Abstract

( Work in progress to update the OpenPGP specification from RFC4880 )

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.

Status of This Memo

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

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

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

This Internet-Draft will expire on 21 April 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.
1. Introduction .................................................. 6
   1.1. Terms ..................................................... 7
2. General functions ............................................. 7
   2.1. Confidentiality via Encryption .......................... 8
   2.2. Authentication via Digital Signature ................... 9
   2.3. Compression .............................................. 9
   2.4. Conversion to Radix-64 .................................. 10
   2.5. Signature-Only Applications ............................. 10
3. Data Element Formats ......................................... 10
   3.1. Scalar Numbers .......................................... 10
   3.2. Multiprecision Integers ................................. 10
       3.2.1. Using MPIs to encode other data .................. 11
   3.3. Key IDs ................................................ 11
   3.4. Text .................................................. 11
   3.5. Time Fields ............................................ 11
   3.6. Keyrings ............................................... 12
   3.7. String-to-Key (S2K) Specifiers ......................... 12
       3.7.1. String-to-Key (S2K) Specifier Types .............. 12
           3.7.1.1. Simple S2K .................................. 12
           3.7.1.2. Salted S2K ................................ 13
           3.7.1.3. Iterated and Salted S2K .................... 13
           3.7.1.4. Argon2 ..................................... 14
       3.7.2. String-to-Key Usage ............................... 15
           3.7.2.1. Secret-Key Encryption ....................... 15
           3.7.2.2. Symmetric-Key Message Encryption ............ 16
4. Packet Syntax ................................................ 16
   4.1. Overview ............................................... 16
   4.2. Packet Headers ......................................... 16
       4.2.1. Old Format Packet Lengths ....................... 17
       4.2.2. New Format Packet Lengths ........................ 18
           4.2.2.1. One-Octet Lengths .......................... 18
           4.2.2.2. Two-Octet Lengths .......................... 18
           4.2.2.3. Five-Octet Lengths .......................... 18
           4.2.2.4. Partial Body Lengths ....................... 19
       4.2.3. Packet Length Examples ............................ 19
   4.3. Packet Tags ........................................... 20
5. Packet Types ................................................ 21
5.1. Public-Key Encrypted Session Key Packets (Tag 1) . . . . 22
  5.1.1. Algorithm-Specific Fields for RSA encryption . . . . 22
  5.1.2. Algorithm-Specific Fields for Elgamal encryption . . 22
  5.1.3. Algorithm-Specific Fields for ECDH encryption . . . 22
  5.1.4. Notes on PKESK . . . . . . . . . . . . . . . . . . . 23
5.2. Signature Packet (Tag 2) . . . . . . . . . . . . . . . . 23
  5.2.1. Signature Types . . . . . . . . . . . . . . . . . . . 24
  5.2.2. Version 3 Signature Packet Format . . . . . . . . . 26
  5.2.3. Version 4 and 5 Signature Packet Formats . . . . . 29
    5.2.3.1. Algorithm-Specific Fields for RSA signatures . . 30
    5.2.3.2. Algorithm-Specific Fields for DSA or ECDSA
               signatures . . . . . . . . . . . . . . . . . . . . . 30
    5.2.3.3. Algorithm-Specific Fields for EdDSA signatures . 30
    5.2.3.4. Notes on Signatures . . . . . . . . . . . . . . . 31
    5.2.3.5. Signature Subpacket Specification . . . . . . . 32
    5.2.3.6. Signature Subpacket Types . . . . . . . . . . . 34
    5.2.3.7. Notes on Self-Signatures . . . . . . . . . . . . 35
    5.2.3.8. Signature Creation Time . . . . . . . . . . . . 36
    5.2.3.9. Issuer . . . . . . . . . . . . . . . . . . . . . 36
    5.2.3.10. Key Expiration Time . . . . . . . . . . . . . . 36
    5.2.3.11. Preferred Symmetric Algorithms . . . . . . . . 36
    5.2.3.12. Preferred Hash Algorithms . . . . . . . . . . . 37
    5.2.3.13. Preferred Compression Algorithms . . . . . . . 37
    5.2.3.14. Signature Expiration Time . . . . . . . . . . . 37
    5.2.3.15. Exportable Certification . . . . . . . . . . . . 37
    5.2.3.16. Revocable . . . . . . . . . . . . . . . . . . . 38
    5.2.3.17. Trust Signature . . . . . . . . . . . . . . . . 38
    5.2.3.18. Regular Expression . . . . . . . . . . . . . . . 38
    5.2.3.19. Revocation Key . . . . . . . . . . . . . . . . 39
    5.2.3.20. Notation Data . . . . . . . . . . . . . . . . . 39
    5.2.3.21. Key Server Preferences . . . . . . . . . . . . 40
    5.2.3.22. Preferred Key Server . . . . . . . . . . . . . 41
    5.2.3.23. Primary User ID . . . . . . . . . . . . . . . . 41
    5.2.3.24. Policy URI . . . . . . . . . . . . . . . . . . . 41
    5.2.3.25. Key Flags . . . . . . . . . . . . . . . . . . . 42
    5.2.3.26. Signer’s User ID . . . . . . . . . . . . . . . 43
    5.2.3.27. Reason for Revocation . . . . . . . . . . . . . 43
    5.2.3.28. Features . . . . . . . . . . . . . . . . . . . . 45
    5.2.3.29. Signature Target . . . . . . . . . . . . . . . 45
    5.2.3.30. Embedded Signature . . . . . . . . . . . . . . 46
    5.2.3.31. Issuer Fingerprint . . . . . . . . . . . . . . . 46
    5.2.3.32. Intended Recipient Fingerprint . . . . . . . . 46
5.2.4. Computing Signatures . . . . . . . . . . . . . . . . . 46
  5.2.4.1. Subpacket Hints . . . . . . . . . . . . . . . . . 49
5.3. Symmetric-Key Encrypted Session Key Packets (Tag 3) . . 49
  5.3.1. No v5 SKESK with SEIPD . . . . . . . . . . . . . . . 51
5.4. One-Pass Signature Packets (Tag 4) . . . . . . . . . . . 51
5.5. Key Material Packet . . . . . . . . . . . . . . . . . . . 52
5.5.1. Key Packet Variants .................................. 52
5.5.1.1. Public-Key Packet (Tag 6) ......................... 52
5.5.1.2. Public-Subkey Packet (Tag 14) .................... 52
5.5.1.3. Secret-Key Packet (Tag 5) ......................... 52
5.5.1.4. Secret-Subkey Packet (Tag 7) ..................... 52
5.5.2. Public-Key Packet Formats ............................. 53
5.5.3. Secret-Key Packet Formats .............................. 54
5.6. Algorithm-specific Parts of Keys ......................... 57
5.6.1. Algorithm-Specific Part for RSA Keys ................ 57
5.6.2. Algorithm-Specific Part for DSA Keys ................ 57
5.6.3. Algorithm-Specific Part for Elgamal Keys ............. 57
5.6.4. Algorithm-Specific Part for ECDSA Keys ............... 58
5.6.5. Algorithm-Specific Part for EdDSA Keys ............... 58
5.6.6. Algorithm-Specific Part for ECDH Keys ............... 59
5.7. Compressed Data Packet (Tag 8) ......................... 59
5.8. Symmetrically Encrypted Data Packet (Tag 9) .......... 60
5.9. Marker Packet (Obsolete Literal Packet) (Tag 10) ...... 61
5.10. Literal Data Packet (Tag 11) ......................... 61
5.11. Trust Packet (Tag 12) .................................. 62
5.12. User ID Packet (Tag 13) ............................... 63
5.13. User Attribute Packet (Tag 17) ....................... 63
5.13.1. The Image Attribute Subpacket ....................... 64
5.14. Sym. Encrypted Integrity Protected Data Packet (Tag 18) ......................................................... 64
5.15. Modification Detection Code Packet (Tag 19) .......... 67
5.16. AEAD Encrypted Data Packet (Tag 20) .................. 68
5.16.1. EAX Mode .............................................. 69
5.16.2. OCB Mode .............................................. 70
6. Radix-64 Conversions ....................................... 70
6.1. An Implementation of the CRC-24 in "C" ................ 71
6.2. Forming ASCII Armor ...................................... 71
6.3. Encoding Binary in Radix-64 ............................. 74
6.4. Decoding Radix-64 ....................................... 76
6.5. Examples of Radix-64 .................................... 76
6.6. Example of an ASCII Armored Message .................... 77
7. Cleartext Signature Framework ............................... 77
7.1. Dash-Escaped Text ....................................... 78
8. Regular Expressions ......................................... 79
9. Constants .................................................. 79
9.1. Public-Key Algorithms .................................. 80
9.2. ECC Curves for OpenPGP ................................. 82
9.2.1. Curve-Specific Wire Formats ......................... 83
9.3. Symmetric-Key Algorithms .............................. 84
9.4. Compression Algorithms ................................. 85
9.5. Hash Algorithms ........................................ 86
9.6. AEAD Algorithms ........................................ 87
10. IANA Considerations ....................................... 87
10.1. New String-to-Key Specifier Types .................... 87
1. Introduction

{ This is work in progress to update OpenPGP. Editorial notes are
enclosed in curly braces. }

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", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119].

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
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. Note that the message is also usually compressed.
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

OpenPGP implementations SHOULD compress the message after applying
the signature but before encryption.

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.

Furthermore, compression has the added side effect that some types of
attacks can be thwarted by the fact that slightly altered, compressed
data rarely uncompresses without severe errors. This is hardly
rigorous, but it is operationally useful. These attacks can be
rigorously prevented by implementing and using Modification Detection
Codes as described in sections following.
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 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 \((\text{MPI.length} + 7) / 8\) + 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 13.2, or an octet string of known, fixed length (see Section 13.3). 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 12 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

String-to-key (S2K) specifiers are used to convert passphrase strings into symmetric-key encryption/decryption keys. 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 three types of S2K specifiers currently supported, and some reserved values:

<table>
<thead>
<tr>
<th>ID</th>
<th>S2K Type</th>
<th>Generate?</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Simple S2K</td>
<td>N</td>
<td>Section 3.7.1.1</td>
</tr>
<tr>
<td>1</td>
<td>Salted S2K</td>
<td>Only when string is high entropy</td>
<td>Section 3.7.1.2</td>
</tr>
<tr>
<td>2</td>
<td>Reserved value</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Iterated and Salted S2K</td>
<td>Y</td>
<td>Section 3.7.1.3</td>
</tr>
<tr>
<td>4</td>
<td>Argon2</td>
<td>Y</td>
<td>Section 3.7.1.4</td>
</tr>
<tr>
<td>100 to 110</td>
<td>Private/Experimental S2K</td>
<td>As appropriate</td>
<td></td>
</tr>
</tbody>
</table>

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 $2^{\text{encoded}_m}$, where "encoded_m" is the encoded memory size in Octet 19. The encoded memory size MUST be a value from $3+\text{ceil}(\log_2(p))$ to 31, such that the decoded memory size $m$ is a value from $8p$ to $2^{31}$.

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 $8p$).

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 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 (e.g., 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 then by the S2K specifier as encoded above.
Therefore, preceding the secret data there will be one of these possibilities:

0:                 secret data is unencrypted (no passphrase)
255, 254, or 253:  followed by algorithm octet and S2K specifier
            Cipher alg:        use Simple S2K algorithm using MD5 hash

This last possibility, the cipher algorithm number with an implicit use of MD5 and IDEA, is provided for backward compatibility; it MAY be understood, but SHOULD NOT be generated, and is deprecated.

These are followed by an Initial Vector of the same length as the block size of the cipher for the decryption of the secret values, if they are encrypted, and then the secret-key values themselves.

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. This is deprecated, but MAY be used 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.

4.2. 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.
Note that the most significant bit is the leftmost bit, called bit 7. A mask for this bit is 0x80 in hexadecimal.

\[
\text{PTag 7 6 5 4 3 2 1 0}
\]

- Bit 7 -- Always one
- Bit 6 -- New packet format if set

PGP 2.6.x only uses old format packets. Thus, software that interoperates with those versions of PGP must only use old format packets. If interoperability is not an issue, the new packet format is RECOMMENDED. Note that old format packets have four bits of packet tags, and new format packets have six; some features cannot be used and still be backward-compatible.

Also note that packets with a tag greater than or equal to 16 MUST use new format packets. The old format packets can only express tags less than or equal to 15.

Old format packets contain:

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

New format packets contain:

- Bits 5-0 -- packet tag

4.2.1. Old Format Packet Lengths

The meaning of the length-type in old 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. In general, an implementation SHOULD NOT use indeterminate-length packets except where the end of the data will be clear from the context, and even then it is better to use a definite length, or a new format header. The new format headers described below have a mechanism for precisely encoding data of indeterminate length.
4.2.2. New Format Packet Lengths

New 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 (0xFFFFFFF) 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.

4.2.2.1. 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:

\[
\text{bodyLen} = \text{1st\_octet};
\]

4.2.2.2. 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:

\[
\text{bodyLen} = ((\text{1st\_octet} - 192) \ll 8) + \text{(2nd\_octet)} + 192
\]

4.2.2.3. 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:

\[
\text{bodyLen} = (\text{2nd\_octet} \ll 24) | (\text{3rd\_octet} \ll 16) | (\text{4th\_octet} \ll 8) | \text{5th\_octet}
\]

This basic set of one, two, and five-octet lengths is also used internally to some packets.
4.2.2.4. 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:

\[
\text{partialBodyLen} = 1 << (1st\_octet \& 0x1F);
\]

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.

4.2.3. Packet Length Examples

These examples show ways that new 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 old format headers can only have tags less than 16, whereas new format headers can have tags as great as 63. The defined tags (in decimal) are as follows:
<table>
<thead>
<tr>
<th>Tag</th>
<th>Packet Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved - a packet tag MUST NOT have this value</td>
</tr>
<tr>
<td>1</td>
<td>Public-Key Encrypted Session Key Packet</td>
</tr>
<tr>
<td>2</td>
<td>Signature Packet</td>
</tr>
<tr>
<td>3</td>
<td>Symmetric-Key Encrypted Session Key Packet</td>
</tr>
<tr>
<td>4</td>
<td>One-Pass Signature Packet</td>
</tr>
<tr>
<td>5</td>
<td>Secret-Key Packet</td>
</tr>
<tr>
<td>6</td>
<td>Public-Key Packet</td>
</tr>
<tr>
<td>7</td>
<td>Secret-Subkey Packet</td>
</tr>
<tr>
<td>8</td>
<td>Compressed Data Packet</td>
</tr>
<tr>
<td>9</td>
<td>Symmetrically Encrypted Data Packet</td>
</tr>
<tr>
<td>10</td>
<td>Marker Packet</td>
</tr>
<tr>
<td>11</td>
<td>Literal Data Packet</td>
</tr>
<tr>
<td>12</td>
<td>Trust Packet</td>
</tr>
<tr>
<td>13</td>
<td>User ID Packet</td>
</tr>
<tr>
<td>14</td>
<td>Public-Subkey Packet</td>
</tr>
<tr>
<td>17</td>
<td>User Attribute Packet</td>
</tr>
<tr>
<td>18</td>
<td>Sym. Encrypted and Integrity Protected Data Packet</td>
</tr>
<tr>
<td>19</td>
<td>Modification Detection Code Packet</td>
</tr>
<tr>
<td>20</td>
<td>AEAD Encrypted Data Packet</td>
</tr>
<tr>
<td>60</td>
<td>Private or Experimental Values</td>
</tr>
</tbody>
</table>

Table 2: Packet type registry

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

Zero or more Public-Key Encrypted Session Key packets and/or Symmetric-Key Encrypted Session Key packets may precede an encryption container (i.e. a Symmetrically Encrypted Integrity Protected Data packet, an AEAD Encrypted 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 consists of:

* A one-octet number giving the version number of the packet type. The currently defined value for packet version is 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.

* A one-octet number giving the public-key algorithm used.

* A string of octets that is the encrypted session key. This string takes up the remainder of the packet, and its contents are dependent on the public-key algorithm used.

5.1.1. Algorithm Specific Fields for RSA encryption

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

5.1.2. 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.

5.1.3. 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 13.5.

5.1.4. Notes on PKESK

The value "m" in the above formulas is derived from the session key as follows. First, the session key is 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. This value is then encoded as described in PKCS#1 block encoding EME-PKCS1-v1_5 in Section 7.2.1 of [RFC3447] to form the "m" value used in the formulas above. See Section 14.1 in this document for notes on OpenPGP’s use of PKCS#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.

An implementation MAY accept or use a Key ID of zero as a "wild card" or "speculative" Key ID. 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. PGP 2.6.x only accepts version 3 signatures.

Implementations MUST generate version 5 signatures when using a version 5 key. Implementations SHOULD generate V4 signatures with version 4 keys. 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. Note that it doesn’t make sense to have a V3 standalone signature.

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 Revocation Key subpacket. It is also appropriate for statements that non-self certifiers want to make about the key itself, rather than the binding between a key and a name.

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 an authorized 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 an authorized 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 an authorized 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 and ECDSA signatures:

* MPI of DSA or ECDSA value r.
* MPI of DSA or ECDSA 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 [RFC3447]. 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:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Hexadecimal Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD5</td>
<td>0x2A, 0x86, 0x48, 0x86, 0xFD, 0x0D, 0x02, 0x05</td>
</tr>
<tr>
<td>RIPEMD-160</td>
<td>0x2B, 0x24, 0x03, 0x02, 0x01</td>
</tr>
<tr>
<td>SHA-1</td>
<td>0x2B, 0x0E, 0x03, 0x02, 0x1A</td>
</tr>
<tr>
<td>SHA224</td>
<td>0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x04</td>
</tr>
<tr>
<td>SHA256</td>
<td>0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01</td>
</tr>
<tr>
<td>SHA384</td>
<td>0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x02</td>
</tr>
<tr>
<td>SHA512</td>
<td>0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x03</td>
</tr>
</tbody>
</table>

Table 3: Hash hexadecimal representations

The ASN.1 Object Identifiers (OIDs) are as follows:
### Table 4: Hash OIDs

The full hash prefixes for these are as follows:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>OID</th>
</tr>
</thead>
<tbody>
<tr>
<td>MD5</td>
<td>1.2.840.113549.2.5</td>
</tr>
<tr>
<td>RIPEMD-160</td>
<td>1.3.36.3.2.1</td>
</tr>
<tr>
<td>SHA-1</td>
<td>1.3.14.3.2.26</td>
</tr>
<tr>
<td>SHA224</td>
<td>2.16.840.1.101.3.4.2.4</td>
</tr>
<tr>
<td>SHA256</td>
<td>2.16.840.1.101.3.4.2.1</td>
</tr>
<tr>
<td>SHA384</td>
<td>2.16.840.1.101.3.4.2.2</td>
</tr>
<tr>
<td>SHA512</td>
<td>2.16.840.1.101.3.4.2.3</td>
</tr>
<tr>
<td>Algorithm</td>
<td>Full Hash Prefix</td>
</tr>
<tr>
<td>--------------</td>
<td>------------------</td>
</tr>
<tr>
<td>MD5</td>
<td>0x30, 0x20, 0x30, 0x0C, 0x06, 0x08, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0xD0, 0x02, 0x05, 0x00, 0x04, 0x10</td>
</tr>
<tr>
<td>RIPEMD-160</td>
<td>0x30, 0x21, 0x30, 0x09, 0x06, 0x05, 0x2B, 0x24, 0x03, 0x02, 0x01, 0x05, 0x00, 0x04, 0x14</td>
</tr>
<tr>
<td>SHA-1</td>
<td>0x30, 0x21, 0x30, 0x09, 0x06, 0x05, 0x2B, 0x0E, 0x03, 0x02, 0x1A, 0x05, 0x00, 0x04, 0x14</td>
</tr>
<tr>
<td>SHA224</td>
<td>0x30, 0x2D, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x04, 0x05, 0x00, 0x04, 0x1C</td>
</tr>
<tr>
<td>SHA256</td>
<td>0x30, 0x31, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x04, 0x05, 0x00, 0x04, 0x20</td>
</tr>
<tr>
<td>SHA384</td>
<td>0x30, 0x41, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x04, 0x05, 0x00, 0x04, 0x30</td>
</tr>
<tr>
<td>SHA512</td>
<td>0x30, 0x51, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x03, 0x05, 0x00, 0x04, 0x40</td>
</tr>
</tbody>
</table>

Table 5: 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.

* Two-octet scalar octet count for following hashed subpacket data. 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).

* Two-octet scalar octet count for the following unhashed subpacket data. 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.

* 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.

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 58 octets: a prefix 0x40 octet, followed by 57 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 difference between a V4 and V5 signature is that the latter includes additional meta data.

The algorithms for converting the hash function result to a signature are described in a section below.
5.2.3.5. 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 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:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Reserved</td>
</tr>
<tr>
<td>1</td>
<td>Reserved</td>
</tr>
<tr>
<td>2</td>
<td>Signature Creation Time</td>
</tr>
<tr>
<td>3</td>
<td>Signature Expiration Time</td>
</tr>
<tr>
<td>4</td>
<td>Exportable Certification</td>
</tr>
<tr>
<td>5</td>
<td>Trust Signature</td>
</tr>
<tr>
<td>----</td>
<td>-----------------</td>
</tr>
<tr>
<td>6</td>
<td>Regular Expression</td>
</tr>
<tr>
<td>7</td>
<td>Revocable</td>
</tr>
<tr>
<td>8</td>
<td>Reserved</td>
</tr>
<tr>
<td>9</td>
<td>Key Expiration Time</td>
</tr>
<tr>
<td>10</td>
<td>Placeholder for backward compatibility</td>
</tr>
<tr>
<td>11</td>
<td>Preferred Symmetric Algorithms</td>
</tr>
<tr>
<td>12</td>
<td>Revocation Key</td>
</tr>
<tr>
<td>13 to 15</td>
<td>Reserved</td>
</tr>
<tr>
<td>16</td>
<td>Issuer</td>
</tr>
<tr>
<td>17 to 19</td>
<td>Reserved</td>
</tr>
<tr>
<td>20</td>
<td>Notation Data</td>
</tr>
<tr>
<td>21</td>
<td>Preferred Hash Algorithms</td>
</tr>
<tr>
<td>22</td>
<td>Preferred Compression Algorithms</td>
</tr>
<tr>
<td>23</td>
<td>Key Server Preferences</td>
</tr>
<tr>
<td>24</td>
<td>Preferred Key Server</td>
</tr>
<tr>
<td>25</td>
<td>Primary User ID</td>
</tr>
<tr>
<td>26</td>
<td>Policy URI</td>
</tr>
<tr>
<td>27</td>
<td>Key Flags</td>
</tr>
<tr>
<td>28</td>
<td>Signer’s User ID</td>
</tr>
<tr>
<td>29</td>
<td>Reason for Revocation</td>
</tr>
<tr>
<td>30</td>
<td>Features</td>
</tr>
<tr>
<td>31</td>
<td>Signature Target</td>
</tr>
<tr>
<td>32</td>
<td>Embedded Signature</td>
</tr>
</tbody>
</table>
Table 6: 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 three preferred algorithm subpackets (11, 21, and 22), 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). 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 symmetric algorithm AES-256, and Bob prefers Camellia-256 or AES-128. If the software locates this key via Alice’s name, then the preferred algorithm is AES-256; if software locates the key via Bob’s name, then the preferred algorithm is Camellia-256. If the key is located by Key ID, the algorithm of the primary User ID of the key provides the preferred symmetric algorithm.

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 5.2.3.27 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 may resolve the ambiguity in any way it sees fit, but it is RECOMMENDED that priority be given to the most recent self-signature.

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

(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.

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. 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 Algorithms

(array of one-octet values)

Symmetric algorithm numbers that indicate which algorithms the key holder prefers to use. 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.
5.2.3.12. 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 symmetric algorithms, the list is ordered. Algorithm numbers are in Section 9.5. This is only found on a self-signature.

5.2.3.13. Preferred Compression Algorithms

(array of one-octet values)

Compression algorithm numbers that indicate which algorithms the key holder prefers to use. Like the preferred symmetric algorithms, the list is ordered. Algorithm numbers are in Section 9.4. If this subpacket is not included, ZIP is preferred. A zero 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.

5.2.3.14. 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.

5.2.3.15. 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.

5.2.3.16. 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.

5.2.3.17. 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, i.e., that it 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.

5.2.3.18. 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.

5.2.3.19. Revocation Key

(1 octet of class, 1 octet of public-key algorithm ID, 20 or 32 octets of fingerprint)

V4 keys use the full 20 octet fingerprint; V5 keys use the full 32 octet fingerprint

Authorizes 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.

5.2.3.20. 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:

First octet:
Table 7: Notation flag registry (first octet)

Other octets: none.

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.

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.

5.2.3.21. 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:
### Table 8: Key server preferences flag registry (first octet)

<table>
<thead>
<tr>
<th>flag</th>
<th>shorthand</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x80</td>
<td>No-modify</td>
<td>The key holder requests that this key only be modified or updated by the key holder or an administrator of the key server.</td>
</tr>
</tbody>
</table>

This is found only on a self-signature.

#### 5.2.3.22. 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.

#### 5.2.3.23. 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.

#### 5.2.3.24. Policy URI

(String)

This subpacket contains a URI of a document that describes the policy under which the signature was issued.
5.2.3.25. 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:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x01</td>
<td>This key may be used to certify other keys.</td>
</tr>
<tr>
<td>0x02</td>
<td>This key may be used to sign data.</td>
</tr>
<tr>
<td>0x04</td>
<td>This key may be used to encrypt communications.</td>
</tr>
<tr>
<td>0x08</td>
<td>This key may be used to encrypt storage.</td>
</tr>
<tr>
<td>0x10</td>
<td>The private component of this key may have been split by a secret-sharing mechanism.</td>
</tr>
<tr>
<td>0x20</td>
<td>This key may be used for authentication.</td>
</tr>
<tr>
<td>0x80</td>
<td>The private component of this key may be in the possession of more than one person.</td>
</tr>
</tbody>
</table>

Table 9: Key flags registry (first octet)

Second octet:

<table>
<thead>
<tr>
<th>Flag</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x04</td>
<td>Reserved (ADSK).</td>
</tr>
<tr>
<td>0x08</td>
<td>Reserved (timestamping).</td>
</tr>
</tbody>
</table>

Table 10: 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.

5.2.3.26. 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.

5.2.3.27. 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:
### Table 11: 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, that is, 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.

<table>
<thead>
<tr>
<th>Code</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>No reason specified (key revocations or cert revocations)</td>
</tr>
<tr>
<td>1</td>
<td>Key is superseded (key revocations)</td>
</tr>
<tr>
<td>2</td>
<td>Key material has been compromised (key revocations)</td>
</tr>
<tr>
<td>3</td>
<td>Key is retired and no longer used (key revocations)</td>
</tr>
<tr>
<td>32</td>
<td>User ID information is no longer valid (cert revocations)</td>
</tr>
<tr>
<td>100-110</td>
<td>Private Use</td>
</tr>
</tbody>
</table>
5.2.3.28. 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                                                  |
+----------+-------------------------------------------------------------+
| 0x01     | Modification Detection (packets 18 and 19)                  |
+----------+-------------------------------------------------------------+
| 0x02     | AEAD Encrypted Data (packet 20)                             |
+----------+-------------------------------------------------------------+
| 0x04     | Reserved                                                    |
+----------+-------------------------------------------------------------+

Table 12: 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.

5.2.3.29. 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.

5.2.3.30. 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.

5.2.3.31. 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 subpacket is also included in the signature, the key ID of the Issuer 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.

5.2.3.32. 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.

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

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.

For binary document signatures (type 0x00), the document data is hashed directly. For text document signatures (type 0x01), the document is canonicalized by converting line endings to <CR><LF>, and the resulting data 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 an old-style 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 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:
  - the signature version (0x04),
  - the signature type,
  - the public-key algorithm,
  - the hash algorithm,
- the hashed subpacket length,
- the hashed subpacket body,
- the two octets 0x04 and 0xFF,
- a four-octet big-endian number that is the length of the hashed data from the Signature packet stopping right before the 0x04, 0xff octets.

The four-octet big-endian number is considered to be an unsigned integer modulo 2**32.

* A 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:

- the signature version (0x05),
- the signature type,
- the public-key algorithm,
- the hash algorithm,
- the hashed subpacket length,
- the hashed subpacket body,
- Only for document signatures (type 0x00 or 0x01) the following three data items are hashed here:
  - the one-octet content format,
  - the file name as a string (one octet length, followed by the file name),
  - a four-octet number that indicates a date,
- the two octets 0x05 and 0xFF,
- a eight-octet big-endian number that is the length of the hashed data from the Signature packet stopping right before the 0x05, 0xff octets.

The three data items hashed for document signatures need to mirror the values of the Literal Data packet. For detached and cleartext signatures 6 zero octets are hashed instead.
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 subpacket for each key, as a way of explicitly tying those keys to the signature.

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 and/or Symmetric-Key Encrypted Session Key packets may precede a an encryption container (i.e. a Symmetrically Encrypted Integrity Protected Data packet, an AEAD Encrypted 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 or the Symmetric-Key Encrypted Session Key packet.

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. This packet type is new and is not generated by PGP 2 or PGP version 5.0.

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, length as defined above.

* 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 or an Iterated-Salted S2K. The salt value will ensure that the decryption key is not repeated even if the passphrase is reused.

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

* A one-octet version number with value 5.

* A one-octet cipher algorithm.

* A one-octet AEAD algorithm.

* A string-to-key (S2K) specifier, length as defined above.

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

* The encrypted session key itself, which is decrypted with the string-to-key object using the given cipher and AEAD mode.

* An authentication tag for the AEAD mode.

The encrypted session key is encrypted using one of the AEAD algorithms specified for the AEAD Encrypted Data Packet. Note that no chunks are used and that there is only one authentication tag. The Packet Tag in new 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 EAX and AES-128 consists of the octets 0xC3, 0x05, 0x07, and 0x01.

5.3.1. No v5 SKESK with SEIPD

Note that unlike the AEAD Encrypted Data Packet (AED, see Section 5.16), the Symmetrically Encrypted Integrity Protected Data Packet (SEIPD, see Section 5.14) does not internally indicate what cipher algorithm to use to decrypt it. Since the v5 SKESK packet’s encrypted payload only indicates the key used, not the choice of cipher algorithm used for the subsequent encrypted data, a v5 SKESK packet can only provide a session key for an AED packet, and MUST NOT be used to provide a session key for a SEIPD Packet.

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.

A One-Pass Signature does not interoperate with PGP 2.6.x or earlier.

The body of this packet consists of:

* A one-octet version number. The current version is 3.

* 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.

* An eight-octet number holding the Key ID of the signing key.

* 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.
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.

Note: in PGP version 2.6, tag 14 was intended to indicate a comment packet. This tag was selected for reuse because no previous version of PGP ever emitted comment packets but they did properly ignore them. Public-Subkey packets are ignored by PGP version 2.6 and do not cause it to fail, providing a limited degree of backward compatibility.

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. Version 3 packets were first generated by PGP version 2.6. Version 4 keys first appeared in PGP 5 and are the preferred key version for OpenPGP.

OpenPGP implementations MUST create keys with version 4 format. V3 keys are deprecated; an implementation MUST NOT generate a V3 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 below for a fuller discussion of Key IDs and fingerprints.

V2 keys are identical to the deprecated V3 keys except for the version number. An implementation MUST NOT generate them and MAY accept or reject them as it sees fit.

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 12.

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 multiprecision integers comprising the key material. This is algorithm-specific and described in Section 5.6.

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 the section "Enhanced Key Formats".

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 5.6.

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:

* 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 or 254 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, a one-octet scalar octet count of the next 4 optional 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] If string-to-key usage octet was 255, 254, or 253, a string-to-key specifier. The length of the string-to-key specifier is implied by its type, as described above.

* [Optional] If string-to-key usage octet was 253 (i.e. the secret data is AEAD-encrypted), an initialization vector (IV) of size specified by the AEAD algorithm (see Section 5.16), 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 (i.e. the secret data is CFB-encrypted), an initialization vector (IV) of the same length as the cipher’s block size.

* Only for a version 5 packet, a four-octet scalar octet count for the following secret key material. This includes the encrypted SHA-1 hash or AEAD tag if the string-to-key usage octet is 254 or 253.

* Plain or encrypted multiprecision integers comprising the secret key data. This is algorithm-specific and described in section Section 5.6. 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 (i.e., 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).

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 (i.e., 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 encrypted MPI values are encrypted as one combined plaintext using one of the AEAD algorithms specified for the AEAD Encrypted Data Packet. Note that no chunks are used and that there is only one authentication tag. As additional data, the Packet Tag in new 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.
5.6. 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.

5.6.1. 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.

5.6.2. 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.

5.6.3. 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.

5.6.4. 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.

5.6.5. 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 13.2.2).

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.

See [RFC8032] for more details about the native octet strings.
5.6.6. 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 13.5 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.

5.7. 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. Note that PGP V2.6 uses 13 bits of compression. If an implementation uses more bits of compression, PGP V2.6 cannot decompress it.

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

BZip2-compressed packets are compressed using the BZip2 [BZ2] algorithm.

5.8. 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.

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 15 for hints on the proper use of this "quick check".

5.9. Marker Packet (Obsolete Literal Packet) (Tag 10)

An experimental version of PGP used this packet as the Literal packet, but no released version of PGP generated Literal packets with this tag. With PGP 5, this packet has been reassigned and is reserved for use as the Marker packet.

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. It may be placed at the beginning of a message that uses features not available in PGP version 2.6 in order to cause that version to report that newer software is necessary to process the message.

5.10. 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 "t" (0x74), then it contains text data, and thus may need line ends converted to local form, or other text-mode changes. The tag "u" (0x75) means the same as "t", but also indicates that implementation believes that the literal data contains UTF-8 text.
Early versions of PGP also defined a value of "1" 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.

If the special name "_CONSOLE" is used, 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.

* 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 is stored with <CR><LF> text endings (i.e., network-normal line endings). These should be converted to native line endings by the receiving software.

Note that V3 and V4 signatures do not include the formatting octet, the file name, and the date field of the literal packet in a signature hash and thus are not protected against tampering in a signed document. In contrast V5 signatures include them.

5.11. 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.
5.12. 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.

5.13. 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:

+----------------+---------------------------------------------------+
<table>
<thead>
<tr>
<th>Type</th>
<th>Attribute Subpacket</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Image Attribute Subpacket</td>
</tr>
<tr>
<td>100-110</td>
<td>Private/Experimental Use</td>
</tr>
</tbody>
</table>
+----------------+---------------------------------------------------+

Table 13: User Attribute type registry

An implementation SHOULD ignore any subpacket of a type that it does not recognize.
5.13.1. 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.

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

The Symmetrically Encrypted Integrity Protected Data packet is a variant of the Symmetrically Encrypted Data packet. It is a new feature created for OpenPGP that addresses the problem of detecting a modification to encrypted data. It is used in combination with a Modification Detection Code packet.

There is a corresponding feature in the features Signature subpacket that denotes that an implementation can properly use this packet type. An implementation MUST support decrypting and generating these packets. An implementation SHOULD specifically denote support for this packet, but it MAY infer it from other mechanisms.
For example, an implementation might infer from the use of a cipher such as Advanced Encryption Standard (AES) or Twofish that a user supports this feature. It might place in the unhashed portion of another user’s key signature a Features subpacket. It might also present a user with an opportunity to regenerate their own self-signature with a Features subpacket.

This packet contains data encrypted with a symmetric-key algorithm and protected against modification by the SHA-1 hash algorithm. When it has been decrypted, it will typically contain other packets (often a Literal Data packet or Compressed Data packet). The last decrypted packet in this packet’s payload MUST be a Modification Detection Code packet.

The body of this packet consists of:

* A one-octet version number. The only currently defined value is 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 14.10 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.
The plaintext of the data to be encrypted is passed through the SHA-1 hash function, and the result of the hash is appended to the plaintext in a Modification Detection Code packet. The input to the hash function includes the prefix data described above; it includes all of the plaintext, and then also includes two octets of values 0xD3, 0x14. These represent the encoding of a Modification Detection Code packet tag and length field of 20 octets.

The resulting hash value is stored in a Modification Detection Code (MDC) packet, which MUST use the two octet encoding just given to represent its tag and length field. The body of the MDC packet is the 20-octet output of the SHA-1 hash.

The Modification Detection Code packet is appended to the plaintext and encrypted along with the plaintext using the same CFB context.

During decryption, the plaintext data should be hashed with SHA-1, including the prefix data as well as the packet tag and length field of the Modification Detection Code packet. The body of the MDC packet, upon decryption, is compared with the result of the SHA-1 hash.

Any failure of the MDC indicates that the message has been modified and MUST be treated as a security problem. Failures include a difference in the hash values, but also the absence of an MDC packet, or an MDC packet in any position other than the end of the plaintext. Any failure SHOULD be reported to the user.

Note: future designs of new versions of this packet should consider rollback attacks since it will be possible for an attacker to change the version back to 1.

NON-NORMATIVE EXPLANATION

The MDC system, as packets 18 and 19 are called, were 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 RIPE-MD/160, for example.)

However, given the present state of hash function cryptanalysis and cryptography, it may be desirable to upgrade the MDC system to a new hash function. See Section 14.12 for guidance.

5.15. Modification Detection Code Packet (Tag 19)

The Modification Detection Code packet contains a SHA-1 hash of plaintext data, which is used to detect message modification. It is only used with a Symmetrically Encrypted Integrity Protected Data packet. The Modification Detection Code packet MUST be the last packet in the plaintext data that is encrypted in the Symmetrically Encrypted Integrity Protected Data packet, and MUST appear in no other place.
A Modification Detection Code packet MUST have a length of 20 octets.

The body of this packet consists of:

* A 20-octet SHA-1 hash of the preceding plaintext data of the Symmetrically Encrypted Integrity Protected Data packet, including prefix data, the tag octet, and length octet of the Modification Detection Code packet.

Note that the Modification Detection Code packet MUST always use a new format encoding of the packet tag, and a one-octet encoding of the packet length. The reason for this is that the hashing rules for modification detection include a one-octet tag and one-octet length in the data hash. While this is a bit restrictive, it reduces complexity.

5.16. AEAD Encrypted Data Packet (Tag 20)

This packet contains data encrypted with an authenticated encryption and additional data (AEAD) construction. When it has been decrypted, it will typically contain other packets (often a Literal Data packet or Compressed Data packet).

The body of this packet starts with:

* A one-octet version number. The only currently defined value is 1.

When the version is 1, it is followed by the following fields:

* A one-octet cipher algorithm.

* A one-octet AEAD algorithm.

* A one-octet chunk size.

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

* 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.

An AEAD encrypted data 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 new format encoding (bits 7 and 6 set, bits 5-0 carry the packet tag), version number, cipher algorithm octet, AEAD algorithm octet, chunk size octet, and an eight-octet, big-endian chunk index as additional data. The index of the first chunk is zero. 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 0xD4, 0x01, 0x07, 0x01, 0x10, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, and 0x00.

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. Please note that the big-endian number representing the chunk index in the additional data is increased accordingly, although it’s not really a chunk.

The chunk size octet specifies the size of chunks using the following formula (in C), where c is the chunk size octet:

\[
\text{chunk\_size} = ((\text{uint64\_t})1 << (c + 6))
\]

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).

A unique, random, unpredictable initialization vector MUST be used for each message. Failure to do so for each message can lead to a catastrophic failure depending on the choice of AEAD mode and symmetric key reuse.

5.16.1. 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 initialization vector is 16 octets long. EAX authentication tags are 16 octets long.
The nonce for EAX mode is computed by treating the initialization vector as a 16-octet, big-endian value and exclusive-oring the low eight octets of it with the chunk index.

5.16.2. 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 initialization vector is 15 octets long. OCB authentication tags are 16 octets long.

The nonce for OCB mode is computed by the exclusive-oring of the initialization vector as a 15-octet, big-endian value, against the chunk index.

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].

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 the next section.

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

Rationale for CRC-24: The size of 24 bits fits evenly into printable base64. The nonzero initialization can detect more errors than a zero initialization.
6.1. 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 Armor Checksum
* 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 MESSAGE, PART X/Y
   Used for multi-part messages, where the armor is split amongst Y parts, and this is the Xth part out of Y.

BEGIN PGP MESSAGE, PART X
   Used for multi-part messages, where this is the Xth part of an unspecified number of parts. Requires the MESSAGE-ID Armor Header to be used.

BEGIN PGP SIGNATURE
   Used for detached signatures, OpenPGP/MIME signatures, and cleartext signatures. Note that PGP 2 uses BEGIN PGP MESSAGE for detached 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 below).

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. OpenPGP should consider improperly formatted Armor Headers to be corruption of the ASCII Armor. Unknown keys should be reported to the user, but OpenPGP 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.

* "MessageID", a 32-character string of printable characters. The string must be the same for all parts of a multi-part message that uses the "PART X" Armor Header. MessageID strings should be unique enough that the recipient of the mail can associate all the parts of a message with each other. A good checksum or cryptographic hash function is sufficient.

The MessageID SHOULD NOT appear unless it is in a multi-part message. If it appears at all, it MUST be computed from the finished (encrypted, signed, etc.) message in a deterministic fashion, rather than contain a purely random value. This is to allow the legitimate recipient to determine that the MessageID cannot serve as a covert means of leaking cryptographic key information.

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

* "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.

\[
\begin{array}{cccccccc}
7 & 6 & 5 & 4 & 3 & 2 & 1 & 07 6 5 4 3 2 1 0 7 6 5 4 3 2 1 0
\end{array}
\]

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.
Table 14: 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
6.6. Example of an ASCII Armored Message

-----BEGIN PGP MESSAGE-----
yDgBO22WxBHv7O8X7O/jygAEzol56iUKiXmV+XmpCtmpqQUkIqrFqclFqUDbovzS
vBSFjNSiVHsuAA==
=jUN
-----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,
* One or more "Hash" 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 there are no such headers, MD5 is used. If MD5 is the only hash used, then an implementation MAY omit this header for improved V2.x compatibility. If more than one message digest is used in the signature, the "Hash" armor header contains a comma-delimited list of used message digests.

Current message digest names are described below with the algorithm IDs.

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 (i.e., 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 (e.g., "[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 14 for more discussion of the algorithms.

### 9.1. Public-Key Algorithms

<table>
<thead>
<tr>
<th>ID</th>
<th>Algorithm</th>
<th>Public Key Format</th>
<th>Secret Key Format</th>
<th>Signature Format</th>
<th>PKESK Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RSA (Encrypt or Sign) [HAC]</td>
<td>MPI(n), [Section 5.6.1]</td>
<td>MPI(d), MPI(p), MPI(q), MPI(u)</td>
<td>MPI(m^d mod n) [Section 5.2.3.1]</td>
<td>MPI(m^e mod n) [Section 5.1.1]</td>
</tr>
<tr>
<td>2</td>
<td>RSA Encrypt-Only [HAC]</td>
<td>MPI(n), [Section 5.6.1]</td>
<td>MPI(d), MPI(p), MPI(q), MPI(u)</td>
<td>N/A</td>
<td>MPI(m^e mod n) [Section 5.1.1]</td>
</tr>
<tr>
<td>3</td>
<td>RSA Sign-Only [HAC]</td>
<td>MPI(n), [Section 5.6.1]</td>
<td>MPI(d), MPI(p), MPI(q), MPI(u)</td>
<td>MPI(m^d mod n) [Section 5.2.3.1]</td>
<td>N/A</td>
</tr>
<tr>
<td>16</td>
<td>Elgamal (Encrypt-Only) [ELGAMAL] [HAC]</td>
<td>MPI(p), MPI(g), MPI(y) [Section 5.6.3]</td>
<td>MPI(x)</td>
<td>N/A</td>
<td>MPI(g^k mod p), MPI(m * y^k mod p) [Section 5.1.2]</td>
</tr>
<tr>
<td>17</td>
<td>DSA (Digital Signature Algorithm) [FIPS186] [HAC]</td>
<td>MPI(p), MPI(q), MPI(g), MPI(y) [Section 5.6.2]</td>
<td>MPI(x)</td>
<td>MPI(r), MPI(s) [Section 5.2.3.2]</td>
<td>N/A</td>
</tr>
<tr>
<td>18</td>
<td>ECDH public key algorithm</td>
<td>OID, MPI(point in curve-specific point format), KDFParams [see Section 9.2.1, Section 5.1.3]</td>
<td>N/A</td>
<td>MPI(point in curve-specific point format), size octet, encoded key [Section 9.2.1, Section 5.1.3]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td>---</td>
<td></td>
</tr>
<tr>
<td>19</td>
<td>ECDSA public key algorithm [FIPS186]</td>
<td>OID, MPI(point in SEC1 format) [Section 5.6.4]</td>
<td>MPI(secret)</td>
<td>MPI(r), MPI(s) [Section 5.2.3.2]</td>
<td>N/A</td>
</tr>
<tr>
<td>20</td>
<td>Reserved (formerly Elgamal Encrypt or Sign)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>21</td>
<td>Reserved for Diffie-Hellman (X9.42, as defined for IETF-S/MIME)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>22</td>
<td>EdDSA [RFC8032]</td>
<td>OID, MPI(point in prefixed native format) [Section 5.6.5]</td>
<td>MPI(value in curve-specific format) [see Section 9.2.1]</td>
<td>MPI, MPI [see Section 9.2.1, Section 5.2.3.3]</td>
<td>N/A</td>
</tr>
<tr>
<td>23</td>
<td>Reserved (AEDH)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>24</td>
<td>Reserved (AEDSA)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>Private/to Experimental algorithm</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 15: Public-key algorithm registry
Implementations MUST implement DSA for signatures, and Elgamal for encryption. Implementations SHOULD implement RSA keys (1). RSA Encrypt-Only (2) and RSA Sign-Only (3) are deprecated and SHOULD NOT be generated, but may be interpreted. See Section 14.5. See Section 14.9 for notes on Elgamal Encrypt or Sign (20), and X9.42 (21). Implementations MAY implement any other algorithm.

A compatible specification of ECDSA is given in [RFC6090] as "KT-I Signatures" and in [SEC1]; EC DH is defined in Section 13.5 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:

<table>
<thead>
<tr>
<th>ASN.1 Identifier</th>
<th>OID len</th>
<th>Curve OID octets in hexadecimal representation</th>
<th>Curve name</th>
<th>Usage</th>
<th>Field Size (fsize)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.2.840.10045.3.1.7</td>
<td>8</td>
<td>2A 86 48 CE 3D 03 01 07</td>
<td>NIST P-256</td>
<td>ECDSA, ECDH</td>
<td>32</td>
</tr>
<tr>
<td>1.3.132.0.34</td>
<td>5</td>
<td>2B 81 04 00 22</td>
<td>NIST P-384</td>
<td>ECDSA, ECDH</td>
<td>48</td>
</tr>
<tr>
<td>1.3.132.0.35</td>
<td>5</td>
<td>2B 81 04 00 23</td>
<td>NIST P-521</td>
<td>ECDSA, ECDH</td>
<td>66</td>
</tr>
<tr>
<td>1.3.6.1.4.1.11591.15.1</td>
<td>9</td>
<td>2B 06 01 04 01 DA 47 0F 01</td>
<td>Ed25519</td>
<td>EdDSA</td>
<td>32</td>
</tr>
<tr>
<td>1.3.101.113</td>
<td>3</td>
<td>2B 65 71</td>
<td>Ed448</td>
<td>EdDSA</td>
<td>57</td>
</tr>
<tr>
<td>1.3.6.1.4.1.3029.1.5.1</td>
<td>10</td>
<td>2B 06 01 04 01 97 55 01 05 01</td>
<td>Curve25519</td>
<td>ECDH</td>
<td>32</td>
</tr>
<tr>
<td>1.3.101.111</td>
<td>3</td>
<td>2B 65 6F</td>
<td>X448</td>
<td>ECDH</td>
<td>56</td>
</tr>
</tbody>
</table>

Table 16: 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.

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.
<table>
<thead>
<tr>
<th>Curve</th>
<th>ECDH Point Format</th>
<th>ECDH Secret Key MPI</th>
<th>EdDSA Secret Key MPI</th>
<th>EdDSA Signature first MPI</th>
<th>EdDSA Signature second MPI</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIST P-256</td>
<td>SEC1 integer</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>NIST P-384</td>
<td>SEC1 integer</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>NIST P-521</td>
<td>SEC1 integer</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>Ed25519</td>
<td>N/A integer</td>
<td>32 octets of secret</td>
<td>32 octets of R</td>
<td>32 octets of S</td>
<td></td>
</tr>
<tr>
<td>Ed448</td>
<td>N/A integer</td>
<td>prefixed 57 octets of secret</td>
<td>prefixed 114 octets of signature</td>
<td>0 [this is an unused placeholder]</td>
<td></td>
</tr>
<tr>
<td>Curve25519</td>
<td>prefixed native integer</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
<tr>
<td>X448</td>
<td>prefixed native integer</td>
<td>prefixed 56 octets of secret</td>
<td>N/A</td>
<td>N/A</td>
<td></td>
</tr>
</tbody>
</table>

Table 17: 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

<table>
<thead>
<tr>
<th>ID</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Plaintext or unencrypted data</td>
</tr>
<tr>
<td>1</td>
<td>IDEA [IDEA]</td>
</tr>
<tr>
<td>2</td>
<td>TripleDES (DES-EDE, [SCHNEIER], [HAC] - 168 bit key derived from 192)</td>
</tr>
<tr>
<td>3</td>
<td>CAST5 (128 bit key, as per [RFC2144])</td>
</tr>
<tr>
<td>ID</td>
<td>Algorithm</td>
</tr>
<tr>
<td>----</td>
<td>------------------------------------------</td>
</tr>
<tr>
<td>4</td>
<td>Blowfish (128 bit key, 16 rounds) [BLOWFISH]</td>
</tr>
<tr>
<td>5</td>
<td>Reserved</td>
</tr>
<tr>
<td>6</td>
<td>Reserved</td>
</tr>
<tr>
<td>7</td>
<td>AES with 128-bit key [AES]</td>
</tr>
<tr>
<td>8</td>
<td>AES with 192-bit key</td>
</tr>
<tr>
<td>9</td>
<td>AES with 256-bit key</td>
</tr>
<tr>
<td>10</td>
<td>Twofish with 256-bit key [TWOFISH]</td>
</tr>
<tr>
<td>11</td>
<td>Camellia with 128-bit key [RFC3713]</td>
</tr>
<tr>
<td>12</td>
<td>Camellia with 192-bit key</td>
</tr>
<tr>
<td>13</td>
<td>Camellia with 256-bit key</td>
</tr>
<tr>
<td>100 to 110</td>
<td>Private/Experimental algorithm</td>
</tr>
<tr>
<td>253, 254 and 255</td>
<td>Reserved to avoid collision with Secret Key Encryption (see Section 3.7.2.1 and Section 5.5.3)</td>
</tr>
</tbody>
</table>

Table 18: Symmetric-key algorithm registry

Implementations MUST implement TripleDES. Implementations SHOULD implement AES-128 and CAST5. Implementations that interoperate with PGP 2.6 or earlier need to support IDEA, as that is the only symmetric cipher those versions use. Implementations MAY implement any other algorithm.

9.4. Compression Algorithms

<table>
<thead>
<tr>
<th>ID</th>
<th>Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Uncompressed</td>
</tr>
<tr>
<td>1</td>
<td>ZIP [RFC1951]</td>
</tr>
<tr>
<td>2</td>
<td>ZLIB [RFC1950]</td>
</tr>
<tr>
<td>3</td>
<td>BZip2 [BZ2]</td>
</tr>
</tbody>
</table>
Implementations MUST implement uncompressed data. Implementations SHOULD implement ZIP. Implementations MAY implement any other algorithm.

### Table 19: Compression algorithm registry

| 100 to 110 | Private/Experimental algorithm |
+------------+--------------------------------+

9.5. Hash Algorithms

<table>
<thead>
<tr>
<th>ID</th>
<th>Algorithm</th>
<th>Text Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MD5 [HAC]</td>
<td>&quot;MD5&quot;</td>
</tr>
<tr>
<td>2</td>
<td>SHA-1 [FIPS180]</td>
<td>&quot;SHA1&quot;</td>
</tr>
<tr>
<td>3</td>
<td>RIPE-MD/160 [HAC]</td>
<td>&quot;RIPEMD160&quot;</td>
</tr>
<tr>
<td>4</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>SHA2-256 [FIPS180]</td>
<td>&quot;SHA256&quot;</td>
</tr>
<tr>
<td>9</td>
<td>SHA2-384 [FIPS180]</td>
<td>&quot;SHA384&quot;</td>
</tr>
<tr>
<td>10</td>
<td>SHA2-512 [FIPS180]</td>
<td>&quot;SHA512&quot;</td>
</tr>
<tr>
<td>11</td>
<td>SHA2-224 [FIPS180]</td>
<td>&quot;SHA224&quot;</td>
</tr>
<tr>
<td>12</td>
<td>SHA3-256 [FIPS202]</td>
<td>&quot;SHA3-256&quot;</td>
</tr>
<tr>
<td>13</td>
<td>Reserved</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>SHA3-512 [FIPS202]</td>
<td>&quot;SHA3-512&quot;</td>
</tr>
<tr>
<td>100 to 110</td>
<td>Private/Experimental algorithm</td>
<td></td>
</tr>
</tbody>
</table>

Table 20: Hash algorithm registry
Implementations MUST implement SHA-1. Implementations MAY implement other algorithms. MD5 is deprecated.

9.6. AEAD Algorithms

<table>
<thead>
<tr>
<th>ID</th>
<th>Algorithm</th>
<th>IV length (octets)</th>
<th>authentication tag length (octets)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>EAX [EAX]</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>2</td>
<td>OCB [RFC7253]</td>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>100 to 110</td>
<td>Private/Experimental algorithm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 21: AEAD algorithm registry

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 5.13. 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 5.13.1. 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 Signature Notation Data type, the name of the Signature Notation Data, its allowed values, and a reference to the defining specification. The initial values for this registry can be found in Section 5.2.3.20. 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", "Description", "Security Recommended", "Interoperability Recommended", and "Reference". The initial values for this registry can be found in Section 5.2.3.20. 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 5.2.3.21. 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 5.2.3.25. 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 5.2.3.27. 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 5.2.3.28. Adding a new feature-implementation flag MUST be done through the SPECIFICATION REQUIRED method, as described in [RFC8126].

Also see Section 14.13 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:

<table>
<thead>
<tr>
<th>ID</th>
<th>Algorithm</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>EdDSA public key algorithm</td>
<td>This doc, Section 14.8</td>
</tr>
</tbody>
</table>

Table 22: New public-Key algorithms registered

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

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:
### 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 13.2. 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 13.3. 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.

11.1. Transferable Public Keys

OpenPGP users may transfer public keys. The essential elements of a transferable public key are as follows:
* One Public-Key packet
* Zero or more revocation signatures
* One or more User ID packets
* After each User ID packet, zero or more Signature packets
  (certifications)
* Zero or more User Attribute packets
* After each User Attribute packet, zero or more Signature packets
  (certifications)
* Zero or more Subkey packets
* After each Subkey packet, one Signature packet, plus optionally a
  revocation

The Public-Key packet occurs first. 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.

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 signature-only key. However, any V4 or V5 key may have subkeys, and the subkeys may be encryption-only keys, signature-only keys, or general-purpose keys. V3 keys MUST NOT have subkeys.

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.

Transferable public-key packet sequences may be concatenated to allow transferring multiple public keys in one operation.

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 are used instead of the public key and public-subkey packets. 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 | AEAD Encrypted 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.

In addition, decrypting a Symmetrically Encrypted and Integrity Protected Data packet, an AEAD Encrypted 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.

Note that some subtle combinations that are formally acceptable by this grammar are nonetheless unacceptable. For example, a v5 SKESK packet cannot effectively precede a SEIPD packet, since that combination does not include any information about the choice of symmetric cipher used for SEIPD (see Section 5.3.1 for more details).

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 a Signature packet stored separately from the data for which they are a signature.

12. Enhanced Key Formats

12.1. Key Structures

The format of an OpenPGP V3 key is as follows. Entries in square brackets are optional and ellipses indicate repetition.

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

The format of an OpenPGP V4 key that uses multiple public keys is similar except that the other keys are added to the end as "subkeys" of the primary key.

Primary-Key
- [Revocation Self Signature]
- [Direct Key Signature...]
- User ID [Signature ...]
- [User ID [Signature ...] ...]
- [User Attribute [Signature ...] ...]
- [[Subkey [Binding-Signature-Revocation ...]
  Subkey-Binding-Signature ...] ...]

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.

In a V4 key, the primary key MUST be a key capable of certification. The subkeys may be keys of any other type. There may be other constructions of V4 keys, too. For example, there may be a single-key RSA key in V4 format, a DSA primary key with an RSA encryption key, or RSA primary key with an Elgamal subkey, etc.

It is also possible to have a signature-only subkey. This permits a primary key that collects certifications (key signatures), but is used only for certifying subkeys that are used for encryption and signatures.

12.2. 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 a DSA 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): 17 = DSA (example);

e) Algorithm-specific fields.

Algorithm-Specific Fields for DSA keys (example):

e.1) MPI of DSA prime p;

e.2) MPI of DSA group order q (q is a prime divisor of p-1);

e.3) MPI of DSA group generator g;

e.4) MPI of DSA public-key value y (= g**x mod p where x is secret).

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 a DSA 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): 17 = DSA (example);

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

f) algorithm-specific fields.
Algorithm-Specific Fields for DSA keys (example):

f.1) MPI of DSA prime p;

f.2) MPI of DSA group order q (q is a prime divisor of p-1);

f.3) MPI of DSA group generator g;

f.4) MPI of DSA public-key value y (= g**x mod p where x is secret).

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 (V3 and V4 key) or 0x9A (V5 key) as the first octet (even though this is not a valid packet ID for a public subkey).

13. 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].

13.1. 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.
13.2. 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.

<table>
<thead>
<tr>
<th>Name</th>
<th>Wire Format</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEC1</td>
<td>0x04</td>
<td></td>
</tr>
<tr>
<td>Prefixed native</td>
<td>0x40</td>
<td></td>
</tr>
</tbody>
</table>

Table 24: Elliptic Curve Point Wire Formats

13.2.1. 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].

13.2.2. 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.
13.2.3. Notes on EC Point Wire Formats

Given the above definitions, the exact size of the MPI payload for an encoded point is 515 bits for "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.

13.3. EC Scalar Wire Formats

Some non-curve values in elliptic curve cryptography (e.g. 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.
13.3.1. 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
13.3.2. 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 13.2.2.

13.4. 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].

13.5. 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 13.4 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 13.5
  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;
* 20 octets representing a recipient encryption subkey or a primary
  key fingerprint identifying the key material that is needed for
  decryption (for version 5 keys the 20 leftmost octets of the
  fingerprint are used).

The size of the KDF parameters sequence, defined above, is either 54
for the NIST curve P-256, 51 for the curves P-384 and P-521, 56 for
Curve25519, or 49 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 15 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 value "m" derived from
the session key, as described in Section 5.1, "Public-Key Encrypted
Session Key Packets (Tag 1)", except that the PKCS #1.5 padding step
is omitted. The result is padded using the method described in
[PKCS5] to an 8-octet granularity. For example, 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 05

The octets s0 and s1 above denote the checksum. 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.
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 \( S = vR \);

* \( m = \text{symm}_\text{alg}_\text{ID} || \text{session key} || \text{checksum} || \text{pkcs5_padding} \);

* \( \text{curve}_\text{OID}_\text{len} = (\text{octet})\text{len}(\text{curve}_\text{OID}) \);

* \( \text{Param} = \text{curve}_\text{OID}_\text{len} || \text{curve}_\text{OID} || \text{public}_\text{key}_\text{alg}_\text{ID} || 03 || 01 || \text{KDF}_\text{hash}_\text{ID} || \text{KEK}_\text{alg}_\text{ID} \text{ for AESKeyWrap} || "\text{Anonymous Sender}" || \text{recipient_fingerprint} \);

* \( Z\text{len} = \text{the key size for the KEK}_\text{alg}_\text{ID} \text{ used with AESKeyWrap} \)

* Compute \( Z = \text{KDF}( S, Z\text{len}, \text{Param}) \);

* Compute \( C = \text{AESKeyWrap}( Z, m) \text{ as per [RFC3394]} \)

* \( \text{VB} = \text{convert point} V \text{ to the octet string} \)

* Output \( (\text{MPI(VB)} || \text{len}(C) || C) \).

The decryption is the inverse of the method given. Note that the recipient obtains the shared secret by calculating

\[ S = rV = rvG, \text{ where } (r,R) \text{ is the recipient’s key pair.} \]
Consistent with Section 5.16 and Section 5.14, AEAD encryption or a Modification Detection Code (MDC) MUST be used anytime the symmetric key is protected by ECDH.

14. Notes on Algorithms

14.1. 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 [RFC3447]. [RFC3447] 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.

14.1.1. 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.
14.1.2. 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 the security note in Section 15 regarding differences in reporting between a decryption error and a padding error.

14.1.3. 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:
   \[ H = \text{Hash}(M). \]

   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 \( \text{emLen} < \text{tLen} + 11 \), output "intended encoded message length too short" and stop.

4. Generate an octet string PS consisting of \( \text{emLen} - \text{tLen} - 3 \) 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
   \[ \text{EM} = 0x00 || 0x01 || \text{PS} || 0x00 || T. \]

6. Output EM.

14.2. 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 TripleDES 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 a TripleDES-only implementation.
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, TripleDES, 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.

14.3. 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.

14.3.1. Compression Preferences

Compression has been an integral part of PGP since its first days. OpenPGP and all previous versions of PGP have offered compression. In this specification, the default is for messages to be compressed, although an implementation is not required to do so. Consequently, the compression preference gives a way for a keyholder to request that messages not be compressed, presumably because they are using a minimal implementation that does not include compression. Additionally, this gives a keyholder a way to state that it can support alternate algorithms.

Like the algorithm preferences, an implementation MUST NOT use an algorithm that is not in the preference vector. If the preferences are not present, then they are assumed to be [ZIP(1), Uncompressed(0)].
Additionally, an implementation MUST implement this preference to the degree of recognizing when to send an uncompresssed message. A robust implementation would satisfy this requirement by looking at the recipient’s preference and acting accordingly. A minimal implementation can satisfy this requirement by never generating a compressed message, since all implementations can handle messages that have not been compressed.

14.3.2. 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 SHA1 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.

14.4. Plaintext

Algorithm 0, "plaintext", may only be used to denote secret keys that are stored in the clear. Implementations MUST NOT use plaintext in Symmetrically Encrypted Data packets; they must use Literal Data packets to encode unencrypted or literal data.

14.5. RSA

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 SHOULD NOT create such a key, but MAY interpret it.

An implementation SHOULD NOT implement RSA keys of size less than 1024 bits.
14.6. DSA

An implementation SHOULD NOT implement DSA keys of size less than 1024 bits. It MUST NOT implement a DSA key with a q size of less than 160 bits. DSA keys MUST also be a multiple of 64 bits, and the q size MUST be a multiple of 8 bits. The Digital Signature Standard (DSS) [FIPS186] specifies that DSA be used in one of the following ways:

* 1024-bit key, 160-bit q, SHA-1, SHA2-224, SHA2-256, SHA2-384, or SHA2-512 hash

* 2048-bit key, 224-bit q, SHA2-224, SHA2-256, SHA2-384, or SHA2-512 hash

* 2048-bit key, 256-bit q, SHA2-256, SHA2-384, or SHA2-512 hash

* 3072-bit key, 256-bit q, SHA2-256, SHA2-384, or SHA2-512 hash

The above key and q size pairs were chosen to best balance the strength of the key with the strength of the hash. Implementations SHOULD use one of the above key and q size pairs when generating DSA keys. If DSS compliance is desired, one of the specified SHA hashes must be used as well. [FIPS186] is the ultimate authority on DSS, and should be consulted for all questions of DSS compliance.

Note that earlier versions of this standard only allowed a 160-bit q with no truncation allowed, so earlier implementations may not be able to handle signatures with a different q size or a truncated hash.

14.7. Elgamal

An implementation SHOULD NOT implement Elgamal keys of size less than 1024 bits.

14.8. 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 Section 5.2.4, "Computing Signatures" 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.

14.9. 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].

14.10. OpenPGP CFB Mode

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 Data Packets; the mechanism used for encrypting secret-key material is similar, and is described in the sections above.

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.
14.11. 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.

14.12. Extension of the MDC System

As described in the non-normative explanation in Section 5.14, the MDC system is uniquely unparameterized in OpenPGP. This was an intentional decision to avoid cross-grade attacks. If the MDC system is extended to a stronger hash function, care must be taken to avoid downgrade and cross-grade attacks.

One simple way to do this is to create new packets for a new MDC. For example, instead of the MDC system using packets 18 and 19, a new MDC could use 20 and 21. This has obvious drawbacks (it uses two packet numbers for each new hash function in a space that is limited to a maximum of 60).

Another simple way to extend the MDC system is to create new versions of packet 18, and reflect this in packet 19. For example, suppose that V2 of packet 18 implicitly used SHA-256. This would require packet 19 to have a length of 32 octets. The change in the version in packet 18 and the size of packet 19 prevent a downgrade attack.

There are two drawbacks to this latter approach. The first is that using the version number of a packet to carry algorithm information is not tidy from a protocol-design standpoint. It is possible that there might be several versions of the MDC system in common use, but this untidiness would reflect untidiness in cryptographic consensus about hash function security. The second is that different versions of packet 19 would have to have unique sizes. If there were two versions each with 256-bit hashes, they could not both have 32-octet packet 19s without admitting the chance of a cross-grade attack.

Yet another, complex approach to extend the MDC system would be a hybrid of the two above -- create a new pair of MDC packets that are fully parameterized, and yet protected from downgrade and cross-grade.
Any change to the MDC system MUST be done through the IETF CONSENSUS method, as described in [RFC8126].

14.13. 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.

15. 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.

* Certain operations in this specification involve the use of random numbers. An appropriate entropy source should be used to generate these numbers (see [RFC4086]).

* 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 26: 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 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 random oracle, and can often decrypt the message.

Compressing data can ameliorate this attack. The incorrectly decrypted data nearly always decompresses in ways that defeat the attack. However, this is not a rigorous fix, and leaves open some small vulnerabilities. For example, if an implementation does not compress a message before encryption (perhaps because it knows it was already compressed), then that message is vulnerable. Because of this happenstance -- that modification attacks can be thwarted by decompression errors -- an implementation SHOULD treat a decompression error as a security problem, not merely a data problem.

This attack can be defeated by the use of Modification Detection, provided that the implementation does not let the user naively return the data to the attacker. An implementation MUST treat an MDC failure as a security problem, not merely a data problem.

In either case, the implementation MAY allow the user access to the erroneous data, but MUST warn the user as to potential security problems should that data be returned to the sender.
While this attack is somewhat obscure, requiring a special set of circumstances to create it, it is nonetheless quite serious as it permits someone to trick a user to decrypt a message. Consequently, it is important that:

1. Implementers treat MDC errors and decompression failures as security problems.

2. Implementers implement Modification Detection with all due speed and encourage its spread.

3. Users migrate to implementations that support Modification Detection with all due speed.

* PKCS#1 has been found to be vulnerable to attacks in which a system that reports errors in padding differently from errors in decryption becomes a random oracle that can leak the private key in mere millions of queries. Implementations must be aware of this attack and prevent it from happening. The simplest solution is to report a single error code for all variants of decryption errors so as not to leak information to an attacker.

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

* In winter 2005, Serge Mister and Robert Zuccherato from Entrust released a paper describing a way that the "quick check" in OpenPGP CFB mode can be used with a random oracle to decrypt two octets of every cipher block [MZ05]. They recommend as prevention not using the quick check at all.

Many implementers have taken this advice to heart for any data that is symmetrically encrypted and for which the session key is public-key encrypted. In this case, the quick check is not needed as the public-key encryption of the session key should guarantee that it is the right session key. In other cases, the implementation should use the quick check with care.

On the one hand, there is a danger to using it if there is a random oracle that can leak information to an attacker. In plainer language, there is a danger to using the quick check if timing information about the check can be exposed to an attacker, particularly via an automated service that allows rapidly repeated queries.
On the other hand, it is inconvenient to the user to be informed that they typed in the wrong passphrase only after a petabyte of data is decrypted. There are many cases in cryptographic engineering where the implementer must use care and wisdom, and this is one.

* Refer to [FIPS186], B.4.1, for the method to generate a uniformly distributed ECC private key.

* The curves proposed in this document correspond to the symmetric key sizes 128 bits, 192 bits, and 256 bits, as described in the table below. This allows a compliant application to offer balanced public key security, which is compatible with the symmetric key strength for each AES algorithm defined here.

The following table defines the hash and the symmetric encryption algorithm that SHOULD be used with a given curve for ECDSA or ECDH. A stronger hash algorithm or a symmetric key algorithm MAY be used for a given ECC curve. However, note that the increase in the strength of the hash algorithm or the symmetric key algorithm may not increase the overall security offered by the given ECC key.

<table>
<thead>
<tr>
<th>Curve name</th>
<th>ECC</th>
<th>RSA strength</th>
<th>Hash size strength, informative</th>
<th>Symmetric key size</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIST P-256</td>
<td>256</td>
<td>3072</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>NIST P-384</td>
<td>384</td>
<td>7680</td>
<td>384</td>
<td>192</td>
</tr>
<tr>
<td>NIST P-521</td>
<td>521</td>
<td>15360</td>
<td>512</td>
<td>256</td>
</tr>
</tbody>
</table>

Table 27: Elliptic Curve cryptographic guidance

* Requirement levels indicated elsewhere in this document lead to the following combinations of algorithms in the OpenPGP profile: MUST implement NIST curve P-256 / SHA2-256 / AES-128, SHOULD implement NIST curve P-521 / SHA2-512 / AES-256, MAY implement NIST curve P-384 / SHA2-384 / AES-256, among other allowed combinations.

Consistent with the table above, the following table defines the KDF hash algorithm and the AES KEK encryption algorithm that SHOULD be used with a given curve for ECDH. A stronger KDF hash algorithm or AES KEK algorithm MAY be used for a given ECC curve.
<table>
<thead>
<tr>
<th>Curve name</th>
<th>Recommended KDF hash algorithm</th>
<th>Recommended KEK encryption algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIST P-256</td>
<td>SHA2-256</td>
<td>AES-128</td>
</tr>
<tr>
<td>NIST P-384</td>
<td>SHA2-384</td>
<td>AES-192</td>
</tr>
<tr>
<td>NIST P-521</td>
<td>SHA2-512</td>
<td>AES-256</td>
</tr>
</tbody>
</table>

Table 28: Elliptic Curve KDF and KEK recommendations

* This document explicitly discourages the use of algorithms other than AES as a KEK algorithm because backward compatibility of the ECDH format is not a concern. The KEK algorithm is only used within the scope of a Public-Key Encrypted Session Key Packet, which represents an ECDH key recipient of a message. Compare this with the algorithm used for the session key of the message, which MAY be different from a KEK algorithm.

Compliant applications SHOULD implement, advertise through key preferences, and use the strongest algorithms specified in this document.

Note that the symmetric algorithm preference list may make it impossible to use the balanced strength of symmetric key algorithms for a corresponding public key. For example, the presence of the symmetric key algorithm IDs and their order in the key preference list affects the algorithm choices available to the encoding side, which in turn may make the adherence to the table above infeasible. Therefore, compliance with this specification is a concern throughout the life of the key starting immediately after the key generation when the key preferences are first added to a key. It is generally advisable to position a symmetric algorithm ID of strength matching the public key at the head of the key preference list.

Encryption to multiple recipients often results in an unordered intersection subset. For example, if the first recipient’s set is \{A, B\} and the second’s is \{B, A\}, the intersection is an unordered set of two algorithms, A and B. In this case, a compliant application SHOULD choose the stronger encryption algorithm.

Resource constraints, such as limited computational power, are a reason why an application might prefer to use the weakest algorithm. On the other side of the spectrum are applications...
that can implement every algorithm defined in this document. Most applications are expected to fall into either of these two categories. A compliant application in the second, or strongest, category SHOULD prefer AES-256 to AES-192.

SHA-1 MUST NOT be used with the ECDSA or the KDF in the ECDH method.

MDC MUST be used when a symmetric encryption key is protected by ECDH. None of the ECC methods described in this document are allowed with deprecated V3 keys.

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.

16. Implementation Nits

This section is a collection of comments to help an implementer, particularly with an eye to backward compatibility. Previous implementations of PGP are not OpenPGP compliant. 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.

* The IDEA algorithm is patented, and yet it is required for PGP 2 interoperability. It is also the de-facto preferred algorithm for a V3 key with a V3 self-signature (or no self-signature).

* When exporting a private key, PGP 2 generates the header "BEGIN PGP SECRET KEY BLOCK" instead of "BEGIN PGP PRIVATE KEY BLOCK". All previous versions ignore the implied data type, and look directly at the packet data type.
* PGP versions 2.0 through 2.5 generated V2 Public-Key packets. These are identical to the deprecated V3 keys except for the version number. An implementation MUST NOT generate them and may accept or reject them as it sees fit. Some older PGP versions generated V2 PKESK packets (Tag 1) as well. An implementation may accept or reject V2 PKESK packets as it sees fit, and MUST NOT generate them.

* PGP version 2.6 will not accept key-material packets with versions greater than 3.

* There are many ways possible for two keys to have the same key material, but different fingerprints (and thus Key IDs). Perhaps the most interesting is an RSA key that has been "upgraded" to V4 format, but 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.

* If an implementation is using zlib to interoperate with PGP 2, then the "windowBits" parameter should be set to -13.

* The 0x19 back signatures were not required for signing subkeys until relatively recently. Consequently, there may be keys in the wild that do not have these back signatures. Implementing software may handle these keys as it sees fit.

* 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. As of 2007 most implementations have an upper bound of 4096 bits.
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.

17. References

17.1. Normative References


Internet-Draft          OpenPGP Message Format          October 2021

DOl 10.17487/RFC2144, May 1997,

DOl 10.17487/RFC2822, April 2001,

[RFC3156] Elkins, M., Del Torto, D., Levien, R., and T. Roessler,
"MIME Security with OpenPGP", RFC 3156,
DOl 10.17487/RFC3156, August 2001,

(AES) Key Wrap Algorithm", RFC 3394, DOl 10.17487/RFC3394,

Standards (PKCS) #1: RSA Cryptography Specifications
Version 2.1", RFC 3447, DOl 10.17487/RFC3447, February

[RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO
10646", STD 63, RFC 3629, DOl 10.17487/RFC3629, November

[RFC3713] Matsui, M., Nakajima, J., and S. Moriai, "A Description of
the Camellia Encryption Algorithm", RFC 3713,
DOl 10.17487/RFC3713, April 2004,

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker,
"Randomness Requirements for Security", BCP 106, RFC 4086,
DOl 10.17487/RFC4086, June 2005,

[RFC7253] Krovetz, T. and P. Rogaway, "The OCB Authenticated-
Encryption Algorithm", RFC 7253, DOl 10.17487/RFC7253, May

for Security", RFC 7748, DOl 10.17487/RFC7748, January

Signature Algorithm (EdDSA)", RFC 8032,
DOl 10.17487/RFC8032, January 2017,

Koch & Wouters   Expires 21 April 2022   [Page 126]


17.2. Informative References


Appendix A. Test vectors

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

A.1. Sample EdDSA 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
A.2. Sample EdDSA signature

The signature is created using the sample key over the input data "OpenPGP" on 2015-09-16 12:24:53 and thus the input to the hash function is:

m: 4f70656e504750040016080006050255f95f9504ff0000000c

Using the SHA2-256 hash algorithm yields the digest:

d: f6220a3f757814f4c2176ffbb68b00249cd4ccdc059c4b34ad871f30b1740280

Which is fed into the EdDSA signature function and yields this signature:

r: 56f90cca98e2102637bd983fbd16c131dfdd7ed82bf4dfe5606e0d756aed3366
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 01 00 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

A.3. Sample AEAD-EAX encryption and decryption

Encryption is performed with the string "Hello, world!", using AES-128 with AEAD-EAX encryption.

A.3.1. Sample Parameters

S2K:

type 3
Iterations:
524288 (144), SHA2-256

Salt:
cd5a9f70fbe0bc65

A.3.2. Sample symmetric-key encrypted session key packet (v5)

Packet header:
c3 3e

Version, algorithms, S2K fields:
05 07 01 03 08 cd 5a 9f 70 fb e0 bc 65 90

AEAD IV:
bc 66 9e 34 e5 00 dc ae dc 5b 32 aa 2d ab 02 35

AEAD encrypted CEK:
9d ee 19 d0 7c 34 46 c4 31 2a 34 ae 19 67 a2 fb

Authentication tag:
7e 92 8e a5 b4 fa 80 12 bd 45 6d 17 38 c6 3c 36

A.3.3. Starting AEAD-EAX decryption of CEK

The derived key is:
b2 55 69 b9 54 32 45 66 45 27 c4 97 6e 7a 5d 6e

Authenticated Data:
c3 05 07 01

Nonce:
bc 66 9e 34 e5 00 dc ae dc 5b 32 aa 2d ab 02 35

Decrypted CEK:
86 f1 ef b8 69 52 32 9f 24 ac d3 bf d0 e5 34 6d
A.3.4. Initial Content Encryption Key

This key would typically be extracted from an SKESK or PKESK. In this example, it is extracted from an SKESK packet, as described above.

CEK:
86 f1 ef b8 69 52 32 9f 24 ac d3 bf d0 e5 34 6d

A.3.5. Sample AEAD encrypted data packet

Packet header:
  d4 4a

Version, AES-128, EAX, Chunk bits (14):
  01 07 01 0e

IV:
  b7 32 37 9f 73 c4 92 8d e2 5f ac fe 65 17 ec 10

AEAD-EAX Encrypted data chunk #0:
  5d c1 1a 81 dc 0c b8 a2 f6 f3 d9 00 16 38 4a 56
  fc 82 1a e1 1a e8

Chunk #0 authentication tag:
  db cb 49 86 26 55 de a8 8d 06 a8 14 86 80 1b 0f

Final (zero-size chunk #1) authentication tag:
  f3 87 bd 2e ab 01 3d e1 25 95 86 90 6e ab 24 76

A.3.6. Decryption of data

Starting AEAD-EAX decryption of data, using the CEK.

Chunk #0:

Authenticated data:
  d4 01 07 01 0e 00 00 00 00 00 00 00 00 00 00

Nonce:
Decrypted chunk #0.

Literal data packet with the string contents "Hello, world!\n".

Nonce:

A.3.7. Complete AEAD-EAX encrypted packet sequence

Symmetric-key encrypted session key packet (v5):

c3 3e 05 07 01 03 08 cd 5a 9f 70 fb e0 bc 65 90
bc 66 9e 34 e5 00 dc ae dc 5b 32 aa 2d ab 02 35
9d ee 19 d0 7c 34 46 c4 31 2a 34 ae 19 67 a2 fb
7e 92 8e a5 b4 fa 80 12 bd 45 6d 17 3c 6c 3c 36

AEAD encrypted data packet:

d4 4a 01 07 01 0e b7 32 37 9f 73 c4 92 8d e2 5f
ac fe 65 17 ec 10 5d c1 1a 81 dc 0c b8 a2 f6 f3
d9 00 16 38 4a 56 fc 82 1a el 1a e8 db cb 49 86
26 55 de a8 8d 06 a8 14 86 80 1b 0f f3 87 bd 2e
ab 01 3d e1 25 95 86 90 6e ab 24 76

A.4. Sample AEAD-OCB encryption and decryption

Encryption is performed with the string "Hello, world!" using AES-128
with AEAD-OCB encryption.

A.4.1. Sample Parameters

S2K:

type 3
Iterations:
524288 (144), SHA2-256

Salt:
9f0b7da3e5ea6477

A.4.2. Sample symmetric-key encrypted session key packet (v5)

Packet header:
c3 3d

Version, algorithms, S2K fields:
05 07 02 03 08 9f 0b 7d a3 e5 ea 64 77 90

AEAD IV:
99 e3 26 e5 40 0a 90 93 6c ef b4 e8 eb a0 8c

AEAD encrypted CEK:
67 73 71 6d 1f 27 14 54 0a 38 fc ac 52 99 49 da

Authentication tag:
c5 29 d3 de 31 e1 5b 4a eb 72 9e 33 00 33 db ed

A.4.3. Starting AEAD-OCB decryption of CEK

The derived key is:
eb 9d a7 8a 9d 5d f8 0e c7 02 05 96 39 9b 65 08

Authenticated Data:
c3 05 07 02

Nonce:
99 e3 26 e5 40 0a 90 93 6c ef b4 e8 eb a0 8c

Decrypted CEK:
d1 f0 1b a3 0e 13 0a a7 d2 58 2c 16 e0 50 ae 44
A.4.4. Initial Content Encryption Key

This key would typically be extracted from an SKESK or PKESK. In this example, it is extracted from an SKESK packet, as described above.

Decrypted CEK:

\[ d1 \ f0 \ 1b \ a3 \ 0e \ 13 \ 0a \ a7 \ d2 \ 58 \ 2c \ 16 \ e0 \ 50 \ ae \ 44 \]

A.4.5. Sample AEAD encrypted data packet

Packet header:

\[ d4 \ 49 \]

Version, AES-128, OCB, Chunk bits (14):

\[ 01 \ 07 \ 02 \ 0e \]

IV:

\[ 5e \ d2 \ bc \ 1e \ 47 \ 0a \ be \ 8f \ 1d \ 64 \ 4c \ 7a \ 6c \ 8a \ 56 \]

AEAD-OCB Encrypted data chunk #0:

\[ 7b \ 0f \ 77 \ 01 \ 19 \ 66 \ 11 \ a1 \ 54 \ ba \ 9c \ 25 \ 74 \ cd \ 05 \ 62 \ 84 \ a8 \ ef \ 68 \ 03 \ 5c \]

Chunk #0 authentication tag:

\[ 62 \ 3d \ 93 \ cc \ 70 \ 8a \ 43 \ 21 \ 1b \ b6 \ ea \ f2 \ b2 \ 7f \ 7c \ 18 \]

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

\[ d5 \ 71 \ bc \ d8 \ 3b \ 20 \ ad \ d3 \ a0 \ 8b \ 73 \ af \ 15 \ b9 \ a0 \ 98 \]

A.4.6. Decryption of data

Starting AEAD-OCB decryption of data, using the CEK.

Chunk #0:

Authenticated data:

\[ d4 \ 01 \ 07 \ 02 \ 0e \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \ 00 \]

Nonce:
Decrypted chunk #0.

Literal data packet with the string contents "Hello, world!\n".

Authenticating final tag:

Authenticated data:

d4 01 07 02 0e 00 00 00 00 00 00 00 00 00 00 00
00 00 00 00 16

Nonce:

5e d2 bc 1e 47 0a be 8f 1d 64 4c 7a 6c 8a 56

A.4.7. Complete AEAD-OCB encrypted packet sequence

Symmetric-key encrypted session key packet (v5):

c3 3d 05 07 02 03 08 9f 0b 7d a3 e5 ea 64 77 90
99 e3 26 e5 40 0a 90 93 6c ef b4 e8 eb a0 8c 67
73 71 6d 1f 27 14 54 0a 38 fc ac 52 99 49 da c5
29 d3 de 31 e1 5b 4a eb 72 9e 33 00 33 db ed

AEAD encrypted data packet:

d4 49 01 07 02 0e 5e d2 bc 1e 47 0a be 8f 1d 64
4c 7a 6c 8a 56 7b 0f 77 01 19 66 11 a1 54 ba 9c
25 74 cd 05 62 84 a8 ef 68 03 5c 62 3d 93 cc 70
8a 43 21 1b b6 ea f2 b2 7f 7c 18 d5 71 bc d8 3b
20 ad d3 a0 8b 73 af 15 b9 a0 98

Appendix B. Acknowledgements

This memo 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.
Appendix C. Document Workflow

This document is built from markdown using ruby-kramdown/rfc2629 (https://rubygems.org/gems/kramdown/rfc2629), and tracked using git (https://git-scm.com/). The markdown source under development can be found in the file "crypto-refresh.md" in the "main" branch of the git repository (https://gitlab.com/openpgp-wg/rfc4880bis). Discussion of this document should take place on the openpgp@ietf.org mailing list (https://www.ietf.org/mailman/listinfo/openpgp).

A non-substantive editorial nit can be submitted directly as a merge request (https://gitlab.com/openpgp-wg/rfc4880bis/-/merge_requests/new). A substantive proposed edit may also be submitted as a merge request, but should simultaneously be sent to the mailing list for discussion.

An open problem can be recorded and tracked as an issue (https://gitlab.com/openpgp-wg/rfc4880bis/-/issues) in the gitlab issue tracker, but discussion of the issue should take place on the mailing list.

[Note to RFC-Editor: Please remove this section on publication.]

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