

openpgp
Internet-Draft
Intended status: Informational
Expires: 28 April 2022

D.K. Gillmor
ACLU
25 October 2021

Stateless OpenPGP Command Line Interface
draft-dkg-openpgp-stateless-cli-03

Abstract

This document defines a generic stateless command-line interface for dealing with OpenPGP messages, known as sop. It aims for a minimal, well-structured API covering OpenPGP object security.

Status of This Memo

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

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

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

This Internet-Draft will expire on 28 April 2022.

Copyright Notice

Copyright (c) 2021 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

Table of Contents

1.	Introduction	3
1.1.	Requirements Language	4
1.2.	Terminology	4
1.3.	Using sop in a Test Suite	4
2.	Examples	5
3.	Subcommands	5
3.1.	version: Version Information	5
3.2.	generate-key: Generate a Secret Key	6
3.3.	extract-cert: Extract a Certificate from a Secret Key	7
3.4.	sign: Create Detached Signatures	7
3.5.	verify: Verify Detached Signatures	8
3.6.	encrypt: Encrypt a Message	9
3.7.	decrypt: Decrypt a Message	11
3.8.	armor: Convert binary to ASCII	13
3.9.	dearmor: Convert ASCII to binary	14
3.10.	detach-inband-signature-and-message: split a clearsinged message	14
4.	Input String Types	15
4.1.	DATE	16
4.2.	USERID	16
5.	Input/Output Indirect Types	16
5.1.	Special Designators for Indirect Types	17
5.2.	CERTS	17
5.3.	KEYS	17
5.4.	CIPHERTEXT	18
5.5.	SIGNATURES	18
5.6.	SESSIONKEY	18
5.7.	MICALG	19
5.8.	PASSWORD	19
5.9.	VERIFICATIONS	19
5.10.	DATA	20
6.	Failure Modes	20
7.	Alternate Interfaces	22
8.	Guidance for Implementers	22
8.1.	One OpenPGP Message at a Time	23
8.2.	Simplified Subset of OpenPGP Message	23
8.3.	Validate Signatures Only from Known Signers	23
8.4.	OpenPGP inputs can be either Binary or ASCII-armored	23
8.5.	Detached Signatures	24
8.6.	Reliance on Supplied Certs and Keys	25
8.7.	Text is always UTF-8	25
8.8.	Passwords are Human-Readable	26
8.9.	Be careful with Special Designators	27
9.	Guidance for Consumers	27
9.1.	Choosing between -as=text and -as=binary	28
9.2.	Special Designators and Unusual Filenames	28

10. Security Considerations	28
10.1. Signature Verification	29
10.2. Compression	30
11. Privacy Considerations	30
11.1. Object Security vs. Transport Security	30
12. Document Considerations	30
12.1. Document History	30
12.2. Future Work	32
13. Acknowledgements	33
14. References	33
14.1. Normative References	33
14.2. Informative References	34
Author's Address	35

1. Introduction

Different OpenPGP implementations have many different requirements, which typically break down in two main categories: key/certificate management and object security.

The purpose of this document is to provide a "stateless" interface that primarily handles the object security side of things, and assumes that secret key management and certificate management will be handled some other way.

Isolating object security from key/certificate management should make it easier to provide interoperability testing for the object security side of OpenPGP implementations, as described in Section 1.3.

This document defines a generic stateless command-line interface for dealing with OpenPGP messages, known here by the placeholder `sop`. It aims for a minimal, well-structured API.

An OpenPGP implementation should not name its executable `sop` to implement this specification. It just needs to provide a program that conforms to this interface.

A `sop` implementation should leave no trace on the system, and its behavior should not be affected by anything other than command-line arguments and input.

Obviously, the user will need to manage their secret keys (and their peers' certificates) somehow, but the goal of this interface is to separate out that task from the task of interacting with OpenPGP messages.

While this document identifies a command-line interface, the rough outlines of this interface should also be amenable to relatively straightforward library implementations in different languages.

1.1. Requirements Language

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.2. Terminology

This document uses the term "key" to refer exclusively to OpenPGP Transferable Secret Keys (see section 11.2 of [RFC4880]).

It uses the term "certificate" to refer to OpenPGP Transferable Public Key (see section 11.1 of [RFC4880]).

"Stateless" in "Stateless OpenPGP" means avoiding secret key and certificate state. The user is responsible for managing all OpenPGP certificates and secret keys themselves, and passing them to sop as needed. The user should also not be concerned that any state could affect the underlying operations.

OpenPGP revocations can have "Reason for Revocation" (section 5.2.3.23 of [RFC4880]), which can be either "soft" or "hard". The set of "soft" reasons is: "Key is superseded" and "Key is retired and no longer used". All other reasons (and revocations that do not state a reason) are "hard" revocations.

1.3. Using sop in a Test Suite

If an OpenPGP implementation provides a sop interface, it can be used to test interoperability (e.g., [OpenPGP-Interoperability-Test-Suite]).

Such an interop test suite can, for example, use custom code (`_not_sop`) to generate a new OpenPGP object that incorporates new primitives, and feed that object to a stable of sop implementations, to determine whether those implementations can consume the new form.

Or, the test suite can drive each sop implementation with a simple input, and observe which cryptographic primitives each implementation chooses to use as it produces output.

2. Examples

These examples show no error checking, but give a flavor of how `sop` might be used in practice from a shell.

The key and certificate files described in them (e.g. `alice.sec`) could be for example those found in [I-D.draft-bre-openpgp-samples-01].

```
sop generate-key "Alice Lovelace <alice@openpgp.example>" > alice.sec
sop extract-cert < alice.sec > alice.pgp
```

```
sop generate-key "Bob Babbage <bob@openpgp.example>" > bob.sec
sop extract-cert < bob.sec > bob.pgp
```

```
sop sign --as=text alice.sec < statement.txt > statement.txt.asc
sop verify statement.txt.asc alice.pgp < statement.txt
```

```
sop encrypt --sign-with=alice.sec --as=mime bob.pgp < msg.eml > ciphertext.asc
sop decrypt bob.sec < ciphertext.asc > cleartext.eml
```

See Section 6 for more information about errors and error handling.

3. Subcommands

`sop` uses a subcommand interface, similar to those popularized by systems like `git` and `svn`.

If the user supplies a subcommand that `sop` does not implement, it fails with `UNSUPPORTED_SUBCOMMAND`. If a `sop` implementation does not handle a supplied option for a given subcommand, it fails with `UNSUPPORTED_OPTION`.

All subcommands that produce OpenPGP material on standard output produce ASCII-armored (section 6 of [I-D.ietf-openpgp-rfc4880bis]) objects by default (except for `sop dearmor`). These subcommands have a `--no-armor` option, which causes them to produce binary OpenPGP material instead.

All subcommands that accept OpenPGP material on input should be able to accept either ASCII-armored or binary inputs (see Section 8.4) and behave accordingly.

See Section 5 for details about how various forms of OpenPGP material are expected to be structured.

3.1. version: Version Information

```
sop version [--backend|--extended]
```

- * Standard Input: ignored

- * Standard Output: version information

This subcommand emits version information as UTF-8-encoded text.

With no arguments, the version string emitted should contain the name of the sop implementation, followed by a single space, followed by the version number. A sop implementation should use a version number that respects an established standard that is easily comparable and parsable, like [SEMVER].

If --backend is supplied, the implementation should produce a comparable line of implementation and version information about the primary underlying OpenPGP toolkit.

If --extended is supplied, the implementation may emit multiple lines of version information. The first line **MUST** match the information produced by a simple invocation, but the rest of the text has no defined structure.

--backend and --extended are mutually-exclusive options.

Example:

```
$ sop version
ExampleSop 0.2.1
$ sop version --backend
LibExamplePGP 3.4.2
$ sop version --extended
ExampleSop 0.2.1
Running on MonkeyScript 4.5
LibExamplePGP 3.4.2
LibExampleCrypto 3.1.1
LibXCompression 4.0.2
See https://pgp.example/sop/ for more information
$
```

3.2. generate-key: Generate a Secret Key

```
sop generate-key [--no-armor] [--] [USERID...]
```

- * Standard Input: ignored

- * Standard Output: KEYS (Section 5.3)

Generate a single default OpenPGP key with zero or more User IDs.

The generated secret key SHOULD be usable for as much of the sop functionality as possible. In particular:

- * It should be possible to extract an OpenPGP certificate from the key in KEYS with `sop extract-cert`.
- * The key in KEYS should be able to create signatures (with `sop sign`) that are verifiable by using `sop verify` with the extracted certificate.
- * The key in KEYS should be able to decrypt messages (with `sop decrypt`) that are encrypted by using `sop encrypt` with the extracted certificate.

The detailed internal structure of the certificate is left to the discretion of the sop implementation.

Example:

```
$ sop generate-key 'Alice Lovelace <alice@openpgp.example>' > alice.sec
$ head -n1 < alice.sec
-----BEGIN PGP PRIVATE KEY BLOCK-----
$
```

3.3. `extract-cert`: Extract a Certificate from a Secret Key

`sop extract-cert [--no-armor]`

- * Standard Input: KEYS (Section 5.3)
- * Standard Output: CERTS (Section 5.2)

The output should contain one OpenPGP certificate in CERTS per OpenPGP Transferable Secret Key found in KEYS. There is no guarantee what order the CERTS will be in.

Example:

```
$ sop extract-cert < alice.sec > alice.pgp
$ head -n1 < alice.pgp
-----BEGIN PGP PUBLIC KEY BLOCK-----
$
```

3.4. `sign`: Create Detached Signatures

```
sop sign [--no-armor] [--micalg-out=MICALG]
        [--as={binary|text}] [--] KEYS [KEYS...]
```

* Standard Input: DATA (Section 5.10)

* Standard Output: SIGNATURES (Section 5.5)

Exactly one signature will be made by each key in the supplied KEYS arguments.

--as defaults to binary. If --as=text and the input DATA is not valid UTF-8 (Section 8.7), sop sign fails with EXPECTED_TEXT.

--as=binary SHOULD result in an OpenPGP signature of type 0x00 ("Signature of a binary document"). --as=text SHOULD result in an OpenPGP signature of type 0x01 ("Signature of a canonical text document"). See section 5.2.1 of [RFC4880] for more details.

When generating PGP/MIME messages ([RFC3156]), it is useful to know what digest algorithm was used for the generated signature. When --micalg-out is supplied, sop sign emits the digest algorithm used to the specified MICALG file in a way that can be used to populate the micalg parameter for the Content-Type (see Section 5.7). If the specified MICALG file already exists in the filesystem, sop sign will fail with OUTPUT_EXISTS.

When signing with multiple keys, sop sign SHOULD use the same digest algorithm for every signature generated in a single run, unless there is some internal constraint on the KEYS objects. If --micalg-out is requested, and multiple incompatibly-constrained KEYS objects are supplied, sop sign MUST emit the empty string to the designated MICALG.

If any key in the KEYS objects is not capable of producing a signature, sop sign will fail with KEY_CANNOT_SIGN.

sop sign MUST NOT produce any extra signatures beyond those from KEYS objects supplied on the command line.

Example:

```
$ sop sign --as=text alice.sec < message.txt > message.txt.asc
$ head -n1 < message.txt.asc
-----BEGIN PGP SIGNATURE-----
$
```

3.5. verify: Verify Detached Signatures


```
sop verify [--not-before=DATE] [--not-after=DATE]
          [--] SIGNATURES CERTS [CERTS...]
```

* Standard Input: DATA (Section 5.10)

* Standard Output: VERIFICATIONS (Section 5.9)

--not-before and --not-after indicate that signatures with dates outside certain range MUST NOT be considered valid.

--not-before defaults to the beginning of time. Accepts the special value - to indicate the beginning of time (i.e. no lower boundary).

--not-after defaults to the current system time (now). Accepts the special value - to indicate the end of time (i.e. no upper boundary).

sop verify only returns OK if at least one certificate included in any CERTS object made a valid signature in the range over the DATA supplied.

For details about the valid signatures, the user MUST inspect the VERIFICATIONS output.

If no CERTS are supplied, sop verify fails with MISSING_ARG.

If no valid signatures are found, sop verify fails with NO_SIGNATURE.

See Section 10.1 for more details about signature verification.

Example:

(In this example, we see signature verification succeed first, and then fail on a modified version of the message.)

```
$ sop verify message.txt.asc alice.pgp < message.txt
2019-10-29T18:36:45Z EB85BB5FA33A75E15E944E63F231550C4F47E38E EB85BB5FA33A75E15E9
44E63F231550C4F47E38E signed by alice.pgp
$ echo $?
0
$ tr a-z A-Z < message.txt | sop verify message.txt.asc alice.pgp
$ echo $?
3
$
```

3.6. encrypt: Encrypt a Message

```
sop encrypt [--as={binary|text|mime}]
  [--no-armor]
  [--with-password=PASSWORD...]
  [--sign-with=KEYS...]
  [--] [CERTS...]
```

* Standard Input: DATA (Section 5.10)

* Standard Output: CIPHERTEXT (Section 5.4)

--as defaults to binary. The setting of --as corresponds to the one octet format field found in the Literal Data packet at the core of the output CIPHERTEXT. If --as is set to binary, the octet is b (0x62). If it is text, the format octet is u (0x75). If it is mime, the format octet is m (0x6d).

--with-password enables symmetric encryption (and can be used multiple times if multiple passwords are desired). If sop encrypt encounters a PASSWORD which is not a valid UTF-8 string (Section 8.7), or is otherwise not robust in its representation to humans, it fails with PASSWORD_NOT_HUMAN_READABLE. If sop encrypt sees trailing whitespace at the end of a PASSWORD, it will trim the trailing whitespace before using the password. See Section 8.8 for more discussion about passwords.

--sign-with creates exactly one signature by for each secret key found in the supplied KEYS object (this can also be used multiple times if signatures from keys found in separate files are desired). If any key in any supplied KEYS objects is not capable of producing a signature, sop sign will fail with KEY_CANNOT_SIGN.

If --as is set to binary, then --sign-with will sign as a binary document (OpenPGP signature type 0x00).

If --as is set to text, then --sign-with will sign as a canonical text document (OpenPGP signature type 0x01). In this case, if the input DATA is not valid UTF-8 (Section 8.7), sop encrypt fails with EXPECTED_TEXT.

sop should only be invoked with --as=mime when the input DATA is a MIME message ([RFC2045]). If --sign-with is supplied for such a message, then if the input data is valid UTF-8, sop SHOULD sign as a canonical text document (OpenPGP signature type 0x01). However, a MIME message itself might not be valid UTF-8, for example, if a MIME subpart contains a raw binary object. If --sign-with is supplied for input DATA that is not valid UTF-8, sop encrypt MAY sign as a binary document (OpenPGP signature type 0x00).

sop encrypt MUST NOT produce any extra signatures beyond those from KEYS objects identified by --sign-with.

The resulting CIPHERTEXT should be decryptable by the secret keys corresponding to every certificate included in all CERTS, as well as each password given with --with-password.

If no CERTS or --with-password options are present, sop encrypt fails with MISSING_ARG.

If at least one of the identified certificates requires encryption to an unsupported asymmetric algorithm, sop encrypt fails with UNSUPPORTED_ASYMMETRIC_ALGO.

If at least one of the identified certificates is not encryption-capable (e.g., revoked, expired, no encryption-capable flags on primary key and valid subkeys), sop encrypt fails with CERT_CANNOT_ENCRYPT.

If sop encrypt fails for any reason, it emits no CIPHERTEXT.

Example:

(In this example, bob.bin is a file containing Bob's binary-formatted OpenPGP certificate. Alice is encrypting a message to both herself and Bob.)

```
$ sop encrypt --as=mime --sign-with=alice.key alice.asc bob.bin < message.eml > encrypted.asc
$ head -n1 encrypted.asc
-----BEGIN PGP MESSAGE-----
$
```

3.7. decrypt: Decrypt a Message

```
sop decrypt [--session-key-out=SESSIONKEY]
  [--with-session-key=SESSIONKEY...]
  [--with-password=PASSWORD...]
  [--verify-out=VERIFICATIONS]
  [--verify-with=CERTS...]
  [--verify-not-before=DATE]
  [--verify-not-after=DATE] ]
  [--] [KEYS...]
```

* Standard Input: CIPHERTEXT (Section 5.4)

* Standard Output: DATA (Section 5.10)

The caller can ask `sop` for the session key discovered during decryption by supplying the `--session-key-out` option. If the specified file already exists in the filesystem, `sop decrypt` will fail with `OUTPUT_EXISTS`. When decryption is successful, `sop decrypt` writes the discovered session key to the specified file.

`--with-session-key` enables decryption of the CIPHERTEXT using the session key directly against the SEIPD packet. This option can be used multiple times if several possible session keys should be tried.

`--with-password` enables decryption based on any SKESK (section 5.3 of [I-D.ietf-openpgp-rfc4880bis]) packets in the CIPHERTEXT. This option can be used multiple times if the user wants to try more than one password.

If `sop decrypt` tries and fails to use a supplied PASSWORD, and it observes that there is trailing UTF-8 whitespace at the end of the PASSWORD, it will retry with the trailing whitespace stripped. See Section 8.8 for more discussion about passwords.

`--verify-out` produces signature verification status to the designated file. If the designated file already exists in the filesystem, `sop decrypt` will fail with `OUTPUT_EXISTS`.

The return code of `sop decrypt` is not affected by the results of signature verification. The caller MUST check the returned VERIFICATIONS to confirm signature status. An empty VERIFICATIONS output indicates that no valid signatures were found.

`--verify-with` identifies a set of certificates whose signatures would be acceptable for signatures over this message.

If the caller is interested in signature verification, both `--verify-out` and at least one `--verify-with` must be supplied. If only one of these arguments is supplied, `sop decrypt` fails with `INCOMPLETE_VERIFICATION`.

`--verify-not-before` and `--verify-not-after` provide a date range for acceptable signatures, by analogy with the options for `sop verify` (see Section 3.5). They should only be supplied when doing signature verification.

See Section 10.1 for more details about signature verification.

If no KEYS or `--with-password` or `--with-session-key` options are present, `sop decrypt` fails with `MISSING_ARG`.

If unable to decrypt, `sop decrypt` fails with `CANNOT_DECRYPT`.

sop decrypt only emits cleartext to Standard Output that was successfully decrypted.

Example:

(In this example, Alice stashes and re-uses the session key of an encrypted message.)

```
$ sop decrypt --session-key-out=session.key alice.sec < ciphertext.asc > cleartext.out
$ ls -l ciphertext.asc cleartext.out
-rw-r--r-- 1 user user  321 Oct 28 01:34 ciphertext.asc
-rw-r--r-- 1 user user  285 Oct 28 01:34 cleartext.out
$ sop decrypt --with-session-key=session.key < ciphertext.asc > cleartext2.out
$ diff cleartext.out cleartext2.out
$
```

3.8. armor: Convert binary to ASCII

```
sop armor [--label={auto|sig|key|cert|message}]
```

- * Standard Input: OpenPGP material (SIGNATURES, KEYS, CERTS, or CIPHERTEXT)
- * Standard Output: the same material with ASCII-armoring added, if not already present

The user can choose to specify the label used in the header and tail of the armoring.

The default for --label is auto, in which case, sop inspects the input and chooses the label appropriately, based on the type of the first OpenPGP packet. If the type of the first OpenPGP packet is:

- * 0x02 (Signature), the packet stream should be parsed as a SIGNATURES input (with Armor Header BEGIN PGP SIGNATURE).
- * 0x05 (Secret-Key), the packet stream should be parsed as a KEYS input (with Armor Header BEGIN PGP PRIVATE KEY BLOCK).
- * 0x06 (Public-Key), the packet stream should be parsed as a CERTS input (with Armor Header BEGIN PGP PUBLIC KEY BLOCK).
- * 0x01 (Public-key Encrypted Session Key) or 0x03 (Symmetric-key Encrypted Session Key), the packet stream should be parsed as a CIPHERTEXT input (with Armor Header BEGIN PGP MESSAGE).

If the input packet stream does not match the expected sequence of packet types, sop armor fails with BAD_DATA.

Since `sop armor` accepts ASCII-armored input as well as binary input, this operation is idempotent on well-structured data. A caller can use this subcommand blindly ensure that any well-formed OpenPGP packet stream is 7-bit clean.

Example:

```
$ sop armor < bob.bin > bob.pgp
$ head -nl bob.pgp
-----BEGIN PGP PUBLIC KEY BLOCK-----
$
```

3.9. `dearmor`: Convert ASCII to binary

`sop dearmor`

- * Standard Input: OpenPGP material (SIGNATURES, KEYS, CERTS, or CIPHERTEXT)
- * Standard Output: the same material with any ASCII-armoring removed

If the input packet stream does not match any of the expected sequence of packet types, `sop dearmor` fails with `BAD_DATA`. See also Section 8.4.

Since `sop dearmor` accepts binary-formatted input as well as ASCII-armored input, this operation is idempotent on well-structured data. A caller can use this subcommand blindly ensure that any well-formed OpenPGP packet stream is in its standard binary representation.

Example:

```
$ sop dearmor < message.txt.asc > message.txt.sig
$
```

3.10. `detach-inband-signature-and-message`: split a clearsigned message

`sop detach-inband-signature-and-message [--no-armor] --signatures-out=SIGNATURES`

- * Standard Input: DATA (clearsigned message)
- * Standard Output: DATA (the message without the cleartext signature framework)

In some contexts, the user may encounter a clearsigned ("inline PGP") message (section 7 of [RFC4880]) rather than a message and its detached signature. This subcommand takes such a clearsigned message on standard input, and splits it into:

- * the potentially signed material on standard output, and
- * a detached signature block to the destination identified by --signatures-out

Note that no cryptographic verification of the signatures is done by this subcommand. Once the clearsigned message is separated, verification of the detached signature can be done with `sop verify`.

If no `--signatures-out` is supplied, `sop detach-inband-signature-and-message` fails with `MISSING_ARG`.

Note that the signature block in a clearsigned message may contain multiple signatures. All signatures found in the signature block will be emitted to the `--signatures-out` destination.

The message body in the clearsigned message will be dash-escaped on standard input (see section 7.1 of [RFC4880]). The output of `sop detach-inband-signature-and-message` will have dash-escaping removed.

If the input DATA contains no clearsigned message, `sop detach-inband-signature-and-message` fails with `BAD_DATA`. If the input DATA contains more than one clearsigned message, `sop detach-inband-signature-and-message` also fails with `BAD_DATA`. A `sop` implementation MAY accept (and discard) leading and trailing data around the inline PGP clearsigned message.

If the file designated by `--signatures-out` already exists in the filesystem, `sop detach-inband-signature-and-message` will fail with `OUTPUT_EXISTS`.

Note that `--no-armor` here governs the data written to the `--signatures-out` destination. Standard output is always the raw message, not an OpenPGP packet.

Example:

```
$ sop detach-inband-signature-and-message --signatures-out=Release.pgp < InRelease >Release
$ sop verify Release.pgp archive-keyring.pgp < Release
$
```

4. Input String Types

Some material is passed to `sop` directly as a string on the command line.

4.1. DATE

An ISO-8601 formatted timestamp with time zone, or the special value now to indicate the current system time.

Examples:

```
now
2019-10-29T12:11:04+00:00
2019-10-24T23:48:29Z
20191029T121104Z
```

In some cases where used to specify lower and upper boundaries, a DATE value can be set to - to indicate "no time limit".

A flexible implementation of sop MAY accept date inputs in other unambiguous forms.

Note that whenever sop emits a timestamp (e.g. in Section 5.9) it MUST produce only a UTC-based ISO-8601 compliant representation.

4.2. USERID

This is an arbitrary UTF-8 string (Section 8.7). By convention, most User IDs are of the form Display Name <email.address@example.com>, but they do not need to be.

5. Input/Output Indirect Types

Some material is passed to sop indirectly, typically by referring to a filename containing the data in question. This type of data may also be passed to sop on Standard Input, or delivered by sop to Standard Output.

If any input data is specified explicitly to be read from a file that does not exist, sop will fail with MISSING_INPUT.

If any input data does not meet the requirements described below, sop will fail with BAD_DATA.

5.1. Special Designators for Indirect Types

An indirect argument or parameter that starts with "@" (COMMERCIAL AT, U+0040) is not treated as a filename, but is reserved for special handling, based on the prefix that follows the @. We describe two of those prefixes (@ENV: and @FD:) here. A sop implementation that receives such a special designator but does not know how to handle a given prefix in that context MUST fail with UNSUPPORTED_SPECIAL_PREFIX.

If the filename for any indirect material used as input has the special form @ENV:xxx, then contents of environment variable \$xxx is used instead of looking in the filesystem. @ENV is for input only: if the prefix @ENV: is used for any output argument, sop fails with UNSUPPORTED_SPECIAL_PREFIX.

If the filename for any indirect material used as either input or output has the special form @FD:nnn where nnn is a decimal integer, then the associated data is read from file descriptor nnn.

See Section 8.9 for more details about safe handling of these special designators.

5.2. CERTS

One or more OpenPGP certificates (section 11.1 of [I-D.ietf-openpgp-rfc4880bis]), aka "Transferable Public Key". May be armored (see Section 8.4).

Although some existing workflows may prefer to use one CERTS object with multiple certificates in it (a "keyring"), supplying exactly one certificate per CERTS input will make error reporting clearer and easier.

5.3. KEYS

One or more OpenPGP Transferable Secret Keys (section 11.2 of [I-D.ietf-openpgp-rfc4880bis]). May be armored (see Section 8.4).

Secret key material should be in cleartext (that is, it should not be locked with a password). If any secret key material is locked with a password, sop may fail with error KEY_IS_PROTECTED.

Although some existing workflows may prefer to use one KEYS object with multiple keys in it (a "secret keyring"), supplying exactly one key per KEYS input will make error reporting clearer and easier.

5.4. CIPHERTEXT

sop accepts only a restricted subset of the arbitrarily-nested grammar allowed by the OpenPGP Messages definition (section 11.3 of [I-D.ietf-openpgp-rfc4880bis]).

In particular, it accepts and generates only:

An OpenPGP message, consisting of a sequence of PKESKs (section 5.1 of [I-D.ietf-openpgp-rfc4880bis]) and SKESKs (section 5.3 of [I-D.ietf-openpgp-rfc4880bis]), followed by one SEIPD (section 5.14 of [I-D.ietf-openpgp-rfc4880bis]).

The SEIPD can decrypt into one of two things:

- * "Maybe Signed Data" (see below), or
- * Compressed data packet that contains "Maybe Signed Data"

"Maybe Signed Data" is a sequence of:

- * N (zero or more) one-pass signature packets, followed by
- * zero or more signature packets, followed by
- * one Literal data packet, followed by
- * N signature packets (corresponding to the outer one-pass signatures packets)

FIXME: does any tool do compression inside signing? Do we need to handle that?

May be armored (see Section 8.4).

5.5. SIGNATURES

One or more OpenPGP Signature packets. May be armored (see Section 8.4).

5.6. SESSIONKEY

This documentation uses the GnuPG defacto ASCII representation:

ALGONUM:HEXKEY

where ALGONUM is the decimal value associated with the OpenPGP Symmetric Key Algorithms (section 9.3 of [I-D.ietf-openpgp-rfc4880bis]) and HEXKEY is the hexadecimal representation of the binary key.

Example AES-256 session key:

9:FCA4BEAF687F48059CACC14FB019125CD57392BAB7037C707835925CBF9F7BCD

5.7. MICALG

This output indicates the cryptographic digest used when making a signature. It is useful specifically when generating signed PGP/MIME objects, which want a micalg= parameter for the multipart/signed content type as described in section 5 of [RFC3156].

It will typically be a string like pgp-sha512, but in some situations (multiple signatures using different digests) it will be the empty string. If the user of sop is assembling a PGP/MIME signed object, and the MICALG output is the empty string, the user should omit the micalg= parameter entirely.

5.8. PASSWORD

This is expected to be a UTF-8 string (Section 8.7), but for sop decrypt, any bytestring that the user supplies will be accepted. Note the details in sop encrypt and sop decrypt about trailing whitespace!

See also Section 8.8 for more discussion.

5.9. VERIFICATIONS

One line per successful signature verification. Each line has three structured fields delimited by a single space, followed by arbitrary text to the end of the line that forms a message describing the verification.

- * ISO-8601 UTC datestamp
- * Fingerprint of the signing key (may be a subkey)
- * Fingerprint of primary key of signing certificate (if signed by primary key, same as the previous field)
- * message describing the verification (free form)

Note that while Section 4.1 permits a sop implementation to accept other unambiguous date representations, its date output here MUST be a strict ISO-8601 UTC date timestamp. In particular:

- * the date and time fields MUST be separated by T, not by whitespace, since whitespace is used as a delimiter
- * the time MUST be emitted in UTC, with the explicit suffix Z

Example:

```
2019-10-24T23:48:29Z C90E6D36200A1B922A1509E77618196529AE5FF8 C4BC2DDB38CCE96485E
BE9C2F20691179038E5C6 certificate from dkg.asc
```

5.10. DATA

Cleartext, arbitrary data. This is either a bytestream or UTF-8 text.

It MUST only be UTF-8 text in the case of input supplied to sop sign --as=text or sop encrypt --as={mime|text}. If sop receives DATA containing non-UTF-8 octets in this case, it will fail (see Section 8.7) with EXPECTED_TEXT.

6. Failure Modes

sop return codes have both mnemonics and numeric values.

When sop succeeds, it will return 0 (OK) and emit nothing to Standard Error. When sop fails, it fails with a non-zero return code, and emits one or more warning messages on Standard Error. Known return codes include:

Value	Mnemonic	Meaning
0	OK	Success
3	NO_SIGNATURE	No acceptable signatures found (sop verify)
13	UNSUPPORTED_ASYMMETRIC_ALGO	Asymmetric algorithm unsupported (sop encrypt)
17	CERT_CANNOT_ENCRYPT	Certificate not encryption-capable (e.g., expired, revoked, unacceptable usage)

		flags) (sop encrypt)
19	MISSING_ARG	Missing required argument
23	INCOMPLETE_VERIFICATION	Incomplete verification instructions (sop decrypt)
29	CANNOT_DECRYPT	Unable to decrypt (sop decrypt)
31	PASSWORD_NOT_HUMAN_READABLE	Non-UTF-8 or otherwise unreliable password (sop encrypt)
37	UNSUPPORTED_OPTION	Unsupported option
41	BAD_DATA	Invalid data type (no secret key where KEYS expected, etc)
53	EXPECTED_TEXT	Non-text input where text expected
59	OUTPUT_EXISTS	Output file already exists
61	MISSING_INPUT	Input file does not exist
67	KEY_IS_PROTECTED	A KEYS input is protected (locked) with a password, and sop cannot unlock it
69	UNSUPPORTED_SUBCOMMAND	Unsupported subcommand
71	UNSUPPORTED_SPECIAL_PREFIX	An indirect parameter is a special designator (it starts with @) but sop does not know how to handle the prefix
73	AMBIGUOUS_INPUT	A indirect input parameter is a special designator (it starts with @), and a filename

		matching the designator is actually present
79	KEY_CANNOT_SIGN	Key not signature- capable (e.g., expired, revoked, unacceptable usage flags) (sop sign and sop encrypt with -- sign-with)

Table 1

If a sop implementation fails in some way not contemplated by this document, it MAY return any non-zero error code, not only those listed above.

7. Alternate Interfaces

This draft primarily defines a command line interface, but future versions may try to outline a comparable idiomatic interface for C or some other widely-used programming language.

Comparable idiomatic interfaces are already active in the wild for different programming languages, in particular:

- * Rust: [RUST-SOP]
- * Java: [SOP-JAVA]
- * Python: [PYTHON-SOP]

These programmatic interfaces are typically coupled with a wrapper that can automatically generate a command-line tool compatible with this draft.

An implementation that uses one of these languages should target the corresponding idiomatic interface for ease of development and interoperability.

8. Guidance for Implementers

sop uses a few assumptions that implementers might want to consider.

8.1. One OpenPGP Message at a Time

sop is intended to be a simple tool that operates on one OpenPGP object at a time. It should be composable, if you want to use it to deal with multiple OpenPGP objects.

FIXME: discuss what this means for streaming. The stdio interface doesn't necessarily imply streamed output.

8.2. Simplified Subset of OpenPGP Message

While the formal grammar for OpenPGP Message is arbitrarily nestable, sop constrains itself to what it sees as a single "layer" (see Section 5.4).

This is a deliberate choice, because it is what most consumers expect. Also, if an arbitrarily-nested structure is parsed with a recursive algorithm, this risks a denial of service vulnerability. sop intends to be implementable with a parser that defensively declines to do recursive descent into an OpenPGP Message.

Note that an implementation of sop decrypt MAY choose to handle more complex structures, but if it does, it should document the other structures it handles and why it chooses to do so. We can use such documentation to improve future versions of this spec.

8.3. Validate Signatures Only from Known Signers

There are generally only a few signers who are relevant for a given OpenPGP message. When verifying signatures, sop expects that the caller can identify those relevant signers ahead of time.

8.4. OpenPGP inputs can be either Binary or ASCII-armored

OpenPGP material on input can be in either ASCII-armored or binary form. This is a deliberate choice because there are typical scenarios where the program can't predict which form will appear. Expecting the caller of sop to detect the form and adjust accordingly seems both redundant and error-prone.

The simple way to detect possible ASCII-armoring is to see whether the high bit of the first octet is set: section 4.2 of [RFC4880] indicates that bit 7 is always one in the first octet of an OpenPGP packet. In standard ASCII-armor, the first character is "-" (HYPHEN-MINUS, U+002D), so the high bit should be cleared.

When considering an input as ASCII-armored OpenPGP material, sop MAY reject an input based on any of the following variations (see section 6.2 of [RFC4880] for precise definitions):

- * An unknown Armor Header Line
- * Any text before the Armor Header Line
- * Malformed lines in the Armor Headers section
- * Any non-whitespace data after the Armor Tail
- * Any Radix-64 encoded line with more than 76 characters
- * Invalid characters in the Radix-64-encoded data
- * An invalid Armor Checksum
- * A mismatch between the Armor Header Line and the Armor Tail

For robustness, sop SHOULD be willing to ignore whitespace after the Armor Tail.

When considering OpenPGP material as input, regardless of whether it is ASCII-armored or binary, sop SHOULD reject any material that doesn't produce a valid stream of OpenPGP packets. For example, sop SHOULD raise an error if an OpenPGP packet header is malformed, or if there is trailing garbage after the end of a packet.

For a given type of OpenPGP input material (i.e., SIGNATURES, CERTS, KEYS, or CIPHERTEXT), sop SHOULD also reject any input that does not conform to the expected packet stream. See Section 5 for the expected packet stream for different types.

8.5. Detached Signatures

sop deals with detached signatures as the baseline form of OpenPGP signatures.

The primary alternative to detached signatures is inline signatures, but handling an inline signature requires parsing to delimit the multiple parts of the document, including at least:

- * any preamble before the message
- * the inline message header (delimiter line, OpenPGP headers)
- * the message itself

- * the divider between the message and the signature (including any OpenPGP headers there)
- * the signature
- * the divider that terminates the signature
- * any suffix after the signature

Note also that the preamble or the suffix might be arbitrary text, and might themselves contain OpenPGP messages (whether signatures or otherwise).

If the parser that does this split differs in any way from the parser that does the verification, or parts of the message are confused, it would be possible to produce a verification status and an actual signed message that don't correspond to one another.

Blurred boundary problems like this can produce ugly attacks similar to those found in [EFAIL].

8.6. Reliance on Supplied Certs and Keys

A truly stateless implementation may find that it spends more time validating the internal consistency of certificates and keys than it does on the actual object security operations.

For performance reasons, an implementation may choose to ignore validation on certificate and key material supplied to it. The security implications of doing so depend on how the certs and keys are managed outside of sop.

8.7. Text is always UTF-8

Various places in this specification require UTF-8 [RFC3629] when encoding text. sop implementations SHOULD NOT consider textual data in any other character encoding.

OpenPGP Implementations MUST already handle UTF-8, because various parts of [RFC4880] require it, including:

- * User ID
- * Notation name
- * Reason for revocation
- * ASCII-armor Comment: header

Dealing with messages in other charsets leads to weird security failures like [Charset-Switching], especially when the charset indication is not covered by any sort of cryptographic integrity check. Restricting textual data to UTF-8 universally across the OpenPGP ecosystem eliminates any such risk without losing functionality, since UTF-8 can encode all known characters.

8.8. Passwords are Human-Readable

Passwords are generally expected to be human-readable, as they are typically recorded and transmitted as human-visible, human-transferable strings. However, they are used in the OpenPGP protocol as bytestrings, so ensuring that there is a reliable bidirectional mapping between strings and bytes. The maximally robust behavior here is for `sop encrypt` to constrain the choice of passwords to strings that have such a mapping, and for `sop decrypt` to try multiple plausible versions of any supplied `PASSWORD`.

When generating material based on a password, `sop encrypt` enforces that the password is actually meaningfully human-transferable (requiring UTF-8, trimming trailing whitespace). Some `sop encrypt` implementations may make even more strict requirements on input to ensure that they are transferable between humans in a robust way.

For example, a more strict `sop encrypt` MAY also:

- * forbid leading whitespace
- * forbid non-printing characters other than SPACE (U+0020), such as ZERO WIDTH NON-JOINER (U+200C) or TAB (U+0009)
- * require the password to be in Unicode Normal Form C ([UNICODE-NORMALIZATION])

Violations of these more-strict policies SHOULD result in an error of `PASSWORD_NOT_HUMAN_READABLE`.

A `sop encrypt` implementation typically SHOULD NOT attempt enforce a minimum "password strength", but in the event that some implementation does, it MUST NOT represent a weak password with `PASSWORD_NOT_HUMAN_READABLE`.

When `sop decrypt` receives a `PASSWORD` input, it sees it as a bytestring. If the bytestring fails to work as a password, but ends in UTF-8 whitespace, it will try again with the trailing whitespace removed. This handles a common pattern of using a file with a final newline, for example. The pattern here is one of robustness in the face of typical errors in human-transferred textual data.

A more robust sop decrypt implementation that finds neither of the above two attempts work for a given PASSWORD MAY try additional variations if they produce a different bytestring, such as:

- * trimming any leading whitespace, if discovered
- * trimming any internal non-printable characters other than SPACE (U+0020)
- * converting the supplied PASSWORD into Unicode Normal Form C ([UNICODE-NORMALIZATION])

A sop decrypt implementation that stages multiple decryption attempts like this SHOULD consider the computational resources consumed by each attempt, to avoid presenting an attack surface for resource exhaustion in the face of a non-standard PASSWORD input.

8.9. Be careful with Special Designators

As documented in Section 5.1, special designators for indirect inputs like @ENV: and @FD: (and indirect outputs using @FD:) warrant some special/cautious handling.

For one thing, it's conceivable that the filesystem could contain a file with these literal names. If sop receives an indirect output parameter that starts with an "@" (COMMERCIAL AT, U+0040) it MUST NOT write to the filesystem for that parameter. A sop implementation that receives such a parameter as input MAY test for the presence of such a file in the filesystem and fail with AMBIGUOUS_INPUT to warn the user of the ambiguity and possible confusion.

These special designators are likely to be used to pass sensitive data (like secret key material or passwords) so that it doesn't need to touch the filesystem. Given this sensitivity, sop should be careful with such an input, and minimize its leakage to other processes. In particular, sop SHOULD NOT leak any environment variable identified by @ENV: or file descriptor identified by @FD: to any subprocess unless the subprocess specifically needs access to that data.

9. Guidance for Consumers

While sop is originally conceived of as an interface for interoperability testing, it's conceivable that an application that uses OpenPGP for object security would want to use it.

FIXME: more guidance for how to use such a tool safely and efficiently goes here.

FIXME: if an encrypted OpenPGP message arrives without metadata, it is difficult to know which signers to consider when decrypting. How do we do this efficiently without invoking `sop decrypt` twice, once without `--verify-*` and again with the expected identity material?

9.1. Choosing between `-as=text` and `-as=binary`

A program that invokes `sop` to generate an OpenPGP signature typically needs to decide whether it is making a text or binary signature.

By default, `sop` will make a binary signature. The caller of `sop sign` should choose `--as=text` only when it knows that: - the data being signed is in fact textual, and encoded in UTF-8, and - the signed data might be transmitted to the recipient (the verifier of the signature) over a channel that has the propensity to transform line-endings.

Examples of such channels include FTP ([RFC0959]) and SMTP ([RFC5321]).

9.2. Special Designators and Unusual Filenames

In some cases, a user of `sop` might want to pass all the files in a given directory as positional parameters (e.g., a list of CERTS files to test a signature against).

If one of the files has a name that starts with `--`, it might be confused by `sop` for an option. If one of the files has a name that starts with `@`, it might be confused by `sop` as a special designator (Section 5.1).

If the user wants to deliberately refer to such an ambiguously-named file in the filesystem, they should prefix the filename with `./` or use an absolute path.

Any specific `@FD:` special designator SHOULD NOT be supplied more than once to an invocation of `sop`. If a `sop` invocation sees multiple copies of a specific `@FD:n` input (e.g., `sop sign @FD:3 @FD:3`), it MAY fail with `MISSING_INPUT` even if file descriptor 3 contains a valid `KEYS`, because the bytestream for the `KEYS` was consumed by the first argument. Doubling up on the same `@FD:` for output (e.g., `sop decrypt --session-key-out=@FD:3 --verify-out=@FD:3`) also results in an ambiguous data stream.

10. Security Considerations

The OpenPGP object security model is typically used for confidentiality and authenticity purposes.

10.1. Signature Verification

In many contexts, an OpenPGP signature is verified to prove the origin and integrity of an underlying object.

When `sop` checks a signature (e.g. via `sop verify` or `sop decrypt --verify-with`), it **MUST NOT** consider it to be verified unless all of these conditions are met:

- * The signature must be made by a signing-capable public key that is present in one of the supplied certificates
- * The certificate and signing subkey must have been created before or at the signature time
- * The certificate and signing subkey must not have been expired at the signature time
- * The certificate and signing subkey must not be revoked with a "hard" revocation
- * If the certificate or signing subkey is revoked with a "soft" revocation, then the signature time must predate the revocation
- * The signing subkey must be properly bound to the primary key, and cross-signed
- * The signature (and any dependent signature, such as the cross-sig or subkey binding signatures) must be made with strong cryptographic algorithms (e.g., not MD5 or a 1024-bit RSA key)

Implementers **MAY** also consider other factors in addition to the origin and authenticity, including application-specific information.

For example, consider the application domain of checking software updates. If software package Foo version 13.3.2 was signed on 2019-10-04, and the user receives a copy of Foo version 12.4.8 that was signed on 2019-10-16, it may be authentic and have a more recent signature date. But it is not an upgrade ($12.4.8 < 13.3.2$), and therefore it should not be applied automatically.

In such cases, it is critical that the application confirms that the other information verified is also protected by the relevant OpenPGP signature.

Signature validity is a complex topic (see for example the discussion at [DISPLAYING-SIGNATURES]), and this documentation cannot list all possible details.

10.2. Compression

The interface as currently specified does not allow for control of compression. Compressing and encrypting data that may contain both attacker-supplied material and sensitive material could leak information about the sensitive material (see the CRIME attack).

Unless an application knows for sure that no attacker-supplied material is present in the input, it should not compress during encryption.

11. Privacy Considerations

Material produced by `sop encrypt` may be placed on an untrusted machine (e.g., sent through the public SMTP network). That material may contain metadata that leaks associational information (e.g., recipient identifiers in PKESK packets (section 5.1 of [I-D.ietf-openpgp-rfc4880bis])). **FIXME:** document things like PURBs and `--hidden-recipient`)

11.1. Object Security vs. Transport Security

OpenPGP offers an object security model, but says little to nothing about how the secured objects get to the relevant parties.

When sending or receiving OpenPGP material, the implementer should consider what privacy leakage is implicit with the transport.

12. Document Considerations

[RFC Editor: please remove this section before publication]

This document is currently edited as markdown. Minor editorial changes can be suggested via merge requests at <https://gitlab.com/dkg/openpgp-stateless-cli> or by e-mail to the authors. Please direct all significant commentary to the public IETF OpenPGP mailing list: openpgp@ietf.org

12.1. Document History

substantive changes between -02 and -03:

- * Added `--micalg-out` parameter to sign
- * Change from KEY to KEYS (permit multiple secret keys in each blob)
- * New error code: KEY_CANNOT_SIGN

- * version now has `--backend` and `--extended` options
- substantive changes between -01 and -02:
- * Added mnemonics for return codes
 - * `decrypt` should fail when asked to output to a pre-existing file
 - * Removed superfluous `--armor` option
 - * Much more specific about what armor `--label=auto` should do
 - * `armor` and `dearmor` are now fully idempotent, but work only well-formed OpenPGP streams
 - * Dropped armor `--allow-nested`
 - * Specified what `encrypt --as=` means
 - * New error code: `KEY_IS_PROTECTED`
 - * Documented expectations around human-readable, human-transferable passwords
 - * New subcommand: `detach-inband-signature-and-message`
 - * More specific guidance about special designators like `@FD:` and `@ENV:`, including new error codes `UNSUPPORTED_SPECIAL_PREFIX` and `AMBIGUOUS_INPUT`

substantive changes between -00 and -01:

- * Changed `generate` subcommand to `generate-key`
- * Changed `convert` subcommand to `extract-cert`
- * Added "Input String Types" section as distinct from indirect I/O
- * Made implicit arguments potentially explicit (e.g. `sop armor --label=auto`)
- * Added `--allow-nested` to `sop armor` to make it idempotent by default
- * Added fingerprint of signing (sub)key to `VERIFICATIONS` output
- * Dropped `--mode` and `--session-key` arguments for `sop encrypt` (no plausible use, not needed for interop)

- * Added `--with-session-key` argument to `sop decrypt` to allow for session-key-based decryption
- * Added examples to each subcommand
- * More detailed error codes for `sop encrypt`
- * Move from CERT to CERTS (each CERTS argument might contain multiple certificates)

12.2. Future Work

- * certificate transformation into popular publication forms:
 - WKD
 - DANE OPENPGPKEY
 - Autocrypt
- * `sop encrypt` - specify compression? (see Section 10.2)
- * `sop encrypt` - specify padding policy/mechanism?
- * `sop decrypt` - how can it more safely handle zip bombs?
- * `sop decrypt` - what should it do when encountering weakly-encrypted (or unencrypted) input?
- * `sop encrypt` - minimize metadata (e.g. `--throw-keyids`)?
- * handling secret keys that are locked with passwords?
- * specify an error if a DATE arrives as input without a time zone?
- * add considerations about what it means for armored CERTS to contain multiple certificates - multiple armorings? one big blob?
- * do we need an interface or option (for performance?) with the semantics that `sop` doesn't validate certificates internally, it just accepts whatever's given as legit data? (see Section 8.6)
- * do we need to be able to assemble a clearsigned message? I'd rather not, given the additional complications.

- * does detach-inband-signature-and-message need to be able to split an OpenPGP signed message that `_isn't_` using the clearsigned framework (e.g., the output of `gpg --sign`, in addition to handling `gpg --clearsign`)?

13. Acknowledgements

This work was inspired by Justus Winter's [OpenPGP-Interoperability-Test-Suite].

The following people contributed helpful feedback and considerations to this draft, but are not responsible for its problems:

- * Allan Nordhoej
- * Antoine Beaupre
- * Edwin Taylor
- * Jameson Rollins
- * Justus Winter
- * Paul Schaub
- * Vincent Breitmoser

14. References

14.1. Normative References

- [I-D.ietf-openpgp-rfc4880bis] Koch, W., carlson, B. M., Tse, R. H., Atkins, D., and D. K. Gillmor, "OpenPGP Message Format", Work in Progress, Internet-Draft, draft-ietf-openpgp-rfc4880bis-10, 31 August 2020, <<https://www.ietf.org/archive/id/draft-ietf-openpgp-rfc4880bis-10.txt>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC3156] Elkins, M., Del Torto, D., Levien, R., and T. Roessler, "MIME Security with OpenPGP", RFC 3156, DOI 10.17487/RFC3156, August 2001, <<https://www.rfc-editor.org/info/rfc3156>>.

- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <<https://www.rfc-editor.org/info/rfc3629>>.
- [RFC4880] Callas, J., Donnerhackle, L., Finney, H., Shaw, D., and R. Thayer, "OpenPGP Message Format", RFC 4880, DOI 10.17487/RFC4880, November 2007, <<https://www.rfc-editor.org/info/rfc4880>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.

14.2. Informative References

- [Charset-Switching]
Gillmor, D.K., "Inline PGP Considered Harmful", 24 February 2014, <<https://dkg.fifthhorseman.net/notes/inline-pgp-harmful/>>.
- [DISPLAYING-SIGNATURES]
Brunschwig, P., "On Displaying Signatures", n.d., <https://admin.hostpoint.ch/pipermail/enigmail-users_enigmail.net/2017-November/004683.html>.
- [EFAIL] Poddebniak, D. and C. Dresen, "Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels", n.d., <<https://efail.de>>.
- [I-D.draft-bre-openpgp-samples-01]
Einarsson, B. R., "juga", and D. K. Gillmor, "OpenPGP Example Keys and Certificates", Work in Progress, Internet-Draft, draft-bre-openpgp-samples-01, 20 December 2019, <<https://www.ietf.org/archive/id/draft-bre-openpgp-samples-01.txt>>.
- [OpenPGP-Interoperability-Test-Suite]
"OpenPGP Interoperability Test Suite", 25 October 2021, <<https://tests.sequoia-pgp.org/>>.
- [PYTHON-SOP]
Gillmor, D., "SOP for python", n.d., <<https://pypi.org/project/sop/>>.
- [RFC0959] Postel, J. and J. Reynolds, "File Transfer Protocol", STD 9, RFC 959, DOI 10.17487/RFC0959, October 1985, <<https://www.rfc-editor.org/info/rfc959>>.

- [RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996, <<https://www.rfc-editor.org/info/rfc2045>>.
- [RFC5321] Klensin, J., "Simple Mail Transfer Protocol", RFC 5321, DOI 10.17487/RFC5321, October 2008, <<https://www.rfc-editor.org/info/rfc5321>>.
- [RUST-SOP] Winter, J., "A Rust implementation of the Stateless OpenPGP Protocol", n.d., <<https://sequoia-pgp.gitlab.io/sop-rs/>>.
- [SEMMVER] Preston-Werner, T., "Semantic Versioning 2.0.0", 18 June 2013, <<https://semver.org/>>.
- [SOP-JAVA] Schaub, P., "Stateless OpenPGP Protocol for Java.", n.d., <<https://github.com/pgpainless/pgpainless/tree/master/sop-java>>.
- [UNICODE-NORMALIZATION] Whistler, K., "Unicode Normalization Forms", 4 February 2019, <<https://unicode.org/reports/tr15/>>.

Author's Address

Daniel Kahn Gillmor
American Civil Liberties Union
125 Broad St.
New York, NY, 10004
United States of America

Email: dkg@fifthhorseman.net

Network Working Group
Internet-Draft
Obsoletes: 4880, 5581, 6637 (if approved)
Intended status: Standards Track
Expires: 8 September 2022

W. Koch, Ed.
GnuPG e.V.
P. Wouters, Ed.
Aiven
7 March 2022

OpenPGP Message Format
draft-ietf-openpgp-crypto-refresh-05

Abstract

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

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

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

Status of This Memo

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

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

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

This Internet-Draft will expire on 8 September 2022.

Copyright Notice

Copyright (c) 2022 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Revised BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Revised BSD License.

Table of Contents

1. Introduction	7
1.1. Terms	7
2. General functions	8
2.1. Confidentiality via Encryption	8
2.2. Authentication via Digital Signature	9
2.3. Compression	10
2.4. Conversion to Radix-64	10
2.5. Signature-Only Applications	10
3. Data Element Formats	10
3.1. Scalar Numbers	10
3.2. Multiprecision Integers	11
3.2.1. Using MPIs to encode other data	11
3.3. Key IDs	11
3.4. Text	12
3.5. Time Fields	12
3.6. Keyrings	12
3.7. String-to-Key (S2K) Specifiers	12
3.7.1. String-to-Key (S2K) Specifier Types	12
3.7.1.1. Simple S2K	13
3.7.1.2. Salted S2K	14
3.7.1.3. Iterated and Salted S2K	14
3.7.1.4. Argon2	15
3.7.2. String-to-Key Usage	16
3.7.2.1. Secret-Key Encryption	16
3.7.2.2. Symmetric-Key Message Encryption	17
4. Packet Syntax	18
4.1. Overview	18
4.2. Packet Headers	18
4.2.1. OpenPGP Format Packet Lengths	19
4.2.1.1. One-Octet Lengths	20
4.2.1.2. Two-Octet Lengths	20
4.2.1.3. Five-Octet Lengths	20

4.2.1.4.	Partial Body Lengths	20
4.2.2.	Legacy Format Packet Lengths	21
4.2.3.	Packet Length Examples	21
4.3.	Packet Tags	22
5.	Packet Types	23
5.1.	Public-Key Encrypted Session Key Packets (Tag 1)	23
5.1.1.	v3 PKESK	23
5.1.2.	v5 PKESK	24
5.1.3.	Algorithm Specific Fields for RSA encryption	25
5.1.4.	Algorithm Specific Fields for Elgamal encryption	25
5.1.5.	Algorithm-Specific Fields for ECDH encryption	25
5.1.6.	Notes on PKESK	25
5.2.	Signature Packet (Tag 2)	26
5.2.1.	Signature Types	26
5.2.2.	Version 3 Signature Packet Format	28
5.2.3.	Version 4 and 5 Signature Packet Formats	31
5.2.3.1.	Algorithm-Specific Fields for RSA signatures	32
5.2.3.2.	Algorithm-Specific Fields for DSA or ECDSA signatures	32
5.2.3.3.	Algorithm-Specific Fields for EdDSA signatures	32
5.2.3.4.	Notes on Signatures	33
5.2.3.5.	Signature Subpacket Specification	34
5.2.3.6.	Signature Subpacket Types	37
5.2.3.7.	Notes on Self-Signatures	37
5.2.3.8.	Signature Creation Time	38
5.2.3.9.	Issuer	38
5.2.3.10.	Key Expiration Time	38
5.2.3.11.	Preferred Symmetric Ciphers for v1 SEIPD	39
5.2.3.12.	Preferred AEAD Ciphersuites	39
5.2.3.13.	Preferred Hash Algorithms	40
5.2.3.14.	Preferred Compression Algorithms	40
5.2.3.15.	Signature Expiration Time	40
5.2.3.16.	Exportable Certification	40
5.2.3.17.	Revocable	41
5.2.3.18.	Trust Signature	41
5.2.3.19.	Regular Expression	42
5.2.3.20.	Revocation Key	42
5.2.3.21.	Notation Data	43
5.2.3.22.	Key Server Preferences	44
5.2.3.23.	Preferred Key Server	44
5.2.3.24.	Primary User ID	45
5.2.3.25.	Policy URI	45
5.2.3.26.	Key Flags	45
5.2.3.27.	Signer's User ID	47
5.2.3.28.	Reason for Revocation	47
5.2.3.29.	Features	49
5.2.3.30.	Signature Target	50
5.2.3.31.	Embedded Signature	50

5.2.3.32. Issuer Fingerprint	50
5.2.3.33. Intended Recipient Fingerprint	50
5.2.4. Computing Signatures	51
5.2.4.1. Subpacket Hints	52
5.3. Symmetric-Key Encrypted Session Key Packets (Tag 3)	53
5.3.1. v4 SKESK	53
5.3.2. v5 SKESK	54
5.4. One-Pass Signature Packets (Tag 4)	55
5.5. Key Material Packet	56
5.5.1. Key Packet Variants	56
5.5.1.1. Public-Key Packet (Tag 6)	56
5.5.1.2. Public-Subkey Packet (Tag 14)	57
5.5.1.3. Secret-Key Packet (Tag 5)	57
5.5.1.4. Secret-Subkey Packet (Tag 7)	57
5.5.2. Public-Key Packet Formats	57
5.5.3. Secret-Key Packet Formats	59
5.6. Algorithm-specific Parts of Keys	61
5.6.1. Algorithm-Specific Part for RSA Keys	62
5.6.2. Algorithm-Specific Part for DSA Keys	62
5.6.3. Algorithm-Specific Part for Elgamal Keys	62
5.6.4. Algorithm-Specific Part for ECDSA Keys	63
5.6.5. Algorithm-Specific Part for EdDSA Keys	63
5.6.6. Algorithm-Specific Part for ECDH Keys	63
5.6.6.1. ECDH Secret Key Material	64
5.7. Compressed Data Packet (Tag 8)	65
5.8. Symmetrically Encrypted Data Packet (Tag 9)	66
5.9. Marker Packet (Tag 10)	67
5.10. Literal Data Packet (Tag 11)	67
5.10.1. Special Filename _CONSOLE (Deprecated)	69
5.11. Trust Packet (Tag 12)	69
5.12. User ID Packet (Tag 13)	70
5.13. User Attribute Packet (Tag 17)	70
5.13.1. The Image Attribute Subpacket	71
5.14. Sym. Encrypted Integrity Protected Data Packet (Tag 18)	71
5.14.1. Version 1 Sym. Encrypted Integrity Protected Data Packet Format	72
5.14.2. Version 2 Sym. Encrypted Integrity Protected Data Packet Format	74
5.14.3. EAX Mode	76
5.14.4. OCB Mode	76
5.14.5. GCM Mode	76
5.15. Padding Packet (Tag 21)	76
6. Radix-64 Conversions	77
6.1. An Implementation of the CRC-24 in "C"	78
6.2. Forming ASCII Armor	78
6.3. Encoding Binary in Radix-64	81
6.4. Decoding Radix-64	83

6.5.	Examples of Radix-64	83
6.6.	Example of an ASCII Armored Message	84
7.	Cleartext Signature Framework	84
7.1.	Dash-Escaped Text	85
8.	Regular Expressions	86
9.	Constants	87
9.1.	Public-Key Algorithms	87
9.2.	ECC Curves for OpenPGP	89
9.2.1.	Curve-Specific Wire Formats	91
9.3.	Symmetric-Key Algorithms	92
9.4.	Compression Algorithms	93
9.5.	Hash Algorithms	93
9.6.	AEAD Algorithms	94
10.	IANA Considerations	95
10.1.	New String-to-Key Specifier Types	95
10.2.	New Packets	95
10.2.1.	User Attribute Types	96
10.2.1.1.	Image Format Subpacket Types	96
10.2.2.	New Signature Subpackets	96
10.2.2.1.	Signature Notation Data Subpackets	96
10.2.2.2.	Signature Notation Data Subpacket Notation Flags	97
10.2.2.3.	Key Server Preference Extensions	97
10.2.2.4.	Key Flags Extensions	97
10.2.2.5.	Reason for Revocation Extensions	97
10.2.2.6.	Implementation Features	97
10.2.3.	New Packet Versions	98
10.3.	New Algorithms	98
10.3.1.	Public-Key Algorithms	98
10.3.2.	Symmetric-Key Algorithms	99
10.3.3.	Hash Algorithms	99
10.3.4.	Compression Algorithms	100
10.3.5.	Elliptic Curve Algorithms	100
10.4.	Elliptic Curve Point and Scalar Wire Formats	100
10.5.	Changes to existing registries	101
11.	Packet Composition	101
11.1.	Transferable Public Keys	101
11.2.	Transferable Secret Keys	103
11.3.	OpenPGP Messages	103
11.3.1.	Unwrapping Encrypted and Compressed Messages	104
11.3.2.	Additional Constraints on Packet Sequences	104
11.3.2.1.	Packet Versions in Encrypted Messages	105
11.4.	Detached Signatures	106
12.	Enhanced Key Formats	106
12.1.	Key Structures	106
12.2.	Key IDs and Fingerprints	107
13.	Elliptic Curve Cryptography	108
13.1.	Supported ECC Curves	109

13.2.	EC Point Wire Formats	109
13.2.1.	SEC1 EC Point Wire Format	109
13.2.2.	Prefixed Native EC Point Wire Format	110
13.2.3.	Notes on EC Point Wire Formats	110
13.3.	EC Scalar Wire Formats	110
13.3.1.	EC Octet String Wire Format	111
13.3.2.	Elliptic Curve Prefixed Octet String Wire Format . .	112
13.4.	Key Derivation Function	112
13.5.	EC DH Algorithm (ECDH)	113
14.	Notes on Algorithms	116
14.1.	PKCS#1 Encoding in OpenPGP	116
14.1.1.	EME-PKCS1-v1_5-ENCODE	116
14.1.2.	EME-PKCS1-v1_5-DECODE	117
14.1.3.	EMSA-PKCS1-v1_5	118
14.2.	Symmetric Algorithm Preferences	119
14.2.1.	Plaintext	119
14.3.	Other Algorithm Preferences	120
14.3.1.	Compression Preferences	120
14.3.1.1.	Uncompressed	120
14.3.2.	Hash Algorithm Preferences	120
14.4.	RSA	121
14.5.	DSA	121
14.6.	Elgamal	121
14.7.	EdDSA	122
14.8.	Reserved Algorithm Numbers	122
14.9.	OpenPGP CFB Mode	122
14.10.	Private or Experimental Parameters	124
14.11.	Meta-Considerations for Expansion	124
15.	Security Considerations	124
15.1.	Avoiding Ciphertext Malleability	128
15.2.	Escrowed Revocation Signatures	130
15.3.	Random Number Generation and Seeding	131
15.4.	Traffic Analysis	131
16.	Implementation Nits	132
17.	References	133
17.1.	Normative References	133
17.2.	Informative References	136
Appendix A.	Test vectors	138
A.1.	Sample EdDSA key	138
A.2.	Sample EdDSA signature	138
A.3.	Sample AEAD-EAX encryption and decryption	139
A.3.1.	Sample Parameters	139
A.3.2.	Sample symmetric-key encrypted session key packet (v5)	139
A.3.3.	Starting AEAD-EAX decryption of the session key . . .	140
A.3.4.	Sample v2 SEIPD packet	140
A.3.5.	Decryption of data	141
A.3.6.	Complete AEAD-EAX encrypted packet sequence	142

A.4.	Sample AEAD-OCB encryption and decryption	142
A.4.1.	Sample Parameters	142
A.4.2.	Sample symmetric-key encrypted session key packet (v5)	143
A.4.3.	Starting AEAD-EAX decryption of the session key . . .	143
A.4.4.	Sample v2 SEIPD packet	144
A.4.5.	Decryption of data	144
A.4.6.	Complete AEAD-EAX encrypted packet sequence	145
A.5.	Sample AEAD-GCM encryption and decryption	146
A.5.1.	Sample Parameters	146
A.5.2.	Sample symmetric-key encrypted session key packet (v5)	146
A.5.3.	Starting AEAD-EAX decryption of the session key . . .	146
A.5.4.	Sample v2 SEIPD packet	147
A.5.5.	Decryption of data	148
A.5.6.	Complete AEAD-EAX encrypted packet sequence	149
A.6.	Sample message encrypted using Argon2	149
Appendix B.	Acknowledgements	150
Appendix C.	Document Workflow	150
Authors' Addresses	150

1. Introduction

This document provides information on the message-exchange packet formats used by OpenPGP to provide encryption, decryption, signing, and key management functions. It is a revision of RFC 4880, "OpenPGP Message Format", which is a revision of RFC 2440, which itself replaces RFC 1991, "PGP Message Exchange Formats" [RFC1991] [RFC2440] [RFC4880].

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

1.1. Terms

- * OpenPGP - This is a term for security software that uses PGP 5 as a basis, formalized in this document.
- * PGP - Pretty Good Privacy. PGP is a family of software systems developed by Philip R. Zimmermann from which OpenPGP is based.
- * PGP 2 - This version of PGP has many variants; where necessary a more detailed version number is used here. PGP 2 uses only RSA, MD5, and IDEA for its cryptographic transforms. An informational RFC, RFC 1991, was written describing this version of PGP.

- * PGP 5 - This version of PGP is formerly known as "PGP 3" in the community. It has new formats and corrects a number of problems in the PGP 2 design. It is referred to here as PGP 5 because that software was the first release of the "PGP 3" code base.
- * GnuPG - GNU Privacy Guard, also called GPG. GnuPG is an OpenPGP implementation that avoids all encumbered algorithms. Consequently, early versions of GnuPG did not include RSA public keys.

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

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC2119] [RFC8174] when, and only when, they appear in all capitals, as shown here.

The key words "PRIVATE USE", "SPECIFICATION REQUIRED", and "RFC REQUIRED" that appear in this document when used to describe namespace allocation are to be interpreted as described in [RFC8126].

2. General functions

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

- * digital signatures
- * encryption
- * compression
- * Radix-64 conversion

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

2.1. Confidentiality via Encryption

OpenPGP combines symmetric-key encryption and public-key encryption to provide confidentiality. When made confidential, first the object is encrypted using a symmetric encryption algorithm. Each symmetric key is used only once, for a single object. A new "session key" is generated as a random number for each object (sometimes referred to as a session). Since it is used only once, the session key is bound

to the message and transmitted with it. To protect the key, it is encrypted with the receiver's public key. The sequence is as follows:

1. The sender creates a message.
2. The sending OpenPGP generates a random number to be used as a session key for this message only.
3. The session key is encrypted using each recipient's public key. These "encrypted session keys" start the message.
4. The sending OpenPGP encrypts the message using the session key, which forms the remainder of the message.
5. The receiving OpenPGP decrypts the session key using the recipient's private key.
6. The receiving OpenPGP decrypts the message using the session key. If the message was compressed, it will be decompressed.

With symmetric-key encryption, an object may be encrypted with a symmetric key derived from a passphrase (or other shared secret), or a two-stage mechanism similar to the public-key method described above in which a session key is itself encrypted with a symmetric algorithm keyed from a shared secret.

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

2.2. Authentication via Digital Signature

The digital signature uses a hash code or message digest algorithm, and a public-key signature algorithm. The sequence is as follows:

1. The sender creates a message.
2. The sending software generates a hash code of the message.
3. The sending software generates a signature from the hash code using the sender's private key.
4. The binary signature is attached to the message.

5. The receiving software keeps a copy of the message signature.
6. The receiving software generates a new hash code for the received message and verifies it using the message's signature. If the verification is successful, the message is accepted as authentic.

2.3. Compression

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

2.4. Conversion to Radix-64

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

Implementations SHOULD provide Radix-64 conversions.

2.5. Signature-Only Applications

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

3. Data Element Formats

This section describes the data elements used by OpenPGP.

3.1. Scalar Numbers

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

3.2. Multiprecision Integers

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

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

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

Examples:

(all numbers are in hexadecimal)

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

Additional rules:

The size of an MPI is $((\text{MPI.length} + 7) / 8) + 2$ octets.

The length field of an MPI describes the length starting from its most significant non-zero bit. Thus, the MPI [00 02 01] is not formed correctly. It should be [00 01 01].

Unused bits of an MPI MUST be zero.

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

3.2.1. Using MPIs to encode other data

Note that MPIs are used in some places used to encode non-integer data, such as an elliptic curve point (see Section 13.2, or an octet string of known, fixed length (see Section 13.3). The wire representation is the same: two octets of length in bits counted from the first non-zero bit, followed by the smallest series of octets that can represent the value while stripping off any leading zero octets.

3.3. Key IDs

A Key ID is an eight-octet scalar that identifies a key. Implementations SHOULD NOT assume that Key IDs are unique. Section 12.2 describes how Key IDs are formed.

3.4. Text

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

3.5. Time Fields

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

3.6. Keyrings

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

3.7. String-to-Key (S2K) Specifiers

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

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

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

ID	S2K Type	Generate?	S2K field size (octets)	Reference
0	Simple S2K	N	2	Section 3.7.1.1
1	Salted S2K	Only when string is high entropy	10	Section 3.7.1.2
2	Reserved value	N		
3	Iterated and Salted S2K	Y	11	Section 3.7.1.3
4	Argon2	Y	20	Section 3.7.1.4
100 to 110	Private/Experimental S2K	As appropriate		

Table 1: S2K type registry

These are described in the subsections below.

3.7.1.1. Simple S2K

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

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

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

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

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

3.7.1.2. Salted S2K

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

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

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

3.7.1.3. Iterated and Salted S2K

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

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

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

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

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

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

Initially, one or more hash contexts are set up as with the other S2K algorithms, depending on how many octets of key data are needed. Then the salt, followed by the passphrase data, is repeatedly hashed until the number of octets specified by the octet count has been hashed. The one exception is that if the octet count is less than the size of the salt plus passphrase, the full salt plus passphrase will be hashed even though that is greater than the octet count. After the hashing is done, the data is unloaded from the hash context(s) as with the other S2K algorithms.

3.7.1.4. Argon2

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

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

The salt SHOULD be unique for each password.

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

The memory size m is $2^{\text{encoded_m}}$ kibibytes of RAM, where "encoded_m" is the encoded memory size in Octet 19. The encoded memory size MUST be a value from $3 + \text{ceil}(\log_2(p))$ to 31, such that the decoded memory size m is a value from $8 \cdot p$ to 2^{31} . Note that memory-hardness size is indicated in kibibytes (KiB), not octets.

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

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

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

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

(where XX represents a random octet of salt).

3.7.2. String-to-Key Usage

Simple S2K and Salted S2K specifiers can be brute-forced when used with a low-entropy string, such as those typically provided by users. In addition, the usage of Simple S2K can lead to key and IV reuse (see Section 5.3). Therefore, when generating S2K specifiers, implementations MUST NOT use Simple S2K, and SHOULD NOT use Salted S2K unless the implementation knows that the string is high-entropy (for example, it generated the string itself using a known-good source of randomness). It is RECOMMENDED that implementations use Argon2.

3.7.2.1. Secret-Key Encryption

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

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

Therefore, the first octet of the secret key material describes how the secret key data is presented.

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

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

Table 2: Secret Key protection details

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

An implementation MUST NOT create and MUST reject as malformed a secret key packet where the S2K usage octet is anything but 253 and the S2K specifier type is Argon2.

3.7.2.2. Symmetric-Key Message Encryption

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

PGP 2 always used IDEA with Simple string-to-key conversion when encrypting a message with a symmetric algorithm. See Section 5.8. This MUST NOT be generated, but MAY be consumed for backward-compatibility.

4. Packet Syntax

This section describes the packets used by OpenPGP.

4.1. Overview

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

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

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

Additionally, some packets contain internal length indicators (for example, a subfield within the packet). In the event that a subfield length indicator within a packet implies inclusion of octets outside the range indicated in the packet header, a parser MUST truncate the subfield at the octet boundary indicated in the packet header. Such a truncation renders the packet malformed and unusable. An implementation MUST NOT interpret octets outside the range indicated in the packet header as part of the contents of the packet.

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.

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

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

Ptag 7 6 5 4 3 2 1 0

Bit 7 -- Always one

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

The Legacy packet format MAY be used when consuming packets to facilitate interoperability with legacy implementations and accessing archived data. The Legacy packet format SHOULD NOT be used to generate new data, unless the recipient is known to only support the Legacy packet format.

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

The current OpenPGP packet format packets contain:

Bits 5 to 0 -- packet tag

Legacy packet format packets contain:

Bits 5 to 2 -- packet tag

Bits 1 to 0 -- length-type

4.2.1. OpenPGP Format Packet Lengths

OpenPGP format packets have four possible ways of encoding length:

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

4.2.1.1. One-Octet Lengths

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

$$\text{bodyLen} = \text{1st_octet};$$

4.2.1.2. Two-Octet Lengths

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

$$\text{bodyLen} = ((\text{1st_octet} - 192) \ll 8) + (\text{2nd_octet}) + 192$$

4.2.1.3. Five-Octet Lengths

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

$$\text{bodyLen} = (\text{2nd_octet} \ll 24) \mid (\text{3rd_octet} \ll 16) \mid (\text{4th_octet} \ll 8) \mid \text{5th_octet}$$

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

4.2.1.4. Partial Body Lengths

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

$$\text{partialBodyLen} = 1 \ll (\text{1st_octet} \& 0\text{x1F});$$

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

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

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

4.2.2. Legacy Format Packet Lengths

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

- 0 The packet has a one-octet length. The header is 2 octets long.
- 1 The packet has a two-octet length. The header is 3 octets long.
- 2 The packet has a four-octet length. The header is 5 octets long.
- 3 The packet is of indeterminate length. The header is 1 octet long, and the implementation must determine how long the packet is. If the packet is in a file, this means that the packet extends until the end of the file. The OpenPGP format headers have a mechanism for precisely encoding data of indeterminate length. An implementation MUST NOT generate a Legacy format packet with indeterminate length. An implementation MAY interpret an indeterminate length Legacy format packet in order to deal with historic data, or data generated by a legacy system.

4.2.3. Packet Length Examples

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

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

A packet with length 1723 may have its length encoded in two octets: 0xC5, 0xFB. This header is followed by the 1723 octets of data.

A packet with length 100000 may have its length encoded in five octets: 0xFF, 0x00, 0x01, 0x86, 0xA0.

It might also be encoded in the following octet stream: 0xEE, first 32768 octets of data; 0xE1, next two octets of data; 0xE0, next one octet of data; 0xF0, next 65536 octets of data; 0xC5, 0xDD, last 1693 octets of data. This is just one possible encoding, and many variations are possible on the size of the Partial Body Length headers, as long as a regular Body Length header encodes the last portion of the data.

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

4.3. Packet Tags

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

Tag	Packet Type
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	Reserved (formerly Modification Detection Code Packet)
20	Reserved (formerly AEAD Encrypted Data Packet)
21	Padding Packet
60 to 63	Private or Experimental Values

Table 3: Packet type registry

5. Packet Types

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

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

The body of this packet starts with a one-octet number giving the version number of the packet type. The currently defined versions are 3 and 5. The remainder of the packet depends on the version.

The versions differ in how they identify the recipient key, and in what they encode. The version of the PKESK packet must align with the version of the SEIPD packet (see Section 11.3.2.1).

5.1.1. v3 PKESK

A version 3 Public-Key Encrypted Session Key (PKESK) packet precedes a version 1 Symmetrically Encrypted Integrity Protected Data (v1 SEIPD, see Section 5.14.1) packet. In historic data, it is sometimes found preceding a deprecated Symmetrically Encrypted Data packet (SED, see Section 5.8). A v3 PKESK packet MUST NOT precede a v2 SEIPD packet (see Section 11.3.2.1).

The v3 PKESK packet consists of:

- * A one-octet version number with value 3.
- * An eight-octet number that gives the Key ID of the public key to which the session key is encrypted. If the session key is encrypted to a subkey, then the Key ID of this subkey is used here instead of the Key ID of the primary key. The Key ID may also be all zeros, for an "anonymous recipient" (see Section 5.1.6).
- * A one-octet number giving the public-key algorithm used.
- * A series of values comprising the encrypted session key. This is algorithm-specific and described below.

When creating a v3 PKESK packet, the session key is first prefixed with a one-octet algorithm identifier that specifies the symmetric encryption algorithm used to encrypt the following encryption container. Then a two-octet checksum is appended, which is equal to the sum of the preceding session key octets, not including the algorithm identifier, modulo 65536.

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

5.1.2. v5 PKESK

A version 5 Public-Key Encrypted Session Key (PKESK) packet precedes a version 2 Symmetrically Encrypted Integrity Protected Data (v2 SEIPD, see Section 5.14.2) packet. A v5 PKESK packet MUST NOT precede a v1 SEIPD packet or a deprecated Symmetrically Encrypted Data packet (see Section 11.3.2.1).

The v5 PKESK packet consists of:

- * A one-octet version number with value 5.
- * A one octet key version number and N octets of the fingerprint of the public key or subkey to which the session key is encrypted. Note that the length N of the fingerprint for a version 4 key is 20 octets; for a version 5 key N is 32. The key version number may also be zero, and the fingerprint omitted (that is, the length N is zero in this case), for an "anonymous recipient" (see Section 5.1.6).
- * A one-octet number giving the public-key algorithm used.
- * A series of values comprising the encrypted session key. This is algorithm-specific and described below.

When creating a V5 PKESK packet, the symmetric encryption algorithm identifier is not included. Before encrypting, a two-octet checksum is appended, which is equal to the sum of the preceding session key octets, modulo 65536.

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

5.1.3. Algorithm Specific Fields for RSA encryption

- * Multiprecision integer (MPI) of RSA-encrypted value $m^e \bmod n$.

The value "m" in the above formula is the plaintext value described above, encoded in the PKCS#1 block encoding EME-PKCS1-v1_5 described in Section 7.2.1 of [RFC8017] (see also Section 14.1). Note that when an implementation forms several PKESKs with one session key, forming a message that can be decrypted by several keys, the implementation MUST make a new PKCS#1 encoding for each key.

5.1.4. Algorithm Specific Fields for Elgamal encryption

- * MPI of Elgamal (Diffie-Hellman) value $g^k \bmod p$.
- * MPI of Elgamal (Diffie-Hellman) value $m * y^k \bmod p$.

The value "m" in the above formula is the plaintext value described above, encoded in the PKCS#1 block encoding EME-PKCS1-v1_5 described in Section 7.2.1 of [RFC8017] (see also Section 14.1). Note that when an implementation forms several PKESKs with one session key, forming a message that can be decrypted by several keys, the implementation MUST make a new PKCS#1 encoding for each key.

5.1.5. Algorithm-Specific Fields for ECDH encryption

- * MPI of an EC point representing an ephemeral public key, in the point format associated with the curve as specified in Section 9.2.
- * A one-octet size, followed by a symmetric key encoded using the method described in Section 13.5.

5.1.6. Notes on PKESK

An implementation MAY accept or use a Key ID of all zeros, or a key version of zero and no key fingerprint, to hide the intended decryption key. In this case, the receiving implementation would try all available private keys, checking for a valid decrypted session key. This format helps reduce traffic analysis of messages.

5.2. Signature Packet (Tag 2)

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

Three versions of Signature packets are defined. Version 3 provides basic signature information, while versions 4 and 5 provide an expandable format with subpackets that can specify more information about the signature.

An implementation **MUST** generate a version 5 signature when signing with a version 5 key. An implementation **MUST** generate a version 4 signature when signing with a version 4 key. Implementations **MUST NOT** create version 3 signatures; they **MAY** accept version 3 signatures.

5.2.1. Signature Types

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

These meanings are as follows:

0x00: Signature of a binary document.

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

0x01: Signature of a canonical text document.

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

0x02: Standalone signature.

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

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

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

0x11: Persona certification of a User ID and Public-Key packet.

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

0x12: Casual certification of a User ID and Public-Key packet.

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

0x13: Positive certification of a User ID and Public-Key packet.

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

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

0x18: Subkey Binding Signature.

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

0x19: Primary Key Binding Signature.

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

0x1F: Signature directly on a key.

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

0x20: Key revocation signature.

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

0x28: Subkey revocation signature.

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

0x30: Certification revocation signature.

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

0x40: Timestamp signature.

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

0x50: Third-Party Confirmation signature.

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

5.2.2. Version 3 Signature Packet Format

The body of a version 3 Signature Packet contains:

- * One-octet version number (3).
- * One-octet length of following hashed material. MUST be 5.
 - One-octet signature type.
 - Four-octet creation time.
- * Eight-octet Key ID of signer.

- * One-octet public-key algorithm.
- * One-octet hash algorithm.
- * Two-octet field holding left 16 bits of signed hash value.
- * One or more multiprecision integers comprising the signature.
This portion is algorithm specific, as described below.

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

Algorithm-Specific Fields for RSA signatures:

- * Multiprecision integer (MPI) of RSA signature value $m^d \bmod n$.

Algorithm-Specific Fields for DSA signatures:

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

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

With RSA signatures, the hash value is encoded using PKCS#1 encoding type EMSA-PKCS1-v1_5 as described in Section 9.2 of [RFC8017]. This requires inserting the hash value as an octet string into an ASN.1 structure. The object identifier for the type of hash being used is included in the structure. The hexadecimal representations for the currently defined hash algorithms are as follows:

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

Table 4: Hash hexadecimal representations

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

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

Table 5: Hash OIDs

The full hash prefixes for these are as follows:

algorithm	full hash prefix
MD5	0x30, 0x20, 0x30, 0x0C, 0x06, 0x08, 0x2A, 0x86, 0x48, 0x86, 0xF7, 0x0D, 0x02, 0x05, 0x05, 0x00, 0x04, 0x10
RIPEMD-160	0x30, 0x21, 0x30, 0x09, 0x06, 0x05, 0x2B, 0x24, 0x03, 0x02, 0x01, 0x05, 0x00, 0x04, 0x14
SHA-1	0x30, 0x21, 0x30, 0x09, 0x06, 0x05, 0x2B, 0x0E, 0x03, 0x02, 0x1A, 0x05, 0x00, 0x04, 0x14
SHA224	0x30, 0x2D, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x04, 0x05, 0x00, 0x04, 0x1C
SHA256	0x30, 0x31, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x01, 0x05, 0x00, 0x04, 0x20
SHA384	0x30, 0x41, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x02, 0x05, 0x00, 0x04, 0x30
SHA512	0x30, 0x51, 0x30, 0x0D, 0x06, 0x09, 0x60, 0x86, 0x48, 0x01, 0x65, 0x03, 0x04, 0x02, 0x03, 0x05, 0x00, 0x04, 0x40

Table 6: Hash hexadecimal prefixes

DSA signatures MUST use hashes that are equal in size to the number of bits of q , the group generated by the DSA key's generator value.

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

5.2.3. Version 4 and 5 Signature Packet Formats

The body of a V4 or V5 Signature packet contains:

- * One-octet version number. This is 4 for V4 signatures and 5 for V5 signatures.
- * One-octet signature type.
- * One-octet public-key algorithm.
- * One-octet hash algorithm.
- * A scalar octet count for following hashed subpacket data. For a V4 signature, this is a two-octet field. For a V5 signature, this is a four-octet field. Note that this is the length in octets of all of the hashed subpackets; a pointer incremented by this number will skip over the hashed subpackets.
- * Hashed subpacket data set (zero or more subpackets).
- * A scalar octet count for the following unhashed subpacket data. For a V4 signature, this is a two-octet field. For a V5 signature, this is a four-octet field. Note that this is the length in octets of all of the unhashed subpackets; a pointer incremented by this number will skip over the unhashed subpackets.
- * Unhashed subpacket data set (zero or more subpackets).
- * Two-octet field holding the left 16 bits of the signed hash value.
- * Only for V5 signatures, a 16 octet field containing random values used as salt.
- * One or more multiprecision integers comprising the signature. This portion is algorithm specific:

5.2.3.1. Algorithm-Specific Fields for RSA signatures

- * Multiprecision integer (MPI) of RSA signature value $m^d \bmod n$.

5.2.3.2. Algorithm-Specific Fields for DSA or ECDSA signatures

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

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

5.2.3.3. Algorithm-Specific Fields for EdDSA signatures

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

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

5.2.3.3.1. Algorithm-Specific Fields for Ed25519 signatures

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

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

5.2.3.3.2. Algorithm-Specific Fields for Ed448 signatures

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

- * The first MPI has a body of 58 octets: a prefix 0x40 octet, followed by 57 octets of the native signature.
- * The second MPI is set to 0 (this is a placeholder, and is unused). Note that an MPI with a value of 0 is encoded on the wire as a pair of zero octets: 00 00.

5.2.3.4. Notes on Signatures

The concatenation of the data being signed and the signature data from the version number through the hashed subpacket data (inclusive) is hashed. The resulting hash value is what is signed. The high 16 bits (first two octets) of the hash are included in the Signature packet to provide a way to reject some invalid signatures without performing a signature verification.

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

The differences between a V4 and V5 signature are two-fold: first, a V5 signature increases the width of the size indicators for the signed data, making it more capable when signing large keys or messages. Second, the hash is salted with 128 bit of random data.

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

5.2.3.5. Signature Subpacket Specification

A subpacket data set consists of zero or more Signature subpackets. In Signature packets, the subpacket data set is preceded by a two-octet (for V4 signatures) or four-octet (for V5 signatures) scalar count of the length in octets of all the subpackets. A pointer incremented by this number will skip over the subpacket data set.

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

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

and is followed by the subpacket-specific data.

The length includes the type octet but not this length. Its format is similar to the "new" format packet header lengths, but cannot have Partial Body Lengths. That is:

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

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

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

The value of the subpacket type octet may be:

Type	Description
0	Reserved
1	Reserved
2	Signature Creation Time
3	Signature Expiration Time

4	Exportable Certification
5	Trust Signature
6	Regular Expression
7	Revocable
8	Reserved
9	Key Expiration Time
10	Placeholder for backward compatibility
11	Preferred Symmetric Ciphers for v1 SEIPD
12	Revocation Key (deprecated)
13 to 15	Reserved
16	Issuer
17 to 19	Reserved
20	Notation Data
21	Preferred Hash Algorithms
22	Preferred Compression Algorithms
23	Key Server Preferences
24	Preferred Key Server
25	Primary User ID
26	Policy URI
27	Key Flags
28	Signer's User ID
29	Reason for Revocation
30	Features
31	Signature Target

32	Embedded Signature
33	Issuer Fingerprint
34	Reserved
35	Intended Recipient Fingerprint
37	Reserved (Attested Certifications)
38	Reserved (Key Block)
39	Preferred AEAD Ciphersuites
100 to 110	Private or experimental

Table 7: Subpacket type registry

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

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

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

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

5.2.3.6. Signature Subpacket Types

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

A subpacket may be found either in the hashed or unhashed subpacket sections of a signature. If a subpacket is not hashed, then the information in it cannot be considered definitive because it is not part of the signature proper.

5.2.3.7. Notes on Self-Signatures

A self-signature is a binding signature made by the key to which the signature refers. There are three types of self-signatures, the certification signatures (types 0x10-0x13), the direct-key signature (type 0x1F), and the subkey binding signature (type 0x18). A cryptographically-valid self-signature should be accepted from any primary key, regardless of what Key Flags (Section 5.2.3.26) apply to the primary key. In particular, a primary key does not need to have 0x01 set in the first octet of Key Flags order to make a valid self-signature.

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

Implementing software should interpret a self-signature's preference subpackets as narrowly as possible. For example, suppose a key has two user names, Alice and Bob. Suppose that Alice prefers the AEAD ciphersuite AES-256 with OCB, and Bob prefers Camellia-256 with GCM. If the software locates this key via Alice's name, then the preferred AEAD ciphersuite is AES-256 with OCB; if software locates the key via Bob's name, then the preferred algorithm is Camellia-256 with GCM. If the key is located by Key ID, the algorithm of the primary User ID of the key provides the preferred AEAD ciphersuite.

Revoking a self-signature or allowing it to expire has a semantic meaning that varies with the signature type. Revoking the self-signature on a User ID effectively retires that user name. The self-signature is a statement, "My name X is tied to my signing key K" and

is corroborated by other users' certifications. If another user revokes their certification, they are effectively saying that they no longer believe that name and that key are tied together. Similarly, if the users themselves revoke their self-signature, then the users no longer go by that name, no longer have that email address, etc. Revoking a binding signature effectively retires that subkey. Revoking a direct-key signature cancels that signature. Please see Section 5.2.3.28 for more relevant detail.

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

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

An implementation that encounters multiple self-signatures on the same object may resolve the ambiguity in any way it sees fit, but it is RECOMMENDED that priority be given to the most recent self-signature.

5.2.3.8. Signature Creation Time

(4-octet time field)

The time the signature was made.

MUST be present in the hashed area.

5.2.3.9. Issuer

(8-octet Key ID)

The OpenPGP Key ID of the key issuing the signature. If the version of that key is greater than 4, this subpacket MUST NOT be included in the signature.

5.2.3.10. Key Expiration Time

(4-octet time field)

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

5.2.3.11. Preferred Symmetric Ciphers for v1 SEIPD

(array of one-octet values)

A series of symmetric cipher algorithm identifiers indicating how the keyholder prefers to receive version 1 Symmetrically Encrypted Integrity Protected Data (Section 5.14.1). The subpacket body is an ordered list of octets with the most preferred listed first. It is assumed that only algorithms listed are supported by the recipient's software. Algorithm numbers are in Section 9.3. This is only found on a self-signature.

When generating a v2 SEIPD packet, this preference list is not relevant. See Section 5.2.3.12 instead.

5.2.3.12. Preferred AEAD Ciphersuites

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

A series of paired algorithm identifiers indicating how the keyholder prefers to receive version 2 Symmetrically Encrypted Integrity Protected Data (Section 5.14.2). Each pair of octets indicates a combination of a symmetric cipher and an AEAD mode that the keyholder prefers to use. The symmetric cipher identifier precedes the AEAD identifier in each pair. The subpacket body is an ordered list of pairs of octets with the most preferred algorithm combination listed first.

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

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

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

09 02 09 03 13 02

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

Note that support for version 2 of the Symmetrically Encrypted Integrity Protected Data packet (Section 5.14.2) in general is indicated by a Feature Flag (Section 5.2.3.29).

This subpacket is only found on a self-signature.

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

5.2.3.13. Preferred Hash Algorithms

(array of one-octet values)

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

5.2.3.14. Preferred Compression Algorithms

(array of one-octet values)

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

5.2.3.15. Signature Expiration Time

(4-octet time field)

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

5.2.3.16. Exportable Certification

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

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

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

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

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

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

5.2.3.17. Revocable

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

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

5.2.3.18. Trust Signature

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

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

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

5.2.3.19. Regular Expression

(null-terminated regular expression)

Used in conjunction with trust Signature packets (of level > 0) to limit the scope of trust that is extended. Only signatures by the target key on User IDs that match the regular expression in the body of this packet have trust extended by the trust Signature subpacket. The regular expression uses the same syntax as the Henry Spencer's "almost public domain" regular expression [REGEX] package. A description of the syntax is found in Section 8.

5.2.3.20. Revocation Key

(1 octet of class, 1 octet of public-key algorithm ID, 20 octets of V4 fingerprint)

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

An application that wants the functionality of delegating revocation SHOULD instead use an escrowed Revocation Signature. See Section 15.2 for more details.

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

This packet was intended to authorize the specified key to issue revocation signatures for this key. Class octet must have bit 0x80 set. If the bit 0x40 is set, then this means that the revocation information is sensitive. Other bits are for future expansion to other kinds of authorizations. This is only found on a direct-key self-signature (type 0x1f). The use on other types of self-signatures is unspecified.

If the "sensitive" flag is set, the keyholder feels this subpacket contains private trust information that describes a real-world sensitive relationship. If this flag is set, implementations SHOULD NOT export this signature to other users except in cases where the data needs to be available: when the signature is being sent to the designated revoker, or when it is accompanied by a revocation signature from that revoker. Note that it may be appropriate to isolate this subpacket within a separate signature so that it is not combined with other subpackets that need to be exported.

5.2.3.21. Notation Data

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

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

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

First octet:

flag	shorthand	definition
0x80	human-readable	This note value is text.

Table 8: Notation flag registry (first octet)

Other octets: none.

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

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

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

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

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

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

5.2.3.22. Key Server Preferences

(N octets of flags)

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

First octet:

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

Table 9: Key server preferences flag registry (first octet)

This is found only on a self-signature.

5.2.3.23. Preferred Key Server

(String)

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

5.2.3.24. Primary User ID

(1 octet, Boolean)

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

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

5.2.3.25. Policy URI

(String)

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

5.2.3.26. Key Flags

(N octets of flags)

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

First octet:

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

Table 10: Key flags registry (first octet)

Second octet:

flag	definition
0x04	Reserved (ADSK).
0x08	Reserved (timestamping).

Table 11: Key flags registry
(second octet)

Usage notes:

The flags in this packet may appear in self-signatures or in certification signatures. They mean different things depending on who is making the statement --- for example, a certification signature that has the "sign data" flag is stating that the certification is for that use. On the other hand, the "communications encryption" flag in a self-signature is stating a preference that a given key be used for communications. Note however, that it is a thorny issue to determine what is "communications" and what is "storage". This decision is left wholly

up to the implementation; the authors of this document do not claim any special wisdom on the issue and realize that accepted opinion may change.

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

5.2.3.27. Signer's User ID

(String)

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

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

5.2.3.28. Reason for Revocation

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

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

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

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

Table 12: Reasons for revocation

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

If a key has been revoked because of a compromise, all signatures created by that key are suspect. However, if it was merely superseded or retired, old signatures are still valid. If the revoked signature is the self-signature for certifying a User ID, a revocation denotes that that user name is no longer in use. Such a revocation SHOULD include a 0x20 code.

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

5.2.3.29. Features

(N octets of flags)

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

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

An implementation SHOULD NOT use a feature listed when sending to a user who does not state that they can use it.

Defined features are as follows:

First octet:

Feature	Definition	Reference
0x01	Symmetrically Encrypted Integrity Protected Data packet version 1	Section 5.14.1
0x02	Reserved	
0x04	Reserved	
0x08	Symmetrically Encrypted Integrity Protected Data packet version 2	Section 5.14.2

Table 13: Features registry

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

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

See Section 15.1 for details about how to use the Features subpacket when generating encryption data.

5.2.3.30. Signature Target

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

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

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

5.2.3.31. Embedded Signature

(1 signature packet body)

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

5.2.3.32. Issuer Fingerprint

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

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

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

5.2.3.33. Intended Recipient Fingerprint

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

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

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

5.2.4. Computing Signatures

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

When a V5 signature is made, the salt is hashed first.

For binary document signatures (type 0x00), the document data is hashed directly. For text document signatures (type 0x01), the document is canonicalized by converting line endings to <CR><LF>, and the resulting data is hashed.

When a V4 signature is made over a key, the hash data starts with the octet 0x99, followed by a two-octet length of the key, and then body of the key packet. When a V5 signature is made over a key, the hash data starts with the octet 0x9a, followed by a four-octet length of the key, and then body of the key packet.

A subkey binding signature (type 0x18) or primary key binding signature (type 0x19) then hashes the subkey using the same format as the main key (also using 0x99 or 0x9a as the first octet). Primary key revocation signatures (type 0x20) hash only the key being revoked. Subkey revocation signature (type 0x28) hash first the primary key and then the subkey being revoked.

A certification signature (type 0x10 through 0x13) hashes the User ID being bound to the key into the hash context after the above data. A V3 certification hashes the contents of the User ID or attribute packet packet, without any header. A V4 or V5 certification hashes the constant 0xB4 for User ID certifications or the constant 0xD1 for User Attribute certifications, followed by a four-octet number giving the length of the User ID or User Attribute data, and then the User ID or User Attribute data.

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

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

- * A V3 signature hashes five octets of the packet body, starting from the signature type field. This data is the signature type, followed by the four-octet signature time.
- * A V4 or V5 signature hashes the packet body starting from its first field, the version number, through the end of the hashed subpacket data and a final extra trailer. Thus, the hashed fields are:
 - An octet indicating the signature version (0x04 for V4, 0x05 for V5),
 - the signature type,
 - the public-key algorithm,
 - the hash algorithm,
 - the hashed subpacket length,
 - the hashed subpacket body,
 - A second version octet (0x04 for V4, 0x05 for V5)
 - A single octet 0xFF,
 - A number representing the length of the hashed data from the Signature packet stopping right before the second version octet. For a V4 signature, this is a four-octet big-endian number, considered to be an unsigned integer modulo 2^{32} . For a V5 signature, this is an eight-octet big-endian number, considered to be an unsigned integer modulo 2^{64} .

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

5.2.4.1. Subpacket Hints

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

be stingy in what they generate.

Some apparent conflicts may actually make sense --- for example, suppose a keyholder has a V3 key and a V4 key that share the same RSA key material. Either of these keys can verify a signature created by the other, and it may be reasonable for a signature to contain an issuer subpacket for each key, as a way of explicitly tying those keys to the signature.

5.3. Symmetric-Key Encrypted Session Key Packets (Tag 3)

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

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

The body of this packet starts with a one-octet number giving the version number of the packet type. The currently defined versions are 4 and 5. The remainder of the packet depends on the version.

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

5.3.1. v4 SKESK

A version 4 Symmetric-Key Encrypted Session Key (SKESK) packet precedes a version 1 Symmetrically Encrypted Integrity Protected Data (v1 SEIPD, see Section 5.14.1) packet. In historic data, it is sometimes found preceding a deprecated Symmetrically Encrypted Data packet (SED, see Section 5.8). A v4 SKESK packet MUST NOT precede a v2 SEIPD packet (see Section 11.3.2.1).

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

- * A one-octet version number with value 4.

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

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

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

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

5.3.2. v5 SKESK

A version 5 Symmetric-Key Encrypted Session Key (SKESK) packet precedes a version 2 Symmetrically Encrypted Integrity Protected Data (v2 SEIPD, see Section 5.14.2) packet. A v5 SKESK packet MUST NOT precede a v1 SEIPD packet or a deprecated Symmetrically Encrypted Data packet (see Section 11.3.2.1).

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

- * A one-octet version number with value 5.
- * A one-octet scalar octet count of the following 5 fields.
- * A one-octet symmetric cipher algorithm identifier.
- * A one-octet AEAD algorithm identifier.
- * A one-octet scalar octet count of the following field.

- * A string-to-key (S2K) specifier. The length of the string-to-key specifier depends on its type (see Section 3.7.1).
- * A starting initialization vector of size specified by the AEAD algorithm.
- * The encrypted session key itself.
- * An authentication tag for the AEAD mode.

HKDF is used with SHA256 as hash algorithm, the key derived from S2K as Initial Keying Material (IKM), no salt, and the Packet Tag in new format encoding (bits 7 and 6 set, bits 5-0 carry the packet tag), the packet version, and the cipher-algo and AEAD-mode used to encrypt the key material, are used as info parameter. Then, the session key is encrypted using the resulting key, with the AEAD algorithm specified for version 2 of the Symmetrically Encrypted Integrity Protected Data packet. Note that no chunks are used and that there is only one authentication tag. The Packet Tag in OpenPGP format encoding (bits 7 and 6 set, bits 5-0 carry the packet tag), the packet version number, the cipher algorithm octet, and the AEAD algorithm octet are given as additional data. For example, the additional data used with AES-128 with OCB consists of the octets 0xC3, 0x05, 0x07, and 0x02.

5.4. One-Pass Signature Packets (Tag 4)

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

The body of this packet consists of:

- * A one-octet version number. The currently defined versions are 3 and 5.
- * A one-octet signature type. Signature types are described in Section 5.2.1.
- * A one-octet number describing the hash algorithm used.
- * A one-octet number describing the public-key algorithm used.
- * Only for V5 packets, a 16 octet field containing random values used as salt. The value must match the salt field of the corresponding Signature packet.

- * Only for V3 packets, an eight-octet number holding the Key ID of the signing key.
- * Only for V5 packets, a one octet key version number and N octets of the fingerprint of the signing key. Note that the length N of the fingerprint for a version 5 key is 32.
- * A one-octet number holding a flag showing whether the signature is nested. A zero value indicates that the next packet is another One-Pass Signature packet that describes another signature to be applied to the same message data.

When generating a one-pass signature, the OPS packet version **MUST** correspond to the version of the associated signature packet, except for the historical accident that v4 keys use a v3 one-pass signature packet (there is no v4 OPS):

Signing key version	OPS packet version	Signature packet version
4	3	4
5	5	5

Table 14: Versions of packets used in a one-pass signature

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

5.5. Key Material Packet

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

5.5.1. Key Packet Variants

5.5.1.1. Public-Key Packet (Tag 6)

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

5.5.1.2. Public-Subkey Packet (Tag 14)

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

5.5.1.3. Secret-Key Packet (Tag 5)

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

5.5.1.4. Secret-Subkey Packet (Tag 7)

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

5.5.2. Public-Key Packet Formats

There are three versions of key-material packets.

OpenPGP implementations SHOULD create keys with version 5 format. V4 keys are deprecated; an implementation SHOULD NOT generate a V4 key, but SHOULD accept it. V3 keys are deprecated; an implementation MUST NOT generate a V3 key, but MAY accept it. V2 keys are deprecated; an implementation MUST NOT generate a V2 key, but MAY accept it.

A version 3 public key or public-subkey packet contains:

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

V3 keys are deprecated. They contain three weaknesses. First, it is relatively easy to construct a V3 key that has the same Key ID as any other key because the Key ID is simply the low 64 bits of the public modulus. Secondly, because the fingerprint of a V3 key hashes the key material, but not its length, there is an increased opportunity for fingerprint collisions. Third, there are weaknesses in the MD5 hash algorithm that make developers prefer other algorithms. See Section 12.2 for a fuller discussion of Key IDs and fingerprints.

V2 keys are identical to the deprecated V3 keys except for the version number.

The version 4 format is similar to the version 3 format except for the absence of a validity period. This has been moved to the Signature packet. In addition, fingerprints of version 4 keys are calculated differently from version 3 keys, as described in Section 12.

A version 4 packet contains:

- * A one-octet version number (4).
- * A four-octet number denoting the time that the key was created.
- * A one-octet number denoting the public-key algorithm of this key.
- * A series of values comprising the key material. This is algorithm-specific and described in Section 5.6.

The version 5 format is similar to the version 4 format except for the addition of a count for the key material. This count helps parsing secret key packets (which are an extension of the public key packet format) in the case of an unknown algorithm. In addition, fingerprints of version 5 keys are calculated differently from version 4 keys, as described in Section 12.

A version 5 packet contains:

- * A one-octet version number (5).
- * A four-octet number denoting the time that the key was created.
- * A one-octet number denoting the public-key algorithm of this key.
- * A four-octet scalar octet count for the following public key material.

- * A series of values comprising the public key material. This is algorithm-specific and described in Section 5.6.

5.5.3. Secret-Key Packet Formats

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

The packet contains:

- * The fields of a Public-Key or Public-Subkey packet, as described above.
- * One octet indicating string-to-key usage conventions. Zero indicates that the secret-key data is not encrypted. 255, 254, or 253 indicates that a string-to-key specifier is being given. Any other value is a symmetric-key encryption algorithm identifier. A version 5 packet MUST NOT use the value 255.
- * Only for a version 5 packet, a one-octet scalar octet count of the next 5 optional fields.
- * [Optional] If string-to-key usage octet was 255, 254, or 253, a one-octet symmetric encryption algorithm.
- * [Optional] If string-to-key usage octet was 253, a one-octet AEAD algorithm.
- * [Optional] Only for a version 5 packet, and if string-to-key usage octet was 255, 254, or 253, an one-octet count of the following field.
- * [Optional] If string-to-key usage octet was 255, 254, or 253, a string-to-key (S2K) specifier. The length of the string-to-key specifier depends on its type (see Section 3.7.1).
- * [Optional] If string-to-key usage octet was 253 (that is, the secret data is AEAD-encrypted), an initialization vector (IV) of size specified by the AEAD algorithm (see Section 5.14.2), which is used as the nonce for the AEAD algorithm.
- * [Optional] If string-to-key usage octet was 255, 254, or a cipher algorithm identifier (that is, the secret data is CFB-encrypted), an initialization vector (IV) of the same length as the cipher's block size.

- * Plain or encrypted multiprecision integers comprising the secret key data. This is algorithm-specific and described in Section 5.6. If the string-to-key usage octet is 253, then an AEAD authentication tag is part of that data. If the string-to-key usage octet is 254, a 20-octet SHA-1 hash of the plaintext of the algorithm-specific portion is appended to plaintext and encrypted with it. If the string-to-key usage octet is 255 or another nonzero value (that is, a symmetric-key encryption algorithm identifier), a two-octet checksum of the plaintext of the algorithm-specific portion (sum of all octets, mod 65536) is appended to plaintext and encrypted with it. (This is deprecated and SHOULD NOT be used, see below.)
- * If the string-to-key usage octet is zero, then a two-octet checksum of the algorithm-specific portion (sum of all octets, mod 65536).

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

Note that the version 5 packet format adds two count values to help parsing packets with unknown S2K or public key algorithms.

Secret MPI values can be encrypted using a passphrase. If a string-to-key specifier is given, that describes the algorithm for converting the passphrase to a key, else a simple MD5 hash of the passphrase is used. Implementations MUST use a string-to-key specifier; the simple hash is for backward compatibility and is deprecated, though implementations MAY continue to use existing private keys in the old format. The cipher for encrypting the MPIs is specified in the Secret-Key packet.

Encryption/decryption of the secret data is done using the key created from the passphrase and the initialization vector from the packet. If the string-to-key usage octet is not 253, CFB mode is used. A different mode is used with V3 keys (which are only RSA) than with other key formats. With V3 keys, the MPI bit count prefix (that is, the first two octets) is not encrypted. Only the MPI non-prefix data is encrypted. Furthermore, the CFB state is resynchronized at the beginning of each new MPI value, so that the CFB block boundary is aligned with the start of the MPI data.

With V4 and V5 keys, a simpler method is used. All secret MPI values are encrypted, including the MPI bitcount prefix.

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

Initial Keying Material (IKM), no salt, and the Packet Tag in OpenPGP format encoding (bits 7 and 6 set, bits 5-0 carry the packet tag), the packet version, and the cipher-algo and AEAD-mode used to encrypt the key material, are used as info parameter. Then, the encrypted MPI values are encrypted as one combined plaintext using one of the AEAD algorithms specified for version 2 of the Symmetrically Encrypted Integrity Protected Data packet. Note that no chunks are used and that there is only one authentication tag. As additional data, the Packet Tag in OpenPGP format encoding (bits 7 and 6 set, bits 5-0 carry the packet tag), followed by the public key packet fields, starting with the packet version number, are passed to the AEAD algorithm. For example, the additional data used with a Secret-Key Packet of version 4 consists of the octets 0xC5, 0x04, followed by four octets of creation time, one octet denoting the public-key algorithm, and the algorithm-specific public-key parameters. For a Secret-Subkey Packet, the first octet would be 0xC7. For a version 5 key packet, the second octet would be 0x05, and the four-octet octet count of the public key material would be included as well (see Section 5.5.2).

The two-octet checksum that follows the algorithm-specific portion is the algebraic sum, mod 65536, of the plaintext of all the algorithm-specific octets (including MPI prefix and data). With V3 keys, the checksum is stored in the clear. With V4 keys, the checksum is encrypted like the algorithm-specific data. This value is used to check that the passphrase was correct. However, this checksum is deprecated; an implementation SHOULD NOT use it, but should rather use the SHA-1 hash denoted with a usage octet of 254. The reason for this is that there are some attacks that involve undetectably modifying the secret key. If the string-to-key usage octet is 253 no checksum or SHA-1 hash is used but the authentication tag of the AEAD algorithm follows.

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

5.6. Algorithm-specific Parts of Keys

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

5.6.1. Algorithm-Specific Part for RSA Keys

The public key is this series of multiprecision integers:

- * MPI of RSA public modulus n ;
- * MPI of RSA public encryption exponent e .

The secret key is this series of multiprecision integers:

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

5.6.2. Algorithm-Specific Part for DSA Keys

The public key is this series of multiprecision integers:

- * MPI of DSA prime p ;
- * MPI of DSA group order q (q is a prime divisor of $p-1$);
- * MPI of DSA group generator g ;
- * MPI of DSA public-key value y ($= g^{**}x \bmod p$ where x is secret).

The secret key is this single multiprecision integer:

- * MPI of DSA secret exponent x .

5.6.3. Algorithm-Specific Part for Elgamal Keys

The public key is this series of multiprecision integers:

- * MPI of Elgamal prime p ;
- * MPI of Elgamal group generator g ;
- * MPI of Elgamal public key value y ($= g^{**}x \bmod p$ where x is secret).

The secret key is this single multiprecision integer:

- * MPI of Elgamal secret exponent x .

5.6.4. Algorithm-Specific Part for ECDSA Keys

The public key is this series of values:

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

The secret key is this single multiprecision integer:

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

5.6.5. Algorithm-Specific Part for EdDSA Keys

The public key is this series of values:

- * A variable-length field containing a curve OID, formatted as follows:
 - A one-octet size of the following field; values 0 and 0xFF are reserved for future extensions,
 - The octets representing a curve OID, defined in Section 9.2;
- * An MPI of an EC point representing a public key Q in prefixed native form (see Section 13.2.2).

The secret key is this single multiprecision integer:

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

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

5.6.6. Algorithm-Specific Part for ECDH Keys

The public key is this series of values:

- * A variable-length field containing a curve OID, which is formatted as follows:

- A one-octet size of the following field; values 0 and 0xFF are reserved for future extensions,
- Octets representing a curve OID, defined in Section 9.2;
- * MPI of an EC point representing a public key, in the point format associated with the curve as specified in Section 9.2.1
- * A variable-length field containing KDF parameters, which is formatted as follows:
 - A one-octet size of the following fields; values 0 and 0xFF are reserved for future extensions,
 - A one-octet value 1, reserved for future extensions,
 - A one-octet hash function ID used with a KDF,
 - A one-octet algorithm ID for the symmetric algorithm used to wrap the symmetric key used for the message encryption; see Section 13.5 for details.

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

The secret key is this single multiprecision integer:

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

5.6.6.1. ECDH Secret Key Material

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

5.6.6.1.1. Curve25519 ECDH Secret Key Material

A Curve25519 secret key is stored as a standard integer in big-endian MPI form. Note that this form is in reverse octet order from the little-endian "native" form found in [RFC7748].

Note also that the integer for a Curve25519 secret key for OpenPGP MUST have the appropriate form: that is, it MUST be divisible by 8, MUST be at least 2^{254} , and MUST be less than 2^{255} . The length of this MPI in bits is by definition always 255, so the two leading octets of the MPI will always be 00 ff and reversing the following 32 octets from the wire will produce the "native" form.

When generating a new Curve25519 secret key from 32 fully-random octets, the following pseudocode produces the MPI wire format (note the similarity to decodeScalar25519 from [RFC7748]):

```
def curve25519_MPI_from_random(octet_list):
    octet_list[0] &= 248
    octet_list[31] &= 127
    octet_list[31] |= 64
    mpi_header = [ 0x00, 0xff ]
    return mpi_header || reversed(octet_list)
```

5.6.6.1.2. X448 ECDH Secret Key Material

An X448 secret key is contained within its MPI as a prefixed octet string (see Section 13.3.2), which encapsulates the native secret key format found in [RFC7748]. The full wire format (as an MPI) will thus be the three octets 01 c7 40 followed by the full 56 octet native secret key.

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

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

5.7. Compressed Data Packet (Tag 8)

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

The body of this packet consists of:

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

A Compressed Data Packet's body contains a block that compresses some set of packets. See Section 11 for details on how messages are formed.

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

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

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

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

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

5.8. Symmetrically Encrypted Data Packet (Tag 9)

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

This packet is obsolete. An implementation MUST NOT create this packet. An implementation MAY process such a packet but it MUST return a clear diagnostic that a non-integrity protected packet has been processed. The implementation SHOULD also return an error in this case and stop processing.

This packet format is impossible to handle safely in general because the ciphertext it provides is malleable. See Section 15.1 about selecting a better OpenPGP encryption container that does not have this flaw.

The body of this packet consists of:

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

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

The data is encrypted in CFB mode, with a CFB shift size equal to the cipher's block size. The Initial Vector (IV) is specified as all zeros. Instead of using an IV, OpenPGP prefixes a string of length equal to the block size of the cipher plus two to the data before it is encrypted. The first block-size octets (for example, 8 octets for a 64-bit block length) are random, and the following two octets are copies of the last two octets of the IV. For example, in an 8-octet block, octet 9 is a repeat of octet 7, and octet 10 is a repeat of octet 8. In a cipher of length 16, octet 17 is a repeat of octet 15 and octet 18 is a repeat of octet 16. As a pedantic clarification, in both these examples, we consider the first octet to be numbered 1.

After encrypting the first block-size-plus-two octets, the CFB state is resynchronized. The last block-size octets of ciphertext are passed through the cipher and the block boundary is reset.

The repetition of 16 bits in the random data prefixed to the message allows the receiver to immediately check whether the session key is incorrect. See Section 15 for hints on the proper use of this "quick check".

5.9. Marker Packet (Tag 10)

The body of this packet consists of:

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

Such a packet MUST be ignored when received.

5.10. Literal Data Packet (Tag 11)

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

The body of this packet consists of:

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

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

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

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

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

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

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

Due to their inherent malleability, an implementation that generates a literal data packet SHOULD avoid storing any significant data in these fields. If the implementation is certain that the data is textual and is encoded with UTF-8 (for example, if it will follow this literal data packet with a signature packet of type 0x01 (see

Section 5.2.1), it MAY set the format octet to u. Otherwise, it SHOULD set the format octet to b. It SHOULD set the filename to the empty string (encoded as a single zero octet), and the timestamp to zero (encoded as four zero octets).

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

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

An implementation that deals with either historic data or data generated by legacy implementations MAY interpret Literal Data packets that use the Legacy format for packet framing.

5.10.1. Special Filename _CONSOLE (Deprecated)

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

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

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

5.11. Trust Packet (Tag 12)

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

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

5.12. User ID Packet (Tag 13)

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

5.13. User Attribute Packet (Tag 17)

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

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

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

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

and is followed by the subpacket specific data.

The following table lists the currently known subpackets:

Type	Attribute Subpacket
1	Image Attribute Subpacket
100-110	Private/Experimental Use

Table 15: User Attribute type registry

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

5.13.1. The Image Attribute Subpacket

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

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

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

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

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

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

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

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

Version 2 of this packet contains data encrypted with an authenticated encryption and additional data (AEAD) construction. This offers a more cryptographically rigorous defense against ciphertext malleability, but may not be as widely supported yet. See Section 15.1 for more details on choosing between these formats.

5.14.1. Version 1 Sym. Encrypted Integrity Protected Data Packet Format

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

- * A one-octet version number with value 1.
- * Encrypted data, the output of the selected symmetric-key cipher operating in Cipher Feedback mode with shift amount equal to the block size of the cipher (CFB-n where n is the block size).

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

The data is encrypted in CFB mode, with a CFB shift size equal to the cipher's block size. The Initial Vector (IV) is specified as all zeros. Instead of using an IV, OpenPGP prefixes an octet string to the data before it is encrypted. The length of the octet string equals the block size of the cipher in octets, plus two. The first octets in the group, of length equal to the block size of the cipher, are random; the last two octets are each copies of their 2nd preceding octet. For example, with a cipher whose block size is 128 bits or 16 octets, the prefix data will contain 16 random octets, then two more octets, which are copies of the 15th and 16th octets, respectively. Unlike the Symmetrically Encrypted Data Packet, no special CFB resynchronization is done after encrypting this prefix data. See Section 14.9 for more details.

The repetition of 16 bits in the random data prefixed to the message allows the receiver to immediately check whether the session key is incorrect.

Two constant octets with the values 0xD3 and 0x14 are appended to the plaintext. Then, the plaintext of the data to be encrypted is passed through the SHA-1 hash function. The input to the hash function includes the prefix data described above; it includes all of the plaintext, including the trailing constant octets 0xD3, 0x14. The 20 octets of the SHA-1 hash are then appended to the plaintext (after the constant octets 0xD3, 0x14) and encrypted along with the plaintext using the same CFB context. This trailing checksum is known as the Modification Detection Code (MDC).

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

NON-NORMATIVE EXPLANATION

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

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

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

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

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

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

Note also that unlike nearly every other OpenPGP subsystem, there are no parameters in the MDC system. It hard-defines SHA-1 as its hash function. This is not an accident. It is an intentional choice to avoid downgrade and cross-grade attacks while making a simple, fast system. (A downgrade attack would be an attack that replaced SHA2-256 with SHA-1, for example. A cross-grade attack would replace SHA-1 with another 160-bit hash, such as RIPEMD/160, for example.)

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

5.14.2. Version 2 Sym. Encrypted Integrity Protected Data Packet Format

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

- * A one-octet version number with value 2.
- * A one-octet cipher algorithm.
- * A one-octet AEAD algorithm.
- * A one-octet chunk size.
- * Thirty-two octets of salt. The salt is used to derive the message key and must be unique.
- * Encrypted data, the output of the selected symmetric-key cipher operating in the given AEAD mode.

* A final, summary authentication tag for the AEAD mode.

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

The KDF mechanism provides key separation between cipher and AEAD algorithms. Furthermore, an implementation can securely reply to a message even if a recipients certificate is unknown by reusing the encrypted session key packets and replying with a different salt yielding a new, unique message key.

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

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

For each chunk, the AEAD construction is given the Packet Tag in OpenPGP format encoding (bits 7 and 6 set, bits 5-0 carry the packet tag), version number, cipher algorithm octet, AEAD algorithm octet, and chunk size octet as additional data. For example, the additional data of the first chunk using EAX and AES-128 with a chunk size of 2**16 octets consists of the octets 0xD2, 0x02, 0x07, 0x01, and 0x10.

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

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

$$\text{chunk_size} = ((\text{uint64_t})1 \ll (c + 6))$$

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

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

5.14.3. EAX Mode

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

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

5.14.4. OCB Mode

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

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

5.14.5. GCM Mode

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

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

5.15. Padding Packet (Tag 21)

The Padding packet contains random data, and can be used to defend against traffic analysis (see Section 15.4) on version 2 SEIPD messages (see Section 5.14.2) and Transferable Public Keys (see Section 11.1).

Such a packet MUST be ignored when received.

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

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

- * At the end of a v5 Transferable Public Key that is transferred over an encrypted channel (see Section 11.1).
- * As the last packet of an Optionally Padded Message within a version 2 Symmetrically Encrypted Integrity Protected Data Packet (see Section 11.3.1).

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

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

6. Radix-64 Conversions

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

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

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

The optional checksum is a 24-bit Cyclic Redundancy Check (CRC) converted to four characters of radix-64 encoding by the same MIME base64 transformation, preceded by an equal sign (=). The CRC is computed by using the generator 0x864CFB and an initialization of 0xB704CE. The accumulation is done on the data before it is converted to radix-64, rather than on the converted data. A sample implementation of this algorithm is in Section 6.1.

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

Rationale for CRC-24: The size of 24 bits fits evenly into printable base64. The nonzero initialization can detect more errors than a zero initialization.

6.1. An Implementation of the CRC-24 in "C"

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

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

6.2. Forming ASCII Armor

When OpenPGP encodes data into ASCII Armor, it puts specific headers around the Radix-64 encoded data, so OpenPGP can reconstruct the data later. An OpenPGP implementation MAY use ASCII armor to protect raw binary data. OpenPGP informs the user what kind of data is encoded in the ASCII armor through the use of the headers.

Concatenating the following data creates ASCII Armor:

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

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

BEGIN PGP MESSAGE

Used for signed, encrypted, or compressed files.

BEGIN PGP PUBLIC KEY BLOCK

Used for armoring public keys.

BEGIN PGP PRIVATE KEY BLOCK

Used for armoring private keys.

BEGIN PGP MESSAGE, PART X/Y

Used for multi-part messages, where the armor is split amongst Y parts, and this is the Xth part out of Y.

BEGIN PGP MESSAGE, PART X

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

BEGIN PGP SIGNATURE

Used for detached signatures, OpenPGP/MIME signatures, and cleartext signatures.

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

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

The format of an Armor Header is that of a key-value pair. A colon (: 0x38) and a single space (0x20) separate the key and value. OpenPGP should consider improperly formatted Armor Headers to be corruption of the ASCII Armor. Unknown keys should be reported to the user, but OpenPGP should continue to process the message.

Note that some transport methods are sensitive to line length. While there is a limit of 76 characters for the Radix-64 data (Section 6.3), there is no limit to the length of Armor Headers. Care should be taken that the Armor Headers are short enough to survive transport. One way to do this is to repeat an Armor Header Key multiple times with different values for each so that no one line is overly long.

Currently defined Armor Header Keys are as follows:

- * "Version", which states the OpenPGP implementation and version used to encode the message. To minimize metadata, implementations SHOULD NOT emit this key and its corresponding value except for debugging purposes with explicit user consent.
- * "Comment", a user-defined comment. OpenPGP defines all text to be in UTF-8. A comment may be any UTF-8 string. However, the whole point of armoring is to provide seven-bit-clean data. Consequently, if a comment has characters that are outside the US-ASCII range of UTF, they may very well not survive transport.
- * "MessageID", a 32-character string of printable characters. The string must be the same for all parts of a multi-part message that uses the "PART X" Armor Header. MessageID strings should be unique enough that the recipient of the mail can associate all the parts of a message with each other. A good checksum or cryptographic hash function is sufficient.

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

- * "Hash", a comma-separated list of hash algorithms used in this message. This is used only in cleartext signed messages.
- * "SaltedHash", a salt and hash algorithm used in this message. This is used only in cleartext signed messages that are followed by a v5 Signature.

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

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

6.3. Encoding Binary in Radix-64

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

```

first octetsecond octetthird octet
7 6 5 4 3 2 1 07 6 5 4 3 2 1 07 6 5 4 3 2 1 0
5 4 3 2 1 05 4 3 2 1 05 4 3 2 1 05 4 3 2 1 0
1.index2.index3.index4.index
```

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

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

Table 16: Encoding for Radix-64

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

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

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

2. The last data group has 16 bits (2 octets). The first two 6-bit groups are processed as above. The third (incomplete) data group has two zero-value bits added to it, and is processed as above. A pad character (=) is added to the output.
3. The last data group has 8 bits (1 octet). The first 6-bit group is processed as above. The second (incomplete) data group has four zero-value bits added to it, and is processed as above. Two pad characters (=) are added to the output.

6.4. Decoding Radix-64

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

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

6.5. Examples of Radix-64

```

Input data: 0x14FB9C03D97E
Hex:      1  4  F  B  9  C      |  0  3  D  9  7  E
8-bit:    00010100 11111011 10011100 | 000000011 11011001 01111110
6-bit:    000101 001111 101110 011100 | 0000000 111101 100101 111110
Decimal:  5      15      46      28    |  0      61      37      62
Output:   F      P      u      c      |  A      9      1      +
Input data: 0x14FB9C03D9
Hex:      1  4  F  B  9  C      |  0  3  D  9
8-bit:    00010100 11111011 10011100 | 000000011 11011001
                                           pad with 00
6-bit:    000101 001111 101110 011100 | 0000000 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 | 000000011
                                           pad with 0000
6-bit:    000101 001111 101110 011100 | 0000000 110000
Decimal:  5      15      46      28    |  0      48
                                           pad with =
Output:   F      P      u      c      |  A      w      =

```

6.6. Example of an ASCII Armored Message

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

```

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

```

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

7. Cleartext Signature Framework

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

The cleartext signed message consists of:

- * The cleartext header -----BEGIN PGP SIGNED MESSAGE----- on a single line,
- * If the message is signed using v3 or v4 Signatures, one or more "Hash" Armor Headers,
- * If the message is signed using v5 Signatures, one or more "SaltedHash" Armor Headers,
- * Exactly one empty line not included into the message digest,
- * The dash-escaped cleartext that is included into the message digest,
- * The ASCII armored signature(s) including the -----BEGIN PGP SIGNATURE----- Armor Header and Armor Tail Lines.

If the "Hash" Armor Header is given, the specified message digest algorithm(s) are used for the signature. If more than one message digest is used in the signature, the "Hash" armor header contains a comma-delimited list of used message digests.

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

Current message digest names are described with the algorithm IDs in Section 9.5.

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

7.1. Dash-Escaped Text

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

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

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

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

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

8. Regular Expressions

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

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

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

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

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

9. Constants

This section describes the constants used in OpenPGP.

Note that these tables are not exhaustive lists; an implementation MAY implement an algorithm not on these lists, so long as the algorithm numbers are chosen from the private or experimental algorithm range.

See Section 14 for more discussion of the algorithms.

9.1. Public-Key Algorithms

ID	Algorithm	Public Key Format	Secret Key Format	Signature Format	PKESK Format
1	RSA (Encrypt or Sign) [HAC]	MPI(n), MPI(e) [Section 5.6.1]	MPI(d), MPI(p), MPI(q), MPI(u)	MPI(m**d mod n) [Section 5.2.3.1]	MPI(m**e mod n) [Section 5.1.3]
2	RSA Encrypt-Only [HAC]	MPI(n), MPI(e) [Section 5.6.1]	MPI(d), MPI(p), MPI(q), MPI(u)	N/A	MPI(m**e mod n) [Section 5.1.3]
3	RSA Sign-Only [HAC]	MPI(n), MPI(e) [Section 5.6.1]	MPI(d), MPI(p), MPI(q), MPI(u)	MPI(m**d mod n) [Section 5.2.3.1]	N/A
16	Elgamal (Encrypt-Only) [ELGAMAL] [HAC]	MPI(p), MPI(g), MPI(y) [Section 5.6.3]	MPI(x)	N/A	MPI(g**k mod p), MPI (m * y**k mod p) [Section 5.1.4]
17	DSA (Digital Signature Algorithm) [FIPS186] [HAC]	MPI(p), MPI(q), MPI(g), MPI(y) [Section 5.6.2]	MPI(x)	MPI(r), MPI(s) [Section 5.2.3.2]	N/A
18	ECDH public key algorithm	OID, MPI(point	MPI(value in curve-	N/A	MPI(point in curve-

		in curve-specific point format), KDFParams [see Section 9.2.1, Section 5.6.6]	specific format) [Section 9.2.1]		specific point format), size octet, encoded key [Section 9.2.1, Section 5.1.5, Section 13.5]
19	ECDSA public key algorithm [FIPS186]	OID, MPI(point in SEC1 format) [Section 5.6.4]	MPI(value)	MPI(r), MPI(s) [Section 5.2.3.2]	N/A
20	Reserved (formerly Elgamal Encrypt or Sign)				
21	Reserved for Diffie-Hellman (X9.42, as defined for IETF-S/MIME)				
22	EdDSA [RFC8032]	OID, MPI(point in prefixed native format) [see Section 13.2.2, Section 5.6.5]	MPI(value in curve-specific format) [see Section 9.2.1]	MPI, MPI [see Section 9.2.1, Section 5.2.3.3]	N/A
23	Reserved (AEDH)				
24	Reserved (AEDSA)				

100	Private/ to Experimental					
110	algorithm					

Table 17: Public-key algorithm registry

Implementations MUST implement EdDSA (19) for signatures, and ECDH (18) for encryption. Implementations SHOULD implement RSA (1) for signatures and encryption.

RSA Encrypt-Only (2) and RSA Sign-Only (3) are deprecated and SHOULD NOT be generated, but may be interpreted. See Section 14.4. See Section 14.8 for notes on Elgamal Encrypt or Sign (20), and X9.42 (21). Implementations MAY implement any other algorithm.

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

A compatible specification of ECDSA is given in [RFC6090] as "KT-I Signatures" and in [SEC1]; ECDH is defined in Section 13.5 of this document.

9.2. ECC Curves for OpenPGP

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

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

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

Table 18: ECC Curve OID and usage registry

The "Field Size (fsize)" column represents the field size of the group in number of octets, rounded up, such that x or y coordinates for a point on the curve, native point representations, or scalars with high enough entropy for the curve can be represented in that many octets.

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

Implementations MUST implement Ed25519 for use with EdDSA, and Curve25519 for use with ECDH. Implementations SHOULD implement Ed448 for use with EdDSA, and X448 for use with ECDH.

9.2.1. Curve-Specific Wire Formats

Some Elliptic Curve Public Key Algorithms use different conventions for specific fields depending on the curve in use. Each field is always formatted as an MPI, but with a curve-specific framing. This table summarizes those distinctions.

Curve	ECDH Point Format	ECDH Secret Key MPI	EdDSA Secret Key MPI	EdDSA Signature first MPI	EdDSA Signature second MPI
NIST P-256	SEC1	integer	N/A	N/A	N/A
NIST P-384	SEC1	integer	N/A	N/A	N/A
NIST P-521	SEC1	integer	N/A	N/A	N/A
Ed25519	N/A	N/A	32 octets of secret	32 octets of R	32 octets of S
Ed448	N/A	N/A	prefixed 57 octets of secret	prefixed 114 octets of signature	0 [this is an unused placeholder]
Curve25519	prefixed native	integer (see Section 5.6.6.1.1)	N/A	N/A	N/A
X448	prefixed native	prefixed 56 octets of secret	N/A	N/A	N/A

Table 19: Curve-specific wire formats

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

9.3. Symmetric-Key Algorithms

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

Table 20: Symmetric-key algorithm registry

Implementations MUST implement AES-128. Implementations SHOULD implement AES-256. Implementations MUST NOT encrypt data with IDEA, TripleDES, or CAST5. Implementations MAY decrypt data that uses IDEA, TripleDES, or CAST5 for the sake of reading older messages or new messages from legacy clients. Implementations MAY implement any other algorithm.

9.4. Compression Algorithms

ID	Algorithm
0	Uncompressed
1	ZIP [RFC1951]
2	ZLIB [RFC1950]
3	BZip2 [BZ2]
100 to 110	Private/Experimental algorithm

Table 21: Compression algorithm registry

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

9.5. Hash Algorithms

ID	Algorithm	Text Name
1	MD5 [HAC]	"MD5"
2	SHA-1 [FIPS180]	"SHA1"
3	RIPE-MD/160 [HAC]	"RIPEMD160"
4	Reserved	
5	Reserved	
6	Reserved	
7	Reserved	

8	SHA2-256 [FIPS180]	"SHA256"
9	SHA2-384 [FIPS180]	"SHA384"
10	SHA2-512 [FIPS180]	"SHA512"
11	SHA2-224 [FIPS180]	"SHA224"
12	SHA3-256 [FIPS202]	"SHA3-256"
13	Reserved	
14	SHA3-512 [FIPS202]	"SHA3-512"
100 to 110	Private/Experimental algorithm	

Table 22: Hash algorithm registry

Implementations MUST implement SHA2-256. Implementations SHOULD implement SHA2-384 and SHA2-512. Implementations MAY implement other algorithms. Implementations SHOULD NOT create messages which require the use of SHA-1 with the exception of computing version 4 key fingerprints and for purposes of the Modification Detection Code (MDC) in version 1 Symmetrically Encrypted Integrity Protected Data packets. Implementations MUST NOT generate signatures with MD5, SHA-1, or RIPE-MD/160. Implementations MUST NOT use MD5, SHA-1, or RIPE-MD/160 as a hash function in an ECDH KDF. Implementations MUST NOT validate any recent signature that depends on MD5, SHA-1, or RIPE-MD/160. Implementations SHOULD NOT validate any old signature that depends on MD5, SHA-1, or RIPE-MD/160 unless the signature's creation date predates known weakness of the algorithm used, and the implementation is confident that the message has been in the secure custody of the user the whole time.

9.6. AEAD Algorithms

ID	Algorithm	IV length (octets)	authentication tag length (octets)
1	EAX [EAX]	16	16
2	OCB [RFC7253]	15	16
3	GCM [SP800-38D]	12	16

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

Table 23: AEAD algorithm registry

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

10. IANA Considerations

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

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

10.1. New String-to-Key Specifier Types

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

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

10.2. New Packets

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

10.2.1. User Attribute Types

The User Attribute packet permits an extensible mechanism for other types of certificate identification. This specification creates a registry of User Attribute types. The registry includes the User Attribute type, the name of the User Attribute, and a reference to the defining specification. The initial values for this registry can be found in Section 5.13. Adding a new User Attribute type MUST be done through the SPECIFICATION REQUIRED method, as described in [RFC8126].

10.2.1.1. Image Format Subpacket Types

Within User Attribute packets, there is an extensible mechanism for other types of image-based User Attributes. This specification creates a registry of Image Attribute subpacket types. The registry includes the Image Attribute subpacket type, the name of the Image Attribute subpacket, and a reference to the defining specification. The initial values for this registry can be found in Section 5.13.1. Adding a new Image Attribute subpacket type MUST be done through the SPECIFICATION REQUIRED method, as described in [RFC8126].

10.2.2. New Signature Subpackets

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

10.2.2.1. Signature Notation Data Subpackets

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

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

10.2.2.2. Signature Notation Data Subpacket Notation Flags

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

10.2.2.3. Key Server Preference Extensions

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

10.2.2.4. Key Flags Extensions

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

10.2.2.5. Reason for Revocation Extensions

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

10.2.2.6. Implementation Features

OpenPGP signatures contain a mechanism for flags to be specified stating which optional features an implementation supports. This specification creates a registry of feature-implementation flags. The registry includes the feature-implementation flags value, the name of the flag, and a reference to the defining specification. The initial values for this registry can be found in Section 5.2.3.29.

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

Also see Section 14.11 for more information about when feature flags are needed.

10.2.3. New Packet Versions

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

10.3. New Algorithms

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

10.3.1. Public-Key Algorithms

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

This document requests IANA register the following new public-key algorithm:

ID	Algorithm	Reference
22	EdDSA public key algorithm	This doc, Section 14.7

Table 24: New public-Key algorithms registered

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

10.3.2. Symmetric-Key Algorithms

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

10.3.3. Hash Algorithms

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

This document requests IANA register the following hash algorithms:

ID	Algorithm	Reference
12	SHA3-256	This doc
13	Reserved	
14	SHA3-512	This doc

Table 25: New hash algorithms registered

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

10.3.4. Compression Algorithms

OpenPGP specifies a number of compression algorithms. This specification creates a registry of compression algorithm identifiers. The registry includes the algorithm name and a reference to the defining specification. The initial values for this registry can be found in Section 9.4. Adding a new compression key algorithm MUST be done through the SPECIFICATION REQUIRED method, as described in [RFC8126].

10.3.5. Elliptic Curve Algorithms

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

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

10.4. Elliptic Curve Point and Scalar Wire Formats

This document requests IANA add a registry of wire formats that represent elliptic curve points. The table's initial headings and values can be found in Section 13.2. Adding a new EC point wire format MUST be done through the SPECIFICATION REQUIRED method, as described in [RFC8126].

This document also requests IANA add a registry of wire formats that represent scalars for use with elliptic curve cryptography. The table's initial headings and values can be found in Section 13.3. Adding a new EC scalar wire format MUST be done through the SPECIFICATION REQUIRED method, as described in [RFC8126].

This document also requests that IANA add a registry mapping curve-specific MPI octet-string encoding conventions for ECDH and EdDSA. The table's initial headings and values can be found in Section 9.2.1. Adding a new elliptic curve algorithm to OpenPGP MUST be done through the SPECIFICATION REQUIRED method, as described in [RFC8126], and requires adding an entry to this table if the curve is to be used with either EdDSA or ECDH.

10.5. Changes to existing registries

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

- * Public Key Format
- * Secret Key Format
- * Signature Format
- * PKESK Format

And populate them with the values found in Section 9.1.

11. Packet Composition

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

11.1. Transferable Public Keys

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

- * One Public-Key packet
- * Zero or more revocation signatures
- * Zero or more User ID packets
- * After each User ID packet, zero or more Signature packets (certifications)
- * Zero or more User Attribute packets
- * After each User Attribute packet, zero or more Signature packets (certifications)

- * Zero or more Subkey packets
- * After each Subkey packet, one Signature packet, plus optionally a revocation
- * An optional Padding packet

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. A transferable public key SHOULD include at least one User ID packet unless storage requirements prohibit this.

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

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

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

After the User ID packet or Attribute packet, there may be zero or more Subkey packets. In general, subkeys are provided in cases where the top-level public key is a signature-only key. However, any V4 or V5 key may have subkeys, and the subkeys may be encryption-only keys, signature-only keys, or general-purpose keys. V3 keys MUST NOT have subkeys.

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

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

The optional trailing Padding packet is a mechanism to defend against traffic analysis (see Section 15.4). For maximum interoperability, if the Public-Key packet is a V4 key, the optional Padding packet SHOULD NOT be present unless the recipient has indicated that they are capable of ignoring it successfully. An implementation that is capable of receiving a transferable public key with a V5 Public-Key primary key MUST be able to accept (and ignore) the trailing optional Padding packet.

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

11.2. Transferable Secret Keys

OpenPGP users may transfer secret keys. The format of a transferable secret key is the same as a transferable public key except that secret-key and secret-subkey packets are used instead of the public key and public-subkey packets. Implementations SHOULD include self-signatures on any User IDs and subkeys, as this allows for a complete public key to be automatically extracted from the transferable secret key. Implementations MAY choose to omit the self-signatures, especially if a transferable public key accompanies the transferable secret key.

11.3. OpenPGP Messages

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

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

Compressed Message :- Compressed Data Packet.

Literal Message :- Literal Data Packet.

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

ESK Sequence :- ESK | ESK Sequence, ESK.

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

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

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

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

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

11.3.1. Unwrapping Encrypted and Compressed Messages

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

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

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

11.3.2. Additional Constraints on Packet Sequences

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

11.3.2.1. Packet Versions in Encrypted Messages

As noted above, an Encrypted Message is a sequence of zero or more PKESKs (Section 5.1) and SKESKs (Section 5.3), followed by an SEIPD (Section 5.14) payload. In some historic data, the payload may be a deprecated SED (Section 5.8) packet instead of SEIPD, though implementations MUST NOT generate SED packets (see Section 15.1). The versions of the preceding ESK packets within an Encrypted Message MUST align with the version of the payload SEIPD packet, as described in this section.

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

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

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

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

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

Table 26: Encrypted Message Packet Version Alignment

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

11.4. Detached Signatures

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

12. Enhanced Key Formats

12.1. Key Structures

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

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

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

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

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

A subkey always has at least one subkey binding signature after it that is issued using the primary key to tie the two keys together. These binding signatures may be in either V3 or V4 format, but SHOULD be V4. Subkeys that can issue signatures MUST have a V4 binding signature due to the REQUIRED embedded primary key binding signature.

In order to create self-signatures (see Section 5.2.3.7), the primary key MUST be an algorithm capable of making signatures (that is, not an encryption-only algorithm). The subkeys may be keys of any type. For example, there may be a single-key RSA key, an EdDSA primary key with an RSA encryption key, or an EdDSA primary key with an ECDH subkey, etc.

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

12.2. Key IDs and Fingerprints

For a V3 key, the eight-octet Key ID consists of the low 64 bits of the public modulus of the RSA key.

The fingerprint of a V3 key is formed by hashing the body (but not the two-octet length) of the MPIs that form the key material (public modulus n , followed by exponent e) with MD5. Note that both V3 keys and MD5 are deprecated.

A V4 fingerprint is the 160-bit SHA-1 hash of the octet 0x99, followed by the two-octet packet length, followed by the entire Public-Key packet starting with the version field. The Key ID is the low-order 64 bits of the fingerprint. Here are the fields of the hash material, with the example of an EdDSA key:

- a.1) 0x99 (1 octet)
- a.2) two-octet, big-endian scalar octet count of (b)-(e)
- b) version number = 4 (1 octet);
- c) timestamp of key creation (4 octets);
- d) algorithm (1 octet): 22 = EdDSA (example);
- e) Algorithm-specific fields.

Algorithm-Specific Fields for EdDSA keys (example):

- e.1) A one-octet size of the following field;
- e.2) The octets representing a curve OID, defined in Section 9.2;
- e.3) An MPI of an EC point representing a public key Q in prefixed native form (see Section 13.2.2).

A V5 fingerprint is the 256-bit SHA2-256 hash of the octet 0x9A, followed by the four-octet packet length, followed by the entire Public-Key packet starting with the version field. The Key ID is the high-order 64 bits of the fingerprint. Here are the fields of the hash material, with the example of an EdDSA key:

a.1) 0x9A (1 octet)

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

b) version number = 5 (1 octet);

c) timestamp of key creation (4 octets);

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

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

f) algorithm-specific fields.

Algorithm-Specific Fields for EdDSA keys (example):

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

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

f.3) An MPI of an EC point representing a public key Q in prefixed native form (see Section 13.2.2).

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

Also note that if V3, V4, and V5 format keys share the same RSA key material, they will have different Key IDs as well as different fingerprints.

Finally, the Key ID and fingerprint of a subkey are calculated in the same way as for a primary key, including the 0x99 (V4 key) or 0x9A (V5 key) as the first octet (even though this is not a valid packet ID for a public subkey).

13. Elliptic Curve Cryptography

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

None of the ECC methods described in this document are allowed with deprecated V3 keys. Refer to [FIPS186], B.4.1, for the method to generate a uniformly distributed ECC private key.

13.1. Supported ECC Curves

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

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

13.2. EC Point Wire Formats

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

Name	Wire Format	Reference
SEC1	0x04 x y	Section 13.2.1
Prefixed native	0x40 native	Section 13.2.2

Table 27: Elliptic Curve Point Wire Formats

13.2.1. SEC1 EC Point Wire Format

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

B = 04 || x || y

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

13.2.2. Prefixed Native EC Point Wire Format

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

$$B = 40 \parallel p$$

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

13.2.3. Notes on EC Point Wire Formats

Given the above definitions, the exact size of the MPI payload for an encoded point is 515 bits for "Curve P-256", 771 for "Curve P-384", 1059 for "Curve P-521", 263 for both "Curve25519" and "Ed25519", 463 for "Ed448", and 455 for "X448". For example, the length of a EdDSA public key for the curve Ed25519 is 263 bits: 7 bits to represent the 0x40 prefix octet and 32 octets for the native value of the public key.

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

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

13.3. EC Scalar Wire Formats

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

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

Type	Description	Reference
integer	An integer, big-endian encoded as a standard OpenPGP MPI	Section 3.2
octet string	An octet string of fixed length, that may be shorter on the wire due to leading zeros being stripped by the MPI encoding, and may need to be zero-padded before usage	Section 13.3.1
prefixed N octets	An octet string of fixed length N, prefixed with octet 0x40 to ensure no leading zero octet	Section 13.3.2

Table 28: Elliptic Curve Scalar Encodings

13.3.1. EC Octet String Wire Format

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

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

To encode X to the wire format, we set the MPI's two-octet bit counter to the value of the highest set bit (bit 26, or 0x001a), and do not transfer the leading all-zero octet to the wire.

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

- * ensure that the MPI's two-octet bitcount is less than or equal to 40 (5 octets of 8 bits)
- * allocate 5 octets, setting all to zero initially
- * copy the MPI data octets (without the two count octets) into the lower octets of the allocated space

13.3.2. Elliptic Curve Prefixed Octet String Wire Format

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

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

To encode the string, we prefix it with the octet 0x40 (whose 7th bit is set), then set the MPI's two-octet bit counter to 47 (0x002f, 7 bits for the prefix octet and 40 bits for the string).

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

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

Note that this is a similar approach to that used in the EC point encodings found in Section 13.2.2.

13.4. Key Derivation Function

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

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

```

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

```

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

13.5. EC DH Algorithm (ECDH)

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

The One-Pass Diffie-Hellman method C(1, 1, ECC CDH) [SP800-56A] MUST be implemented with the following restrictions: the ECC CDH primitive employed by this method is modified to always assume the cofactor is 1, the KDF specified in Section 13.4 is used, and the KDF parameters specified below are used.

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

- * A variable-length field containing a curve OID, which is formatted as follows:
 - A one-octet size of the following field,
 - The octets representing a curve OID defined in Section 9.2;
- * A one-octet public key algorithm ID defined in Section 9.1;
- * A variable-length field containing KDF parameters, which are identical to the corresponding field in the ECDH public key, and are formatted as follows:
 - A one-octet size of the following fields; values 0 and 0xFF are reserved for future extensions,

- A one-octet value 0x01, reserved for future extensions,
 - A one-octet hash function ID used with the KDF,
 - A one-octet algorithm ID for the symmetric algorithm used to wrap the symmetric key for message encryption; see Section 13.5 for details;
- * 20 octets representing the UTF-8 encoding of the string Anonymous Sender , which is the octet sequence 41 6E 6F 6E 79 6D 6F 75 73 20 53 65 6E 64 65 72 20 20 20 20;
- * A variable-length field containing the fingerprint of the recipient encryption subkey or a primary key fingerprint identifying the key material that is needed for decryption. For version 4 keys, this field is 20 octets. For version 5 keys, this field is 32 octets.

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

The key wrapping method is described in [RFC3394]. The KDF produces a symmetric key that is used as a key-encryption key (KEK) as specified in [RFC3394]. Refer to Section 15 for the details regarding the choice of the KEK algorithm, which SHOULD be one of three AES algorithms. Key wrapping and unwrapping is performed with the default initial value of [RFC3394].

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

For example, in a V4 Public-Key Encrypted Session Key packet, the following AES-256 session key, in which 32 octets are denoted from k0 to k31, is composed to form the following 40 octet sequence:

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

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

In a V5 Public-Key Encrypted Session Key packet, the symmetric algorithm is not included, as described in Section 5.1. For example, an AES-256 session key would be composed as follows:

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

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

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

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

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

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

- * Obtain the authenticated recipient public key `R`
- * Generate an ephemeral key pair $\{v, V=vG\}$
- * Compute the shared point $S = vR$;
- * `m = symm_alg_ID || session key || checksum || pkcs5_padding`;

```

* curve_OID_len = (octet)len(curve_OID);

* Param = curve_OID_len || curve_OID || public_key_alg_ID || 03 ||
  01 || KDF_hash_ID || KEK_alg_ID for AESKeyWrap || Anonymous
  Sender      || recipient_fingerprint;

* Z_len = the key size for the KEK_alg_ID used with AESKeyWrap

* Compute Z = KDF( S, Z_len, Param );

* Compute C = AESKeyWrap( Z, m ) as per [RFC3394]

* VB = convert point V to the octet string

* Output (MPI(VB) || len(C) || C).

```

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

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

Consistent with Section 5.14, AEAD encryption or a Modification Detection Code (MDC) MUST be used anytime the symmetric key is protected by ECDH.

14. Notes on Algorithms

14.1. PKCS#1 Encoding in OpenPGP

This standard makes use of the PKCS#1 functions EME-PKCS1-v1_5 and EMSA-PKCS1-v1_5. However, the calling conventions of these functions has changed in the past. To avoid potential confusion and interoperability problems, we are including local copies in this document, adapted from those in PKCS#1 v2.1 [RFC8017]. [RFC8017] should be treated as the ultimate authority on PKCS#1 for OpenPGP. Nonetheless, we believe that there is value in having a self-contained document that avoids problems in the future with needed changes in the conventions.

14.1.1. EME-PKCS1-v1_5-ENCODE

Input:

k = the length in octets of the key modulus.

M = message to be encoded, an octet string of length $mLen$, where $mLen \leq k - 11$.

Output:

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

Error: "message too long".

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

$$EM = 0x00 \parallel 0x02 \parallel PS \parallel 0x00 \parallel M.$$

4. Output EM.

14.1.2. EME-PKCS1-v1_5-DECODE

Input:

EM = encoded message, an octet string

Output:

M = message, an octet string.

Error: "decryption error".

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

$$EM = 0x00 \parallel 0x02 \parallel PS \parallel 0x00 \parallel M.$$

If the first octet of EM does not have hexadecimal value 0x00, if the second octet of EM does not have hexadecimal value 0x02, if there is no octet with hexadecimal value 0x00 to separate PS from M, or if the length of PS is less than 8 octets, output "decryption error" and stop. See also the security note in Section 15 regarding differences in reporting between a decryption error and a padding error.

14.1.1.3. EMSA-PKCS1-v1_5

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

Option:

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

Input:

M = message to be encoded.

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

Output:

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

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

Steps:

1. Apply the hash function to the message M to produce a hash value H:
$$H = \text{Hash}(M) .$$

If the hash function outputs "message too long," output "message too long" and stop.
2. Using the list in Section 5.2.2, produce an ASN.1 DER value for the hash function used. Let T be the full hash prefix from the list, and let tLen be the length in octets of T.
3. If $\text{emLen} < \text{tLen} + 11$, output "intended encoded message length too short" and stop.
4. Generate an octet string PS consisting of $\text{emLen} - \text{tLen} - 3$ octets with hexadecimal value 0xFF. The length of PS will be at least 8 octets.
5. Concatenate PS, the hash prefix T, and other padding to form the encoded message EM as

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

6. Output EM.

14.2. Symmetric Algorithm Preferences

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

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

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

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

14.2.1. Plaintext

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

14.3. Other Algorithm Preferences

Other algorithm preferences work similarly to the symmetric algorithm preference, in that they specify which algorithms the keyholder accepts. There are two interesting cases that other comments need to be made about, though, the compression preferences and the hash preferences.

14.3.1. Compression Preferences

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

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

14.3.1.1. Uncompressed

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

14.3.2. Hash Algorithm Preferences

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

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

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

14.4. RSA

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

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

14.5. DSA

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

- * 1024-bit key, 160-bit q , SHA-1, SHA2-224, SHA2-256, SHA2-384, or SHA2-512 hash
- * 2048-bit key, 224-bit q , SHA2-224, SHA2-256, SHA2-384, or SHA2-512 hash
- * 2048-bit key, 256-bit q , SHA2-256, SHA2-384, or SHA2-512 hash
- * 3072-bit key, 256-bit q , SHA2-256, SHA2-384, or SHA2-512 hash

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

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

14.6. Elgamal

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

14.7. EdDSA

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

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

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

14.8. Reserved Algorithm Numbers

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

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

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

14.9. OpenPGP CFB Mode

When using a version 1 Symmetrically Encrypted Integrity Protected Data packet (Section 5.14.1) or --- for historic data --- a Symmetrically Encrypted Data packet (Section 5.8), OpenPGP does symmetric encryption using a variant of Cipher Feedback mode (CFB mode). This section describes the procedure it uses in detail. This mode is what is used for Symmetrically Encrypted Integrity Protected Data Packets (and the dangerously malleable --- and deprecated --- Symmetrically Encrypted Data Packets). Some mechanisms for encrypting secret-key material also use CFB mode, as described in Section 3.7.2.1.

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

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

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

Step by step, here is the procedure:

1. The feedback register (FR) is set to the IV, which is all zeros.
2. FR is encrypted to produce FRE (FR Encrypted). This is the encryption of an all-zero value.
3. FRE is xored with the first BS octets of random data prefixed to the plaintext to produce C[1] through C[BS], the first BS octets of ciphertext.
4. FR is loaded with C[1] through C[BS].
5. FR is encrypted to produce FRE, the encryption of the first BS octets of ciphertext.
6. The left two octets of FRE get xored with the next two octets of data that were prefixed to the plaintext. This produces C[BS+1] and C[BS+2], the next two octets of ciphertext.
7. (The resynchronization step) FR is loaded with C[3] through C[BS+2].
8. FR is encrypted to produce FRE.
9. FRE is xored with the first BS octets of the given plaintext, now that we have finished encrypting the BS+2 octets of prefixed data. This produces C[BS+3] through C[BS+(BS+2)], the next BS octets of ciphertext.

10. FR is loaded with C[BS+3] to C[BS + (BS+2)] (which is C11-C18 for an 8-octet block).
11. FR is encrypted to produce FRE.
12. FRE is xored with the next BS octets of plaintext, to produce the next BS octets of ciphertext. These are loaded into FR, and the process is repeated until the plaintext is used up.

14.10. Private or Experimental Parameters

S2K specifiers, Signature subpacket types, User Attribute types, image format types, and algorithms described in Section 9 all reserve the range 100 to 110 for private and experimental use. Packet types reserve the range 60 to 63 for private and experimental use. These are intentionally managed with the PRIVATE USE method, as described in [RFC8126].

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

14.11. Meta-Considerations for Expansion

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

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

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

15. Security Considerations

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

- * This specification uses Public-Key Cryptography technologies. It is assumed that the private key portion of a public-private key pair is controlled and secured by the proper party or parties.
- * The MD5 hash algorithm has been found to have weaknesses, with collisions found in a number of cases. MD5 is deprecated for use in OpenPGP. Implementations MUST NOT generate new signatures using MD5 as a hash function. They MAY continue to consider old signatures that used MD5 as valid.
- * SHA2-224 and SHA2-384 require the same work as SHA2-256 and SHA2-512, respectively. In general, there are few reasons to use them outside of DSS compatibility. You need a situation where one needs more security than smaller hashes, but does not want to have the full 256-bit or 512-bit data length.
- * Many security protocol designers think that it is a bad idea to use a single key for both privacy (encryption) and integrity (signatures). In fact, this was one of the motivating forces behind the V4 key format with separate signature and encryption keys. If you as an implementer promote dual-use keys, you should at least be aware of this controversy.
- * The DSA algorithm will work with any hash, but is sensitive to the quality of the hash algorithm. Verifiers should be aware that even if the signer used a strong hash, an attacker could have modified the signature to use a weak one. Only signatures using acceptably strong hash algorithms should be accepted as valid.
- * As OpenPGP combines many different asymmetric, symmetric, and hash algorithms, each with different measures of strength, care should be taken that the weakest element of an OpenPGP message is still sufficiently strong for the purpose at hand. While consensus about the strength of a given algorithm may evolve, NIST Special Publication 800-57 [SP800-57] recommends the following list of equivalent strengths:

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

Table 29: Key length equivalences

- * There is a somewhat-related potential security problem in signatures. If an attacker can find a message that hashes to the same hash with a different algorithm, a bogus signature structure can be constructed that evaluates correctly.

For example, suppose Alice DSA signs message M using hash algorithm H. Suppose that Mallet finds a message M' that has the same hash value as M with H'. Mallet can then construct a signature block that verifies as Alice's signature of M' with H'. However, this would also constitute a weakness in either H or H' or both. Should this ever occur, a revision will have to be made to this document to revise the allowed hash algorithms.

- * If you are building an authentication system, the recipient may specify a preferred signing algorithm. However, the signer would be foolish to use a weak algorithm simply because the recipient requests it.
- * Some of the encryption algorithms mentioned in this document have been analyzed less than others. For example, although CAST5 is presently considered strong, it has been analyzed less than TripleDES. Other algorithms may have other controversies surrounding them.

- * In late summer 2002, Jallad, Katz, and Schneier published an interesting attack on older versions of the OpenPGP protocol and some of its implementations [JKS02]. In this attack, the attacker modifies a message and sends it to a user who then returns the erroneously decrypted message to the attacker. The attacker is thus using the user as a random oracle, and can often decrypt the message. This attack is a particular form of ciphertext malleability. See Section 15.1 for information on how to defend against such an attack using more recent versions of OpenPGP.
- * PKCS#1 has been found to be vulnerable to attacks in which a system that reports errors in padding differently from errors in decryption becomes a random oracle that can leak the private key in mere millions of queries. Implementations must be aware of this attack and prevent it from happening. The simplest solution is to report a single error code for all variants of decryption errors so as not to leak information to an attacker.
- * Some technologies mentioned here may be subject to government control in some countries.
- * In winter 2005, Serge Mister and Robert Zuccherato from Entrust released a paper describing a way that the "quick check" in OpenPGP CFB mode can be used with a random oracle to decrypt two octets of every cipher block [MZ05]. They recommend as prevention not using the quick check at all.

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

On the one hand, there is a danger to using it if there is a random oracle that can leak information to an attacker. In plainer language, there is a danger to using the quick check if timing information about the check can be exposed to an attacker, particularly via an automated service that allows rapidly repeated queries.

On the other hand, it is inconvenient to the user to be informed that they typed in the wrong passphrase only after a petabyte of data is decrypted. There are many cases in cryptographic engineering where the implementer must use care and wisdom, and this is one.

- * An implementation SHOULD only use an AES algorithm as a KEK algorithm, since backward compatibility of the ECDH format is not a concern. The KEK algorithm is only used within the scope of a Public-Key Encrypted Session Key Packet, which represents an ECDH key recipient of a message. Compare this with the algorithm used for the session key of the message, which MAY be different from a KEK algorithm.

Side channel attacks are a concern when a compliant application's use of the OpenPGP format can be modeled by a decryption or signing oracle, for example, when an application is a network service performing decryption to unauthenticated remote users. ECC scalar multiplication operations used in ECDSA and ECDH are vulnerable to side channel attacks. Countermeasures can often be taken at the higher protocol level, such as limiting the number of allowed failures or time-blinding of the operations associated with each network interface. Mitigations at the scalar multiplication level seek to eliminate any measurable distinction between the ECC point addition and doubling operations.

- * V5 signatures include a 128 bit salt that is hashed first. This makes OpenPGP signatures non-deterministic and protects against a broad class of attacks that depend on creating a signature over a predictable message. Hashing the salt first means that there is no attacker controlled hashed prefix. An example of this kind of attack is described in the paper [SHA-1 Is A Shambles](#) (see [SHAMBLES]), which leverages a chosen prefix collision attack against SHA-1.

15.1. Avoiding Ciphertext Malleability

If ciphertext can be modified by an attacker but still subsequently decrypted to some new plaintext, it is considered "malleable". A number of attacks can arise in any cryptosystem that uses malleable encryption, so modern OpenPGP offers mechanisms to defend against it. However, legacy OpenPGP data may have been created before these mechanisms were available. Because OpenPGP implementations deal with historic stored data, they may encounter malleable ciphertexts.

When an OpenPGP implementation discovers that it is decrypting data that appears to be malleable, it MUST indicate a clear error message that the integrity of the message is suspect, SHOULD NOT release decrypted data to the user, and SHOULD halt with an error. An implementation that encounters malleable ciphertext MAY choose to release cleartext to the user if it is known to be dealing with historic archived legacy data, and the user is aware of the risks.

Any of the following OpenPGP data elements indicate that malleable ciphertext is present:

- * all Symmetrically Encrypted Data packets (Section 5.8).
- * within any encrypted container, any Compressed Data packet (Section 5.7) where there is a decompression failure.
- * any version 1 Symmetrically Encrypted Integrity Protected Data packet (Section 5.14.1) where the internal Modification Detection Code does not validate.
- * any version 2 Symmetrically Encrypted Integrity Protected Data packet (Section 5.14.2) where the authentication tag of any chunk fails, or where there is no final zero-octet chunk.
- * any Secret Key packet with encrypted secret key material (Section 3.7.2.1) where there is an integrity failure, based on the value of the secret key protection octet:
 - value 255 or raw cipher algorithm: where the trailing 2-octet checksum does not match.
 - value 254: where the SHA1 checksum is mismatched.
 - value 253: where the AEAD authentication tag is invalid.

To avoid these circumstances, an implementation that generates OpenPGP encrypted data SHOULD select the encrypted container format with the most robust protections that can be handled by the intended recipients. In particular:

- * The SED packet is deprecated, and MUST NOT be generated.
- * When encrypting to one or more public keys:
 - all recipient keys indicate support for version 2 of the Symmetrically Encrypted Integrity Protected Data packet in their Features subpacket (Section 5.2.3.29), or are v5 keys without a Features subpacket, or the implementation can otherwise infer that all recipients support v2 SEIPD packets, the implementation MUST encrypt using a v2 SEIPD packet.
 - If one of the recipients does not support v2 SEIPD packets, then the message generator MAY use a v1 SEIPD packet instead.

- * Password-protected secret key material in a V5 Secret Key or V5 Secret Subkey packet SHOULD be protected with AEAD encryption (S2K usage octet 253) unless it will be transferred to an implementation that is known to not support AEAD.

Implementers should implement AEAD (v2 SEIPD and S2K usage octet 253) promptly and encourage its spread.

Users should migrate to AEAD with all due speed.

15.2. Escrowed Revocation Signatures

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

The preferred way for her to do this is produce a specific Revocation Signature (signature types 0x20, 0x28, or 0x30) and distribute it securely to her preferred revoker who can hold it in escrow. The preferred revoker can then publish the escrowed Revocation Signature at whatever time is deemed appropriate, rather than generating a revocation signature themselves.

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

- * The keyholder can constrain what types of revocation the preferred revoker can issue, by only escrowing those specific signatures.
- * There is no public/visible linkage between the keyholder and the preferred revoker.
- * Third parties can verify the revocation without needing to find the key of the preferred revoker.
- * The preferred revoker doesn't even need to have a public OpenPGP key if some other secure transport is possible between them and the keyholder.
- * Implementation support for enforcing a revocation from an authorized Revocation Key subpacket is uneven and unreliable.
- * If the fingerprint mechanism suffers a cryptanalytic flaw, the escrowed Revocation Signature is not affected.

A Revocation Signature may also be split up into shares and distributed among multiple parties, requiring some subset of those parties to collaborate before the escrowed Revocation Signature is recreated.

15.3. Random Number Generation and Seeding

OpenPGP requires a cryptographically secure pseudorandom number generator (CSPRNG). In most cases, the operating system provides an appropriate facility such as a `getrandom()` syscall, which should be used absent other (for example, performance) concerns. It is RECOMMENDED to use an existing CSPRNG implementation in preference to crafting a new one. Many adequate cryptographic libraries are already available under favorable license terms. Should those prove unsatisfactory, [RFC4086] provides guidance on the generation of random values.

OpenPGP uses random data with three different levels of visibility:

- * in publicly-visible fields such as nonces, IVs, public padding material, or salts,
- * in shared-secret values, such as session keys for encrypted data or padding material within an encrypted packet, and
- * in entirely private data, such as asymmetric key generation.

With a properly functioning CSPRNG, this does not present a security problem, as it is not feasible to determine the CSPRNG state from its output. However, with a broken CSPRNG, it may be possible for an attacker to use visible output to determine the CSPRNG internal state and thereby predict less-visible data like keying material, as documented in [CHECKOWAY].

An implementation can provide extra security against this form of attack by using separate CSPRNGs to generate random data with different levels of visibility.

15.4. Traffic Analysis

When sending OpenPGP data through the network, the size of the data may leak information to an attacker. There are circumstances where such a leak could be unacceptable from a security perspective.

For example, if possible cleartext messages for a given protocol are known to be either yes (three octets) and no (two octets) and the messages are sent within a Symmetrically-Encrypted Integrity Protected Data packet, the length of the encrypted message will reveal the contents of the cleartext.

In another example, sending an OpenPGP Transferable Public Key over an encrypted network connection might reveal the length of the certificate. Since the length of an OpenPGP certificate varies based on the content, an external observer interested in metadata (who is trying to contact who) may be able to guess the identity of the certificate sent, if its length is unique.

In both cases, an implementation can adjust the size of the compound structure by including a Padding packet (see Section 5.15).

16. Implementation Nits

This section is a collection of comments to help an implementer, particularly with an eye to backward compatibility. Often the differences are small, but small differences are frequently more vexing than large differences. Thus, this is a non-comprehensive list of potential problems and gotchas for a developer who is trying to be backward-compatible.

- * There are many ways possible for two keys to have the same key material, but different fingerprints (and thus Key IDs). For example, since a V4 fingerprint is constructed by hashing the key creation time along with other things, two V4 keys created at different times, yet with the same key material will have different fingerprints.
- * OpenPGP does not put limits on the size of public keys. However, larger keys are not necessarily better keys. Larger keys take more computation time to use, and this can quickly become impractical. Different OpenPGP implementations may also use different upper bounds for public key sizes, and so care should be taken when choosing sizes to maintain interoperability.
- * ASCII armor is an optional feature of OpenPGP. The OpenPGP working group strives for a minimal set of mandatory-to-implement features, and since there could be useful implementations that only use binary object formats, this is not a "MUST" feature for an implementation. For example, an implementation that is using OpenPGP as a mechanism for file signatures may find ASCII armor unnecessary. OpenPGP permits an implementation to declare what features it does and does not support, but ASCII armor is not one of these. Since most implementations allow binary and armored

objects to be used indiscriminately, an implementation that does not implement ASCII armor may find itself with compatibility issues with general-purpose implementations. Moreover, implementations of OpenPGP-MIME [RFC3156] already have a requirement for ASCII armor so those implementations will necessarily have support.

- * What this document calls Legacy packet format Section 4.2.2 is what older documents called the "old packet format". It is the packet format of the legacy PGP 2 implementation. Older RFCs called the current OpenPGP packet format Section 4.2.1 the "new packet format".

17. References

17.1. Normative References

- [AES] NIST, "FIPS PUB 197, Advanced Encryption Standard (AES)", November 2001,
<<http://csrc.nist.gov/publications/fips/fips197/fips-197.{ps,pdf}>>.
- [BLOWFISH] Schneier, B., "Description of a New Variable-Length Key, 64-Bit Block Cipher (Blowfish)", Fast Software Encryption, Cambridge Security Workshop Proceedings Springer-Verlag, 1994, pp191-204, December 1993,
<<http://www.counterpane.com/bfsverlag.html>>.
- [BZ2] Seward, J., "The Bzip2 and libbzip2 home page", 2010,
<<http://www.bzip.org/>>.
- [EAX] Bellare, M., Rogaway, P., and D. Wagner, "A Conventional Authenticated-Encryption Mode", April 2003.
- [ELGAMAL] Elgamal, T., "A Public-Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms", IEEE Transactions on Information Theory v. IT-31, n. 4, 1985, pp. 469-472, 1985.
- [FIPS180] National Institute of Standards and Technology, U.S. Department of Commerce, "Secure Hash Standard (SHS), FIPS 180-4", August 2015,
<<http://dx.doi.org/10.6028/NIST.FIPS.180-4>>.
- [FIPS186] National Institute of Standards and Technology, U.S. Department of Commerce, "Digital Signature Standard (DSS), FIPS 186-4", July 2013,
<<http://dx.doi.org/10.6028/NIST.FIPS.186-4>>.

- [FIPS202] National Institute of Standards and Technology, U.S. Department of Commerce, "SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, FIPS 202", August 2015, <<http://dx.doi.org/10.6028/NIST.FIPS.202>>.
- [HAC] Menezes, A.J., Oorschot, P.v., and S. Vanstone, "Handbook of Applied Cryptography", 1996.
- [IDEA] Lai, X., "On the design and security of block ciphers", ETH Series in Information Processing, J.L. Massey (editor) Vol. 1, Hartung-Gorre Verlag Konstanz, Technische Hochschule (Zurich), 1992.
- [ISO10646] International Organization for Standardization, "Information Technology - Universal Multiple-octet coded Character Set (UCS) - Part 1: Architecture and Basic Multilingual Plane", ISO Standard 10646-1, May 1993.
- [JFIF] CA, E.H.M., "JPEG File Interchange Format (Version 1.02).", September 1996.
- [PKCS5] RSA Laboratories, "PKCS #5 v2.0: Password-Based Cryptography Standard", 25 March 1999.
- [RFC1950] Deutsch, P. and J-L. Gailly, "ZLIB Compressed Data Format Specification version 3.3", RFC 1950, DOI 10.17487/RFC1950, May 1996, <<https://www.rfc-editor.org/info/rfc1950>>.
- [RFC1951] Deutsch, P., "DEFLATE Compressed Data Format Specification version 1.3", RFC 1951, DOI 10.17487/RFC1951, May 1996, <<https://www.rfc-editor.org/info/rfc1951>>.
- [RFC2045] Freed, N. and N. Borenstein, "Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies", RFC 2045, DOI 10.17487/RFC2045, November 1996, <<https://www.rfc-editor.org/info/rfc2045>>.
- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC2144] Adams, C., "The CAST-128 Encryption Algorithm", RFC 2144, DOI 10.17487/RFC2144, May 1997, <<https://www.rfc-editor.org/info/rfc2144>>.

- [RFC2822] Resnick, P., Ed., "Internet Message Format", RFC 2822, DOI 10.17487/RFC2822, April 2001, <<https://www.rfc-editor.org/info/rfc2822>>.
- [RFC3156] Elkins, M., Del Torto, D., Levien, R., and T. Roessler, "MIME Security with OpenPGP", RFC 3156, DOI 10.17487/RFC3156, August 2001, <<https://www.rfc-editor.org/info/rfc3156>>.
- [RFC3394] Schaad, J. and R. Housley, "Advanced Encryption Standard (AES) Key Wrap Algorithm", RFC 3394, DOI 10.17487/RFC3394, September 2002, <<https://www.rfc-editor.org/info/rfc3394>>.
- [RFC3629] Yergeau, F., "UTF-8, a transformation format of ISO 10646", STD 63, RFC 3629, DOI 10.17487/RFC3629, November 2003, <<https://www.rfc-editor.org/info/rfc3629>>.
- [RFC3713] Matsui, M., Nakajima, J., and S. Moriai, "A Description of the Camellia Encryption Algorithm", RFC 3713, DOI 10.17487/RFC3713, April 2004, <<https://www.rfc-editor.org/info/rfc3713>>.
- [RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <<https://www.rfc-editor.org/info/rfc4086>>.
- [RFC7253] Krovetz, T. and P. Rogaway, "The OCB Authenticated-Encryption Algorithm", RFC 7253, DOI 10.17487/RFC7253, May 2014, <<https://www.rfc-editor.org/info/rfc7253>>.
- [RFC7748] Langley, A., Hamburg, M., and S. Turner, "Elliptic Curves for Security", RFC 7748, DOI 10.17487/RFC7748, January 2016, <<https://www.rfc-editor.org/info/rfc7748>>.
- [RFC8017] Moriarty, K., Ed., Kaliski, B., Jonsson, J., and A. Rusch, "PKCS #1: RSA Cryptography Specifications Version 2.2", RFC 8017, DOI 10.17487/RFC8017, November 2016, <<https://www.rfc-editor.org/info/rfc8017>>.
- [RFC8032] Josefsson, S. and I. Liusvaara, "Edwards-Curve Digital Signature Algorithm (EdDSA)", RFC 8032, DOI 10.17487/RFC8032, January 2017, <<https://www.rfc-editor.org/info/rfc8032>>.

- [RFC8126] Cotton, M., Leiba, B., and T. Narten, "Guidelines for Writing an IANA Considerations Section in RFCs", BCP 26, RFC 8126, DOI 10.17487/RFC8126, June 2017, <<https://www.rfc-editor.org/info/rfc8126>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC9106] Biryukov, A., Dinu, D., Khovratovich, D., and S. Josefsson, "Argon2 Memory-Hard Function for Password Hashing and Proof-of-Work Applications", RFC 9106, DOI 10.17487/RFC9106, September 2021, <<https://www.rfc-editor.org/info/rfc9106>>.
- [SCHNEIER] Schneier, B., "Applied Cryptography Second Edition: protocols, algorithms, and source code in C", 1996.
- [SP800-38D] Dworkin, M., "Recommendation for Block Cipher Modes of Operation: Galois/Counter Mode (GCM) and GMAC", NIST Special Publication 800-38D, November 2007.
- [SP800-56A] Barker, E., Johnson, D., and M. Smid, "Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography", NIST Special Publication 800-56A Revision 1, March 2007.
- [TWOFISH] Schneier, B., Kelsey, J., Whiting, D., Wagner, D., Hall, C., and N. Ferguson, "The Twofish Encryption Algorithm", 1999.

17.2. Informative References

- [BLEICHENBACHER] Bleichenbacher, D., "Generating ElGamal Signatures Without Knowing the Secret Key", Lecture Notes in Computer Science Volume 1070, pp. 10-18, 1996.
- [CHECKOWAY] Checkoway, S., Maskiewicz, J., Garman, C., Fried, J., Cohn, S., Green, M., Heninger, N., Weinmann, R., Rescorla, E., and H. Shacham, "A Systematic Analysis of the Juniper Dual EC Incident", Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, DOI 10.1145/2976749.2978395, October 2016, <<https://doi.org/10.1145/2976749.2978395>>.

- [JKS02] Jallad, K., Katz, J., and B. Schneier, "Implementation of Chosen-Ciphertext Attacks against PGP and GnuPG", 2002, <<http://www.counterpane.com/pgp-attack.html>>.
- [KOBLITZ] Koblitz, N., "A course in number theory and cryptography, Chapter VI. Elliptic Curves", ISBN 0-387-96576-9, 1997.
- [MZ05] Mister, S. and R. Zuccherato, "An Attack on CFB Mode Encryption As Used By OpenPGP", IACR ePrint Archive Report 2005/033, 8 February 2005, <<http://eprint.iacr.org/2005/033>>.
- [PAX] The Open Group, "IEEE Standard for Information Technology--Portable Operating System Interface (POSIX(R)) Base Specifications, Issue 7: pax - portable archive interchange", IEEE Standard 1003.1-2017, DOI 10.1109/IEEESTD.2018.8277153, 2018, <<https://pubs.opengroup.org/onlinepubs/9699919799/utilities/pax.html>>.
- [REGEX] Friedl, J., "Mastering Regular Expressions", ISBN 0-596-00289-0, August 2002.
- [RFC1991] Atkins, D., Stallings, W., and P. Zimmermann, "PGP Message Exchange Formats", RFC 1991, DOI 10.17487/RFC1991, August 1996, <<https://www.rfc-editor.org/info/rfc1991>>.
- [RFC2440] Callas, J., Donnerhacke, L., Finney, H., and R. Thayer, "OpenPGP Message Format", RFC 2440, DOI 10.17487/RFC2440, November 1998, <<https://www.rfc-editor.org/info/rfc2440>>.
- [RFC4880] Callas, J., Donnerhacke, L., Finney, H., Shaw, D., and R. Thayer, "OpenPGP Message Format", RFC 4880, DOI 10.17487/RFC4880, November 2007, <<https://www.rfc-editor.org/info/rfc4880>>.
- [RFC5869] Krawczyk, H. and P. Eronen, "HMAC-based Extract-and-Expand Key Derivation Function (HKDF)", RFC 5869, DOI 10.17487/RFC5869, May 2010, <<https://www.rfc-editor.org/info/rfc5869>>.
- [RFC6090] McGrew, D., Igoe, K., and M. Salter, "Fundamental Elliptic Curve Cryptography Algorithms", RFC 6090, DOI 10.17487/RFC6090, February 2011, <<https://www.rfc-editor.org/info/rfc6090>>.
- [SEC1] Standards for Efficient Cryptography Group, "SEC 1: Elliptic Curve Cryptography", September 2000.

[SHAMBLES] Leurent, G. and T. Peyrin, "Sha-1 is a shambles: First chosen-prefix collision on sha-1 and application to the PGP web of trust", 2020, <<https://sha-mbles.github.io/>>.

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

Appendix A. Test vectors

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

A.1. Sample EdDSA key

The secret key used for this example is:

D: 1a8b1ff05ded48e18bf50166c664ab023ea70003d78d9e41f5758a91d850f8d2

Note that this is the raw secret key used as input to the EdDSA signing operation. The key was created on 2014-08-19 14:28:27 and thus the fingerprint of the OpenPGP key is:

C959 BDBA FA32 A2F8 9A15 3B67 8CFD E121 9796 5A9A

The algorithm specific input parameters without the MPI length headers are:

oid: 2b06010401da470f01

q: 403f098994bdd916ed4053197934e4a87c80733a1280d62f8010992e43ee3b2406

The entire public key packet is thus:

98 33 04 53 f3 5f 0b 16 09 2b 06 01 04 01 da 47
0f 01 01 07 40 3f 09 89 94 bd d9 16 ed 40 53 19
79 34 e4 a8 7c 80 73 3a 12 80 d6 2f 80 10 99 2e
43 ee 3b 24 06

A.2. Sample EdDSA signature

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

m: 4f70656e504750040016080006050255f95f9504ff0000000c

Using the SHA2-256 hash algorithm yields the digest:

d: f6220a3f757814f4c2176ffbb68b00249cd4ccdc059c4b34ad871f30b1740280

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

r: 56f90cca98e2102637bd983fdb16c131dfd27ed82bf4dde5606e0d756aed3366

s: d09c4fa11527f038e0f57f2201d82f2ea2c9033265fa6ceb489e854bae61b404

The entire signature packet is thus:

```
88 5e 04 00 16 08 00 06 05 02 55 f9 5f 95 00 0a
09 10 8c fd e1 21 97 96 5a 9a f6 22 01 00 56 f9
0c ca 98 e2 10 26 37 bd 98 3f db 16 c1 31 df d2
7e d8 2b f4 dd e5 60 6e 0d 75 6a ed 33 66 01 00
d0 9c 4f a1 15 27 f0 38 e0 f5 7f 22 01 d8 2f 2e
a2 c9 03 32 65 fa 6c eb 48 9e 85 4b ae 61 b4 04
```

A.3. Sample AEAD-EAX encryption and decryption

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

A.3.1. Sample Parameters

S2K:

Iterated and Salted S2K

Iterations:

65011712 (255), SHA2-256

Salt:

a5 ae 57 9d 1f c5 d8 2b

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

Packet header:

c3 40

Version, algorithms, S2K fields:

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

Nonce:

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

Encrypted session key and AEAD tag:

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

A.3.3. Starting AEAD-EAX decryption of the session key

The derived key is:

15 49 67 e5 90 aa 1f 92 3e 1c 0a c6 4c 88 f2 3d

HKDF info:

c3 05 07 01

HKDF output:

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

Authenticated Data:

c3 05 07 01

Nonce:

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

Decrypted session key:

38 81 ba fe 98 54 12 45 9b 86 c3 6f 98 cb 9a 5e

A.3.4. Sample v2 SEIPD packet

Packet header:

d2 69

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

02 07 01 06

Salt:

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

Chunk #0 encrypted data:

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

Chunk #0 authentication tag:

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

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

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

A.3.5. Decryption of data

Starting AEAD-EAX decryption of data, using the session key.

HKDF info:

```
d2 02 07 01 06
```

HKDF output:

```
b5 04 22 ac 1c 26 be 9d dd 83 1d 5b bb 36 b6 4f
78 b8 33 f2 e9 4a 60 c0
```

Message key:

```
b5 04 22 ac 1c 26 be 9d dd 83 1d 5b bb 36 b6 4f
```

Initialization vector:

```
78 b8 33 f2 e9 4a 60 c0
```

Chunk #0:

Nonce:

```
78 b8 33 f2 e9 4a 60 c0 00 00 00 00 00 00 00 00
```

Additional authenticated data:

d2 02 07 01 06

Decrypted chunk #0.

Literal data packet with the string contents Hello, world!:

cb 13 62 00 00 00 00 00 48 65 6c 6c 6f 2c 20 77
6f 72 6c 64 21

Padding packet:

d5 0e ae 5b f0 cd 67 05 50 03 55 81 6c b0 c8 ff

Authenticating final tag:

Final nonce:

78 b8 33 f2 e9 4a 60 c0 00 00 00 00 00 00 00 01

Final additional authenticated data:

d2 02 07 01 06 00 00 00 00 00 00 00 25

A.3.6. Complete AEAD-EAX encrypted packet sequence

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

w0AFHgCbcwMIpa5XnR/F2Cv/aSJPkZmTslBvo7WaanPP+Np0a4jjV+iuVOuH4dcF
ddcvYCMpkFI+mlkJSSJAa+HD0mkCBwEGn/kOOzIZZPOkKRPI3MZhkyUBUifvt+rq
pJ8EwuZ0F1lKPSJulq/LnKmsEiwUCOEcy9TAqyQcapOK1Iv5mlqZuQu6gyXeYQR1
QCWkt5Wala0FHdqW6xVDHf719eIlXKeCYVRuM5o=
=wG7F

-----END PGP MESSAGE-----

A.4. Sample AEAD-OCB encryption and decryption

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

A.4.1. Sample Parameters

S2K:

Iterated and Salted S2K

Iterations:

65011712 (255), SHA2-256

Salt:

56 a2 98 d2 f5 e3 64 53

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

Packet header:

c3 3f

Version, algorithms, S2K fields:

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

Nonce:

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

Encrypted session key and AEAD tag:

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

A.4.3. Starting AEAD-EAX decryption of the session key

The derived key is:

e8 0d e2 43 a3 62 d9 3b 9d c6 07 ed e9 6a 73 56

HKDF info:

c3 05 07 02

HKDF output:

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

Authenticated Data:

c3 05 07 02

Nonce:

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

Decrypted session key:

28 e7 9a b8 23 97 d3 c6 3d e2 4a c2 17 d7 b7 91

A.4.4. Sample v2 SEIPD packet

Packet header:

d2 69

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

02 07 02 06

Salt:

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

Chunk #0 encrypted data:

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

Chunk #0 authentication tag:

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

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

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

A.4.5. Decryption of data

Starting AEAD-OCB decryption of data, using the session key.

HKDF info:

d2 02 07 02 06

HKDF output:

71 66 2a 11 ee 5b 4e 08 14 4e 6d e8 83 a0 09 99
eb de 12 bb 57 0d cf

Message key:

71 66 2a 11 ee 5b 4e 08 14 4e 6d e8 83 a0 09 99

Initialization vector:

eb de 12 bb 57 0d cf

Chunk #0:

Nonce:

eb de 12 bb 57 0d cf 00 00 00 00 00 00 00 00

Additional authenticated data:

d2 02 07 02 06

Decrypted chunk #0.

Literal data packet with the string contents Hello, world!:

cb 13 62 00 00 00 00 00 48 65 6c 6c 6f 2c 20 77
6f 72 6c 64 21

Padding packet:

d5 0e ae 6a a1 64 9b 56 aa 83 5b 26 13 90 2b d2

Authenticating final tag:

Final nonce:

eb de 12 bb 57 0d cf 00 00 00 00 00 00 00 01

Final additional authenticated data:

d2 02 07 02 06 00 00 00 00 00 00 00 25

A.4.6. Complete AEAD-EAX encrypted packet sequence

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

wz8FHQcCCwMIVqKY0vXjZFP/z8xcEWZO2520JZDX3EaweMXAQZzFGzpGh8sy5bcD
HOFGaXV2W1wh2SrvTMBcP+rSaQIHAgYgpmH3MfyaMDK1YjMmAn46XY21dI6+/wsM
WRDQns3WQf+f04VidYA1vEl1TOG/P/+n2tCjuBBPUTPPQqQQCoPu9MobSAGohGv0
K82nyM6dZeIS8wHLzZj9yt5pSod61CRzI/boVw==
=K/pk

-----END PGP MESSAGE-----

A.5. Sample AEAD-GCM encryption and decryption

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

A.5.1. Sample Parameters

S2K:

Iterated and Salted S2K

Iterations:

65011712 (255), SHA2-256

Salt:

e9 d3 97 85 b2 07 00 08

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

Packet header:

c3 3c

Version, algorithms, S2K fields:

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

Nonce:

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

Encrypted session key and AEAD tag:

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

A.5.3. Starting AEAD-EAX decryption of the session key

The derived key is:

25 02 81 71 5b ba 78 28 ef 71 ef 64 c4 78 47 53

HKDF info:

c3 05 07 03

HKDF output:

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

Authenticated Data:

c3 05 07 03

Nonce:

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

Decrypted session key:

19 36 fc 85 68 98 02 74 bb 90 0d 83 19 36 0c 77

A.5.4. Sample v2 SEIPD packet

Packet header:

d2 69

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

02 07 03 06

Salt:

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

Chunk #0 encrypted data:

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

Chunk #0 authentication tag:

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

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

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

A.5.5. Decryption of data

Starting AEAD-GCM decryption of data, using the session key.

HKDF info:

d2 02 07 03 06

HKDF output:

ea 14 38 80 3c b8 a4 77 40 ce 9b 54 c3 38 77 8d
4d 2b dc 2b

Message key:

ea 14 38 80 3c b8 a4 77 40 ce 9b 54 c3 38 77 8d

Initialization vector:

4d 2b dc 2b

Chunk #0:

Nonce:

4d 2b dc 2b 00 00 00 00 00 00 00 00

Additional authenticated data:

d2 02 07 03 06

Decrypted chunk #0.

Literal data packet with the string contents Hello, world!:

cb 13 62 00 00 00 00 00 48 65 6c 6c 6f 2c 20 77
6f 72 6c 64 21

Padding packet:

d5 0e 1c e2 26 9a 9e dd ef 81 03 21 72 b7 ed 7c

Authenticating final tag:

Final nonce:

4d 2b dc 2b 00 00 00 00 00 00 00 01

Final additional authenticated data:

d2 02 07 03 06 00 00 00 00 00 00 25

A.5.6. Complete AEAD-EAX encrypted packet sequence

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

wzwFGgcDCwMI6dOXhbiHAAj/tC58SD70iERXyzcmDAXL8/LNbLe244tb8zRnwccZ
 RN1ZA0ZmL1reYf+EvODSaQIHAWb8uUSQvLmLvcnRBsYJAmAUD3LontwhtVlrFXax
 Ae0Pn/xvxtZbv9JNzQeQlm5tHoWjAFN4TLHYtqBpnevEhVaeyrWJYUxtXZR/Xd3kS
 +pXjXZtAIW9ppMJI2yj/QzHxYykHOZ5v+Q==
 =ClBe

-----END PGP MESSAGE-----

A.6. Sample message encrypted using Argon2

These messages are the literal data "Hello, world!" encrypted using Argon2 and the passphrase "password", using different session key sizes. In all cases, the Argon2 parameters are t = 1, p = 4, and m = 21.

AES-128:

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

Comment: Encrypted using AES with 128-bit key
 Comment: Session key: 01FE16BBACFD1E7B78EF3B865187374F

wyCEBwScUvg8J/leUNU1RA7N/zE2AQQVn1L8rSLPP5VlQsunlO+ECxHSPgGYGKY+
 YJz4u6F+DD1DBOr5NRQXt/KJIf4m4mOlKyC/uqLbpnLJZMnTq3o79GxBTdIdOzhH
 XfA3pqV4mTzF
 =uIks

-----END PGP MESSAGE-----

AES-192:

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

Comment: Encrypted using AES with 192-bit key
 Comment: Session key: 27006DAE68E509022CE45A14E569E91001C2955AF8DFE194

wy8ECAThTKxHFTRZGKli3KNH4UP4AQQVhzLJ2va3FG8/pmpIPd/H/mdoVS5VBLLw
 F9I+AdJlSw56PRYiKZjCvHg+2bnq02s33AJJoyBexBI4QKATFRkyez2gldJldRys
 LVg77Mwwfgl2n/d572WciAM=
 =n8Ma

-----END PGP MESSAGE-----

AES-256:

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

Comment: Encrypted using AES with 256-bit key

Comment: Session key: BBEDA55B9AAE63DAC45D4F49D89DACF4AF37FEFC13BAB2F1F8E18FB74580D8B0

wzcECQS4eJUgIG/3mcaILEJFpmJ8AQQVnZ9l7KtagdCIm9UaQ/Z6M/5roklSGpGu
623YmaXezGj80j4B+KulsgTdJo87XlWrup7l0wJypZls2lUwd67m9koF60eefH/K
95DlusliXOEm8ayQJQmZrjf6K6v9FWwqMQ==
=1fB/

-----END PGP MESSAGE-----

Appendix B. Acknowledgements

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

Appendix C. Document Workflow

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

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

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

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

Authors' Addresses

Werner Koch (editor)
GnuPG e.V.
Rochusstr. 44
40479 Duesseldorf
Germany

Email: wk@gnupg.org
URI: <https://gnupg.org/verein>

Paul Wouters (editor)
Aiven
Email: paul.wouters@aiven.io