Network Working Group
Category: INTERNET-DRAFT
draft-ietf-openpgp-rfc2440bis-05.txt
Expires Oct 2002
April 2002

Jon Callas
Wave Systems Corporation

Lutz Donnerhacke
IN-Root-CA Individual Network e.V.

Hal Finney
Network Associates

Rodney Thayer

OpenPGP Message Format

draft-ietf-openpgp-rfc2440bis-05.txt

Status of this Memo

This document is an Internet-Draft and is in full conformance with all provisions of Section 10 of RFC2026.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

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."

The list of current Internet-Drafts can be accessed at http://www.ietf.org/ietf/1id-abstracts.txt

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

IESG Note

This document defines many tag values, yet it doesn't describe a mechanism for adding new tags (for new features). Traditionally the Internet Assigned Numbers Authority (IANA) handles the allocation of new values for future expansion and RFCs usually define the procedure to be used by the IANA. However there are subtle (and not so subtle) interactions that may occur in this protocol between new features and existing features which result in a significant reduction in over all security. Therefore this document does not define an extension procedure. Instead requests to define new tag values (say for new encryption algorithms for example) should be forwarded to the IESG Security Area Directors for consideration or

forwarding to the appropriate IETF Working Group for consideration.

Abstract

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.

OpenPGP software uses a combination of strong public-key and symmetric cryptography to provide security services for electronic communications and data storage. These services include confidentiality, key management, authentication, and digital signatures. This document specifies the message formats used in OpenPGP.

Table of Contents

1. Introduction

This document provides information on the message-exchange packet formats used by OpenPGP to provide encryption, decryption, signing, and key management functions. It is a revision of RFC2440, "OpenPGP Message Format", which itself replaces RFC 1991, "PGP Message Exchange Formats."

1.1. Terms

* OpenPGP - This is a definition for security software that uses PGP 5.x as a basis, formalized in RFC 2440 and this document.

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

* PGP 2.6.x - This version of PGP has many variants, hence the term PGP 2.6.x. It used only RSA, MD5, and IDEA for its cryptographic transforms. An informational RFC, RFC1991, was written describing this version of PGP.

* PGP 5.x - This version of PGP is formerly known as "PGP 3" in the community and also in the predecessor of this document, RFC1991. It has new formats and corrects a number of problems in the PGP 2.6.x design. It is referred to here as PGP 5.x because that software was the first release of the "PGP 3" code base.

* GPG - GNU Privacy Guard, also called GNUpg. GPG is an OpenPGP implementation that avoids all encumbered algorithms. Consequently, early versions of GPG did not include RSA public keys. GPG may or may not have (depending on version) support for IDEA or other encumbered algorithms.

"PGP", "Pretty Good", and "Pretty Good Privacy" are trademarks of Network Associates, Inc. and are used with permission.

This document uses the terms "MUST", "SHOULD", and "MAY" as defined in RFC2119, along with the negated forms of those terms.

2. General functions

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

- digital signatures

- encryption

- compression

- radix-64 conversion

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

2.1. Confidentiality via Encryption

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

1. The sender creates a message.

2. The sending OpenPGP generates a random number to be used as a session key for this message only.

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

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

Note that while all past implementations of PGP properly handle messages that have not been compressed, they all have compressed messages by default. If an implementation does not implement compression, its authors should be aware that most PGP messages in the world are compressed. Thus, it may even be wise for a space-constrained implementation to implement decompression, but not compression.

2.4. Conversion to Radix-64

OpenPGP's underlying native representation for encrypted messages, signature certificates, and keys is a stream of arbitrary octets. Some systems only permit the use of blocks consisting of seven-bit, printable text. For transporting OpenPGP's native raw binary octets through channels that are not safe to raw binary data, a printable encoding of these binary octets is needed. OpenPGP provides the service of converting the raw 8-bit binary octet stream to a stream of printable ASCII characters, called Radix-64 encoding or ASCII Armor.

Implementations SHOULD provide Radix-64 conversions.

2.5. Signature-Only Applications

OpenPGP is designed for applications that use both encryption and signatures, but there are a number of problems that are solved by a signature-only implementation. Although this specification requires both encryption and signatures, it is reasonable for there to be subset implementations that are non-comformant 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. Multi-Precision Integers

Multi-Precision 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 ((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].

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

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. The section, "Enhanced Key Formats" below describes how Key IDs are formed.

3.4. Text

Unless otherwise specified, the character set for text is the UTF-8 [RFC2279] 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, as follows:

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.

3.7.2. String-to-key usage

Implementations SHOULD use salted or iterated-and-salted S2K specifiers, as simple S2K specifiers are more vulnerable to dictionary attacks.

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 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 255 is stored in the position where the hash 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 pass phrase)
255:         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.

PGP 2.X 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 left-most bit, called bit 7. A mask for this bit is 0x80 in hexadecimal.

```
     +---------------+
PTag |7 6 5 4 3 2 1 0|
     +---------------+
Bit 7 -- Always one
Bit 6 -- 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, either format may be used. Note that old format packets have four bits of content 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 -- content tag
```

Bits 1-0 - length-type

New format packets contain:

Bits 5-0 -- content 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 (0xFFFFFFFF) octets in length. (This actually encodes a four-octet scalar number.)

4. When the length of the packet body is not known in advance by the issuer, Partial Body Length headers encode a packet of indeterminate length, effectively making it a stream.

4.2.2.1. One-Octet Lengths

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

```
bodyLen = 1st_octet;
```

4.2.2.2. Two-Octet Lengths

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

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

4.2.2.3. 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:

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

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

4.2.2.4. 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:

```
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 (of one of the three types -- one octet, two-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.

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 coded 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. 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.

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:

```
0        -- Reserved - a packet tag must not have this value
1        -- Public-Key Encrypted Session Key Packet
2        -- Signature Packet
3        -- Symmetric-Key Encrypted Session Key Packet
4        -- One-Pass Signature Packet
5        -- Secret Key Packet
6        -- Public Key Packet
7        -- Secret Subkey Packet
8        -- Compressed Data Packet
9        -- Symmetrically Encrypted Data Packet
10       -- Marker Packet
11       -- Literal Data Packet
12       -- Trust Packet
13       -- User ID Packet
14       -- Public Subkey Packet
17       -- User Attribute Packet
18       -- Sym. Encrypted and Integrity Protected Data Packet
19       -- Modification Detection Code Packet
60 to 63 -- Private or Experimental Values
```

5. Packet Types

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

A Public-Key Encrypted Session Key packet holds the session key used to encrypt a message. Zero or more Encrypted Session Key packets (either Public-Key or Symmetric-Key) may precede 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 Symmetrically Encrypted Data Packet 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 implementation should accept, but not generate a version of 2, which is equivalent to V3 in all other respects.

- An eight-octet number that gives the key ID of the public key that the session key is encrypted to. 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.

Algorithm Specific Fields for RSA encryption

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

Algorithm Specific Fields for Elgamal encryption:

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

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

The value "m" in the above 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 Symmetrically Encrypted Data Packet. 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 [RFC2437] to form the "m" value used in the formulas above.

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 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.

Two versions of signature packets are defined. Version 3 provides basic signature information, while version 4 provides an expandable format with subpackets that can specify more information about the signature. PGP 2.6.x only accepts version 3 signatures.

Implementations MUST accept V3 signatures. Implementations SHOULD generate V4 signatures. Implementations MAY generate a V3 signature that can be verified by PGP 2.6.x.

Note that if an implementation is creating an encrypted and signed message that is encrypted to a V3 key, it is reasonable to create a V3 signature.

5.2.1. Signature Types

There are a number of possible meanings for a signature, which are specified in a signature type octet in any given signature. These meanings are:

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> and trailing blanks removed.

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. Note that all PGP "key signatures" are this type of

certification.

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.

Please note that the vagueness of these certification claims is not a flaw, but a feature of the system. Because PGP places final authority for validity upon the receiver of a certification, it may be that one authority's casual certification might be more rigorous than some other authority's positive certification. These classifications allow a certification authority to issue fine-grained claims.

0x18: Subkey Binding Signature
This signature is a statement by the top-level signing key indicates that it owns the subkey. This signature is calculated directly on the subkey itself, 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). It should be issued by the same key that issued the revoked signature or an authorized revocation key The signature should have a later creation date than the signature it revokes.

0x40: Timestamp signature.
This signature is only meaningful for the timestamp contained in it.

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 multi-precision integers comprising the signature. This portion is algorithm specific, as described below.

The data being signed is hashed, and then the signature type and creation time from the signature packet are hashed (5 additional octets). 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 quick test to reject some invalid signatures.

Algorithm Specific Fields for RSA signatures:

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

Algorithm Specific Fields for DSA signatures:

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

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

Algorithm Specific Fields for ElGamal signatures:

- MPI of ElGamal value a = g**k mod p.

- MPI of ElGamal value b = (h-a*x)/k mod p-1.

The hash h is PKCS-1 padded exactly the same way as for the above

described RSA signatures.

With RSA signatures, the hash value is encoded as described in PKCS-1 section 9.2.1 encoded using PKCS-1 encoding type EMSA-PKCS1-v1_5 [RFC2437]. 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:

- MD2: 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x02, 0x02

- MD5: 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x02, 0x05

- RIPEMD-160: 0x2B, 0x24, 0x03, 0x02, 0x01

- SHA-1: 0x2B, 0x0E, 0x03, 0x02, 0x1A

- SHA256: 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01

- SHA384: 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x02

- SHA512: 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x03

The ASN.1 OIDs are:

- MD2: 1.2.840.113549.2.2

- MD5: 1.2.840.113549.2.5

- RIPEMD-160: 1.3.36.3.2.1

- SHA-1: 1.3.14.3.2.26

- SHA256: 2.16.840.1.101.3.4.2.1

- SHA384: 2.16.840.1.101.3.4.2.2

- SHA512: 2.16.840.1.101.3.4.2.3

The full hash prefixes for these are:

MD2: 0x30, 0x20, 0x30, 0x0C, 0x06, 0x08, 0x2A, 0x86,
0x48, 0x86, 0xF7, 0x0D, 0x02, 0x02, 0x05, 0x00,
0x04, 0x10

MD5: 0x30, 0x20, 0x30, 0x0C, 0x06, 0x08, 0x2A, 0x86,
0x48, 0x86, 0xF7, 0x0D, 0x02, 0x05, 0x05, 0x00,
0x04, 0x10

RIPEMD-160: 0x30, 0x21, 0x30, 0x09, 0x06, 0x05, 0x2B, 0x24,
0x03, 0x02, 0x01, 0x05, 0x00, 0x04, 0x14

SHA-1: 0x30, 0x21, 0x30, 0x09, 0x06, 0x05, 0x2b, 0x0E,
0x03, 0x02, 0x1A, 0x05, 0x00, 0x04, 0x14

SHA256: 0x30, 0x31, 0x30, 0x0d, 0x06, 0x09, 0x60, 0x86,
0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01, 0x05,
0x00, 0x04, 0x20

SHA384: 0x30, 0x41, 0x30, 0x0d, 0x06, 0x09, 0x60, 0x86,
0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x02, 0x05,
0x00, 0x04, 0x30

SHA512: 0x30, 0x51, 0x30, 0x0d, 0x06, 0x09, 0x60, 0x86,
0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x03, 0x05,
0x00, 0x04, 0x40

DSA signatures MUST use hashes with a size of 160 bits, to match q, the size of the group generated by the DSA key's generator value. The hash function result is treated as a 160 bit number and used directly in the DSA signature algorithm.

5.2.3. Version 4 Signature Packet Format

The body of a version 4 Signature Packet contains:

- One-octet version number (4).
- 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. (two or more subpackets)

- Two-octet scalar octet count for 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. (zero or more subpackets)

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

- One or more multi-precision integers comprising the signature. This portion is algorithm specific, as described above.

The data being signed is hashed, and then the signature data from the version number through the hashed subpacket data (inclusive) is hashed. The resulting hash value is what is signed. The left 16 bits of the hash are included in the signature packet to provide a quick test to reject some invalid signatures.

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

5.2.3.1. Signature Subpacket Specification

The subpacket fields consist of zero or more signature subpackets. Each set of subpackets is preceded by a two-octet scalar count of the length of the set of 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. Note that this is the same encoding used for signature subpackets

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:

```
    2 = signature creation time
    3 = signature expiration time
    4 = exportable certification
    5 = trust signature
    6 = regular expression
    7 = revocable
    9 = key expiration time
    10 = placeholder for backward compatibility
    11 = preferred symmetric algorithms
    12 = revocation key
    16 = issuer key ID
    20 = notation data
    21 = preferred hash algorithms
    22 = preferred compression algorithms
    23 = key server preferences
    24 = preferred key server
    25 = primary user id
    26 = policy URL
    27 = key flags
    28 = signer's user id
    29 = reason for revocation
    30 = features
    31 = revocation identification
```

100 to 110 = internal or user-defined

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 "preferences" and the "reason for revocation" subpackets. 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.

5.2.3.2. 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 user id 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.

5.2.3.3. Notes on Self-Signatures

A self-signature is a binding signature made by the key the signature refers to. 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 username, 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 usernames, Alice and Bob. Suppose that Alice prefers the symmetric algorithm CAST5, and Bob prefers IDEA or Triple-DES. If the software locates this key via Alice's name, then the preferred algorithm is CAST5, if software locates the key via Bob's name, then the preferred algorithm is IDEA. If the key is located by key id, then algorithm of the default user id of the key provides the default symmetric algorithm.

Revoking a self-signature has a defined semantic meaning. 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 user themselves revokes their self-signature, it means the user no longer goes by that name, no

longer has that email address, etc. Revoking a binding signature effectively retires that subkey. Please see the "Reason for Revocation" subpacket below 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.

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.

5.2.3.4. Signature creation time

(4 octet time field)

The time the signature was made.

MUST be present in the hashed area.

5.2.3.5. Issuer

(8 octet key ID)

The OpenPGP key ID of the key issuing the signature.

5.2.3.6. 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.

5.2.3.7. Preferred symmetric algorithms

(sequence 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 in section 9. This is only found on a self-signature.

5.2.3.8. 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 6. This is only found on a self-signature.

5.2.3.9. 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 6. 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.

5.2.3.10. 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.

5.2.3.11. 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.

5.2.3.12. Revocable

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

Signature's revocability status. 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.

5.2.3.13. 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.

5.2.3.14. 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 package. A description of the syntax is found in a section below.

5.2.3.15. Revocation key

(1 octet of class, 1 octet of algid, 20 octets of 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 found on a self-signature.

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.

5.2.3.16. 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:

```
First octet: 0x80 = human-readable. This note value is text, a
                    note from one person to another, and need
                    not have meaning to software.
Other octets: none.
```

Notation names are arbitrary strings encoded in UTF-8. They reside two name spaces: The IETF name space and the user name space.

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

Names in the user name space 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 of bad form to create a new name in a DNS space that you don't own.

Since the user name space 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.

5.2.3.17. Key server preferences

(N octets of flags)

This is a list of 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: 0x80 = No-modify
the key holder requests that this key only be modified or updated by the key holder or an administrator of the key server.

This is found only on a self-signature.

5.2.3.18. Preferred key server

(String)

This is a URL 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 URL, the key server can actually be a copy of the key retrieved by ftp, http, finger, etc.

5.2.3.19. 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.

5.2.3.20. Policy URL

(String)

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

5.2.3.21. 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:

First octet:

0x01 - This key may be used to certify other keys.

0x02 - This key may be used to sign data.

0x04 - This key may be used to encrypt communications.

0x08 - This key may be used to encrypt storage.

0x10 - The private component of this key may have been split by a secret-sharing mechanism.

0x80 - The private component of this key may be in the possession of more than one person.

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.

5.2.3.22. 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.

5.2.3.23. 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:

- 0x00 - No reason specified (key revocations or cert revocations)
- 0x01 - Key is superceded (key revocations)
- 0x02 - Key material has been compromised (key revocations)
- 0x03 - Key is retired and no longer used (key revocations)
- 0x20 - User id information is no longer valid (cert revocations)

Following the revocation code is a string of octets which 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 superceded 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 an 0x20 subpacket.

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 no longer is a part of validity calculations.

5.2.3.24. 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.

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:

First octet:

0x01 - Modification Detection (packets 18 and 19)

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

5.2.3.25. Revocation Target

(1 octet PK algorithm, 1 octet hash algorithm, N octets hash)

This subpacket identifies a specific target signature that a revocation signature revokes. This provides explicit designation of a revocation. All arguments are an identifier of that 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.

5.2.4. Computing Signatures

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

The signature data is simple to compute for document signatures (types 0x00 and 0x01), for which the document itself is the data. For standalone signatures, this is a null string.

When a 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. (Note that this is an old-style packet header for a key packet with two-octet length.) A subkey signature (type 0x18) then hashes the subkey, using the same format as the main key (also using 0x99 as the first octet). Key revocation signatures (types 0x20 and 0x28) hash only the key 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 name packet,

without any header. A V4 certification hashes the constant 0xb4 for user ID certifications or the constant 0xd1 for User Attribute certifications (which are old-style packet headers with the length-of-length set to zero), 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.

Once the data body is hashed, then a trailer is hashed. 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. Thus, the fields hashed are the signature version, the signature type, the public key algorithm, the hash algorithm, the hashed subpacket length, and the hashed subpacket body.

V4 signatures also hash in a final trailer of six octets: the version of the signature packet, i.e. 0x04; 0xFF; a four-octet, big-endian number that is the length of the hashed data from the signature packet (note that this number does not include these final six octets.

After all this has been hashed, 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

An implementation SHOULD put the two mandatory subpackets, creation time and issuer, as the first subpackets in the subpacket list, simply to make it easier for the implementer to find them.

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 an 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 packet holds the symmetric-key encryption of a session key used to encrypt a message. Zero or more Encrypted Session Key packets and/or Symmetric-Key Encrypted Session Key packets may precede 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 Symmetrically Encrypted Data Packet 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 pass phrases. This packet type is new, and is not generated by PGP 2.x or PGP 5.0.

The body of this packet consists of:

- A one-octet version number. The only currently defined version is 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 file, 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 Symmetrically Encrypted Data Packet, 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 insure that the decryption key is not repeated even if the passphrase is reused.

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 2.6.x, 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 2.6.x 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 two versions of key-material packets. Version 3 packets were first generated by PGP 2.6. Version 2 packets are identical in format to Version 3 packets, but are generated by PGP 2.5 or before. V2 packets are deprecated and they MUST NOT be generated.

PGP 5.0 introduced version 4 packets, with new fields and semantics. PGP 2.6.x will not accept key-material packets with versions greater than 3.

OpenPGP implementations SHOULD create keys with version 4 format. An implementation MAY generate a V3 key to ensure interoperability with old software; note, however, that V4 keys correct some security deficiencies in V3 keys. These deficiencies are described below. An implementation MUST NOT create a V3 key with a public key algorithm other than RSA.

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 multi-precision integers comprising the key material:

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

- an MPI of RSA public encryption exponent e.

V3 keys SHOULD only be used for backward compatibility because of three weaknesses in them. 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, which increases the opportunity for fingerprint collisions. Third, there are minor weaknesses in the MD5 hash algorithm that make developers prefer other algorithms. See below for a fuller discussion of key IDs and fingerprints.

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 "Enhanced Key Formats."

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 multi-precision integers comprising the key material. This algorithm-specific portion is:

  Algorithm Specific Fields for RSA public keys:

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

  - MPI of RSA public encryption exponent e.

  Algorithm Specific Fields for DSA public keys:

  - 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).

  Algorithm Specific Fields for Elgamal public keys:

- 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).

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, in encrypted form.

The packet contains:

- A Public Key or Public Subkey packet, as described above

- One octet indicating string-to-key usage conventions. 0 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 specifier.

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

- [Optional] If string-to-key usage octet was 255 or 254, a string-to-key specifier. The length of the string-to-key specifier is implied by its type, as described above.

- [Optional] If secret data is encrypted, Initial Vector (IV) of the same length as the cipher's block size.

- Encrypted multi-precision integers comprising the secret key data. These algorithm-specific fields are as described below.

- If the string-to-key usage octet was 255, then a two-octet checksum of the plaintext of the algorithm-specific portion (sum of all octets, mod 65536). If the string-to-key usage octet was 254, then a 20-octet SHA-1 hash of the plaintext of the algorithm-specific portion. This checksum or hash is encrypted together with the algorithm-specific fields.

  Algorithm Specific Fields for RSA secret keys:

  - multiprecision integer (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.

Algorithm Specific Fields for DSA secret keys:

- MPI of DSA secret exponent x.

Algorithm Specific Fields for Elgamal secret keys:

- MPI of Elgamal secret exponent x.

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 SHOULD use a string-to-key specifier; the simple hash is for backward compatibility. The cipher for encrypting the MPIs is specified in the secret key packet.

Encryption/decryption of the secret data is done in CFB mode using the key created from the passphrase and the Initial Vector from the packet. 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 keys, a simpler method is used. All secret MPI values are encrypted in CFB mode, including the MPI bitcount prefix.

The 16-bit 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 on the private key that can undetectably modify the secret key. Using a SHA-1 hash prevents this.

5.6. Compressed Data Packet (Tag 8)

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

The body of this packet consists of:

- One octet that gives the algorithm used to compress the packet.

- The remainder of the packet is compressed data.

A Compressed Data Packet's body contains an block that compresses some set of packets. See section "Packet Composition" 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.

5.7. Symmetrically Encrypted Data Packet (Tag 9)

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

The body of this packet consists of:

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

The symmetric cipher used may be specified in an 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.

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-length 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.

5.8. 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.x, this packet has been re-assigned 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 2.6.x in order to cause that version to report that newer software is necessary to process the message.

5.9. Literal Data Packet (Tag 11)

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

The body of this packet consists of:

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

If it is a 'b' (0x62), then the literal packet contains binary data. If it is a 't' (0x74), then it contains text data, and thus may need line ends converted to local form, or other text-mode changes.

Early versions of PGP also defined a value of 'l' as a 'local' mode for machine-local conversions. RFC 1991 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 file name), if the encrypted data should be saved as a file.

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 the modification date of the file, or the creation time of the packet, or a zero that indicates the present 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.

5.10. Trust Packet (Tag 12)

The Trust packet is used only within keyrings and is not normally exported. Trust packets contain data that record the user's specifications of which key holders are trustworthy introducers, along with other information that implementing software uses for trust information.

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

5.11. User ID Packet (Tag 13)

A User ID packet consists of data that is intended to represent the name and email address of the key holder. By convention, it includes an RFC822 mail name, but there are no restrictions on its content. The packet length in the header specifies the length of the user id. If it is text, it is encoded in UTF-8.

5.12. User Attribute Packet (Tag 17)

The User Attribute packet is a variation of the User ID packet. It is capable of storing more types of data than the User ID packet which is (by convention) 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 only currently defined subpacket type is 1, signifying an image. An implementation SHOULD ignore any subpacket of a type that it does not recognize. Subpacket types 100 through 110 are reserved for private or experimental use.

5.12.1. The Image Attribute Subpacket

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

The image attribute subpacket begins with an image header. The first two octets of the image header contain the length of the image header. Note that unlike other multi-byte numerical values in this document, due to an 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 and determine the type of an image by examination of the image data if it is unable to handle a particular version of the image header or if a specified encoding format value is not recognized.

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

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 SHOULD NOT use this packet when encrypting to a recipient that does not state it can use this packet, and SHOULD prefer this to older Symmetrically Encrypted Data Packet when possible.

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 literal data packets or compressed data packets). 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 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.

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. Specifically, the input to the hash function does not include the prefix data described above; it includes all of the plaintext, and then also

includes two octets of values 0xD0, 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 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, not including the prefix data but including 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 difference in hash values is an indication that the message has been modified and SHOULD be reported to the user. Likewise, the absence of an MDC packet, or an MDC packet in any position other than the end of the plaintext, also represent message modifications and SHOULD also be reported.

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.

5.14. 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 which 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, not including prefix data but including the tag and length byte 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.

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 a checksum. The base64 encoding is identical to the MIME base64 content-transfer-encoding [RFC 2045]. An OpenPGP implementation MAY use ASCII Armor to protect the raw binary data.

The checksum is a 24-bit CRC converted to four characters of radix-64 encoding by the same MIME base64 transformation, preceded by an equals 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.

The checksum with its leading equal sign MAY appear on the first 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_POLY 0x1864cfbL

typedef 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 ^= CRC24_POLY;
                }
```

```
        }
        return crc & 0xffffffL;
    }
```

6.2. Forming ASCII Armor

When OpenPGP encodes data into ASCII Armor, it puts specific headers around the data, so OpenPGP can reconstruct the data later. 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 signatures following clearsigned messages. Note that PGP 2.x

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 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.

Currently defined Armor Header Keys are:

- "Version", that states the OpenPGP Version used to encode the message.

- "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. Consquently, if a comment has character 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.

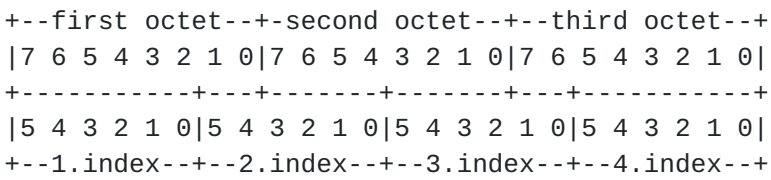
- "Hash", a comma-separated list of hash algorithms used in this message. This is used only in clear-signed messages.

- "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.



Each 6-bit group is used as an index into an array of 64 printable characters from the table below. The character referenced by the index is placed in the output string.

Value	Encoding	Value	Encoding	Value	Encoding	Value	Encoding
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4

6 G	23 X	40 o	57 5
7 H	24 Y	41 p	58 6
8 I	25 Z	42 q	59 7

9 J	26 a	43 r	60 8
10 K	27 b	44 s	61 9
11 L	28 c	45 t	62 +
12 M	29 d	46 u	63 /
13 N	30 e	47 v	
14 O	31 f	48 w	(pad) =
15 P	32 g	49 x	
16 Q	33 h	50 y	

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

Any characters outside of the base64 alphabet are ignored in Radix-64 data. Decoding software must ignore all line breaks or other characters not found in the table above.

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.

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

6.5. Examples of Radix-64

```
Input data:  0x14fb9c03d97e
Hex:     1   4    f   b    9   c     | 0   3    d   9    7   e
```

```
       8-bit:   00010100 11111011 10011100  | 00000011 11011001
       11111110
```

```
      6-bit:   000101 001111 101110 011100 | 000000 111101 100111
      111110
      Decimal: 5      15     46     28       0      61     37     62
      Output:  F      P      u      c        A      9      l      +

      Input data:  0x14fb9c03d9
      Hex:     1   4    f   b    9   c     | 0   3    d   9
      8-bit:   00010100 11111011 10011100  | 00000011 11011001
                                             pad with 00
      6-bit:   000101 001111 101110 011100 | 000000 111101 100100
      Decimal: 5      15     46     28       0      61     36
                                                  pad with =
      Output:  F      P      u      c        A      9      k      =

      Input data:  0x14fb9c03
      Hex:     1   4    f   b    9   c     | 0   3
      8-bit:   00010100 11111011 10011100  | 00000011
                                             pad with 0000
      6-bit:   000101 001111 101110 011100 | 000000 110000
      Decimal: 5      15     46     28       0      48
                                                pad with =      =
      Output:  F      P      u      c        A      w      =      =
```

6.6. Example of an ASCII Armored Message

```
-----BEGIN PGP MESSAGE-----
Version: OpenPrivacy 0.99

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

Note that this example is indented by two spaces.

7. Cleartext signature framework

It is desirable 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. (Note that RFC 3156 defines another way to clear sign 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 is used for the signature. If there are no such headers, MD5 is used, an implementation MAY omit them for 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.

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 dash '-' (0x2D) is prefixed by the sequence dash '-' (0x2D) and space ' ' (0x20). This prevents the parser from recognizing armor headers of the cleartext itself. 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.

Also, any trailing whitespace (spaces, and tabs, 0x09) at the end of any line is ignored when the cleartext signature is calculated.

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.

See the section "Notes on Algorithms" below for more discussion of the algorithms.

9.1. Public Key Algorithms

ID	Algorithm
1	- RSA (Encrypt or Sign)
2	- RSA Encrypt-Only
3	- RSA Sign-Only
16	- Elgamal (Encrypt-Only), see [ELGAMAL]
17	- DSA (Digital Signature Standard) [SCHNEIER]
18	- Reserved for Elliptic Curve
19	- Reserved for ECDSA
20	- Elgamal (Encrypt or Sign)
21	- Reserved for Diffie-Hellman (X9.42, as defined for IETF-S/MIME)
100 to 110	- Private/Experimental algorithm.

Implementations MUST implement DSA for signatures, and Elgamal for encryption. Implementations SHOULD implement RSA keys. Implementations MAY implement any other algorithm.

9.2. Symmetric Key Algorithms

ID		Algorithm
0	-	Plaintext or unencrypted data
1	-	IDEA [IDEA]
2	-	Triple-DES (DES-EDE, [SCHNEIER] - 168 bit key derived from 192)
3	-	CAST5 (128 bit key, as per RFC2144)
4	-	Blowfish (128 bit key, 16 rounds) [BLOWFISH]
5	-	SAFER-SK128 (13 rounds) [SAFER]
6	-	Reserved for DES/SK
7	-	AES with 128-bit key [AES]
8	-	AES with 192-bit key
9	-	AES with 256-bit key
10	-	Twofish with 256-bit key [TWOFISH]
100 to 110	-	Private/Experimental algorithm.

Implementations MUST implement Triple-DES. 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.

9.3. Compression Algorithms

ID		Algorithm
0	-	Uncompressed
1	-	ZIP (RFC1951)
2	-	ZLIB (RFC1950)
100 to 110	-	Private/Experimental algorithm.

Implementations MUST implement uncompressed data. Implementations SHOULD implement ZIP. Implementations MAY implement ZLIB.

9.4. Hash Algorithms

ID		Algorithm	Text Name
1	-	MD5	"MD5"
2	-	SHA-1	"SHA1"
3	-	RIPE-MD/160	"RIPEMD160"
4	-	Reserved for double-width SHA (experimental, obviated)	
5	-	MD2	"MD2"
6	-	Reserved for TIGER/192	"TIGER192"
7	-	Reserved for HAVAL (5 pass, 160-bit)	"HAVAL-5-160"
8	-	SHA256	"SHA256"

```
       9          - SHA384                                  "SHA384"
       10         - SHA512                                  "SHA512"
       100 to 110 - Private/Experimental algorithm.
```

Implementations MUST implement SHA-1. Implementations SHOULD implement MD5.

10. 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.

10.1. Transferable Public Keys

OpenPGP users may transfer public keys. The essential elements of a transferable public key are:

- 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, 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 packets there may be one 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 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.

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.

10.2. OpenPGP Messages

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

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

Compressed Message :- Compressed Data Packet.

Literal Message :- Literal Data Packet.

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

ESK Sequence :- ESK | ESK Sequence, ESK.

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

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

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

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

In addition, decrypting a Symmetrically Encrypted Data Packet or a Symmetrically Encrypted Integrity Protected Data Packet as well as

decompressing a Compressed Data packet must yield a valid OpenPGP Message.

10.3. 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 that they are a signature of.

11. Enhanced Key Formats

11.1. 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.

The format of an OpenPGP V4 key that uses two 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 Self Signature...]
            User ID [Signature ...]
           [User ID [Signature ...] ...]
           [User Attribute [Signature ...] ...]
           [[Subkey [Binding-Signature-Revocation]
                   Primary-Key-Binding-Signature] ...]
```

A subkey always has a single signature after it that is issued using the primary key to tie the two keys together. This binding signature may be in either V3 or V4 format, but V4 is preferred, of course.

In the above diagram, if the binding signature of a subkey has been revoked, the revoked key may be removed, leaving only one key.

In a key that has a main key and subkeys, 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 used for certifying subkeys that are used for encryption and signatures.

11.2. 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.

A V4 fingerprint is the 160-bit SHA-1 hash of the one-octet Packet Tag, 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) high order length octet of (b)-(f) (1 octet)

a.3) low order length octet of (b)-(f) (1 octet)

b) version number = 4 (1 octet);

c) time stamp 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).

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 and V4 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 as the first byte (even though this is not a valid packet ID for a public subkey).

12. Notes on Algorithms

12.1. 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 different preferences. For example, Alice may have TripleDES only specified for "alice@work.com" but CAST5, Blowfish, and TripleDES 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.)

An implementation that is striving for backward compatibility MAY consider a V3 key with a V3 self-signature to be an implicit preference for IDEA, and no ability to do TripleDES. This is technically non-compliant, but an implementation MAY violate the above rule in this case only and use IDEA to encrypt the message, provided that the message creator is warned. Ideally, though, the implementation would follow the rule by actually generating two messages, because it is possible that the OpenPGP user's implementation does not have IDEA, and thus could not read the message. Consequently, an implementation MAY, but SHOULD NOT use IDEA in an algorithm conflict with a V3 key.

12.2. 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.

12.2.1. 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 uncompressed 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.

12.2.2. 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.

12.3. 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.

12.4. RSA

There are algorithm types for RSA-signature-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 768 bits.

It is permissible for an implementation to support RSA merely for backward compatibility; for example, such an implementation would support V3 keys with IDEA symmetric cryptography. Note that this is an exception to the other MUST-implement rules. An implementation that supports RSA in V4 keys MUST implement the MUST-implement features.

12.5. Elgamal

If an Elgamal key [ELGAMAL] is to be used for both signing and encryption, extra care must be taken in creating the key.

An Elgamal key consists of a generator g, a prime modulus p, a secret exponent x, and a public value y = g^x mod p.

The generator and prime must be chosen so that solving the discrete log problem is intractable. The group g should generate the multiplicative group mod p-1 or a large subgroup of it, and the order of g should have at least one large prime factor. A good choice is to use a "strong" Sophie-Germain prime in choosing p, so that both p and (p-1)/2 are primes. In fact, this choice is so good that implementers SHOULD do it, as it avoids a small subgroup attack.

In addition, a result of Bleichenbacher [BLEICHENBACHER] shows that

if the generator g has only small prime factors, and if g divides the order of the group it generates, then signatures can be forged.

In particular, choosing g=2 is a bad choice if the group order may be even. On the other hand, a generator of 2 is a fine choice for an encryption-only key, as this will make the encryption faster.

While verifying Elgamal signatures, note that it is important to test that r and s are less than p. If this test is not done then signatures can be trivially forged by using large r values of approximately twice the length of p. This attack is also discussed in the Bleichenbacher paper.

Details on safe use of Elgamal signatures may be found in [MENEZES], which discusses all the weaknesses described above. Please note that Elgamal signatures are controversial; because of the care that must be taken with Elgamal keys, many implementations forego them.

If an implementation allows Elgamal signatures, then it MUST use the algorithm identifier 20 for an Elgamal public key that can sign.

An implementation SHOULD NOT implement Elgamal keys of size less than 768 bits. For long-term security, Elgamal keys should be 1024 bits or longer.

12.6. DSA

An implementation SHOULD NOT implement DSA keys of size less than 768 bits. Note that present DSA is limited to a maximum of 1024 bit keys, which are recommended for long-term use. Also, DSA keys MUST be an even multiple of 64 bits long.

12.7. 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 the Public Algorithms section as "(reserved for)".

The reserved public key algorithms, Elliptic Curve (18), ECDSA (19), and X9.42 (21) do not have the necessary parameters, parameter order, or semantics defined.

The reserved symmetric key algorithm, DES/SK (6), does not have semantics defined.

The reserved hash algorithms, TIGER192 (6), and HAVAL-5-160 (7), do not have OIDs. The reserved algorithm number 4, reserved for a double-width variant of SHA1, is not presently defined.

12.8. 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, but described in those 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 blocksize 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 "resync" 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 blocksize 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 resync step) FR is loaded with C[3] through C[BS+2].

8. FR is encrypted to produce FRE.

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

octets of ciphertext.

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

11. FR is encrypted to produce FRE.

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

13. 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. Possession of the private key portion of a public-private key pair is assumed to be controlled 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 RFC 1750.

* The MD5 hash algorithm has been found to have weaknesses (pseudo-collisions in the compress function) that make some people deprecate its use. They consider the SHA-1 algorithm better.

* 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 160-bit hash, but it is sensitive to the quality of the hash algorithm, if the hash algorithm is broken, it can leak the secret key. The Digital Signature Standard (DSS) specifies that DSA be used with SHA-1. RIPEMD-160 is considered by many cryptographers to be as strong. An implementation should take care which hash algorithms are used with DSA, as a weak hash can not only allow a signature to be forged, but could leak the secret key. These same considerations about the quality of the hash algorithm apply to Elgamal signatures.

* 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 Triple-DES. Other algorithms may have other controversies surrounding them.

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

14. 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.

* PGP 5.x does not accept V4 signatures for anything other than key material.

* PGP 5.x does not recognize the "five-octet" lengths in new-format headers or in signature subpacket lengths.

* PGP 5.0 rejects an encrypted session key if the keylength differs from the S2K symmetric algorithm. This is a bug in its validation function.

* PGP 5.0 does not handle multiple one-pass signature headers and trailers. Signing one will compress the one-pass signed literal and prefix a V3 signature instead of doing a nested one-pass signature.

* When exporting a private key, PGP 2.x 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.

* In a clear-signed signature, PGP 5.0 will figure out the correct hash algorithm if there is no "Hash:" header, but it will reject a mismatch between the header and the actual algorithm used. The "standard" (i.e. Zimmermann/Finney/et al.) version of PGP 2.x rejects the "Hash:" header and assumes MD5. There are a number of enhanced variants of PGP 2.6.x that have been modified for SHA-1 signatures.

* PGP 5.0 can read an RSA key in V4 format, but can only recognize it with a V3 key id, and can properly use only a V3 format RSA key.

* Neither PGP 5.x nor PGP 6.0 recognize Elgamal Encrypt and Sign keys. They only handle Elgamal Encrypt-only keys.

* 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.x, then the "windowBits" parameter should be set to -13.

* PGP 2.6.X and 5.0 do not trim trailing whitespace from a "canonical text" signature. They only remove it from cleartext signatures. These signatures are not OpenPGP compliant -- OpenPGP requires trimming the whitespace. If you wish to interoperate with PGP 2.6.X or PGP 5, you may wish to accept these non-compliant signatures.

15. Authors and Working Group Chair

The working group can be contacted via the current chair:

John W. Noerenberg, II
Qualcomm, Inc
6455 Lusk Blvd
San Diego, CA 92131 USA
Email: jwn2@qualcomm.com
Tel: +1 (619) 658-3510

The principal authors of this draft are:

Jon Callas
Wave Systems Corp.
1601 S. DeAnza Blvd, Suite 200
Cupertino, CA 95014, USA

Email: jon@callas.org, jcallas@wavesys.com
Tel: +1 (408) 448-6801

Lutz Donnerhacke
IKS GmbH
Wildenbruchstr. 15
07745 Jena, Germany

EMail: lutz@iks-jena.de
Tel: +49-3641-675642

Hal Finney
Network Associates, Inc.
3965 Freedom Circle
Santa Clara, CA 95054, USA

Email: hal@pgp.com

Rodney Thayer

Email: rodney@tillerman.to

This memo also draws on much previous work from a number of other authors who include: Derek Atkins, Charles Breed, Dave Del Torto, Marc Dyksterhouse, Gail Haspert, Gene Hoffman, Paul Hoffman, Raph Levien, Colin Plumb, Will Price, David Shaw, William Stallings, Mark Weaver, and Philip R. Zimmermann.

16. References

[AES] Advanced Encryption Standards Questions and Answers <http://csrc.nist.gov/encryption/aes/round2/aesfact.html>

<http://csrc.nist.gov/encryption/aes/round2/r2algs.html#Rijndael>

[BLEICHENBACHER] Bleichenbacher, Daniel, "Generating ElGamal signatures without knowing the secret key," Eurocrypt 96. Note that the version in the proceedings has an error. A revised version is available at the time of writing from <ftp://ftp.inf.ethz.ch/pub/publications/papers/ti/isc/ElGamal.ps>

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

[DONNERHACKE] Donnerhacke, L., et. al, "PGP263in - an improved international version of PGP", ftp://ftp.iks-jena.de/mitarb/lutz/crypt/software/pgp/

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

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

[ISO10646] ISO/IEC 10646-1:1993. International Standard -- Information technology -- Universal Multiple-Octet Coded Character Set (UCS) -- Part 1: Architecture and Basic Multilingual Plane.

[JFIF] JPEG File Interchange Format (Version 1.02). Eric Hamilton, C-Cube Microsystems, Milpitas, CA, September 1, 1992.

[MENEZES] Alfred Menezes, Paul van Oorschot, and Scott Vanstone, "Handbook of Applied Cryptography," CRC Press, 1996.

[RFC822] Crocker, D., "Standard for the format of ARPA Internet text messages", STD 11, RFC 822, August 1982.

[RFC1423] Balenson, D., "Privacy Enhancement for Internet Electronic Mail: Part III: Algorithms, Modes, and Identifiers", RFC 1423, October 1993.

[RFC1641] Goldsmith, D. and M. Davis, "Using Unicode with MIME", RFC 1641, July 1994.

[RFC1750] Eastlake, D., Crocker, S. and J. Schiller, "Randomness Recommendations for Security", RFC 1750, December 1994.

[RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3.", RFC 1951, May 1996.

[RFC1983] Malkin, G., "Internet Users' Glossary", FYI 18, RFC 1983, August 1996.

[RFC1991] Atkins, D., Stallings, W. and P. Zimmermann, "PGP Message Exchange Formats", RFC 1991, August 1996.

[RFC2045] Borenstein, N. and N. Freed, "Multipurpose Internet

Mail Extensions (MIME) Part One: Format of Internet Message Bodies.", RFC 2045, November 1996.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Level", BCP 14, RFC 2119, March 1997.

[RFC2144] Adams, C., "The CAST-128 Encryption Algorithm", RFC 2144, May 1997.

[RFC2279] Yergeau., F., "UTF-8, a transformation format of Unicode and ISO 10646", RFC 2279, January 1998.

[RFC2437] B. Kaliski and J. Staddon, " PKCS #1: RSA Cryptography Specifications Version 2.0", RFC 2437, October 1998.

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

[SAFER] Massey, J.L. "SAFER K-64: One Year Later", B. Preneel, editor, Fast Software Encryption, Second International Workshop (LNCS 1008) pp212-241, Springer-Verlag 1995

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

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

17. Full Copyright Statement