Table 9: Signature Notation Data Subpacket Notation Flag 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

      Table 10: Signature Notation Data Subpacket registry

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.

5.2.3.22. Key Server Preferences

(N octets of flags)

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

First octet:

```
   +======+===========+==============================================+
   | flag | shorthand | definition                                   |
   +======+===========+==============================================+
   | 0x80 | No-modify | The key holder requests that this key only   |
   |      |           | be modified or updated by the key holder     |
   |      |           | or an administrator of the key server.       |
   +------+-----------+----------------------------------------------+
```

        Table 11: Key server preferences flag registry (first octet)

This is found only on a self-signature.

5.2.3.23. Preferred Key Server

(String)

This is a URI of a key server that the key holder 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 ftp, http,
finger, etc.

5.2.3.24. 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, etc. 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 to say, there are two different and
independent "primaries" --- one for User IDs, and one for User
Attributes.

5.2.3.25. Policy URI

(String)

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

5.2.3.26. 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. This is so 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:

First octet:

```
+======+=========================================================+
| flag | definition                                              |
+======+=========================================================+
| 0x01 | This key may be used to make User ID certifications     |
|      | (signature types 0x10-0x13) or direct-key               |
|      | signatures (signature type 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.                |
+------+----------------------------------------------------+
```

                Table 12: Key flags registry (first octet)

    Second octet:

                    +======+==========================+
                    | flag | definition |
                    +======+==========================+
                    | 0x04 | Reserved (ADSK). |
                    +------+--------------------------+
                    | 0x08 | Reserved (timestamping). |
                    +------+--------------------------+

                        Table 13: Key flags registry
                               (second octet)

    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. Note
    however, that it is a thorny issue to determine what is
    "communications" and what is "storage". 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 0x1F) or a subkey signature (type 0x18), one that refers to the
    key the flag applies to.

5.2.3.27. 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.

5.2.3.28. 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
certificate 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 14: Reasons for revocation

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
revocation SHOULD include a 0x20 code.

Note that any signature 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.

. 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 backwards-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.

   Defined features are as follows:

   First octet:

```
      +=========+====================================+===========+
      | Feature | Definition                         | Reference |
      +=========+====================================+===========+
      | 0x01    | Symmetrically Encrypted Integrity  | Section   |
      |         | Protected Data packet version 1    | 5.13.1    |
      +---------+------------------------------------+-----------+
      | 0x02    | Reserved                           |           |
      +---------+------------------------------------+-----------+
      | 0x04    | Reserved                           |           |
      +---------+------------------------------------+-----------+
      | 0x08    | Symmetrically Encrypted Integrity  | Section   |
      |         | Protected Data packet version 2    | 5.13.2    |
      +---------+------------------------------------+-----------+
```

                    Table 15: Features registry

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

   An implementation may freely infer features from other suitable
   implementation-dependent mechanisms.

   See Section 14.7 for details about how to use the Features subpacket
   when generating encryption data.

5.2.3.30. 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 hash of the
   signature. For example, a target signature with a SHA-1 hash MUST
   have 20 octets of hash data.

5.2.3.31. 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.

5.2.3.32. 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 5.2.3.9)
   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 5 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).

5.2.3.33. 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 14.11).

   Note that the length N of the fingerprint for a version 4 key is 20
   octets; for a version 5 key N is 32.

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

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 a v5 signature is made, the salt is hashed first.

   For binary document signatures (type 0x00), the document data is
   hashed directly. For text document signatures (type 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 bytestream 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, and then body
   of the key packet. When a v5 signature is made over a key, the hash
   data starts with the octet 0x9a, followed by a four-octet length of
   the key, and then body of the key packet.

   A subkey binding signature (type 0x18) or primary key binding
   signature (type 0x19) then hashes the subkey using the same format as
   the main key (also using 0x99 or 0x9a as the first octet). Primary

key revocation signatures (type 0x20) hash only the key being
revoked. Subkey revocation signature (type 0x28) hash first the
primary key and then the subkey being revoked.

A certification signature (type 0x10 through 0x13) hashes the User ID
being bound to the key into the hash context after the above data. A
v3 certification hashes the contents of the User ID or attribute
packet packet, without any header. A v4 or v5 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, and then the User
ID or User Attribute data.

When a signature is made over a Signature packet (type 0x50, "Third-
Party Confirmation signature"), the hash data starts with 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 time.

* A v4 or v5 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, 0x05
      for v5),

   - 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, 0x05 for v5)

- A single octet 0xFF,

- A number representing the length of the hashed data from the
  Signature packet stopping right before the second version
  octet. For a v4 signature, this is a four-octet big-endian
  number, considered to be an unsigned integer modulo 2^{32}. For
  a v5 signature, this is an eight-octet big-endian number,
  considered to be an unsigned integer modulo 2^{64}.

After all this has been hashed in a single hash context, the
resulting hash field is used in the signature algorithm and placed at
the end of the Signature packet.

5.2.4.1. Subpacket Hints

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
signature, but 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 5.2.3.9) for each key, as a way of
explicitly tying those keys to the signature.

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 5.2.3.5](#)) 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 known-weak hash algorithm (e.g. MD5)

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 Packets (Tag 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 may precede a 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 5. The remainder of the packet depends on the version.

The versions differ in how they encrypt the session key with the
password, and in what they encode. The version of the SKESK packet
must align with the version of the SEIPD packet (see
[Section 11.3.2.1](#)).

[5.3.1](#). v4 SKESK

A version 4 Symmetric-Key Encrypted Session Key (SKESK) packet
precedes a version 1 Symmetrically Encrypted Integrity Protected Data
(v1 SEIPD, see [Section 5.13.1](#)) packet. In historic data, it is
sometimes found preceding a deprecated Symmetrically Encrypted Data
packet (SED, see [Section 5.7](#)). A v4 SKESK packet MUST NOT precede a
v2 SEIPD packet (see [Section 11.3.2.1](#)).

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.

   * A string-to-key (S2K) specifier. The length of the string-to-key
      specifier depends on its type (see [Section 3.7.1](#)).

*   Optionally, the encrypted session key itself, which is decrypted
        with the string-to-key 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, either a Salted S2K, an Iterated-
    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](). v5 SKESK

    A version 5 Symmetric-Key Encrypted Session Key (SKESK) packet
    precedes a version 2 Symmetrically Encrypted Integrity Protected Data
    (v2 SEIPD, see [Section 5.13.2]()) packet. A v5 SKESK packet MUST NOT
    precede a v1 SEIPD packet or a deprecated Symmetrically Encrypted
    Data packet (see [Section 11.3.2.1]()).

    A version 5 Symmetric-Key Encrypted Session Key packet consists of:

    *   A one-octet version number with value 5.

    *   A one-octet scalar octet count of the following 5 fields.

    *   A one-octet symmetric cipher algorithm identifier.

    *   A one-octet AEAD algorithm identifier.

    *   A one-octet scalar octet count of the following field.

* A string-to-key (S2K) specifier. The length of the string-to-key
      specifier depends on its type (see [Section 3.7.1](#)).

   * A starting initialization vector of size specified by the AEAD
      algorithm.

   * The encrypted session key itself.

   * An authentication tag for the AEAD mode.

   HKDF is used with SHA256 as hash algorithm, the key derived from S2K
   as Initial Keying Material (IKM), no salt, and the Packet Tag in new
   format encoding (bits 7 and 6 set, bits 5-0 carry the packet tag),
   the packet version, and the cipher-algo and AEAD-mode used to encrypt
   the key material, are used as info parameter. 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 Tag in OpenPGP format
   encoding (bits 7 and 6 set, bits 5-0 carry the packet tag), the
   packet version number, the cipher algorithm octet, and the AEAD
   algorithm octet are given as additional data. For example, the
   additional data used with AES-128 with OCB consists of the octets
   0xC3, 0x05, 0x07, and 0x02.

5.4. One-Pass Signature Packets (Tag 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 5.

   * A one-octet signature type. 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 v5 packets, a 16 octet field containing random values
      used as salt. 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 v5 packets, a one octet key version number and N octets
      of the fingerprint of the signing key. Note that the length N of
      the fingerprint for a version 5 key is 32. Since a v5 signature
      can only be made by a v5 key, the key version number MUST be 5.
      An application that encounters a v5 One-Pass Signature packet
      where the key version number is not 5 MUST treat the signature as
      invalid (see [Section 5.2.5](#)).

   * 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):

   +====================+====================+================+
   | Signing key version | OPS packet version | Signature |
   | | | packet version |
   +====================+====================+================+
   | 4 | 3 | 4 |
   +--------------------+--------------------+----------------+
   | 5 | 5 | 5 |
   +--------------------+--------------------+----------------+

         Table 16: Versions of packets used in a one-pass signature

   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 Packet

   A key material packet contains all the information about a public or
   private key. There are four variants of this packet type, and two
   major versions. Consequently, this section is complex.

[5.5.1](#). Key Packet Variants

[5.5.1.1](#). Public-Key Packet (Tag 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 (Tag 14)

   A Public-Subkey packet (tag 14) has exactly the same format as a
   Public-Key packet, but denotes a subkey. One or more subkeys may be
   associated with a top-level key. By convention, the top-level key
   provides signature services, and the subkeys provide encryption
   services.

[5.5.1.3](#). Secret-Key Packet (Tag 5)

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

[5.5.1.4](#). Secret-Subkey Packet (Tag 7)

   A Secret-Subkey packet (tag 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 three versions of key-material packets.

   OpenPGP implementations SHOULD create keys with version 5 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.

   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 this key is
       valid. If this number is zero, then it does not expire.

   *   A one-octet number denoting the public-key algorithm of this key.

   *   A series of multiprecision integers comprising the key material:

       -   a multiprecision integer (MPI) of RSA public modulus n;

       -   an 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. Secondly, 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 make developers prefer other algorithms. See
   Section 5.5.4 for a fuller discussion of Key IDs and fingerprints.

   V2 keys are identical to the deprecated v3 keys except for the
   version number.

   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 this key.

   *   A series of values comprising the key material. This is

algorithm-specific and described in [Section 5.5.5](#).

   The version 5 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 5 keys are calculated differently from
   version 4 keys, as described in [Section 5.5.4](#).

   A version 5 packet contains:

   * A one-octet version number (5).

   * A four-octet number denoting the time that the key was created.

   * A one-octet number denoting the public-key algorithm of this 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 indicating string-to-key usage conventions. Zero
      indicates that the secret-key data is not encrypted. 255, 254, or
      253 indicates that a string-to-key specifier is being given. Any
      other value is a symmetric-key encryption algorithm identifier. A
      version 5 packet MUST NOT use the value 255.

   * Only for a version 5 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 optional string-to-key parameter fields.

   * [Optional] If string-to-key usage octet was 255, 254, or 253, a
     one-octet symmetric encryption algorithm.

   * [Optional] If string-to-key usage octet was 253, a one-octet AEAD
     algorithm.

   * [Optional] Only for a version 5 packet, and if string-to-key usage
     octet was 255, 254, or 253, an one-octet count of the following
     field.

   * [Optional] If string-to-key usage octet was 255, 254, or 253, a
     string-to-key (S2K) specifier. The length of the string-to-key
     specifier depends on its type (see [Section 3.7.1](#)).

   * [Optional] If string-to-key usage octet was 253 (that is, the
     secret data is AEAD-encrypted), an initialization vector (IV) of
     size specified by the AEAD algorithm (see [Section 5.13.2](#)), which
     is used as the nonce for the AEAD algorithm.

   * [Optional] If string-to-key usage octet was 255, 254, or a cipher
     algorithm identifier (that is, the secret data is CFB-encrypted),
     an initialization vector (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, then an
     AEAD authentication tag is part of that data. If the string-to-
     key usage octet is 254, 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 or
     another nonzero 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.)

   * If the string-to-key usage octet is zero, then 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 [Section 3.7.2.1](#).

    Note that the version 5 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, that describes the algorithm for
    converting the passphrase to a key, else a simple MD5 hash of the
    passphrase is used. Implementations MUST use a string-to-key
    specifier; the simple hash is for backward compatibility and is
    deprecated, 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 v5 keys, a simpler method is used. All secret MPI values
    are encrypted, including the MPI bitcount prefix.

    If the string-to-key usage octet is 253, the key encryption key is
    derived using HKDF (see [[RFC5869](#)]) to provide key separation. HKDF
    is used with SHA256 as hash algorithm, the key derived from S2K as

    Initial Keying Material (IKM), no salt, and the Packet Tag in OpenPGP
    format encoding (bits 7 and 6 set, bits 5-0 carry the packet tag),
    the packet version, and the cipher-algo and AEAD-mode used to encrypt
    the key material, are used as info parameter. 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 Tag in OpenPGP format encoding (bits 7 and 6 set,
bits 5-0 carry the packet tag), 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 5
key packet, the second octet would be 0x05, 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 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; an implementation SHOULD NOT use it, but should rather
use the SHA-1 hash denoted with a usage octet of 254. The reason for
this is that there are some attacks that involve undetectably
modifying the secret key. 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 as unusable any secret key material whose
cleartext length does not align with the lengths of the unwrapped MPI
objects.

5.5.4. Key IDs and Fingerprints

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.

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, with the example of an EdDSA 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): 22 = EdDSA (example);

e) Algorithm-specific fields.

Algorithm-Specific Fields for EdDSA keys (example):

e.1) A one-octet size of the following field;

e.2) The octets representing a curve OID, defined in Section 9.2;

e.3) An MPI of an EC point representing a public key Q in prefixed native form (see Section 12.2.2).

A v5 fingerprint is the 256-bit SHA2-256 hash of the octet 0x9A, 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, with the example of an EdDSA key:

a.1) 0x9A (1 octet)

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

b) version number = 5 (1 octet);

c) timestamp of key creation (4 octets);

d) algorithm (1 octet): 22 = EdDSA (example);

   e) four-octet scalar octet count for the following key material;

   f) algorithm-specific fields.

   Algorithm-Specific Fields for EdDSA keys (example):

   f.1) A one-octet size of the following field;

   f.2) The octets representing a curve OID, defined in Section 9.2;

   f.3) An MPI of an EC point representing a public key Q in prefixed
   native form (see Section 12.2.2).

   Note that it is possible for there to be collisions of Key IDs ---
   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 v5 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 0x9A
   (v5 key) as the first octet (even though this is not a valid packet
   ID for a public subkey).

5.5.5. Algorithm-specific Parts of Keys

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

5.5.5.1. Algorithm-Specific Part 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);

   * MPI of u, the multiplicative inverse of p, mod q.

5.5.5.2. Algorithm-Specific Part 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;

   * 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

   The public key is this series of multiprecision integers:

   * MPI of Elgamal prime p;

   * MPI of Elgamal group generator g;

   * 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

   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 (defined in [Section 9.2](#));

   * MPI of an EC point representing a public key.

   The secret key is this single multiprecision integer:

   * 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 EdDSA Keys

   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, defined in [Section 9.2](#);

   * An MPI of an EC point representing a public key Q in prefixed
     native form (see [Section 12.2.2](#)).

   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 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 [[RFC8032](#)] for more
   details about how to use the native octet strings ([section 5.1.5](#) for
   Ed25519 and 5.2.5 for Ed448). The value stored in an OpenPGP EdDSA
   secret key packet is the original sequence of random octets.

Note that a ECDH secret key over the equivalent curve instead stores
the curve-specific secret scalar itself, rather than the sequence of
random octets stored in an EdDSA secret key.

5.5.5.6. Algorithm-Specific Part 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,

    -   Octets representing a curve OID, defined in Section 9.2;

*   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 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 used to
        wrap the symmetric key used for the message encryption; see
        Section 12.5 for details.

Observe that an ECDH public key is composed of the same sequence of
fields that define an ECDSA key plus the KDF parameters field.

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 P-256, P-384, or P-521 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. Curve25519 ECDH Secret Key Material

   A Curve25519 secret key is stored as a standard integer in big-endian
   MPI form. 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 Curve25519 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 Curve25519 secret key from 32 fully-random
   octets, the following pseudocode produces the MPI wire format (note
   the similarity to decodeScalar25519 from [RFC7748]):

```
   def curve25519_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)
```

5.5.5.6.1.2. X448 ECDH Secret Key Material

   An X448 secret key is contained within its MPI as a prefixed octet
   string (see Section 12.3.2), which encapsulates the native secret key
   format found in [RFC7748]. The full wire format (as an MPI) will
   thus be the three octets 01 c7 40 followed by the full 56 octet
   native secret key.

   When generating a new X448 secret key from 56 fully-random octets,
   the following pseudocode produces the MPI wire format:

```
def X448_MPI_from_random(octet_list):
    prefixed_header = [ 0x01, 0xc7, 0x40 ]
    return prefixed_header || octet_list
```

5.6. Compressed Data Packet (Tag 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 an block that compresses
some set of packets. See Section 11 for details on how messages are
formed.

ZIP-compressed packets are compressed with raw [RFC1951] DEFLATE
blocks.

ZLIB-compressed packets are compressed with [RFC1950] ZLIB-style
blocks.

BZip2-compressed packets are 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 MAY interpret Compressed Data
packets that use the Legacy format for packet framing.

5.7. Symmetrically Encrypted Data Packet (Tag 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 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 MAY process such a packet but it MUST
return a clear diagnostic that a non-integrity protected packet has
been processed. The implementation SHOULD also return an error in
this case and stop processing.

This packet format is impossible to handle safely in general because
the ciphertext it provides is malleable. See Section 14.7 about
selecting a better OpenPGP encryption container that does not have
this flaw.

The body of this packet consists of:

*   Encrypted data, the output of the selected symmetric-key cipher
    operating in OpenPGP's variant of Cipher Feedback (CFB) mode.

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 octet 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, with a CFB shift size equal to the
cipher's block size. The Initial Vector (IV) is specified as all
zeros. Instead of using an IV, OpenPGP prefixes a string of length
equal to the block size of the cipher plus two to the data before it
is encrypted. The first block-size octets (for example, 8 octets for
a 64-bit block length) are random, and the following two octets are
copies of the last two octets of the IV. For example, in an 8-octet

block, octet 9 is a repeat of octet 7, and octet 10 is a repeat of
octet 8. In a cipher of length 16, octet 17 is a repeat of octet 15
and octet 18 is a repeat of octet 16. As a pedantic clarification,
in both these examples, we consider the first octet to be numbered 1.

After encrypting the first block-size-plus-two octets, 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 16 bits in the random data prefixed to the message
allows the receiver to immediately check whether the session key is
incorrect. See Section 14.4 for hints on the proper use of this
"quick check".

5.8. Marker Packet (Tag 10)

The body of this 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 (Tag 11)

A Literal Data packet contains the body of a message; 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.

   Older versions of OpenPGP 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.

   Early versions of PGP also defined a value of l as a 'local' mode
   for machine-local conversions. [RFC1991] incorrectly stated this
   local mode flag as 1 (ASCII numeral one). Both of these local
   modes are deprecated.

   * 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, this 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
      software.

   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 and thus 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 either when 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
   SHOULD 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 legacy implementations 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 (Tag 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 key holders are trustworthy introducers,
   along with other information that implementing software 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 (Tag 13)

   A User ID packet consists of UTF-8 text that is intended to represent
   the name and email address of the key holder. By convention, it
   includes an [RFC2822] mail name-addr, 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 (Tag 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
standard, 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 (1 octet)

and is followed by the subpacket specific data.

The following table lists the currently known subpackets:

```
             +=========+===========================+
             |    Type | Attribute Subpacket       |
             +=========+===========================+
             |       1 | Image Attribute Subpacket |
             +---------+---------------------------+
             | 100-110 | Private/Experimental Use  |
             +---------+---------------------------+
```

                Table 17: User Attribute type registry

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

5.12.1. The 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.

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 types 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.

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. Sym. Encrypted Integrity Protected Data Packet (Tag 18)

This packet contains integrity protected and encrypted data. When it has been decrypted, it will contain other packets forming an OpenPGP Message (see Section 11.3).

The first octet of this packet is always used to indicate the version number, but different versions contain differently-structured ciphertext. Version 1 of this packet contains data encrypted with a

symmetric-key algorithm and protected against modification by the
SHA-1 hash algorithm. This is a legacy OpenPGP mechanism that offers
some protections against ciphertext malleability.

Version 2 of this packet contains data encrypted with an
authenticated encryption and additional data (AEAD) construction.
This offers a more cryptographically rigorous defense against
ciphertext malleability, but may not be as widely supported yet. See
Section 14.7 for more details on choosing between these formats.

5.13.1. Version 1 Sym. 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 Cipher Feedback mode with shift amount equal to the
   block size of the cipher (CFB-n where n is the block size).

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 octet is prefixed to the session key
before it is encrypted.

The data is encrypted in CFB mode, with a CFB shift size equal to the
cipher's block size. The Initial Vector (IV) is specified as all
zeros. Instead of using an IV, OpenPGP prefixes an octet string to
the data before it is encrypted. 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 Symmetrically Encrypted Data Packet, no
special CFB resynchronization is done after encrypting this prefix
data. See Section 13.9 for more details.

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.

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 Modification Detection Code (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.

   It is a limitation of CFB encryption that 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 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 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 would be an attack that

replaced 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 Sym. 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.

*   A one-octet AEAD algorithm.

*   A one-octet chunk size.

*   Thirty-two octets of salt. The salt is used to derive the message
    key and must be unique.

*   Encrypted data, 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 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 left-most M bits are used as symmetric algorithm
key, the remaining N - 64 bits are used as initialization vector.
HKDF is used with SHA256 as hash algorithm, the session key as
Initial Keying Material (IKM), the salt as salt, and the Packet Tag
in OpenPGP format encoding (bits 7 and 6 set, bits 5-0 carry the
packet tag), version number, cipher algorithm octet, AEAD algorithm
octet, 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 recipients certificate is unknown by reusing the encrypted session key packets and replying with a different salt yielding a new, unique message key.

A v2 SEIPD packet consists of one or more chunks of data. The plaintext of each chunk is of a size specified using 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; it is nevertheless followed by a full authentication tag.

For each chunk, the AEAD construction is given the Packet Tag in OpenPGP format encoding (bits 7 and 6 set, bits 5-0 carry the packet tag), version number, cipher algorithm octet, AEAD algorithm octet, 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**16 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 (in C), where c is the chunk size octet:

   chunk_size = ((uint64_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 left-most N - 64 bits are the initialization vector derived using HKDF. The right-most 64 bits are the chunk index as 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 (Tag 21)

   The Padding packet contains random data, and can be used to defend
   against traffic analysis (see Section 14.10) on version 2 SEIPD
   messages (see Section 5.13.2) and Transferable Public Keys (see
   Section 11.1).

   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 v5 Transferable Public Key that is transferred
      over an encrypted channel (see Section 11.1).

   * As the last packet of an Optionally Padded Message within a
      version 2 Symmetrically Encrypted Integrity Protected Data Packet
      (see Section 11.3.1).

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. Radix-64 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 standard specifies one such printable encoding scheme to ensure interoperability.

OpenPGP's Radix-64 encoding is composed of two parts: a base64 encoding of the binary data and an optional checksum. The base64 encoding is identical to the MIME base64 content-transfer-encoding [RFC2045].

6.1. Optional checksum

The optional checksum is a 24-bit Cyclic Redundancy Check (CRC) converted to four characters of radix-64 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 radix-64, 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
context or by the OpenPGP object being encoded that the consuming
implementation accepts Radix-64 encoded blocks without CRC24 footer.
Notably:

*   An ASCII-armored Encrypted Message packet sequence that ends in an
    v2 SEIPD packet MUST NOT contain a CRC24 footer.

*   An ASCII-armored sequence of Signature packets that only includes
    v5 Signature packets MUST NOT contain a CRC24 footer.

*   An ASCII-armored Transferable Public Key packet sequence of a v5
    key MUST NOT contain a CRC24 footer.

*   An ASCII-armored keyring consisting of only v5 keys MUST NOT
    contain a CRC24 footer.

Rationale: Previous versions of this document state 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 CRC-24 in "C"

```
#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 Radix-64 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

   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 upon the type of
   data that is being encoded in Armor, and how it is being encoded.
   Header line texts include the following strings:

   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.
```

Note that all these Armor Header Lines are to consist of a complete
line. That is to say, there is always a line ending preceding the
starting five dashes, and following the ending five dashes. The
header lines, therefore, MUST start at the beginning of a line, and
MUST NOT have text other than whitespace following them on the same
line. These line endings are considered a part of the Armor Header
Line for the purposes of determining the content they delimit. This
is particularly important when computing a cleartext signature (see
[Section 7](#)).

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 a
part of 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
(: 0x38) and a single space (0x20) separate the key and value. An
OpenPGP implementation may consider improperly formatted Armor
Headers to be corruption of the ASCII Armor, but 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. While
there is a limit of 76 characters for the Radix-64 data
([Section 6.3](#)), there is no limit to the length of Armor Headers.
Care should be taken 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:

* "Version", which states 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.

* "Comment", 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.

* "Hash", a comma-separated list of hash algorithms used in this
  message. This is used only in cleartext signed messages.

* "SaltedHash", a salt and hash algorithm used in this message.
  This is used only in cleartext signed messages that are followed
  by a v5 Signature.

* "Charset", a description of the character set that the plaintext
  is in. Please note that OpenPGP defines text to be in UTF-8. An
  implementation will get 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.

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".

6.3. Encoding Binary in Radix-64

The encoding process represents 24-bit groups of input bits as output

strings of 4 encoded characters. Proceeding from left to right, a
24-bit input group is formed by concatenating three 8-bit input
groups. These 24 bits are then treated as four concatenated 6-bit
groups, each of which is translated into a single digit in the
Radix-64 alphabet. When encoding a bit stream with the Radix-64
encoding, the bit stream must be presumed to be ordered with the most
significant bit first. That is, the first bit in the stream will be
the high-order bit in the first 8-bit octet, and the eighth bit will
be the low-order bit in the first 8-bit octet, and so on.

```
 ┌─first octet──┬─second octet─┬─third octet──┐
 │7 6 5 4 3 2 1 0│7 6 5 4 3 2 1 0│7 6 5 4 3 2 1 0│
 │               │               │               │
 │5 4 3 2 1 0│5 4 3 2 1 0│5 4 3 2 1 0│5 4 3 2 1 0│
 └──1.index──┴──2.index──┴──3.index──┴──4.index──┘
```
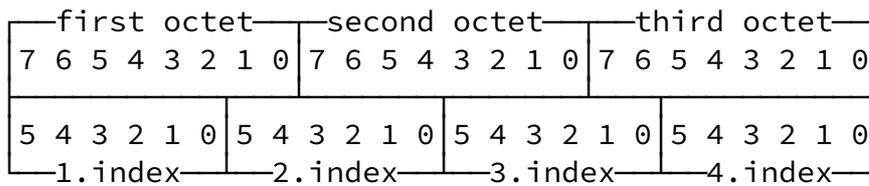
Each 6-bit group is used as an index into an array of 64 printable
characters from the table below. The character referenced by the
index is placed in the output string.

```
+=====+========++=====+=========++=====+==========++=====+==========+
|Value|Encoding||Value|Encoding ||Value| Encoding ||Value| Encoding |
+=====+========++=====+=========++=====+==========++=====+==========+
|    0|A       ||   17|R        ||   34| i        ||   51| z        |
+-----+--------++-----+---------++-----+----------++-----+----------+
|    1|B       ||   18|S        ||   35| j        ||   52| 0        |
+-----+--------++-----+---------++-----+----------++-----+----------+
|    2|C       ||   19|T        ||   36| k        ||   53| 1        |
+-----+--------++-----+---------++-----+----------++-----+----------+
|    3|D       ||   20|U        ||   37| l        ||   54| 2        |
+-----+--------++-----+---------++-----+----------++-----+----------+
|    4|E       ||   21|V        ||   38| m        ||   55| 3        |
+-----+--------++-----+---------++-----+----------++-----+----------+
|    5|F       ||   22|W        ||   39| n        ||   56| 4        |
+-----+--------++-----+---------++-----+----------++-----+----------+
|    6|G       ||   23|X        ||   40| o        ||   57| 5        |
+-----+--------++-----+---------++-----+----------++-----+----------+
|    7|H       ||   24|Y        ||   41| p        ||   58| 6        |
+-----+--------++-----+---------++-----+----------++-----+----------+
|    8|I       ||   25|Z        ||   42| q        ||   59| 7        |
+-----+--------++-----+---------++-----+----------++-----+----------+
|    9|J       ||   26|a        ||   43| r        ||   60| 8        |
+-----+--------++-----+---------++-----+----------++-----+----------+
|   10|K       ||   27|b        ||   44| s        ||   61| 9        |
+-----+--------++-----+---------++-----+----------++-----+----------+
|   11|L       ||   28|c        ||   45| t        ||   62| +        |
+-----+--------++-----+---------++-----+----------++-----+----------+
|   12|M       ||   29|d        ||   46| u        ||   63| /        |
+-----+--------++-----+---------++-----+----------++-----+----------+
|   13|N       ||   30|e        ||   47| v        ||     |          |
+-----+--------++-----+---------++-----+----------++-----+----------+
|   14|O       ||   31|f        ||   48| w        ||(pad)| =        |
+-----+--------++-----+---------++-----+----------++-----+----------+
|   15|P       ||   32|g        ||   49| x        ||     |          |
+-----+--------++-----+---------++-----+----------++-----+----------+
|   16|Q       ||   33|h        ||   50| y        ||     |          |
+-----+--------++-----+---------++-----+----------++-----+----------+
```

Table 18: Encoding for Radix-64

The encoded output stream must be represented in lines of no more
than 76 characters each.

Special processing is performed if fewer than 24 bits are available
at the end of the data being encoded. There are three possibilities:

1. The last data group has 24 bits (3 octets). No special
    processing is needed.

2. The last data group has 16 bits (2 octets). The first two 6-bit
    groups are processed as above. The third (incomplete) data group
    has two zero-value bits added to it, and is processed as above.
    A pad character (=) is added to the output.

3. The last data group has 8 bits (1 octet). The first 6-bit group
    is processed as above. The second (incomplete) data group has
    four zero-value bits added to it, and is processed as above. Two
    pad characters (=) are added to the output.

6.4. Decoding Radix-64

In Radix-64 data, characters other than those in the table, line
breaks, and other white space probably indicate a transmission error,
about which a warning message or even a message rejection might be
appropriate under some circumstances. Decoding software must ignore
all white space.

Because it is used only for padding at the end of the data, the
occurrence of any "=" characters may be taken as evidence that the
end of the data has been reached (without truncation in transit). No
such assurance is possible, however, when the number of octets
transmitted was a multiple of three and no "=" characters are
present.

6.5. Examples of Radix-64

```
   Input data:  0x14FB9C03D97E
   Hex:      1    4    F    B    9    C    | 0    3    D    9    7    E
   8-bit:    00010100 11111011 10011100   | 00000011 11011001 01111110
   6-bit:    000101 001111 101110 011100  | 000000 111101 100101 111110
   Decimal:  5        15     46     28       0        61     37     62
   Output:   F        P      u      c        A        9      l      +
   Input data:  0x14FB9C03D9
   Hex:      1    4    F    B    9    C    | 0    3    D    9
   8-bit:    00010100 11111011 10011100   | 00000011 11011001
                                                         pad with 00
   6-bit:    000101 001111 101110 011100  | 000000 111101 100100
   Decimal:  5        15     46     28       0        61     36
                                                           pad with =
   Output:   F        P      u      c        A        9      k      =
   Input data:  0x14FB9C03
   Hex:      1    4    F    B    9    C    | 0    3
   8-bit:    00010100 11111011 10011100   | 00000011
                                              pad with 0000
   6-bit:    000101 001111 101110 011100  | 000000 110000
   Decimal:  5        15     46     28       0        48
                                                  pad with =      =
   Output:   F        P      u      c        A        w      =      =
```

6.6. Example of an ASCII Armored Message

```
   -----BEGIN PGP MESSAGE-----

   yDgBO22WxBHv7O8X7O/jygAEzol56iUKiXmV+XmpCtmpqQUKiQrFqclFqUDBovzS
   vBSFjNSiVHsuAA==
   -----END PGP MESSAGE-----
```

   Note that this example has extra indenting; an actual armored message
   would have no leading whitespace.

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 without special software. In order to bind a signature to
such a cleartext, this 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.)

The cleartext signed message consists of:

   * The cleartext header -----BEGIN PGP SIGNED MESSAGE----- on a
      single line,

   * If the message is signed using v3 or v4 Signatures, one or more
      "Hash" Armor Headers,

   * If the message is signed using v5 Signatures, one or more
      "SaltedHash" Armor Headers,

   * Exactly one empty line not included into the message digest,

   * The dash-escaped cleartext that is included into the message
      digest,

   * The ASCII armored signature(s) including the -----BEGIN PGP
      SIGNATURE----- Armor Header and Armor Tail Lines.

If the "Hash" Armor Header is given, the specified message digest
algorithm(s) are used for the signature. If more than one message
digest is used in the signatures, each digest algorithm has to be
specified. To that end, the "Hash" Armor Header contains a comma-
delimited list of used message digests, and the "Hash" Armor Header
can be given multiple times.

If the "SaltedHash" Armor Header is given, the specified message
digest algorithm and salt are used for a signature. The message
digest name is followed by a colon (:) followed by 22 characters of
Radix-64 encoded salt without padding. Note: The "SaltedHash" Armor
Header contains digest algorithm and salt for a single signature; a

second signature requires a second "SaltedHash" Armor Header.

If neither a "Hash" nor a "SaltedHash" Armor Header is given, or the message digest algorithms (and salts) used in the signatures do not match the information in the headers, the signature MUST be considered invalid.

Current message digest names are described with the algorithm IDs in [Section 9.5](#).

An implementation SHOULD add a line break after the cleartext, but MAY omit it if the cleartext ends with a line break. This is for visual clarity.

[7.1](#). 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 "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.

As with binary signatures on text documents, a cleartext signature is calculated on the text using canonical <CR><LF> line endings. The line ending (that is, the <CR><LF>) before the -----BEGIN PGP SIGNATURE----- line that terminates the signed text is not considered part of the signed text.

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

Also, any trailing whitespace --- spaces (0x20) and tabs (0x09) --- at the end of any line is removed when the cleartext signature is generated.

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
   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 character (matching that character), or a
   single 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 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 ASCII characters between them
   (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 not on these lists, so long as the
   algorithm numbers are chosen from the private or experimental
   algorithm range.

See [Section 13](#) for more discussion of the algorithms.

9.1. Public-Key Algorithms

ID	Algorithm	Public Key Format	Secret Key Format	Signature Format	PKESK Format
1	RSA (Encrypt or Sign) [HAC]	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 [HAC]	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 [HAC]	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
16	Elgamal (Encrypt-Only) [ELGAMAL] [HAC]	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)	MPI(p), MPI(q), MPI(g),	MPI(x)	MPI(r), MPI(s) [Section	N/A

ID	Algorithm	Public Key Format	Secret Key Format	Signature Format	PKESK Format
	[FIPS186] [HAC]	MPI(y) [Section 5.5.5.2]		5.2.3.2]	
18	ECDH public key algorithm	OID, MPI(point in curve-specific	MPI(value in curve-specific format)	N/A	MPI(point in curve-specific point

		point format), KDFParams [see Section 9.2.1, Section 5.5.5.6]	[Section 9.2.1]		format), size octet, encoded key [Section 9.2.1, Section 5.1.5, Section 12.5]
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	EdDSA [RFC8032]	OID, MPI(point in prefixed native format) [see Section 12.2.2, Section 5.5.5.5]	MPI(value in curve-specific format) [see Section 9.2.1]	MPI, MPI [see Section 9.2.1, Section 5.2.3.3]	N/A

| 23 | Reserved | | | | |

```
|   |(AEDH)      |          |          |          |          |          |
+---+------------+----------+----------+----------+----------+----------+
| 24|Reserved    |          |          |          |          |          |
|   |(AEDSA)     |          |          |          |          |          |
+---+------------+----------+----------+----------+----------+----------+
|100|Private/    |          |          |          |          |          |
| to|Experimental|          |          |          |          |          |
|110|algorithm   |          |          |          |          |          |
+---+------------+----------+----------+----------+----------+----------+
```

               Table 19: Public-key algorithm registry

   Implementations MUST implement EdDSA (19) for signatures, and ECDH
   (18) for encryption.

   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 13.4. Elgamal
   (16) keys are deprecated and MUST NOT be generated (see
   Section 13.6). DSA (17) keys are deprecated and MUST NOT be
   generated (see Section 13.5). See Section 13.8 for notes on Elgamal
   Encrypt or Sign (20), and X9.42 (21). Implementations MAY implement
   any other algorithm.

   Note that an implementation conforming to the previous version of
   this standard ([RFC4880]) have only DSA (17) and Elgamal (16) as its
   MUST-implement algorithms.

   A compatible specification of ECDSA is given in [RFC6090] as "KT-I
   Signatures" and in [SEC1]; ECDH is defined in Section 12.5 of this
   document.

9.2. ECC Curves for OpenPGP

   The parameter curve OID is an array of octets that define 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 in hexadecimal representation	Curve name	Usage	Field Size (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
1.3.6.1.4.1.11591.15.1	9	2B 06 01 04 01 DA 47 0F 01	Ed25519	EdDSA	32
1.3.101.113	3	2B 65 71	Ed448	EdDSA	57
1.3.6.1.4.1.3029.1.5.1	10	2B 06 01 04 01 97 55 01 05 01	Curve25519	ECDH	32
1.3.101.111	3	2B 65 6F	X448	ECDH	56

Table 20: 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, native point representations, or scalars
with high enough entropy for the curve can be represented in that
many octets.

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.

Implementations MUST implement Ed25519 for use with EdDSA, and

Curve25519 for use with ECDH. Implementations SHOULD implement Ed448
for use with EdDSA, and X448 for use with ECDH.

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
Ed25519	N/A	N/A	32 octets of secret	32 octets of R	32 octets of S
Ed448	N/A	N/A	prefixed 57 octets of secret	prefixed 114 octets of signature	0 [this is an unused placeholder]
Curve25519	prefixed native	integer (see Section 5.5.5.6.1.1)	N/A	N/A	N/A
X448	prefixed native	prefixed 56 octets of secret	N/A	N/A	N/A

                   Table 21: Curve-specific wire formats

For the native octet-string forms of EdDSA values, see [RFC8032].
For the native octet-string forms of ECDH 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, [SCHNEIER], [HAC] - 168 bit key derived from 192)
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 to   | Private/Experimental algorithm                   |
| 110      |                                                  |
+----------+------------------------------------------------------+
| 253, 254 | Reserved to avoid collision with Secret Key      |
| and 255  | Encryption (see Section 3.7.2.1 and Section 5.5.3) |
+----------+------------------------------------------------------+
```

                Table 22: Symmetric-key algorithm 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 legacy clients. 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                   |
+-----------+--------------------------------+
|         1 | ZIP [RFC1951]                  |
+-----------+--------------------------------+
|         2 | ZLIB [RFC1950]                 |
+-----------+--------------------------------+
|         3 | BZip2 [BZ2]                    |
+-----------+--------------------------------+
| 100 to 110 | Private/Experimental algorithm |
+-----------+--------------------------------+
```

                Table 23: Compression algorithm 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   |
+===========+===================================+=============+
|         1 | MD5 [HAC]                         | "MD5"       |
+-----------+-----------------------------------+-------------+
|         2 | SHA-1 [FIPS180], Section 14.1     | "SHA1"      |
+-----------+-----------------------------------+-------------+
|         3 | RIPEMD-160 [HAC]                  | "RIPEMD160" |
+-----------+-----------------------------------+-------------+
|         4 | Reserved                          |             |
+-----------+-----------------------------------+-------------+
|         5 | Reserved                          |             |
+-----------+-----------------------------------+-------------+
|         6 | Reserved                          |             |
+-----------+-----------------------------------+-------------+
|         7 | Reserved                          |             |
+-----------+-----------------------------------+-------------+
|         8 | SHA2-256 [FIPS180]                | "SHA256"    |
```

```
+-----------+-----------------------------------+-------------+
|         9 | SHA2-384 [FIPS180]                | "SHA384"    |
+-----------+-----------------------------------+-------------+
|        10 | SHA2-512 [FIPS180]                | "SHA512"    |
+-----------+-----------------------------------+-------------+
|        11 | SHA2-224 [FIPS180]                | "SHA224"    |
+-----------+-----------------------------------+-------------+
|        12 | SHA3-256 [FIPS202]                | "SHA3-256"  |
+-----------+-----------------------------------+-------------+
|        13 | Reserved                          |             |
+-----------+-----------------------------------+-------------+
|        14 | SHA3-512 [FIPS202]                | "SHA3-512"  |
+-----------+-----------------------------------+-------------+
| 100 to 110 | Private/Experimental algorithm   |             |
+-----------+-----------------------------------+-------------+
```

Table 24: Hash algorithm registry

Implementations MUST implement SHA2-256. Implementations SHOULD
implement SHA2-384 and SHA2-512. Implementations MAY implement other
algorithms. Implementations SHOULD NOT create messages which require
the use of SHA-1 with the exception of computing version 4 key
fingerprints and for purposes of the Modification Detection Code
(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 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	Algorithm	IV length (octets)	authentication tag length (octets)
1	EAX [EAX]	16	16
2	OCB [RFC7253]	15	16
3	GCM [SP800-38D]	12	16
100 to 110	Private/Experimental algorithm		

Table 25: AEAD algorithm registry

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

10. IANA Considerations

Because this document obsoletes [RFC4880], IANA is requested to
update all registration information that references [RFC4880] to

instead reference this RFC.

OpenPGP is highly parameterized, and consequently there are a number
of considerations for allocating parameters for extensions. This
section describes how IANA should look at extensions to the protocol
as described in this document.

10.1. New String-to-Key Specifier Types

OpenPGP S2K specifiers contain a mechanism for new algorithms to turn
a string into a key. This specification creates a registry of S2K
specifier types. The registry includes the S2K type, the name of the
S2K, and a reference to the defining specification. The initial
values for this registry can be found in Section 3.7.1. Adding a new
S2K specifier MUST be done through the SPECIFICATION REQUIRED method,
as described in [RFC8126].

IANA should add a column "Generate?" to the S2K type registry, with
initial values taken from Section 3.7.1.

10.2. New Packets

Major new features of OpenPGP are defined through new packet types.
This specification creates a registry of packet types. The registry
includes the packet type, the name of the packet, and a reference to
the defining specification. The initial values for this registry can
be found in Section 4.3. Adding a new packet type MUST be done
through the RFC REQUIRED method, as described in [RFC8126].

10.2.1. User Attribute Types

The User Attribute packet permits an extensible mechanism for other
types of certificate identification. This specification creates a
registry of User Attribute types. The registry includes the User
Attribute type, the name of the User Attribute, and a reference to
the defining specification. The initial values for this registry can
be found in Section 5.12. Adding a new User Attribute type MUST be

done through the SPECIFICATION REQUIRED method, as described in
[RFC8126].

10.2.1.1. Image Format Subpacket Types

Within User Attribute packets, there is an extensible mechanism for
other types of image-based User Attributes. This specification
creates a registry of Image Attribute subpacket types. The registry
includes the Image Attribute subpacket type, the name of the Image
Attribute subpacket, and a reference to the defining specification.
The initial values for this registry can be found in Section 5.12.1.
Adding a new Image Attribute subpacket type MUST be done through the
SPECIFICATION REQUIRED method, as described in [RFC8126].

10.2.2. New Signature Subpackets

OpenPGP signatures contain a mechanism for signed (or unsigned) data
to be added to them for a variety of purposes in the Signature
subpackets as discussed in Section 5.2.3.5. This specification
creates a registry of Signature subpacket types. The registry
includes the Signature subpacket type, the name of the subpacket, and
a reference to the defining specification. The initial values for
this registry can be found in Section 5.2.3.5. Adding a new
Signature subpacket MUST be done through the SPECIFICATION REQUIRED
method, as described in [RFC8126].

10.2.2.1. Signature Notation Data Subpackets

OpenPGP signatures further contain a mechanism for extensions in
signatures. These are the Notation Data subpackets, which contain a
key/value pair. Notations contain a user space that is completely
unmanaged and an IETF space.

This specification creates a registry of Signature Notation Data
types. The registry includes the name of the Signature Notation
Data, the Signature Notation Data type, its allowed values, and a
reference to the defining specification. The initial values for this
registry can be found in Section 5.2.3.21. Adding a new Signature
Notation Data subpacket MUST be done through the SPECIFICATION
REQUIRED method, as described in [RFC8126].

10.2.2.2. Signature Notation Data Subpacket Notation Flags

   This specification creates a new registry of Signature Notation Data
   Subpacket Notation Flags. The registry includes the columns "Flag",
   "Shorthand", "Description", "Security Recommended", "Interoperability
   Recommended", and "Reference". The initial values for this registry
   can be found in Section 5.2.3.21. Adding a new item MUST be done
   through the SPECIFICATION REQUIRED method, as described in [RFC8126].

10.2.2.3. Key Server Preference Extensions

   OpenPGP signatures contain a mechanism for preferences to be
   specified about key servers. This specification creates a registry
   of key server preferences. The registry includes the key server
   preference, the name of the preference, and a reference to the
   defining specification. The initial values for this registry can be
   found in Section 5.2.3.22. Adding a new key server preference MUST
   be done through the SPECIFICATION REQUIRED method, as described in
   [RFC8126].

10.2.2.4. Key Flags Extensions

   OpenPGP signatures contain a mechanism for flags to be specified
   about key usage. This specification creates a registry of key usage
   flags. The registry includes the key flags value, the name of the
   flag, and a reference to the defining specification. The initial
   values for this registry can be found in Section 5.2.3.26. Adding a
   new key usage flag MUST be done through the SPECIFICATION REQUIRED
   method, as described in [RFC8126].

10.2.2.5. Reason for Revocation Extensions

   OpenPGP signatures contain a mechanism for flags to be specified
   about why a key was revoked. This specification creates a registry
   of "Reason for Revocation" flags. The registry includes the "Reason
   for Revocation" flags value, the name of the flag, and a reference to
   the defining specification. The initial values for this registry can
   be found in Section 5.2.3.28. Adding a new feature flag MUST be done
   through the SPECIFICATION REQUIRED method, as described in [RFC8126].

10.2.2.6. Implementation Features

   OpenPGP signatures contain a mechanism for flags to be specified
   stating which optional features an implementation supports. This
   specification creates a registry of feature-implementation flags.
   The registry includes the feature-implementation flags value, the
   name of the flag, and a reference to the defining specification. The

initial values for this registry can be found in Section 5.2.3.29.

Adding a new feature-implementation flag MUST be done through the
SPECIFICATION REQUIRED method, as described in [RFC8126].

Also see Section 13.11 for more information about when feature flags
are needed.

10.2.3. New Packet Versions

The core OpenPGP packets all have version numbers, and can be revised
by introducing a new version of an existing packet. This
specification creates a registry of packet types. The registry
includes the packet type, the number of the version, and a reference
to the defining specification. The initial values for this registry
can be found in Section 5. Adding a new packet version MUST be done
through the RFC REQUIRED method, as described in [RFC8126].

10.3. New Algorithms

Section 9 lists the core algorithms that OpenPGP uses. Adding in a
new algorithm is usually simple. For example, adding in a new
symmetric cipher usually would not need anything more than allocating
a constant for that cipher. If that cipher had other than a 64-bit
or 128-bit block size, there might need to be additional
documentation describing how OpenPGP-CFB mode would be adjusted.
Similarly, when DSA was expanded from a maximum of 1024-bit public
keys to 3072-bit public keys, the revision of FIPS 186 contained
enough information itself to allow implementation. Changes to this
document were made mainly for emphasis.

10.3.1. Public-Key Algorithms

OpenPGP specifies a number of public-key algorithms. This
specification creates a registry of public-key algorithm identifiers.
The registry includes the algorithm name, its key sizes and
parameters, and a reference to the defining specification. The
initial values for this registry can be found in Section 9.1. Adding
a new public-key algorithm MUST be done through the SPECIFICATION
REQUIRED method, as described in [RFC8126].

This document requests IANA register the following new public-key

algorithm:

```
      +====+=============================+=======================+
      | ID | Algorithm                   | Reference             |
      +====+=============================+=======================+
      | 22 | EdDSA public key algorithm  | This doc, Section 13.7 |
      +----+-----------------------------+-----------------------+
```

                Table 26: New public-Key algorithms registered

   [Note to RFC-Editor: Please remove the table above on publication.]

10.3.1.1. Elliptic Curve Algorithms

   Some public key algorithms use Elliptic Curves. In particular,
   ECDH/EdDSA/ECDSA public key algorithms all allow specific curves to
   be used, as indicated by OID. To register a new elliptic curve for
   use with OpenPGP, its OID needs to be registered in Table 20, its
   wire format needs to be documented in Table 21, and if used for ECDH,
   its KDF and KEK parameters must be populated in Table 31.

10.3.2. Symmetric-Key Algorithms

   OpenPGP specifies a number of symmetric-key algorithms. This
   specification creates a registry of symmetric-key algorithm
   identifiers. The registry includes the algorithm name, its key sizes
   and block size, and a reference to the defining specification. The
   initial values for this registry can be found in Section 9.3. Adding
   a new symmetric-key algorithm MUST be done through the SPECIFICATION
   REQUIRED method, as described in [RFC8126].

10.3.3. Hash Algorithms

   OpenPGP specifies a number of hash algorithms. This specification
   creates a registry of hash algorithm identifiers. The registry

includes the algorithm name, a text representation of that name, its
block size, an OID hash prefix, and a reference to the defining
specification. The initial values for this registry can be found in
[Section 9.5](#) for the algorithm identifiers and text names, and
[Section 5.2.2](#) for the OIDs and expanded signature prefixes. Adding a
new hash algorithm MUST be done through the SPECIFICATION REQUIRED
method, as described in [[RFC8126](#)].

This document requests IANA register the following hash algorithms:

```
+====+==========+===========+
| ID | Algorithm | Reference |
+====+==========+===========+
| 12 | SHA3-256  | This doc  |
+----+----------+-----------+
| 13 | Reserved  |           |
+----+----------+-----------+
| 14 | SHA3-512  | This doc  |
+----+----------+-----------+
```

Table 27: New hash
algorithms registered

[Notes to RFC-Editor: Please remove the table above on publication.
It is desirable not to reuse old or reserved algorithms because some
existing tools might print a wrong description. The ID 13 has been
reserved so that the SHA3 algorithm IDs align nicely with their SHA2
counterparts.]

[10.3.4](#). Compression Algorithms

OpenPGP specifies a number of compression algorithms. This
specification creates a registry of compression algorithm
identifiers. The registry includes the algorithm name and a
reference to the defining specification. The initial values for this
registry can be found in [Section 9.4](#). Adding a new compression key

algorithm MUST be done through the SPECIFICATION REQUIRED method, as
described in [RFC8126].

10.3.5. Elliptic Curve Algorithms

This document requests IANA add a registry of elliptic curves for use
in OpenPGP.

Each curve is identified on the wire by OID, and is acceptable for
use in certain OpenPGP public key algorithms. The table's initial
headings and values can be found in Section 9.2. Adding a new
elliptic curve algorithm to OpenPGP MUST be done through the
SPECIFICATION REQUIRED method, as described in [RFC8126]. If the new
curve can be used for ECDH or EdDSA, it must also be added to the
"Curve-specific wire formats" table described in Section 9.2.1.

10.4. Elliptic Curve Point and Scalar Wire Formats

This document requests IANA add a registry of wire formats that
represent elliptic curve points. The table's initial headings and
values can be found in Section 12.2. Adding a new EC point wire
format MUST be done through the SPECIFICATION REQUIRED method, as
described in [RFC8126].

This document also requests IANA add a registry of wire formats that
represent scalars for use with elliptic curve cryptography. The
table's initial headings and values can be found in Section 12.3.
Adding a new EC scalar wire format MUST be done through the
SPECIFICATION REQUIRED method, as described in [RFC8126].

This document also requests that IANA add a registry mapping curve-
specific MPI octet-string encoding conventions for ECDH and EdDSA.
The table's initial headings and values can be found in
Section 9.2.1. Adding a new elliptic curve algorithm to OpenPGP MUST
be done through the SPECIFICATION REQUIRED method, as described in

[RFC8126], and requires adding an entry to this table if the curve is
to be used with either EdDSA or ECDH.

10.5. Changes to existing registries

This document requests IANA add the following wire format columns to
the OpenPGP public-key algorithm registry:

*   Public Key Format

*   Secret Key Format

*   Signature Format

*   PKESK Format

And populate them with the values found in Section 9.1.

11. Packet 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 11.1) and its close counterpart,
    Transferable Secret Keys (Section 11.2)

*   OpenPGP Messages (Section 11.3)

*   Detached Signatures (Section 11.4)

Each sequence has an explicit grammar of what packet types (Table 4)
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 4.3.1). 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 strict.

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, a legacy
implementation 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.

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

11.1. Transferable Public Keys

OpenPGP users may transfer public keys. This section describes the
structure of public keys in transit to ensure interoperability.

11.1.1. OpenPGP v5 Key Structure

The format of an OpenPGP v5 key 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 v5 key uses a Direct-Key Signature to store algorithm
   preferences.

   Every subkey for a v5 primary key MUST be a v5 subkey.

   When a primary v5 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.

11.1.2. OpenPGP v4 Key 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 SHOULD
   be v4. 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 v5 revocation certificate MUST include the primary key
packet, as described above.

11.1.3. 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.

11.1.4. Common requirements

The Public-Key packet occurs first.

In order to create self-signatures (see Section 5.2.3.7), the primary
key MUST be an algorithm capable of making signatures (that is, not
an encryption-only algorithm). The subkeys may be keys of any type.
For example, there may be a single-key RSA key, an EdDSA primary key
with an RSA encryption key, or an EdDSA primary key with an ECDH
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 his or her
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, so long as the signatures that follow them are
maintained on the proper User Attribute or User ID packet.

After the User ID packet or Attribute packet, there may be zero or
more Subkey packets. In general, subkeys are provided in cases where
the top-level public key is a certification-only key. However, any
v4 or v5 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 14.10). 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 v5 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](#)).

[11.2](#). 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](#)). Implementations 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. Implementations MAY choose to omit the self-signatures, especially if a transferable public key accompanies the transferable secret key.

[11.3](#). OpenPGP Messages

An OpenPGP message is a packet or sequence of packets that corresponds to the following grammatical rules (comma represents sequential composition, and 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.

11.3.1. 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 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.

11.3.2. Additional Constraints on Packet Sequences

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

11.3.2.1. 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 (Section 5.7) packet instead of SEIPD, though
implementations MUST NOT generate SED packets (see Section 14.7).
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 in their cleartext the
symmetric cipher algorithm identifier in addition to the session key
for the subsequent SEIPD packet. Since a v1 SEIPD does not contain a
symmetric algorithm identifier, so all ESK packets preceding a v1
SEIPD payload MUST be either v3 PKESK or v4 SKESK.

On the other hand, the cleartext of the v5 ESK packets (either PKESK
or SKESK) do not contain a symmetric cipher algorithm identifier, so
they cannot be used in combination with a v1 SEIPD payload. The
payload following any v5 PKESK or v5 SKESK packet MUST be a v2 SEIPD.

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

The acceptable versions of packets in an Encrypted Message are
summarized in the following table:

Version of Encrypted Data payload	Version of preceding Symmetric-Key ESK (if any)	Version of preceding Public-Key ESK (if any)
v1 SEIPD	v4 SKESK	v3 PKESK
v2 SEIPD	v5 SKESK	v5 PKESK

Table 28: Encrypted Message Packet Version Alignment

   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.

11.4. 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.

12. 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], B.4.1, for the method to
   generate a uniformly distributed ECC private key.

12.1. Supported ECC Curves

   This document references three named prime field curves defined in
   [FIPS186] as "Curve P-256", "Curve P-384", and "Curve P-521". These
   three [FIPS186] curves can be used with ECDSA and ECDH public key
   algorithms. Additionally, curve "Curve25519" and "Curve448" are
   referenced for use with Ed25519 and Ed448 (EdDSA signing, see
   [RFC8032]); and X25519 and X448 (ECDH encryption, see [RFC7748]).

   The named curves are referenced as a sequence of octets in this
   document, called throughout, curve OID. Section 9.2 describes in
   detail how this sequence of octets is formed.

12.2. 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 12.2.1 |
+----------------+----------------+----------------+
| Prefixed native | 0x40 || native | Section 12.2.2 |
+----------------+----------------+----------------+
```

Table 29: Elliptic Curve Point Wire Formats

12.2.1. 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].

12.2.2. 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.

. 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 "Curve P-256", 771 for "Curve P-384",
   1059 for "Curve P-521", 263 for both "Curve25519" and "Ed25519", 463
   for "Ed448", and 455 for "X448". For example, the length of a EdDSA
   public key for the curve Ed25519 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 than the wire format
   associated with the curve.

12.3. 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
   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, big-endian encoded as a  | Section   |
|          | standard OpenPGP MPI                 | 3.2       |
+----------+--------------------------------------+-----------+
| octet    | An octet string of fixed length,     | Section   |
| string   | that may be shorter on the wire due  | 12.3.1    |
|          | to leading zeros being stripped by   |           |
|          | the MPI encoding, and may need to    |           |
|          | be zero-padded before usage          |           |
+----------+--------------------------------------+-----------+
| prefixed | An octet string of fixed length N,   | Section   |
| N octets | prefixed with octet 0x40 to ensure   | 12.3.2    |
|          | no leading zero octet                |           |
+----------+--------------------------------------+-----------+
```

Table 30: Elliptic Curve Scalar Encodings

12.3.1. 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, we 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 can take the following steps:

*   ensure that the MPI's two-octet bitcount 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

12.3.2. Elliptic Curve Prefixed Octet String Wire Format

Another way to ensure that a fixed-length bytestring is encoded
simply to the wire while remaining in MPI format is to prefix the
bytestring with a dedicated non-zero octet. This specification uses
0x40 as the prefix octet. This is represented in this standard as
MPI(prefixed N octets of X), where N is the known bytestring 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, we prefix it with the octet 0x40 (whose 7th bit
is set), 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 12.2.2.

12.4. 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].

12.5. 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 CDH primitive
   employed by this method is modified to always assume the cofactor is
   1, the KDF specified in Section 12.4 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 bitstring [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 defined in [Section 9.2](#);

     * A one-octet public key algorithm ID 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
        are 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 used to
          wrap the symmetric key for message encryption; see [Section 12.5](#)
          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 or a primary key fingerprint
        identifying the key material that is needed for decryption. For
        version 4 keys, this field is 20 octets. For version 5 keys, this
        field is 32 octets.

   The size in octets of the KDF parameters sequence, defined above, for
   encrypting to a v4 key is either 54 for curve P-256, 51 for curves
   P-384 and P-521, 56 for Curve25519, or 49 for X448. For encrypting
   to a v5 key, the size of the sequence is either 66 for curve P-256,
   63 for curves P-384 and P-521, 68 for Curve25519, or 61 for X448.

   The key wrapping method is described in [[RFC3394](#)]. The KDF produces
   a symmetric key that is used as a key-encryption key (KEK) as
   specified in [[RFC3394](#)]. Refer to [Section 12.5.1](#) for the details
   regarding the choice of the KEK algorithm, which SHOULD be one of
   three AES algorithms. Key wrapping and unwrapping is performed with

the default initial value of [RFC3394].

The input to the key wrapping method is the plaintext described in
Section 5.1, "Public-Key Encrypted Session Key Packets (Tag 1)",
padded using the method described in [PKCS5] to an 8-octet
granularity.

For example, in a v4 Public-Key Encrypted Session Key packet, the
following AES-256 session key, in which 32 octets are denoted from k0
to k31, is composed to form the following 40 octet sequence:

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

The octets s0 and s1 above 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 v5 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 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) as per [RFC3394]

* 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 obtains the shared secret by calculating

S = rV = rvG, where (r,R) is the recipient's key pair.

Consistent with [Section 5.13](#), AEAD encryption or a Modification
Detection Code (MDC) MUST be used anytime the symmetric key is
protected by ECDH.

12.5.1. ECDH Parameters

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

For v5 keys, the following algorithms MUST be used depending on the
curve. An implementation MUST NOT generate a v5 ECDH key over any
listed curve that uses different KDF or KEK parameters. An
implementation MUST NOT encrypt any message to a v5 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

```
             +-----------+--------------+--------------------+
             | Curve25519 | SHA2-256     | AES-128            |
             +-----------+--------------+--------------------+
             | X448       | SHA2-512     | AES-256            |
             +-----------+--------------+--------------------+
```

             Table 31: ECDH KDF and KEK parameters

13. Notes on Algorithms

13.1. PKCS#1 Encoding in OpenPGP

   This standard makes use of the PKCS#1 functions EME-PKCS1-v1_5 and
   EMSA-PKCS1-v1_5. However, the calling conventions of these functions
   has 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.

13.1.1. EME-PKCS1-v1_5-ENCODE

   Input:

   k = the length in octets of the key modulus.

   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

pseudo-randomly generated nonzero 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.

13.1.2. 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 nonzero 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, if the
      second octet of EM does not have hexadecimal value 0x02, if there is
      no octet with hexadecimal value 0x00 to separate PS from M, or if the
      length of PS is less than 8 octets, output "decryption error" and
      stop. See also Section 14.5 regarding differences in reporting
      between a decryption error and a padding error.

13.1.3. EMSA-PKCS1-v1_5

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

      Option:

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

      Input:

   M = message to be encoded.

   emLen = intended length in octets of the encoded message, 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 a 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 5.2.2, 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.

13.2. 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 MUST-implement algorithm, 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. Note further that
   implementations conforming to previous versions of this standard
   [RFC4880] have TripleDES as its only MUST-implement algorithm.

   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
   MUST-implement algorithm, AES-128, ensures that the intersection is
   not null. 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 MUST warn the keyholder that the protocol has
   been violated. For example, suppose that Alice, above, has software
   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 software
   warns her that someone sent her an IDEA-encrypted message, but it
   would ideally decrypt it anyway.

13.2.1. Plaintext

   Algorithm 0, "plaintext", may only be used to denote secret keys that

are stored in the clear. Implementations MUST NOT use plaintext in
encrypted data packets; they must use Literal Data packets to encode
unencrypted literal data.

13.3. 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 that other comments need to
be made about, though, the compression preferences and the hash
preferences.

13.3.1. 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.

13.3.1.1. Uncompressed

Algorithm 0, "uncompressed", may only be used to denote a preference
for uncompressed data. Implementations MUST NOT use uncompressed in
Compressed Data packets; they must use Literal Data packets to encode
uncompressed literal data.

13.3.2. Hash Algorithm Preferences

Typically, the choice of a hash algorithm is something the signer
does, rather than the verifier, because a signer rarely knows who is

going to be verifying the signature. This preference, though, 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 software uses.
This preference allows Bob to state in his key which algorithms Alice
may use.

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

13.4. 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
generalizes it to all algorithms. An implementation MUST NOT create
such a key, but MAY interpret it.

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

13.5. DSA

DSA is expected to be deprecated in [FIPS186-5]. Therefore, an
implementation MUST NOT generate DSA keys.

An implementation MUST NOT sign or verify using DSA keys.

13.6. 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.

13.7. EdDSA

Although the EdDSA algorithm allows arbitrary data as input, its use with OpenPGP requires that a digest of the message is 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.

13.8. Reserved Algorithm Numbers

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 in Section 9.1 as "reserved for".

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 OpenPGP permitted Elgamal [ELGAMAL] signatures with a public-key identifier of 20. These are no longer permitted. An implementation MUST NOT generate such keys. An implementation MUST NOT generate Elgamal signatures. See [BLEICHENBACHER].

13.9. OpenPGP CFB Mode

When using a version 1 Symmetrically Encrypted Integrity Protected Data packet (Section 5.13.1) or --- for historic data --- a Symmetrically Encrypted Data packet (Section 5.7), OpenPGP does symmetric encryption using a variant of Cipher Feedback mode (CFB mode). This section describes the procedure it uses in detail. This mode is what is used for Symmetrically Encrypted Integrity Protected Data Packets (and the dangerously malleable --- and deprecated --- Symmetrically Encrypted Data Packets). Some mechanisms for encrypting secret-key material also use CFB mode, as described in Section 3.7.2.1.

In the description below, the value BS is the block size in octets of the cipher. Most ciphers have a block size of 8 octets. The AES and Twofish have a block size of 16 octets. Also note that the description below assumes that the IV and CFB arrays start with an index of 1 (unlike the C language, which assumes arrays start with a zero index).

OpenPGP CFB mode uses an initialization vector (IV) of all zeros, and prefixes the plaintext with BS+2 octets of random data, such that octets BS+1 and BS+2 match octets BS-1 and BS. It does a CFB resynchronization after encrypting those BS+2 octets.

Thus, for an algorithm that has a block size of 8 octets (64 bits), the IV is 10 octets long and octets 7 and 8 of the IV are the same as octets 9 and 10. For an algorithm with a block size of 16 octets (128 bits), the IV is 18 octets long, and octets 17 and 18 replicate octets 15 and 16. Those extra two octets are an easy check for a correct key.

Step by step, here is the procedure:

1. The feedback register (FR) is set to the IV, which is all zeros.

2. FR is encrypted to produce FRE (FR Encrypted). This is the
    encryption of an all-zero value.

3. FRE is xored with the first BS octets of random data prefixed to
    the plaintext to produce C[1] through C[BS], the first BS octets
    of ciphertext.

4. FR is loaded with C[1] through C[BS].

5. FR is encrypted to produce FRE, the encryption of the first BS
    octets of ciphertext.

6. The left two octets of FRE get xored with the next two octets of
    data that were prefixed to the plaintext. This produces C[BS+1]
    and C[BS+2], the next two octets of ciphertext.

7. (The resynchronization step) FR is loaded with C[3] through
    C[BS+2].

8. FR is encrypted to produce FRE.

9. FRE is xored with the first BS octets of the given plaintext,
    now that we have finished encrypting the BS+2 octets of prefixed
    data. This produces C[BS+3] through C[BS+(BS+2)], the next BS
    octets of ciphertext.

10. FR is loaded with C[BS+3] to C[BS + (BS+2)] (which is C11-C18
    for an 8-octet block).

11. FR is encrypted to produce FRE.

12. FRE is xored with the next BS octets of plaintext, to produce
    the next BS octets of ciphertext. These are loaded into FR, and
    the process is repeated until the plaintext is used up.

13.10. Private or Experimental Parameters

   S2K specifiers, Signature subpacket types, User Attribute types,
   image format types, and algorithms described in Section 9 all reserve
   the range 100 to 110 for private and experimental use. Packet types
   reserve the range 60 to 63 for private and experimental use. These
   are intentionally managed with 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.

13.11. Meta-Considerations for Expansion

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

   We cannot state definitively what extensions will not be upwards-
   compatible, but typically new algorithms are upwards-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.

14. Security Considerations

   * As with any technology involving cryptography, you should check
      the current literature to determine if any algorithms used here
      have been found to be vulnerable to attack.

   * 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 hash algorithm has been found to have weaknesses, with
    collisions found in a number of cases. MD5 is deprecated for use
    in OpenPGP. Implementations MUST NOT generate new signatures
    using MD5 as a hash function. They MAY continue to consider old
    signatures that used MD5 as valid.

*   SHA2-224 and SHA2-384 require the same work as SHA2-256 and
    SHA2-512, respectively. In general, there are few reasons to use
    them outside of DSS compatibility. You need a situation where one
    needs more security than smaller hashes, but does not want to have
    the full 256-bit or 512-bit data length.

*   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. If you as an implementer promote dual-use keys, you should
    at least be aware of this controversy.

*   The DSA algorithm will work with any hash, but 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 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] recommends the following list of
    equivalent strengths:

Asymmetric key size	Hash size	Symmetric key size
1024	160	80
2048	224	112
3072	256	128
7680	384	192
15360	512	256

Table 32: Key length equivalences

_____

   * 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 CAST5 is
      presently considered strong, it has been analyzed less than
      TripleDES. Other algorithms may have other controversies
      surrounding them.

   * In late summer 2002, Jallad, Katz, and Schneier published an
      interesting attack on older 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 14.7 for information on how to defend
      against such an attack using more recent versions of OpenPGP.

   * Some technologies mentioned here may be subject to government
      control in some countries.

14.1. 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 and reasoning about 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
deliberately reject the hash input 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].

14.2. Advantages of Salted Signatures

V5 signatures include a 128 bit salt that is hashed first. This
makes v5 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, signatures are not predictable.

When the material to be signed may 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 (see [SHAMBLES]), which leverages a chosen prefix
collision attack against SHA-1.

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, and mitigates this risk.

14.3. 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 of 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.

14.4. 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 OpenPGP
   CFB mode (used by v1 SEIPD and SED packets) can be 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 to 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 legacy encrypted data whose session key is protected
   by a passphrase (v4 SKESK), while the quick check may be convenient
   to the user to be informed early on that they typed the wrong
   passphrase, 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 14.7.

14.5. 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, 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 includes also session key parsing errors, such as on invalid cipher algorithm for v3 PKESK, or session key size mismatch for v5 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.

14.6. Fingerprint Usability

This specification uses fingerprints in several places on the wire (e.g., Section 5.2.3.20, Section 5.2.3.32, and Section 5.2.3.33), and in processing (e.g., in ECDH KDF Section 12.5). 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. We have 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
fingerprint.

The version 5 fingerprint makes the challenge for a human verifier
even worse. At 256 bits (compared to v4's 160 bit fingerprint), a v5
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 v5 fingerprint for
this discouraged use case.

NOTE: the topic of interoperable human-in-the-loop key verification
needs more work, probably in a separate document.

14.7. 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 modern OpenPGP offers mechanisms to defend against it.
   However, legacy OpenPGP data may have been created before these
   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 indicate a clear error message

that the integrity of the message is suspect, SHOULD NOT attempt to
parse nor release decrypted data to the user, and SHOULD halt with an
error. Parsing or releasing decrypted data before having confirmed
its integrity can leak the decrypted data [EFAIL], [MRLG15].

An implementation that encounters malleable ciphertext MAY choose to
release cleartext to the user if it is known to be dealing with
historic archived legacy data, and the user is aware of the risks.

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 or raw cipher algorithm: where the trailing 2-octet
      checksum does not match.

   - value 254: where the SHA1 checksum is mismatched.

   - value 253: 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:

       - all recipient keys indicate support for version 2 of the
          Symmetrically Encrypted Integrity Protected Data packet in
          their Features subpacket ([Section 5.2.3.29](#)), or are v5 keys
          without a Features subpacket, or the implementation can
          otherwise infer that all recipients support v2 SEIPD packets,
          the implementation MUST 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.

    * Password-protected secret key material in a v5 Secret Key or v5
       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. Implementations
       should be aware that, in scenarios where an attacker has 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 should migrate to AEAD with all due speed.

[14.8](#). Escrowed Revocation Signatures

    A keyholder Alice may wish to designate a third party to be able to
    revoke Alice's own key.

    The preferred way for her to do this is produce a specific Revocation
    Signature (signature types 0x20, 0x28, or 0x30) and distribute it
    securely to her 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 a
    revocation signature themselves.

    There are multiple advantages of using an escrowed Revocation
    Signature over the deprecated Revocation Key subpacket
    ([Section 5.2.3.20](#)):

*   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.

14.9. 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, which should be
used absent other (for example, performance) concerns. It is
RECOMMENDED to use an existing CSPRNG implementation in preference 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, and

*   in entirely private data, such as asymmetric key generation.

With a properly functioning CSPRNG, this 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.

14.10. 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) and 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 who) 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).

14.11. 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, implementations SHOULD implement the Intended Recipient
Fingerprint signature subpacket (Section 5.2.3.33).

15. Implementation Nits

   This section is a collection of comments to help an implementer,
   particularly with an eye to 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 be backward-compatible.

   * There are many ways possible for two keys to have the same key
      material, but different fingerprints (and thus 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, and 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 Legacy packet format Section 4.2.2 is
what older documents called the "old packet format". It is the
packet format of the legacy PGP 2 implementation. Older RFCs
called the current OpenPGP packet format Section 4.2.1 the "new
packet format".

15.1. Constrained Legacy Fingerprint Storage for v5 Keys

Some OpenPGP implementations have fixed length constraints for key
fingerprint storage that will not fit all 32 octets of a v5
fingerprint. For example, [OPENPGPCARD] reserves 20 octets for each
stored fingerprint.

An OpenPGP implementation MUST NOT attempt to map any part of a v5
fingerprint to such a constrained field unless the relevant spec for
the constrained environment has explicit guidance for storing a v5
fingerprint that distinguishes it from a v4 fingerprint. An
implementation interacting with such a constrained field SHOULD
directly calculate the v5 fingerprint from public key material and
associated metadata instead of relying on the constrained field.

16. References

16.1. Normative References

[AES]   "Advanced encryption standard (AES)", National Institute
of Standards and Technology report,
DOI 10.6028/nist.fips.197, November 2001,
<https://doi.org/10.6028/nist.fips.197>.

[BLOWFISH] Schneier, B., "Description of a New Variable-Length Key,
64-Bit Block Cipher (Blowfish)", Fast Software Encryption,
Cambridge Security Workshop Proceedings Springer-Verlag,
1994, pp191-204, December 1993,

                   <http://www.counterpane.com/bfsverlag.html>.

   [BZ2]   Seward, J., "The Bzip2 and libbzip2 home page", 2010,
                   <http://www.bzip.org/>.

   [EAX]   Bellare, M., Rogaway, P., and D. Wagner, "A Conventional
                   Authenticated-Encryption Mode", April 2003.

   [ELGAMAL]   Elgamal, T., "A Public-Key Cryptosystem and a Signature
                   Scheme Based on Discrete Logarithms", IEEE Transactions on
                   Information Theory v. IT-31, n. 4, 1985, pp. 469-472,
                   1985.

   [FIPS180]   Dang, Q., "Secure Hash Standard", National Institute of
                   Standards and Technology report,
                   DOI 10.6028/nist.fips.180-4, July 2015,
                   <https://doi.org/10.6028/nist.fips.180-4>.

   [FIPS186]   "Digital Signature Standard (DSS)", National Institute of
                   Standards and Technology report,
                   DOI 10.6028/nist.fips.186-4, July 2013,
                   <https://doi.org/10.6028/nist.fips.186-4>.

   [FIPS202]   Dworkin, M., "SHA-3 Standard: Permutation-Based Hash and
                   Extendable-Output Functions", National Institute of
                   Standards and Technology report,
                   DOI 10.6028/nist.fips.202, July 2015,
                   <https://doi.org/10.6028/nist.fips.202>.

   [HAC]   Menezes, A.J., Oorschot, P.v., and S. Vanstone, "Handbook
                   of Applied Cryptography", 1996.

   [IDEA]   Lai, X., "On the design and security of block ciphers",
                   ETH Series in Information Processing, J.L. Massey
                   (editor) Vol. 1, Hartung-Gorre Verlag Konstanz, Technische
                   Hochschule (Zurich), 1992.

   [ISO10646]   International Organization for Standardization,
                   "Information Technology - Universal Multiple-octet coded

                    Character Set (UCS) - Part 1: Architecture and Basic
                    Multilingual Plane", ISO Standard 10646-1, May 1993.

     [JFIF]   CA, E.H.M., "JPEG File Interchange Format (Version
                    1.02).", September 1996.

     [PKCS5]   RSA Laboratories, "PKCS #5 v2.0: Password-Based
                    Cryptography Standard", 25 March 1999.

     [RFC1950]   Deutsch, P. and J-L. 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>.

     [RFC2045]   Freed, N. and N. Borenstein, "Multipurpose Internet Mail
                    Extensions (MIME) Part One: Format of Internet Message
                    Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996,
                    <https://www.rfc-editor.org/info/rfc2045>.

     [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>.

     [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>.

   [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>.

[SCHNEIER] Schneier, B., "Applied Cryptography Second Edition:
           protocols, algorithms, and source code in C", 1996.

[SP800-38D]
           Dworkin, M., "Recommendation for Block Cipher Modes of
           Operation: Galois/Counter Mode (GCM) and GMAC", NIST
           Special Publication 800-38D, November 2007.

[SP800-56A]
           Barker, E., Johnson, D., and M. Smid, "Recommendation for
           Pair-Wise Key Establishment Schemes Using Discrete
           Logarithm Cryptography", NIST Special Publication 800-56A
           Revision 1, March 2007.

[TWOFISH] Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall,
           C., and N. Ferguson, "The Twofish Encryption Algorithm",
           1999.

16.2. Informative References

[BLEICHENBACHER]
           Bleichenbacher, D., "Generating ElGamal Signatures Without
           Knowing the Secret Key", Lecture Notes in Computer
           Science Volume 1070, pp. 10-18, 1996.

[BLEICHENBACHER-PKCS1]
           Bleichenbacher, D., "Chosen Ciphertext Attacks Against
           Protocols Based on the RSA Encryption Standard PKCS \#1",
           1998, <http://archiv.infsec.ethz.ch/education/fs08/secsem/
           Bleichenbacher98.pdf>.

[CHECKOWAY]
          Checkoway, S., Maskiewicz, J., Garman, C., Fried, J.,
          Cohney, S., Green, M., Heninger, N., Weinmann, R.,
          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 Conference on Security Symposium, August
          2018, Pages 549-566 , 2018,
          <https://www.usenix.org/system/files/conference/
          usenixsecurity18/sec18-poddebniak.pdf>.

[FIPS186-5]
          Regenscheid, A., "Digital Signature Standard (DSS):
          Elliptic Curve Domain Parameters", National Institute of
          Standards and Technology (NIST) posted-content,
          DOI 10.6028/nist.fips.186-5-draft, October 2019,
          <https://doi.org/10.6028/nist.fips.186-5-draft>.

[JKS02]   Jallad, K., Katz, J., and B. Schneier, "Implementation of
          Chosen-Ciphertext Attacks against PGP and GnuPG", 2002,
          <http://www.counterpane.com/pgp-attack.html>.

[KOBLITZ] Koblitz, N., "A course in number theory and cryptography,
          Chapter VI. Elliptic Curves", ISBN 0-387-96576-9, 1997.

[KOPENPGP] Bruseghini, L., Paterson, K.G., and D. Huigens, "Victory
          by KO: Attacking OpenPGP Using Key Overwriting",
          Proceedings of the 29th ACM Conference on Computer and
          Communications Security, November 2022 (to appear) , 2022,
          <https://www.kopenpgp.com/>.

[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, Report 2002/076 , 2002,
          <https://eprint.iacr.org/2002/076>.

   [MRLG15]   Maury, F., Reinhard, J., Levillain, O., and H. Gilbert,
              "Format Oracles on OpenPGP", CT-RSA 2015 Topics in
              Cryptology -- CT-RSA 2015 pp 220-236,
              DOI 10.1007/978-3-319-16715-2_12, 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", IACR ePrint Archive Report
              2005/033, 8 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)", 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", October 2017,
              <https://eprint.iacr.org/2017/1014>.

   [REGEX]   Friedl, J., "Mastering Regular Expressions",
              ISBN 0-596-00289-0, August 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>.

   [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>.

[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>.

[SEC1]   Standards for Efficient Cryptography Group, "SEC 1:
            Elliptic Curve Cryptography", September 2000.

[SHA1CD]   Stevens, M. and D. Shumow, "sha1collisiondetection", 2017,
            <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", 2020, <https://sha-mbles.github.io/>.

[SP800-131A]
            Barker, E. and A. Roginsky, "Transitioning the Use of
            Cryptographic Algorithms and Key Lengths", NIST Special
            Publication 800-131A Revision 2, March 2019,
            <https://nvlpubs.nist.gov/nistpubs/SpecialPublications/
            NIST.SP.800-131Ar2.pdf>.

[SP800-57] NIST, "Recommendation on Key Management", NIST Special
            Publication 800-57, March 2007,
            <http://csrc.nist.gov/publications/nistpubs/800-57/
            SP800-57-Part{1,2}.pdf>.

[STEVENS2013]
            Stevens, M., "Counter-cryptanalysis", June 2013,
            <https://eprint.iacr.org/2013/358>.

. Test vectors

   To help implementing this specification a non-normative example for
   the EdDSA algorithm is given.

A.1. Sample v4 Ed25519 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 Ed25519 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: 4f70656e50475004004001608000060505025ff95f9504ff0000000c

Using the SHA2-256 hash algorithm yields the digest:

d: f6220a3f757814f4c2176ffbb68b00249cd4ccdc059c4b34ad871f30b1740280

Which is fed into the EdDSA signature function and yields this
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 v5 Certificate (Transferable Public Key)

Here is a Transferable Public Key consisting of:

* a v5 Ed25519 Public-Key packet

* a v5 direct key self-signature

* a v5 Curve25519 Public-Subkey packet

   * a v5 subkey binding signature

   -----BEGIN PGP PUBLIC KEY BLOCK-----

   xjcFYiDQVxYAAAAtCSsGAQQB2kcPAQEHQLVQ/UIL3goq8tqYyAhqx19AG5YH
   uMyAHjCOTyUpVKtRwqgFHxYKAAAAIwUCYiDQVwMVCAoEFgACAQIbAwIeCQ0n
   CQMHAwkBBwEJAgcCAAAAIyIhBRtEKdW2+mmb5MgIz7teOE83FiJh8l1/FwE4
   zi0wDN9LAe6QPJVjW4F4PVc/MnGWVpABAQDII7BN+BLRKYzNOhbcPvfYF4z1
   eV8v9ZpnrKBtyU2VegEA4IBoRJBIBupzrKXL497Z1/H4t/zWsNOwx9Gk/NQN
   7QbOPAViINBXEgAAADIKKwYBBAGXVQEFAQEHQOwq6DFNBJ25z8Z/WKRA92BG
   lwBQnfJnGYBF7hPBMl1/AwEIB8KOBRgWCAAAAAkFAmIg0FcCGwwAAAAjIiEF
   G0Qp1bb6aZvkyAjPu144TzcWImHyXX8XATjOLTAM30t2vVIiqtITHHtzmroU
   10kwplUBANrkpE2T3XCNqLYnFEfpj0+eyNjUDX4LZye4k5SICcIkAPwNFfvq
   wyg7rLV+WXlG27Z7S2gNpt1VbZSBs6IxjzXABg==
   -----END PGP PUBLIC KEY BLOCK-----

   The corresponding Transferable Secret Key can be found in
   [Appendix A.4](#).

[A.4](#). Sample v5 Secret Key (Transferable Secret Key)

   Here is a Transferable Secret Key consisting of:

   * a v5 Ed25519 Secret-Key packet

   * a v5 direct key self-signature

   * a v5 Curve25519 Secret-Subkey packet

   * a v5 subkey binding signature

   -----BEGIN PGP PRIVATE KEY BLOCK-----

   xV0FYiDQVxYAAAAtCSsGAQQB2kcPAQEHQLVQ/UIL3goq8tqYyAhqx19AG5YH
   uMyAHjCOTyUpVKtRAAAABAKmpvxTlZ9KQ6j+aOEk8fYe/h0L8K5pJsuAYhvSV
   mL28EbTCqAUfFgoAAAAjBQJiINBXAxUICgQWAAIBAhsDAh4JDScJAwcDCQEH
   AQkCBwIAAAAjIiEFG0Qp1bb6aZvkyAjPu144TzcWImHyXX8XATjOLTAM30sB
   7pA8lWNbgXg9Vz8ycZZWkAEBAMgjsE34EtEpjM06Ftw+99gXjPV5Xy/1mmes

```
oG3JTZV6AQDggGhEkEgG6nOspcvj3tnX8fi3/Naw07DH0aT81A3tBsdiBWIg
0FcSAAAAMgorBgEEAZdVAQUBAQdA7CroMU0EnbnPxn9YpED3YEaXAFCd8mcZ
gEXuE8EyXX8DAQgHAAAA/1b9bxwV0acRUcifrLiKHd0VVifoISz2PSVd4q5I
c1+gD+HCjgUYFggAAAAJBQJiINBXAhsMAAAAIyIhBRtEKdW2+mmb5MgIz7te
OE83FiJh8l1/FwE4zi0wDN9Ldr1SIqrSExx7c5q6FNdJMKZVAQDa5KRNk91w
jai2JxRH6Y9PnsjY1A1+C2cnuJOUiAnCJAD8DRX76sMoO6y1fll5Rtu2e0to
DabdVW2UgbOiMY81wAY=
-----END PGP PRIVATE KEY BLOCK-----
```

   The corresponding Transferable Public Key can be found in
   [Appendix A.3](#).

A.5. Sample AEAD-EAX encryption and decryption

   This example encrypts the cleartext string Hello, world! with the
   password password, using AES-128 with AEAD-EAX encryption.

A.5.1. Sample Parameters

   S2K:

      Iterated and Salted S2K

   Iterations:

      65011712 (255), SHA2-256

   Salt:

      a5 ae 57 9d 1f c5 d8 2b

A.5.2. Sample symmetric-key encrypted session key packet (v5)

   Packet header:

      c3 40

   Version, algorithms, S2K fields:

```
         05 1e 07 01 0b 03 08 a5 ae 57 9d 1f c5 d8 2b ff
         69 22
```

   Nonce:

```
         69 22 4f 91 99 93 b3 50 6f a3 b5 9a 6a 73 cf f8
```

   Encrypted session key and AEAD tag:

```
         da 74 6b 88 e3 57 e8 ae 54 eb 87 e1 d7 05 75 d7
         2f 60 23 29 90 52 3e 9a 59 09 49 22 40 6b e1 c3
```

A.5.3. 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 05 07 01
```

   HKDF output:

```
         74 f0 46 03 63 a7 00 76 db 08 c4 92 ab f2 95 52
```

   Authenticated Data:

```
         c3 05 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.5.4. Sample v2 SEIPD packet

Packet header:

      d2 69

   Version, AES-128, EAX, Chunk size octet:

      02 07 01 06

   Salt:

      9f f9 0e 3b 32 19 64 f3 a4 29 13 c8 dc c6 61 93
      25 01 52 27 ef b7 ea ea a4 9f 04 c2 e6 74 17 5d

   Chunk #0 encrypted data:

      4a 3d 22 6e d6 af cb 9c a9 ac 12 2c 14 70 e1 1c
      63 d4 c0 ab 24 1c 6a 93 8a d4 8b f9 9a 5a 99 b9
      0b ba 83 25 de

   Chunk #0 authentication tag:

      61 04 75 40 25 8a b7 95 9a 95 ad 05 1d da 96 eb

   Final (zero-sized chunk #1) authentication tag:

      15 43 1d fe f5 f5 e2 25 5c a7 82 61 54 6e 33 9a

A.5.5. 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 25
```

A.5.6. Complete AEAD-EAX encrypted packet sequence

```
   -----BEGIN PGP MESSAGE-----

   w0AFHgcBCwMIpa5XnR/F2Cv/aSJPkZmTs1Bvo7WaanPP+Np0a4jjV+iuVOuH4dcF
   ddcvYCMpkFI+mlkJSSJAa+HD0mkCBwEGn/kOOzIZZPOkKRPI3MZhkyUBUifvt+rq
   pJ8EwuZ0F11KPSJu1q/LnKmsEiwUcOEcY9TAqyQcapOK1Iv5mlqZuQu6gyXeYQR1
   QCWKt5Wala0FHdqW6xVDHf719eIlXKeCYVRuM5o=
   -----END PGP MESSAGE-----
```

A.6. Sample AEAD-OCB encryption and decryption

   This example encrypts the cleartext string Hello, world! with the

password password, using AES-128 with AEAD-OCB encryption.

. Sample Parameters

    S2K:

        Iterated and Salted S2K

    Iterations:

        65011712 (255), SHA2-256

    Salt:

        56 a2 98 d2 f5 e3 64 53

. Sample symmetric-key encrypted session key packet (v5)

    Packet header:

        c3 3f

    Version, algorithms, S2K fields:

        05 1d 07 02 0b 03 08 56 a2 98 d2 f5 e3 64 53 ff
        cf cc

    Nonce:

        cf cc 5c 11 66 4e db 9d b4 25 90 d7 dc 46 b0

    Encrypted session key and AEAD tag:

        78 c5 c0 41 9c c5 1b 3a 46 87 cb 32 e5 b7 03 1c
        e7 c6 69 75 76 5b 5c 21 d9 2a ef 4c c0 5c 3f ea

. 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 05 07 02

   HKDF output:

      20 62 fb 76 31 ef be f4 df 81 67 ce d7 f3 a4 64
```

```
   Authenticated Data:

      c3 05 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.6.4. Sample v2 SEIPD packet

```
   Packet header:

      d2 69

   Version, AES-128, OCB, Chunk size octet:

      02 07 02 06

   Salt:

      20 a6 61 f7 31 fc 9a 30 32 b5 62 33 26 02 7e 3a
      5d 8d b5 74 8e be ff 0b 0c 59 10 d0 9e cd d6 41

   Chunk #0 encrypted data:

      ff 9f d3 85 62 75 80 35 bc 49 75 4c e1 bf 3f ff
      a7 da d0 a3 b8 10 4f 51 33 cf 42 a4 10 0a 83 ee
      f4 ca 1b 48 01
```

Chunk #0 authentication tag:

      a8 84 6b f4 2b cd a7 c8 ce 9d 65 e2 12 f3 01 cb

   Final (zero-sized chunk #1) authentication tag:

      cd 98 fd ca de 69 4a 87 7a d4 24 73 23 f6 e8 57

A.6.5. 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.6.6. Complete AEAD-OCB encrypted packet sequence

```
-----BEGIN PGP MESSAGE-----

wz8FHQcCCwMIVqKY0vXjZFP/z8xcEWZO2520JZDX3EaweMXAQZzFGzpGh8sy5bcD
HOfGaXV2W1wh2SrvTMBcP+rSaQIHAgYgpmH3MfyaMDK1YjMmAn46XY21dI6+/wsM
WRDQns3WQf+f04VidYA1vEl1TOG/P/+n2tCjuBBPUTPPQqQQCoPu9MobSAGohGv0
K82nyM6dZeIS8wHLzZj9yt5pSod61CRzI/boVw==
-----END PGP MESSAGE-----
```

A.7. Sample AEAD-GCM encryption and decryption

This example encrypts the cleartext string Hello, world! with the
password password, using AES-128 with AEAD-GCM encryption.

A.7.1. Sample Parameters

S2K:

   Iterated and Salted S2K

Iterations:

     65011712 (255), SHA2-256

   Salt:

     e9 d3 97 85 b2 07 00 08

A.7.2. Sample symmetric-key encrypted session key packet (v5)

   Packet header:

     c3 3c

   Version, algorithms, S2K fields:

     05 1a 07 03 0b 03 08 e9 d3 97 85 b2 07 00 08 ff
     b4 2e

   Nonce:

     b4 2e 7c 48 3e f4 88 44 57 cb 37 26

   Encrypted session key and AEAD tag:

     0c 0c 4b f3 f2 cd 6c b7 b6 e3 8b 5b f3 34 67 c1
     c7 19 44 dd 59 03 46 66 2f 5a de 61 ff 84 bc e0

A.7.3. 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 05 07 03

   HKDF output:

```
     de ec e5 81 8b c0 aa b9 0f 8a fb 02 fa 00 cd 13
```

   Authenticated Data:

```
     c3 05 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.7.4. Sample v2 SEIPD packet

   Packet header:

```
     d2 69
```

   Version, AES-128, GCM, Chunk size octet:

```
     02 07 03 06
```

   Salt:

```
     fc b9 44 90 bc b9 8b bd c9 d1 06 c6 09 02 66 94
     0f 72 e8 9e dc 21 b5 59 6b 15 76 b1 01 ed 0f 9f
```

   Chunk #0 encrypted data:

```
     fc 6f c6 d6 5b bf d2 4d cd 07 90 96 6e 6d 1e 85
     a3 00 53 78 4c b1 d8 b6 a0 69 9e f1 21 55 a7 b2
     ad 62 58 53 1b
```

   Chunk #0 authentication tag:

```
     57 65 1f d7 77 79 12 fa 95 e3 5d 9b 40 21 6f 69
```

   Final (zero-sized chunk #1) authentication tag:

```
      a4 c2 48 db 28 ff 43 31 f1 63 29 07 39 9e 6f f9
```

A.7.5. 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.7.6.  Complete AEAD-GCM encrypted packet sequence

      -----BEGIN PGP MESSAGE-----

      wzwFGgcDCwMI6dOXhbIHAAj/tC58SD70iERXyzcmDAxL8/LNbLe244tb8zRnwccZ
      RN1ZA0ZmL1reYf+EvODSaQIHAwb8uUSQvLmLvcnRBsYJAmaUD3LontwhtVlrFXax
      Ae0Pn/xvxtZbv9JNzQeQlm5tHoWjAFN4TLHYtqBpnvEhVaeyrWJYUxtXZR/Xd3kS
      +pXjXZtAIW9ppMJI2yj/QzHxYykHOZ5v+Q==
      -----END PGP MESSAGE-----

A.8.  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.8.1.  v4 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.8.2.  v4 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.8.3](#).  v4 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](#). Acknowledgements

   Thanks to the openpgp design team for working on this document to
   prepare it for working group consumption: Stephen Farrell, Daniel
   Kahn Gillmor, Daniel Huigens, Jeffrey Lau, Yutaka Niibe, Justus
   Winter and Paul Wouters.

   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, David Shaw, William Stallings,
   Mark Weaver, and Philip R. Zimmermann.

Authors' Addresses

   Werner Koch (editor)
   GnuPG e.V.
   Rochusstr. 44

        40479 Duesseldorf
        Germany
        Email: wk@gnupg.org
        URI: https://gnupg.org/verein

        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