openpgp
Internet-Draft
Intended status: Informational
Expires: 9 November 2025

D. K. Gillmor
ACLU

8 May 2025

Stateless OpenPGP Command Line Interface
draft-dkg-openpgp-stateless-cli-14

Abstract

This document defines a generic stateless command-line interface for dealing with OpenPGP messages, certificates, and secret key material, known as sop. It aims for a minimal, well-structured API covering OpenPGP object security and maintenance of credentials and secrets.

About This Document

This note is to be removed before publishing as an RFC.

The latest revision of this draft can be found at https://dkg.gitlab.io/openpgp-stateless-cli/. Status information for this document may be found at https://datatracker.ietf.org/doc/draft-dkg-openpgp-stateless-cli/.

Discussion of this document takes place on the OpenPGP Working Group mailing list (mailto:openpgp@ietf.org), which is archived at https://mailarchive.ietf.org/arch/browse/openpgp/. Subscribe at https://www.ietf.org/mailman/listinfo/openpgp/.

Source for this draft and an issue tracker can be found at https://gitlab.com/dkg/openpgp-stateless-cli/.

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 9 November 2025.

Table of Contents

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 can handle both object security side and key and certificate management in a way that would be usable by applications across the full OpenPGP lifecycle.

A priority for this interface is to facilitate interoperability testing for OpenPGP implementations, for example as described in Section 1.3.

This document defines a generic stateless command-line interface for dealing with OpenPGP messages, secret keys, and certificates, 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.

Inputs to sop are immutable inputs. Any named files that it receives as input should only need read access, and it must not write to or modify any of its inputs. The only places a sop implementation should write to are standard output and (in some special cases) a location specified by an --*-out= argument.

Obviously, the user (or consuming application) will need to manage persistent secret keys and certificates somehow, but the goal of this interface is to separate out that task from the task of processing and handling OpenPGP objects.

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.

Appendix B offers a preliminary sketch of a C library interface that also has no implicit state.

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 10.2 of [RFC9580]).

It uses the term "certificate" to refer to OpenPGP Transferable Public Key (see Section 10.1 of [RFC9580]).

"Stateless" in "Stateless OpenPGP" means avoiding any sort of persistent and implicit 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 5.2.3.31 of [RFC9580]), 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.

This document refers to a special verification-only subset of the sop command-line interface as sopv (see Section 3 for more details).

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.

A simple self-test can be found in Appendix C.

1.4. Semantics vs. Wire Format

The semantics of sop are deliberately simple and very high-level compared to the vast complexity and nuance available within the OpenPGP specification. This reflects the perspective of nearly every piece of tooling that relies on OpenPGP to accomplish its task: most toolchains don't care about the specifics, they just want the high-level object security properties.

Given this framing, this document generally tries to avoid overconstraining the details of the wire format objects emitted, or what kinds of wire format structures should be acceptable or unacceptable. This allows a test suite to evaluate and contrast the wire format choices made by different implementations in as close to their native configuration as possible. It also makes it easier to promote interoperability by ensuring that the native wire formats emitted by one implementation can be consumed by another, without relying on their choices of wire format being constrained by this draft.

Where this draft does identify specific wire format requirements, that might be due to an ambiguity in the existing specifications (which maybe needs fixing elsewhere), or to a bug in this specification that could be improved.

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 bob.pgp < msg.eml > ciphertext.asc
sop decrypt bob.sec < ciphertext.asc > cleartext.eml
```

See Section 8 for more information about errors and error handling.

3. sopv Subset

While the primary goal of this document is to provide a full sop interface, as a special case, an implementer may choose to produce a version of the command-line interface that only supports OpenPGP signature verification. As a shorthand, this document refers to such an interface as sopv, or "the sopv subset". This can be useful for constrained environments where the only thing needed is signature verification, for example, system installation or update media.

A full implementation of sop by definition provides sopv, of course.

Only the following subcommands and their associated options MUST be implemented for sopv:

* version (Section 5.1.1)

* verify (Section 5.4.2)

* inline-verify (Section 5.4.6)

3.1. sopv Versioning

The abstract sopv interface is itself versioned using [SEMVER]. The definition of the relevant subcommands and options specified in this document is known as sopv version 1.0.

If backward-incompatible changes are made to the sopv subset, the major version number will be increased. If the sopv subset is extended without backward-incompatible changes, the minor version number will be increased.

Elements of the CLI relevant to sopv are annotated in this document with the sopv version in which they were introduced.

See also Appendix A for enumerated version history.

4. Universal Options

Every invocation of sop or sopv MAY use the options described in this section, even though they are not specified in the synopsis for any specific subcommand.

4.1. --debug: emit more verbose output

When the --debug option is present, sop MAY emit implementation-specific debugging information to standard error.

A locale-aware, internationalized sop implementation will localize this debugging information.

5. 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 [RFC9580]) 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 11.4) and behave accordingly.

See Section 7 for details about how various forms of OpenPGP material are expected to be structured.

5.1. Meta Subcommands

The subcommands grouped in this section are related to the sop implementation itself.

5.1.1. version: Version Information

sop version [--backend|--extended|--sop-spec|--sopv]

* Standard Input: ignored

* Standard Output: version information

* sopv Introduction: 1.0

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.

If --sop-spec is supplied, the implementation should emit a single line of text indicating the latest version of this draft that it targets, for example, draft-dkg-openpgp-stateless-cli-06. If the implementation targets a specific draft but the implementer knows the implementation is incomplete, it should prefix the draft title with a "~" (TILDE, U+007E), for example: ~draft-dkg-openpgp-stateless-cli-06. The implementation MAY emit additional text about its relationship to the targeted draft on the lines following the versioned title.

If --sopv is supplied, the implementation should produce a single line with the implemented [SEMVER] version of the sopv interface subset (see Section 3) that this implementation provides complete coverage for. If the implementation does not provide complete coverage for any sopv interface, it should emit nothing on standard out and return UNSUPPORTED_OPTION.

--backend, --extended, --sop-spec, and --sopv 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
$ sop version --sop-spec
~draft-dkg-openpgp-stateless-cli-06

This implementation does not handle @FD: special designators for output.
$ sop version --sopv
1.0
$
```

5.1.2. list-profiles: Describe Available Profiles

sop list-profiles SUBCOMMAND

* Standard Input: ignored

* Standard Output: PROFILELIST (Section 7.12)

* sop Introduction: 1.0

This subcommand emits a list of profiles supported by the identified subcommand. The first profile listed is the default profile, as described in Section 7.12.

If the indicated SUBCOMMAND does not accept a --profile option, it returns UNSUPPORTED_PROFILE.

Example:

```
$ sop list-profiles generate-key
default: use the implementer’s recommendations
security: higher-security, maybe reduced performance
performance: higher-performance, maybe reduced security
rfc4880: use algorithms from RFC 4880 (alias: compatibility)
$
```

5.2. Key and Certificate Management Subcommands

The subcommands grouped in this section are primarily intended to manipulate keys and certificates.

5.2.1. generate-key: Generate a Secret Key

```
sop generate-key [--no-armor]
    [--with-key-password=PASSWORD]
    [--profile=PROFILE]
    [--signing-only]
    [--] [USERID...]
```

* Standard Input: ignored

* Standard Output: KEYS (Section 7.3)

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.

* Unless the --signing-only parameter is supplied, 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.

If the --with-key-password option is supplied, the generated key will be password-protected (locked) with the supplied password. Note that PASSWORD is an indirect data type from which the actual password is acquired (Section 7). See also the guidance on ensuring that the password is human-readable in Section 11.8.1.

If no --with-key-password option is supplied, the generated key will be unencrypted.

If the --profile argument is supplied and the indicated PROFILE is not supported by the implementation, sop will fail with UNSUPPORTED_PROFILE.

The presence of the --signing-only option is intended to create a key that is only capable of signing, not decrypting. This is useful for deployments where only signing and verification are necessary.

If any of the USERID options are not valid UTF-8, sop generate-key fails with EXPECTED_TEXT.

If the implementation rejects any USERID option that is valid UTF-8 (e.g., due to internal policy, see Section 6.2), sop generate-key fails with BAD_DATA.

Example:

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

5.2.2. change-key-password: Update a Key's Password

```
sop change-key-password [--no-armor]
    [--new-key-password=PASSWORD]
    [--old-key-password=PASSWORD...]
```

* Standard Input: KEYS (Section 7.3)

* Standard Output: KEYS (Section 7.3)

The output will be the same set of OpenPGP Transferable Secret Keys as the input, but with all secret key material locked according to the password indicated by the --new-key-password option (or, with no password at all, if --new-key-password is absent). Note that --old-key-password can be supplied multiple times, and each supplied password will be tried as a means to unlock any locked key material encountered. It will normalize a Transferable Secret Key to use a single password even if it originally had distinct passwords locking each of the subkeys.

If any secret key packet is locked but cannot be unlocked with any of the supplied --old-key-password arguments, this subcommand should fail with KEY_IS_PROTECTED.

Example:

```
# adding a password to an unlocked key:
$ sop change-key-password --new-key-password=@ENV:keypass \
  < unlocked.key > locked.key
# removing a password:
$ sop change-key-password --old-key-password=@ENV:keypass \
  < locked.key > unlocked.key
# changing a password:
$ sop change-key-password --old-key-password=@ENV:keypass \
  --new-key-password=@ENV:newpass < locked.key > refreshed.key
$
```

5.2.3. revoke-key: Create a Revocation Certificate

```
sop revoke-key [--no-armor]
    [--with-key-password=PASSWORD...]
```

* Standard Input: KEYS (Section 7.3)

* Standard Output: CERTS (Section 7.2)

Generate a revocation certificate for each Transferable Secret Key found. See Section 10.1.2 of [RFC9580] for a discussion of common forms of revocation certificate.

Example:

```
$ sop revoke-key < alice.key > alice-revoked.pgp
$
```

5.2.4. extract-cert: Extract a Certificate from a Secret Key

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

* Standard Input: KEYS (Section 7.3)

* Standard Output: CERTS (Section 7.2)

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.

sop extract-cert SHOULD work even if any of the keys in KEYS is password-protected.

Example:

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

5.2.5. update-key: Keep a Secret Key Up-To-Date

```
sop update-key [--no-armor]
    [--signing-only]
    [--no-added-capabilities]
    [--with-key-password=PASSWORD...]
    [--merge-certs=CERTS...]
```

* Standard Input: KEYS (Section 7.3)

* Standard Output: KEYS (Section 7.3)

The input OpenPGP Transferable Secret Keys that arrive on standard input will be updated by the implementation, and their updated forms will be produced on standard output. This update will "fix" everything that the implementation knows how to fix to bring each Transferable Secret Key up to reasonable modern practice. Each Transferable Secret Key output must be capable of signing, and (unless --signing-only is provided) capable of decryption. The primary key of each Transferable Secret Key will not be changed in any way that affects its fingerprint.

One important aspect of sop update-key is how it handles advertisement of support for various OpenPGP capabilities (algorithms, mechanisms, etc). All capabilities that the implementation knows it does not support, or knows to be weak and/or deprecated MUST be removed from the output Transferable Secret Keys. This includes unknown/deprecated flags in the Features subpacket, and any unknown/deprecated algorithm IDs in algorithm preferences subpackets. For example, an implementation compliant with [RFC9580] will never emit a Transferable Secret Key with a Preferred Hash Preferences subpacket that explicitly indicates support for MD5, RIPEMD160, or SHA1.

If --no-added-capabilities is not present, then any capability that the implementation supports and encourages that was not advertised in the input Transferable Secret Key MAY be added to the advertisements in the output Transferable Secret Key. If --no-added-capabilities is present, then new capabilities MUST NOT be added to the advertised sets during the update.

Beyond cleanup of the advertised capabilities, --signing-only, and --no-added-capabilities, the choice of exactly what updates to do are up to the implementation. It is expected that an implementer will document and describe the specific considerations and updates they make for this operation. It is acceptable to propagate any non-critical unknown subpackets from old self-signatures to the new, replacement self-signatures.

Possible updates might include:

* Refresh or replace any subkey approaching expiry.

* Refresh any self-signature (including cross-sigs) that is approaching expiry.

* Refresh any self-signature (including cross-sigs) that is made using weak or risky algorithms.

* Correct any mistaken 2-octet hash prefix found in a signature (see Section 5.2.3 of [RFC9580]).

* Ensure proof of "aliveness": if no self-signatures are more recent than some cutoff in the recent past, re-issue the same self-signatures.

If there is nothing to be updated because all the incoming Transferable Secret Keys are already in good shape, then the same set of Transferable Secret Keys will be emitted to standard output and sop update-key succeeds.

If any Transferable Secret Key cannot be fixed (for example, because its primary key uses a weak algorithm, or because the whole certificate is hard-revoked), sop update-key fails with PRIMARY_KEY_BAD, emits an explanation on stderr, and nothing on stdout.

If any secret key that needs to make a signature to update the key cannot be unlocked with any of the supplied PASSWORD objects, sop update-key fails with KEY_IS_PROTECTED, emits an explanation on stderr, and nothing on stdout.

If --merge-certs is supplied, and any of the CERTS objects correspond to the Transferable Secret Keys being updated, then any additional elements found in the corresponding CERTS are merged into the Transferable Secret Key before it is emitted. This can be used, for example, to absorb a third-party certification into the Transferable Secret Key.

Example (keeping certificates fresh):

```
$ sop update-key < alice.key > alice-updated.key
$ mv alice-updated.key alice.key
$ sop extract-cert < alice.key > alice.pgp
$
```

Example (advertising the intersection of features supported by two Stateless OpenPGP implementations, rendered here as sop1 and sop2):

```
$ sop1 update-key < alice.key | sop2 update-key | \
  sop1 --no-added-capabilities update-key > alice-updated.key
$ mv alice-updated.key alice.key
$ sop1 extract-cert < alice.key > alice.pgp
$
```

5.2.6. merge-certs: Merge OpenPGP Certificates

```
sop merge-certs [--no-armor]
    [--] CERTS [CERTS...]
```

* Standard Input: CERTS (Section 7.2)

* Standard Output: CERTS (Section 7.2)

The OpenPGP certificates on standard input will be produced on standard output, merged with the corresponding elements of any of the CERTS objects named on the command line.

This can be used, for example, to absorb a third-party certification into a certificate, or to update a certificate's feature advertisements without losing local annotations.

The certificates produced on standard output are only the certificates received on standard input. If any certificate found via named command line parameters does not share a primary key with any standard input certificate, the certificate from the command line is ignored.

If any of the OpenPGP certificates on standard input share the same primary key, they are also merged and de-deduplicated on standard output. If multiple OpenPGP certificates named on the command line share a primary key with one of the certificates on standard input, their certificate updates are cumulatively merged for output.

Example:

```
$ sop merge-certs alice-certified-by-bob.pgp \
  < alice.pgp > alice-updated.pgp
$ mv alice-updated.pgp alice.pgp
$
```

5.3. User Identity Subcommands

The subcommands in this section handle OpenPGP user identities. OpenPGP certificates contain cryptographic certifications which bind text-based "User IDs" to primary key material, which is in turn cryptographically bound to additional key material.

These subcommands are related to the network of cryptographic identity assertions that has traditionally been called the "Web of Trust". Note also the similarity in structure between these subcommands and sop sign (Section 5.4.1) and sop verify (Section 5.4.2)

5.3.1. certify-userid: Certify OpenPGP Certificate User IDs

```
sop certify-userid [--no-armor]
    --userid=USERID
    [--userid=USERID...]
    [--with-key-password=PASSWORD...]
    [--no-require-self-sig]
    [--] KEYS [KEYS...]
```

* Standard Input: CERTS (Section 7.2)

* Standard Output: CERTS (Section 7.2)

With each Transferable Secret Key in all KEYS objects, add a third-party certification to CERTS found on standard input, and emit the updated OpenPGP certificates (including the new certification(s)) on standard output.

If the caller does not specify at least one --userid=USERID option, sop certify-userid fails with MISSING_ARG.

If the certification-capable key of any Transferable Secret Key in KEYS is locked and cannot be unlocked by any of the supplied PASSWORDs, sop certify-userid fails with KEY_IS_PROTECTED.

If any incoming CERTS object does not already have all of the specified User IDs as valid, self-signed User IDs, then sop certify-userid fails with CERT_USERID_NO_MATCH, unless --no-require-self-sig is supplied.

If --no-require-self-sig is supplied, then each incoming OpenPGP certificate will have each specified User ID added to it (if it did not have it already), and certified directly, regardless of self-signatures. This may be useful for associating a certificate with a specific identity even in cases where the certificate does not itself advertise the identity.

If any key in the KEYS objects is not capable of producing a certification, sop certify-userid will fail with KEY_CANNOT_CERTIFY.

Example:

```
$ sop certify-userid \
  --userid="Alice Lovelace <alice@openpgp.example>" \
  bob.key < alice.pgp > alice-signed-by-bob.pgp
$
```

Example (adding a User ID to your own certificate):

```
$ sop certify-userid \
  --userid="Alice Lovelace <lovelace@business.example>" \
  alice.key < alice.pgp > alice-updated.pgp
$ sop update-key --merge-certs alice-updated.pgp \
  < alice.key > alice-updated.key
$ mv alice-updated.key alice.key
$ rm alice-updated.pgp
$ sop extract-cert < alice.key > alice.pgp
$
```

5.3.2. validate-userid: Validate a User ID in an OpenPGP Certificate

```
sop validate-userid
    [--addr-spec-only]
    [--validate-at=DATE]
    [--] USERID CERTS [CERTS...]
```

* Standard Input: CERTS (Section 7.2)

* Standard Output: none

* sopv Introduction: 1.2

Given a set of authority OpenPGP certificates on the command line, succeed if and only if all OpenPGP certificates on standard input are correctly bound by at least one valid signature from one authority to the USERID in question.

If --addr-spec-only is present, then the USERID is treated as an e-mail address, and matched only against the e-mail address part of each correctly bound User ID. The rest of each correctly bound User ID is ignored. If any correctly bound User ID is not a conventional OpenPGP User ID, it will not match with --addr-spec-only at all. Note that [RFC9580] (and [RFC4880] and [RFC2440] before it) mislabeled an OpenPGP User ID as a name-addr, but that is likely to be wrong.

If --validate-at is present, then evaluate the validity of the User ID at the specified time. If --validate-at is not present (or if it is present with the literal value now), the User ID validity is evaluated at the current time.

If any OpenPGP certificate in the CERTS on standard input does not have a correctly bound User ID that matches USERID, sop validate-userid fails with CERT_USERID_NO_MATCH.

Example:

```
$ if sop validate-userid "Alice Lovelace <alice@openpgp.example>" \
  bob.pgp < alice.pgp; then echo Good; fi
Good
$ if sop validate-userid --addr-spec-only "alice@openpgp.example" \
  bob.pgp < alice.pgp; then echo Good; fi
Good
$
```

5.4. Messaging Subcommands

The subcommands in this section handle OpenPGP messages: encrypting, decrypting, signing, and verifying.

5.4.1. sign: Create Detached Signatures

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

* Standard Input: DATA (Section 7.11)

* Standard Output: SIGNATURES (Section 7.6)

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 11.7), sop sign fails with EXPECTED_TEXT.

--as=binary SHOULD result in OpenPGP signatures of type 0x00 ("Signature of a binary document"). --as=text SHOULD result in OpenPGP signatures 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 7.8). If the specified MICALG file already exists in the filesystem, sop sign will fail with OUTPUT_EXISTS. When --micalg-out is supplied, the DATA on standard input should already be in canonical text form (7-bit clean, CRLF line endings, no trailing whitespace), as specified in Section 3 of [RFC3156]. If the incoming DATA does not already meet these requirements, sop sign will fail with EXPECTED_TEXT, regardless of any argument supplied for --as. When --micalg-out is supplied, and multiple signatures are made but they do not all use the same digest algorithm, sop sign MUST emit the empty string to the designated MICALG.

If the signing key material in any key in the KEYS objects is password-protected, sop sign SHOULD try all supplied --with-key-password options to unlock the key material until it finds one that enables the use of the key for signing. If none of the PASSWORD options unlock the key (or if no such option is supplied), sop sign will fail with KEY_IS_PROTECTED. Note that PASSWORD is an indirect data type from which the actual password is acquired (Section 7). Note also the guidance for retrying variants of a non-human-readable password in Section 11.8.2.

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-----
$
```

5.4.2. verify: Verify Detached Signatures

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

* Standard Input: DATA (Section 7.11)

* Standard Output: VERIFICATIONS (Section 7.10)

* sopv Introduction: 1.0 (VERIFICATIONS output augmented in 1.1)

--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 time window specified 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. In this case, sop verify MAY emit some human-readable explanation to standard error about why no valid signatures were found, see Section 13.1.1.

See Section 13.1 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 EB85BB5FA33A75E15E944E63F23
1550C4F47E38E mode:text {"signers": ["alice.pgp"]}
$ echo $?
0
$ tr a-z A-Z < message.txt | sop verify message.txt.asc alice.pgp
$ echo $?
3
$
```

5.4.3. encrypt: Encrypt a Message

```
sop encrypt [--as={binary|text}]
    [--no-armor]
    [--with-password=PASSWORD...]
    [--sign-with=KEYS...]
    [--with-key-password=PASSWORD...]
    [--profile=PROFILE]
    [--session-key-out=SESSIONKEY]
    [--] [CERTS...]
```

* Standard Input: DATA (Section 7.11)

* Standard Output: CIPHERTEXT (Section 7.4)

--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).

--with-password enables symmetric encryption (and can be used multiple times if multiple passwords are desired).

--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 object is not capable of producing a signature, sop sign will fail with KEY_CANNOT_SIGN. If any signing key material in any supplied KEYS object is password-protected, sop encrypt SHOULD try all supplied --with-key-password options to unlock the key material until it finds one that enables the use of the key for signing. If none of the --with-key-password=PASSWORD options can unlock any locked signing key material (or if no such option is supplied), sop encrypt will fail with KEY_IS_PROTECTED. All signatures made must be placed inside the encryption produced by sop encrypt.

Note that both --with-password and --with-key-password supply PASSWORD arguments, but they do so in different contexts which are not interchangeable. A PASSWORD supplied for symmetric encryption (--with-password) MUST NOT be used to try to unlock a signing key (--with-key-password) and a PASSWORD supplied to unlock a signing key MUST NOT be used to symmetrically encrypt the message. Regardless of context, each PASSWORD argument is presented as an indirect data type from which the actual password is acquired (Section 7). If sop encrypt encounters a password which is not a valid UTF-8 string (Section 11.7), 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 11.8 for more discussion about passwords.

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 11.7), sop encrypt fails with EXPECTED_TEXT.

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 the --profile argument is supplied and the indicated PROFILE is not supported by the implementation, sop will fail with UNSUPPORTED_PROFILE. The use of a profile for this subcommand allows an implementation faced with parametric or algorithmic choices to make a decision coarsely guided by the operator. For example, when encrypting with a password, there is no knowledge about the capabilities of the recipient, and an implementation may prefer cryptographically modern algorithms, or it may prefer more broad compatibility. In the event that a known recipient (i.e., one of the CERTS) explicitly indicates a lack of support for one of the features preferred by the indicated profile, the implementation SHOULD conform to the recipient's advertised capabilities where possible.

If --session-key-out argument is supplied, the session key generated for this encrypted will be written to the indicated location. This can be useful, for example, when Alice encrypts a message to Bob, but also wants to retain the ability to read it without having any of her own secret key material available (see Section 9.1 of [I-D.ietf-lamps-e2e-mail-guidance-11]).

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=text --sign-with=alice.key \
  alice.asc bob.bin < message.eml > encrypted.asc
$ head -n1 encrypted.asc
-----BEGIN PGP MESSAGE-----
$
```

5.4.4. decrypt: Decrypt a Message

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

* Standard Input: CIPHERTEXT (Section 7.4)

* Standard Output: DATA (Section 7.11)

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. SESSIONKEY is an indirect data type from which the actual sessionkey value is acquired (Section 7).

--with-password enables decryption based on any SKESK (Section 5.3 of [RFC9580]) packets in the CIPHERTEXT. This option can be used multiple times if the user wants to try more than one password.

--with-key-password lets the user use password-protected (locked) secret key material. If the decryption-capable secret key material in any key in the KEYS objects is password-protected, sop decrypt SHOULD try all supplied --with-key-password options to unlock the key material until it finds one that enables the use of the key for decryption. If none of the --with-key-password options unlock the key (or if no such option is supplied), and the message cannot be decrypted with any other KEYS, --with-session-key, or --with-password options, sop decrypt will fail with KEY_IS_PROTECTED.

Note that the two kinds of PASSWORD options are for different domains: --with-password is for unlocking an SKESK, and --with-key-password is for unlocking secret key material in KEYS. sop decrypt SHOULD NOT apply the --with-key-password argument to any SKESK, or the --with-password argument to any KEYS.

Each PASSWORD argument is an indirect data type from which the actual password is acquired (Section 7). If sop decrypt tries and fails to use a password supplied by a 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 11.8.2 for more discussion about consuming password-protected key material.

--verifications-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.

If no valid signatures were found, but --verifications-out was supplied, sop decrypt MAY emit some human-readable explanation to standard error about why no valid signatures were found, see Section 13.1.1.

--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 --verifications-out and at least one --verify-with must be supplied. If only one of these options 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 5.4.2). They should only be supplied when doing signature verification.

See Section 13.1 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 reuses 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
$
```

5.4.4.1. Historic Options for sop decrypt

The sop decrypt option --verifications-out used to be named --verify-out. An implementation SHOULD accept either form of this option, and SHOULD produce a deprecation warning to standard error if the old form is used.

5.4.5. inline-detach: Split Signatures from an Inline-Signed Message

sop inline-detach [--no-armor] --signatures-out=SIGNATURES

* Standard Input: INLINESIGNED

* Standard Output: DATA (the message without any signatures)

In some contexts, the user may expect an inline-signed message of some form or another (INLINESIGNED, see Section 7.5) rather than a message and its detached signature. sop inline-detach takes such an inline-signed 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 inline-signed message is separated, verification of the detached signature can be done with sop verify.

If no --signatures-out is supplied, sop inline-detach fails with MISSING_ARG.

Note that there may be more than one Signature packet in an inline-signed message. All signatures found in the inline-signed message will be emitted to the --signatures-out destination.

If the inline-signed message uses the Cleartext Signature Framework, it may be dash-escaped (see Section 7.2 of [RFC9580]). The output of sop detach-inband-signature-and-message will have any dash-escaping removed.

If the input is not an INLINESIGNED message, sop inline-detach fails with BAD_DATA. If the input contains more than one object that could be interpreted as an INLINESIGNED message, sop inline-detach also fails with BAD_DATA. A sop implementation MAY accept (and discard) leading and trailing data when the incoming INLINESIGNED message uses the Cleartext Signature Framework.

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 inline-detach --signatures-out=Release.pgp < InRelease >Release
$ sop verify Release.pgp archive-keyring.pgp < Release
$
```

5.4.6. inline-verify: Verify an Inline-Signed Message

```
sop inline-verify [--not-before=DATE] [--not-after=DATE]
    [--verifications-out=VERIFICATIONS]
    [--] CERTS [CERTS...]
```

* Standard Input: INLINESIGNED (Section 7.5)

* Standard Output: DATA (Section 7.11)

* sopv Introduction: 1.0 (VERIFICATIONS output augmented in 1.1)

This command is similar to sop verify (Section 5.4.2) except that it takes an INLINESIGNED message (see Section 7.5) and produces the message body (without signatures) on standard output. It is also similar to sop inline-detach (Section 5.4.5) except that it actually performs signature verification.

--not-before and --not-after indicate that signatures with dates outside certain range MUST NOT be considered valid. See Section 5.4.2 for their syntax and defaults.

sop inline-verify only returns OK if INLINESIGNED contains at least one valid signature made during the time window specified by a certificate included in any CERTS object.

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

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

If no valid signatures are found, sop inline-verify fails with NO_SIGNATURE and emits nothing on standard output. In this case, sop inline-verify MAY emit some human-readable explanation to standard error about why no valid signatures were found, see Section 13.1.1.

See Section 13.1 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 inline-verify -- alice.pgp < message.txt
Hello, world!
$ echo $?
0
$ sed s/Hello/Goodbye/ < message.txt | sop inline-verify -- alice.pgp
$ echo $?
3
$
```

5.4.7. inline-sign: Create an Inline-Signed Message

```
sop inline-sign [--no-armor]
     [--with-key-password=PASSWORD...]
     [--as={binary|text|clearsigned}]
     [--] KEYS [KEYS...]
```

* Standard Input: DATA (Section 7.11)

* Standard Output: INLINESIGNED (Section 7.5)

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

The generated output stream will be an inline-signed message, by default producing an OpenPGP "Signed Message" packet stream.

--as defaults to binary. If --as= is set to either text or clearsigned, and the input DATA is not valid UTF-8 (Section 11.7), sop inline-sign fails with EXPECTED_TEXT.

--as=binary SHOULD result in OpenPGP signatures of type 0x00 ("Signature of a binary document", see Section 5.2.1.1 of [RFC9580]). --as=text SHOULD result in an OpenPGP signature of type 0x01 ("Signature of a canonical text document" see Section 5.2.1.2 of [RFC9580]). --as=clearsigned SHOULD behave the same way as --as=text except that it produces an output stream using the Cleartext Signature Framework (see Section 7 of [RFC9580] and Section 11.5).

If both --no-armor and --as=clearsigned are supplied, sop inline-sign fails with INCOMPATIBLE_OPTIONS.

If the signing key material in any key in the KEYS objects is password-protected, sop inline-sign SHOULD try all supplied --with-key-password options to unlock the key material until it finds one that enables the use of the key for signing. If none of the PASSWORD options unlock the key (or if no such option is supplied), sop inline-sign will fail with KEY_IS_PROTECTED. Note that PASSWORD is an indirect data type from which the actual password is acquired (Section 7). Note also the guidance for retrying variants of a non-human-readable password in Section 11.8.2.

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

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

Example:

```
$ sop inline-sign --as=clearsigned alice.sec \
  < message.txt > message-signed.txt
$ head -n5 < message-signed.txt
-----BEGIN PGP SIGNED MESSAGE-----
Hash: SHA256

This is the message.
-----BEGIN PGP SIGNATURE-----
$
```

5.5. Transport Subcommands

The commands in this section handle manipulating OpenPGP objects for transport: armoring and dearmoring for 7-bit cleanness and compactness, respectively.

5.5.1. armor: Convert Binary to ASCII

sop armor

* Standard Input: OpenPGP material (SIGNATURES, KEYS, CERTS, CIPHERTEXT, or INLINESIGNED)

* Standard Output: the same material with ASCII-armoring added, if not already present

sop armor inspects the input and chooses the label appropriately, based on the OpenPGP packets encountered.

sop armor ought to be able to correctly re-armor any of the packet streams that are produced by sop with the --no-armor option.

For example, if the type of the first OpenPGP packet is:

* 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).

* 0x04 (One-Pass Signature), the packet stream should be parsed as an INLINESIGNED input (with Armor Header BEGIN PGP MESSAGE).

* 0x02 (Signature), the packet stream may be either a SIGNATURES input or an INLINESIGNED input. If the packet stream contains only Signature packets, it should be parsed as aSIGNATURES input (with Armor Header BEGIN PGP SIGNATURE). If it contains any packet other than a Signature packet, it should be parsed as an INLINESIGNED input (with Armor Header BEGIN PGP MESSAGE).

If the input packet stream does not match any 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 to ensure that any well-formed OpenPGP packet stream is 7-bit clean.

FIXME: what to do if the input is a CSF INLINESIGNED message? Three choices:

* Leave it untouched -- this violates the claim about blindly ensuring 7-bit clean, since UTF-8-encoded message text is not necessarily 7-bit clean.

* Convert to ASCII-armored INLINESIGNED -- this requires synthesis of OPS packet (from signatures block) and Literal Data packet (from the message body).

* Raise a specific error.

Example:

```
$ sop armor < bob.bin > bob.pgp
$ head -n1 bob.pgp
-----BEGIN PGP PUBLIC KEY BLOCK-----
$
```

5.5.1.1. Historic Options for sop armor

sop armor used to be specified as having a --label option, with an argument that took one of the following values: auto, sig, key, cert, or message, which allowed the user to specify the label used in the header and tail of the armoring.

The default value for --label was auto, which matches the currently specified behavior. This option is now deprecated, as it offers no useful functionality.

5.5.2. dearmor: Convert ASCII to Binary

sop dearmor

* Standard Input: OpenPGP material (SIGNATURES, KEYS, CERTS, CIPHERTEXT, or INLINESIGNED)

* 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 11.4.

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.

FIXME: what to do if the input is a CSF INLINESIGNED? Three choices:

* Leave it untouched -- output data is not really in binary format.

* Convert to binary-format INLINESIGNED -- this requires synthesis of OPS packet (from CSF Hash header) and Literal Data packet (from the message body).

* Raise a specific error.

Example:

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

6. Input String Types

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

6.1. 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 7.10) it MUST produce only a UTC-based ISO-8601 compliant representation with a resolution of one second, using the literal Z suffix to indicate timezone.

6.2. USERID

This is an arbitrary UTF-8 string (Section 11.7). By convention, most User IDs are of the form Display Name <email.address@example.com>, but they do not need to be.

By internal policy, an implementation MAY reject a USERID if there are certain UTF-8 strings it declines to work with as a User ID. For example, an implementation may reject the empty string, or a string with characters in it that it considers problematic. Of course, refusing to create a particular User ID does not prevent an implementation from encountering such a User ID in its input.

6.3. SUBCOMMAND

This is an ASCII string that matches the name of one of the subcommands listed in Section 5.

6.4. PROFILE

Some sop subcommands can accept a --profile option, which takes as an argument the name of a profile.

A profile name is a UTF-8 string that has no whitespace in it.

Which profiles are available depends on the sop implementation.

Similar to OpenPGP Notation names, profile names are divided into two namespaces: the IETF namespace and the user namespace. A profile name in the user namespace ends with the @ character (0x40) followed by a DNS domain name. A profile name in the IETF namespace does not have an @ character.

A profile name in the user space is owned and controlled by the owner of the domain in the suffix. A sop implementation that implements a user profile but does not own the domain in question SHOULD hew as closely as possible to the semantics described by the owner of the domain.

A profile name in the IETF namespace that begins with the string rfc should have semantics that hew as closely as possible to the referenced RFC. Similarly, a profile name in the IETF namespace that begins with the string draft- should have semantics that hew as closely as possible to the referenced Internet Draft.

The reserved profile name default in the IETF namespace simply refers to the implementation's default choices. It is not mandatory to name the default profile default. The first profile listed in the list-profiles output is considered the default configuration, as specified in Section 7.12.

The reserved profile names security, performance, and compatibility refer to the implementation's choices when increased emphasis on security, performance or compatibility is required, respectively. It is not mandatory to name any profile security, performance, or compatibility; in that case, those profile names MUST act as aliases of another profile name. They are also allowed to be aliases of the default profile.

Note that this profile mechanism is intended to provide a limited way for an implementation to select among a small set of options that the implementer has vetted and is satisfied with. It is not intended to provide an arbitrary channel for complex configuration, and a sop implementation MUST NOT use it in that way.

7. 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.

7.1. 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.

See Section 11.9 for more details about safe handling of these special designators.

7.1.1. @ENV: Special Designator for Environment Variable

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.

The sopv subset (see Section 3) MUST be capable of supporting the @ENV special designator for all relevant inputs starting at sopv version 1.0.

7.1.2. @FD: Special Designator for File Descriptor

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.

On platforms which support file descriptors, the sopv subset (see Section 3) MUST be capable of supporting the @FD special designator for all relevant inputs and outputs starting at sopv version 1.0.

7.2. CERTS

One or more OpenPGP certificates (Section 10.1 of [RFC9580]), aka "Transferable Public Key". May be armored (see Section 11.4).

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.

If any CERTS input contains secret key material, sop MUST fail with BAD_DATA. This strictness is intended to keep the consumer of the sop interface clear about what material they are dealing with in what locations. This should reduce the consumer’s risk of accidentally exposing secret key material where they meant to expose a CERTS object.

7.3. KEYS

One or more OpenPGP Transferable Secret Keys (Section 10.2 of [RFC9580]). May be armored (see Section 11.4).

Secret key material is often locked with a password to ensure that it cannot be simply copied and reused. If any secret key material is locked with a password and no --with-key-password option is supplied, sop may fail with error KEY_IS_PROTECTED. However, when a cleartext

secret key (that is, one not locked with a password) is available, sop should always be able to use it, whether a --with-key-password option is supplied or not.

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.

7.4. CIPHERTEXT

sop accepts only a restricted subset of the arbitrarily-nested grammar allowed by the OpenPGP Messages definition (Section 10.3 of [RFC9580]).

In particular, it accepts and generates only:

An OpenPGP message, consisting of a sequence of PKESKs (Section 5.1 of [RFC9580]) and SKESKs (Section 5.3 of [RFC9580]), followed by one SEIPD (Section 5.13 of [RFC9580]).

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 11.4).

7.5. INLINESIGNED

An inline-signed message may take any one of three different forms:

* A binary sequence of OpenPGP packets that matches a subset of the "Signed Message" element in the grammar in Section 10.3 of [RFC9580]

* The same sequence of packets, but ASCII-armored (see Section 11.4)

* A message using the Cleartext Signature Framework described in Section 7 of [RFC9580]

The subset of the packet grammar expected in the first two forms consists of either:

* a series of Signature packets followed by a Literal Data packet

* a series of One-Pass Signature (OPS) packets, followed by one Literal Data packet, followed by an equal number of Signature packets corresponding to the OPS packets

When the message is in the third form (Cleartext Signature Framework), it has the following properties:

* The stream SHOULD consist solely of UTF-8 text

* Every Signature packet found in the stream SHOULD have Signature Type 0x01 (canonical text document).

* It SHOULD NOT contain leading text (before the -----BEGIN PGP SIGNED MESSAGE----- cleartext header) or trailing text (after the -----END PGP SIGNATURE----- armor tail).

While some OpenPGP implementations MAY produce more complicated inline signed messages, a sop implementation SHOULD limit itself to producing these straightforward forms.

7.6. SIGNATURES

One or more OpenPGP Signature packets. May be armored (see Section 11.4).

7.7. 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 [RFC9580]) and HEXKEY is the hexadecimal representation of the binary key.

Example AES-256 session key:

9:FCA4BEAF687F48059CACC14FB019125CD57392BAB7037C707835925CBF9F7BCD

A sop implementation SHOULD produce session key data in this format, with a trailing newline. When consuming such a session key, sop SHOULD be willing to accept either upper or lower case hexadecimal digits, and to gracefully ignore any trailing whitespace.

7.8. MICALG

This output-only type 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.

7.9. PASSWORD

This input-only is expected to be a UTF-8 string (Section 11.7), 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 11.8 for more discussion.

7.10. VERIFICATIONS

This output-only type consists of one line per successful signature verification. Each line has four structured fields delimited by a single space, followed by a single-line JSON object or arbitrary text to the end of the line.

* ISO-8601 UTC datestamp of the signature, to one second precision, using the Z suffix

* 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)

* A string describing the mode of the signature, either mode:text or mode:binary

* A JSON object or free-form message describing the verification (see Section 7.10.1)

Note that while Section 6.1 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

* the time MUST be emitted with one-second precision

Example:

```
2019-10-24T23:48:29Z C90E6D36200A1B922A1509E77618196529AE5FF8 C4BC2DDB38CCE96485EBE9C2F20
691179038E5C6 mode:binary {"signers": ["dkg.asc"]}
```

7.10.1. VERIFICATIONS extension JSON

The final field of each VERIFICATIONS line is either JSON data or arbitrary text.

If the final field begins and ends with curly brackets ("{" (LEFT CURLY BRACKET, U+007B) and "}" (RIGHT CURLY BRACKET, U+007D), it is JSON data. Otherwise, the final field is arbitrary text (whose content and structure are up to the discretion of the implementation).

JSON data allows for sophisticated future extensions, and is the preferred form of this field. Arbitrary text is deprecated. The rest of this subsection describes the JSON data.

The JSON data is a single JSON object, coerced into a one-line representation (there are no literal LINE FEED (U+000A) characters in it, though there may be appropriately escaped LINE FEED characters within the JSON text).

The JSON object MAY contain the following keys:

* signers: a list the supplied CERTS objects that could have issued the signature, identified by the name given on the command line. If this key is present, as long as any OpenPGP certificate in a given CERTS object could have issued the signature, that CERTS object MUST be listed here. If multiple CERTS objects contain certificates that could have issued the signature, each CERTS MUST be listed here.

* comment: Free-form UTF-8-encoded text describing the verification. An internationalized, locale-aware sop implementation should localize this field.

* ext: A "extensions" JSON object containing arbitrary, implementation-specific data.

To avoid collisions with future definitions, the top-level JSON object MUST NOT contain any other keys. For forward compatibility, when consuming a JSON object produced by a SOP implementation, unknown keys MUST be ignored.

7.11. 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=text. If sop receives DATA containing non-UTF-8 octets in this case, it will fail (see Section 11.7) with EXPECTED_TEXT.

7.12. PROFILELIST

This output-only type consists of simple UTF-8 textual output, with one line per profile. Each line consists of the profile name optionally followed by a colon (0x31), a space (0x20), and a brief human-readable description of the intended semantics of the profile. Each line may be at most 1000 bytes, and no more than 4 profiles may be listed.

These limits are intended to force sop implementers to make hard decisions and to keep things simple.

The first profile MAY be explicitly named default. If it is not named default, then default is an alias for the first profile listed. No profile after the first listed may be named default.

Any of the profiles MAY be explicitly named security, performance, or compatibility. If none of the listed profiles have (some of) these names, the profiles of which they are an alias should indicate as much in the human-readable description.

See Section 6.4 for more discussion about the namespace and intended semantics of each profile.

8. 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
1	UNSPECIFIED_FAILURE	An otherwise unspecified failure occurred
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, sop generate-key)
37	UNSUPPORTED_OPTION	Unsupported option
41	BAD_DATA	Invalid data type (no secret key where KEYS expected, secret key where CERTS 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 password-protected (locked), and sop cannot unlock it with any of the --with-key-password (or --old-key-password) options
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)
83	INCOMPATIBLE_OPTIONS	Options were supplied that are incompatible with each other
89	UNSUPPORTED_PROFILE	The requested profile is unsupported (sop generate-key, sop encrypt), or the indicated subcommand does not accept profiles (sop list-profiles)
97	NO_HARDWARE_KEY_FOUND	The sop implementation supports some form of

		hardware-backed secret keys, but could not identify the hardware device (see Section 11.10)
101	HARDWARE_KEY_FAILURE	The sop implementation tried to use a hardware-backed secret key, but the cryptographic hardware refused the operation for some reason other than a bad PIN or password (see Section 11.10)
103	PRIMARY_KEY_BAD	The primary key of a KEYS object is too weak or revoked
107	CERT_USERID_NO_MATCH	The CERTS object has no matching User ID
109	KEY_CANNOT_CERTIFY	Key not certification-capable (e.g., expired, revoked, unacceptable usage flags) (sop certify-userid)

Table 1: Error return codes

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

9. Known Implementations

The following implementations are known at the time of this draft:

Project name	cli name	notes
dkg-sop	dkg-sop	Implemented in C++ using the LibTMCG library ([DKG-SOP])
gosop	gosop	Implemented in golang (Go) using GOpenPGP ([GOSOP])
gpgme-sop	gpgme-sop	A Rust wrapper around the gpgme C library ([GPGME-SOP])
PGPainless SOP	pgpainless-cli	Implemented in Java using PGPainless ([PGPAINLESS-CLI])
RNP-sop	rnp-sop	A Rust wrapper around the librnp C library ([RNP-SOP])
rsop	rsop	Implemented in Rust using the rpgpie crate ([RSOP])
Sequoia SOP	sqop	Implemented in Rust using the sequoia-openpgp crate ([SQOP])
sop-openpgp.js	sop-openpgp.js	Implemented in JavaScript using OpenPGP.js ([SOP-OPENPGPJS])
sopgpy	sopgpy	Implemented in Python using PGPy ([SOPGPY])

Table 2: Known implementations

10. 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.

11. Guidance for Implementers

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

11.1. 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.

11.2. 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 7.4).

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.

11.3. 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.

11.4. 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 [RFC9580] 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 [RFC9580] 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
* More than one ASCII-armored object in the input

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

For any plural data type (i.e.,SIGNATURES, CERTS, or KEYS), the unarmored form is trivially concatenatable with another object of the same type (e.g., with Unix's cat utility). But the armored forms are not concatenatable without first dearmoring. To avoid inconsistent behavior, a sop implementation SHOULD reject anything that appears to be a concatenated series of ASCII-armored objects.

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, INLINESIGNED, or CIPHERTEXT), sop SHOULD also reject any input that does not conform to the expected packet stream. See Section 7 for the expected packet stream for different types.

11.5. Complexities of the Cleartext Signature Framework

sop prefers a detached signature as the baseline form of OpenPGP signature, but provides affordances for dealing with inline-signed messages (see INLINESIGNED, Section 7.5) as well.

The most complex form of inline-signed messages is the Cleartext Signature Framework (CSF). Handling the CSF structure 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].

A user of sop that receives an inline-signed message (whether the message uses the CSF or not) can detach the signature from the message with sop inline-detach (see Section 5.4.5).

Alternately, the user can send the message through sop inline-verify to confirm required signatures, and then (if signatures are valid) supply its output to the consumer of the signed message.

11.6. 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.

11.7. 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 [RFC9580] 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.

11.8. 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 it is important to ensure that there is a reliable bidirectional mapping between strings and bytes. The maximally robust behavior here is for sop encrypt and sop generate-key (that is, commands that use a password to encrypt) to constrain the choice of passwords to strings that have such a mapping, and for sop decrypt and sop sign (and sop inline-sign, as well assop encrypt when decrypting a signing key; that is, commands that use a password to decrypt) to try multiple plausible versions of any password supplied by PASSWORD.

11.8.1. Generating Material with Human-Readable Passwords

When generating material based on a password, sop encrypt and sop generate-key enforce that the password is actually meaningfully human-transferable. In particular, an implementation generating material based on a new password SHOULD apply the following considerations to the supplied password:

* require UTF-8

* trim trailing whitespace

Some sop encrypt and sop generate-key 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 or sop generate-key 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 or sop generate-key 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.

11.8.2. Consuming Password-protected Material

When sop decrypt receives a PASSWORD input, either from a --with-key-password or --with-password option, it sees its content as a bytestring. sop sign also sees the content of any PASSWORD input supplied to its --with-key-password option 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 or sop sign 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 or sop sign 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.

11.9. Be Careful with Special Designators

As documented in Section 7.1, 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.

11.10. Nuances for Hardware-backed Secret Key Material

There are a number of limitations and nuances to be aware of for hardware-backed secret key support in this interface. Some sop implementations will simply not support hardware-backed secret key material. Other implementations might support only a single kind of hardware-backing (e.g., an OpenPGP Smartcard [OPENPGP-SMARTCARD] but not a TPM, or vice versa).

There is no formally adopted OpenPGP standard for identifying that a given secret key is backed by hardware based on the OpenPGP wire format. [I-D.dkg-openpgp-hardware-secrets] proposes one simple and straightforward approach for how the wire format could cover this use case. This simple mechanism is deliberately agnostic about the specific kind of cryptographic hardware, but it does imply a sort of rough shape of what the interface to the hardware would permit. In particular, it will work best with hardware that has the following properties:

* The hardware does specific asymmetric secret key operations, using secret keys that it does not release.

* The user can ask the hardware to provide a list of corresponding public key material (or OpenPGP key fingerprints) for any of the secret keys held by the device.

* The hardware MAY require the provision of a PIN or password to enable secret key operation, but does not require a PIN or password for the list of public key material.

The sop interface does not currently provide for provisioning cryptographic hardware with secret key material, or for changing the PIN or password for the cryptographic hardware. Users of cryptographic hardware need to do provisioning and PIN or password setting outside of sop.

If a user has two attached hardware tokens that both hold the same secret key, and they are both password-locked, and they use different passwords, sop offers no way for the user to clearly indicate which password belongs to which device. Some cryptographic hardware is designed to lock the device if the wrong password is entered too many times, so users in this configuration are at risk of accidental lockout. The easiest resolution for this is for the user to detach any duplicate devices before invoking sop.

Note that some OpenPGP implementations use the private codepoint ranges in the OpenPGP specification within an OpenPGP Transferable Secret Key (e.g., [GNUPG-SECRET-STUB]) to indicate that the secret key can be found on a smartcard.

While hardware-backed secret key operations can be significantly slower than modern computers, and physical affordances like button-presses or NFC tapping can themselves incur delay, it’s bad form for an invocation of sop to hang forever. This specification doesn’t define a specific maximum allowable delay, but if an implementation calls into a hardware device either for public key listing or for secret key operations, it should not allow the cryptographic hardware to take an arbitrary amount of time to respond.

11.11. Statelessness exemptions

While this specification strives to define all operations as stateless implementers MAY, for practical reasons, rely on the global state of the system.

For example, the following items constitute a system state but are not considered to violate the stateless rule:

* current time

Implementers are advised to document which global state items they rely on to help in troubleshooting issues for consumers.

12. 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?

12.1. 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]).

12.2. 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 7.1).

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 --verifications-out=@FD:3) also results in an ambiguous data stream.

13. Security Considerations

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

13.1. 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 over data (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)
* The signature must be of type 0x00 ("Signature of a binary document") or 0x01 ("Signature of a canonical text document"); other signature types are inappropriate for data signatures

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.

13.1.1. Explaining Non-Verification on Standard Error

When verifying OpenPGP signatures, sometimes no valid signatures are found. This will cause the verifying subcommand to produce an empty VERIFICATIONS output, and for some subcommands (sop verify and sop inline-verify in particular) will also cause the subcommand to fail with NO_SIGNATURE.

When this happens, some consumers will want to know more details about the verification failure, since some verification failures may be indications that something is wrong with the verifier’s setup, such as outdated OpenPGP implementations (which can be upgraded), expired signing certificates (which can be refreshed), and so on.

To address this, when no valid signatures are found at all, sop MAY emit a human-readable explanation to standard error.

Some example explanations for complete signature validation failure include:

* Version 7 signature found, but FooPGP 2.0.3 only supports versions 4 and 6.

* Version 3 signature found, but BarPGP 0.9.7 rejects all version 3 signatures.

* Signature from pubkey algorithm 94 found, but BazPGP 1.1 does not support this pubkey algorithm.

* Signature using hash algorithm 22 found, but QuxPGP 19.0.5 does not support this hash algorithm.

* Two signatures found, both made by unknown OpenPGP certificates.

* Signature does not match hash prefix.

* No OpenPGP signatures found.

In some cases (such as when two OpenPGP signatures are discovered, and they both fail to validate for different reasons), a sop implementation may choose to emit a more complex warning.

Unless --debug is present, sop SHOULD NOT emit any such warning (even if true for one of the OpenPGP signatures found) if another signature was found in the same SIGNATURES object or INLINESIGNED message that does verify correctly. This keeps the upgrade path smooth for the whole ecosystem. As the ecosystem evolves, signatures using new versions and algorithms, or signatures simply using new signing keys, are typically introduced as a second signature distributed alongside the first. A warning about a signature with a new or unknown algorithm (or key) when an accompanying signature still verifies from a known key with a known algorithm will discourage signers from adopting new algorithms or keys. And introducing a warning about a signature using a deprecated algorithm (or key), when an accompanying signature still verifies using a more modern algorithm or key will discourage a verifier from upgrading their OpenPGP implementation or dropping old, deprecated keys.

Implementers should avoid emitting dangerous explanations. For example, an explanation like "Signature from 0xDEADBEEF found, but not in list of acceptable signers" might encourage a user to go hunting for any certificate with short key ID 0xDEADBEEF and start using it to verify signatures. This would be a very dangerous explanation, in particular because short key IDs are trivially forgeable. But it would also be nearly as dangerous to use a full fingerprint (instead of a short Key ID) in such a message because then all an attacker has to do is to get their signature to appear in the place where the verifier is looking for a signature, and then the warning will encourage the verifier go look up the attacker’s certificate by fingerprint.

An internationalized, locale-aware sop implementation should localize these warning messages.

13.2. 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.

14. 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 [RFC9580])). FIXME: document things like PURBs and --hidden-recipient)

14.1. 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.

15. References

15.1. Normative References

[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/rfc/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/rfc/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/rfc/rfc3629>.

[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/rfc/rfc8174>.

[RFC9580] Wouters, P., Ed., Huigens, D., Winter, J., and Y. Niibe, "OpenPGP", RFC 9580, DOI 10.17487/RFC9580, July 2024, <https://www.rfc-editor.org/rfc/rfc9580>.

15.2. 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>.

[DKG-SOP] Stamer, H., "dkg-sop", n.d., <https://git.savannah.nongnu.org/cgit/dkgpg.git/tree/tools/dkg-sop.cc>.

[EFAIL] Poddebniak, D. and C. Dresen, "Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels", n.d., <https://efail.de>.

[GNUPG-SECRET-STUB]
Koch, W., "GNU Extensions to the S2K algorithm", 4 July 2023, <https://dev.gnupg.org/source/gnupg/browse/master/doc/DETAILS;gnupg-2.4.3$1511>.

[GOSOP] Proton, "gosop", n.d., <https://github.com/ProtonMail/gosop>.

[GPGME-SOP]
Winter, J., "gpgme-sop", n.d., <https://gitlab.com/sequoia-pgp/gpgme-sop>.

[I-D.dkg-openpgp-hardware-secrets]
Gillmor, D. K., "OpenPGP Hardware-Backed Secret Keys", Work in Progress, Internet-Draft, draft-dkg-openpgp-hardware-secrets-02, 19 April 2024, <https://datatracker.ietf.org/doc/html/draft-dkg-openpgp-hardware-secrets-02>.

[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://datatracker.ietf.org/doc/html/draft-bre-openpgp-samples-01>.

[I-D.ietf-lamps-e2e-mail-guidance-11]
Gillmor, D. K., Hoeneisen, B., and A. Melnikov, "Guidance on End-to-End E-mail Security", Work in Progress,

Internet-Draft, draft-ietf-lamps-e2e-mail-guidance-11, 8 August 2023, <https://datatracker.ietf.org/doc/html/draft-ietf-lamps-e2e-mail-guidance-11>.

[OpenPGP-Interoperability-Test-Suite]
"OpenPGP Interoperability Test Suite", 25 October 2021, <https://tests.sequoia-pgp.org/>.

[OPENPGP-SMARTCARD]
Pietig, A., "Functional Specification of the OpenPGP application on ISO Smart Card Operating Systems, Version 3.4", 18 March 2020, <https://www.gnupg.org/ftp/specs/OpenPGP-smart-card-application-3.4.pdf>.

[PGPAINLESS-CLI]
Schaub, P., "pgpainless-cli", n.d., <https://codeberg.org/PGPainless/pgpainless/src/branch/master/pgpainless-sop>.

[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/rfc/rfc959>.

[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/rfc/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/rfc/rfc4880>.

[RFC5321] Klensin, J., "Simple Mail Transfer Protocol", RFC 5321, DOI 10.17487/RFC5321, October 2008, <https://www.rfc-editor.org/rfc/rfc5321>.

[RNP-SOP] Winter, J., "rnp-sop", n.d., <https://gitlab.com/sequoia-pgp/rnp-sop>.

[RSOP] Schaefer, H., "rsop", n.d., <https://codeberg.org/heiko/rsop>.

[RUST-SOP] Winter, J., "A Rust implementation of the Stateless OpenPGP Protocol", n.d., <https://sequoia-pgp.gitlab.io/sop-rs/>.

[SEMVER] 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/sop-java>.

[SOP-OPENPGPJS] Proton, "sop-openpgp.js", n.d., <https://github.com/openpgpjs/sop-openpgpjs>.

[SOPGPY] Gillmor, D. K., "sopgpy", n.d., <https://github.com/SecurityInnovation/PGPy/pull/440>.

[SQOP] Sequoia PGP, "sqop", n.d., <https://gitlab.com/sequoia-pgp/sequoia-sop>.

[UNICODE-NORMALIZATION] Whistler, K., "Unicode Normalization Forms", 4 February 2019, <https://unicode.org/reports/tr15/>.

Appendix A. sopv Version Changelog

This is a reverse-chronological order changelog for the sopv subset. This versioning scheme aims for compliance with [SEMVER].

A.1. sopv Version 1.2

sopv 1.2 adds the certificate verification, via the following subcommand:

* sop validate-userid

A.2. sopv Version 1.1

* VERIFICATIONS output always includes the fourth mode: field

* VERIFICATIONS output always uses JSON format for the trailer of each line, and always populates the signers member (see Section 7.10.1)

A.3. sopv Version 1.0

The following subcommands:

* sop version
* sop verify
* sop inline-verify

And the following features:

* Special designators @FD: and @ENV: as input for any CERTS object or SIGNATURES object
* Special designator @FD: as possible output for the VERIFICATIONS object in sopv inline-verify --verifications-out
* Multiple certificates in each CERTS object
* --not-before and --not-after constraints

Appendix B. C Library API (Tentative)

As specified in this draft, SOP is a command-line tool.

However, it can also be useful to have a comparable API exposed as a C library. This library can be implemented as a shared object (e.g., .so, .dll, or .dylib depending on the platform) or as a statically linked object. This interface can be reused in many different places, as most modern programming languages offer "bindings" to C libraries.

A proposed interface to a C library follows here as a C header file.

The primary goal of this shared object interface is to make it easy to implement the command-line interface described in this document. That said, it is also intended to be relatively ergonomic to use in plausible OpenPGP workflows where the caller has access to all of the explicit state.

If there is a plausible OpenPGP workflow that is not supported by this library API, please propose improvements and explain the specific workflow.

The verification-only subset is defined in sopv.h:

```
   /* -*- mode: c; fill-column: 60; -*- */
   #ifndef __SOPV_H__
   #define __SOPV_H__

   #include <stdlib.h>
   #include <stdint.h>
   #include <stdbool.h>
   #include <string.h>
   #include <limits.h>

   /* C API for Stateless OpenPGP Verification-only Subset */

   /* Depends on C99 */


   /* statically-defined, non-opaque definitions */

   typedef enum {
     SOP_OK = 0,
     SOP_INTERNAL_ERROR = 1, /* Not part of sop CLI */
     SOP_INVALID_ARG = 2, /* Not part of sop CLI */
     SOP_NO_SIGNATURE = 3,
     SOP_OPERATION_ALREADY_EXECUTED = 4, /* Not part of sop CLI */
     SOP_UNSUPPORTED_ASYMMETRIC_ALGO = 13,
     SOP_CERT_CANNOT_ENCRYPT = 17,
     SOP_MISSING_ARG = 19,
     SOP_INCOMPLETE_VERIFICATION = 23,
     SOP_CANNOT_DECRYPT = 29,
     SOP_PASSWORD_NOT_HUMAN_READABLE = 31,
     SOP_UNSUPPORTED_OPTION = 37,
     SOP_BAD_DATA = 41,
     SOP_EXPECTED_TEXT = 53,
     SOP_OUTPUT_EXISTS = 59,
     SOP_MISSING_INPUT = 61,
     SOP_KEY_IS_PROTECTED = 67,
     SOP_UNSUPPORTED_SUBCOMMAND = 69,
     SOP_UNSUPPORTED_SPECIAL_PREFIX = 71,
     SOP_AMBIGUOUS_INPUT = 73,
     SOP_KEY_CANNOT_SIGN = 79,
     SOP_INCOMPATIBLE_OPTIONS = 83,
     SOP_UNSUPPORTED_PROFILE = 89,
     SOP_NO_HARDWARE_KEY_FOUND = 97,
     SOP_HARDWARE_KEY_FAILURE = 101,
     SOP_PRIMARY_KEY_BAD = 103,
     SOP_CERT_USERID_NO_MATCH = 107,
     SOP_KEY_CANNOT_CERTIFY = 109,

     /* ensures a stable size for the enum -- do not use! */
```

```
    SOP_MAX_ERR = INT_MAX,
  } sop_err;


  typedef enum {
    SOP_SIGN_AS_BINARY = 0,
    SOP_SIGN_AS_TEXT = 1,

    /* ensures a stable size for the enum -- do not use! */
    SOP_SIGN_AS_MAX = INT_MAX,
  } sop_sign_as;


  /* timestamps */
  /* sop_time represents the number of seconds since the UNIX
   * epoch (1970-01-01T00:00:00Z) in a 64-bit signed integer.
   *
   * OpenPGP wire format timestamps (through RFC 9580 at
   * least) are internally unsigned 32-bit integers, which all
   * fall well within the range of sop_time.
   *
   * Despite sharing similar semantics to the traditional
   * time_t, sop_time doesn't use it explicitly, because on
   * some architectures time_t is a signed 32-bit integer,
   * which will roll over in 2038 and cannot express the range
   * of valid OpenPGP wire format timestamps.
   *
   * We also want sop_time to be able to explicitly represent
   * a "none" value as well as a "now" value without those
   * choices aliasing some legitimate value in the explicitly
   * supported wire format range. */
  typedef int64_t sop_time;
  #define sop_time_none ((sop_time)(INT64_MIN))
  #define sop_time_now ((sop_time)(INT64_MIN+1))


  /* Context object
   *
   * Each SOP object is bound back to a context object, and,
   * when used in combination with other SOP objects, all SOP
   * objects should come from the same context.
   *
   * A SOP context object need not be thread-safe; it should
   * probably not be used across multiple threads.  See "Zero
   * global state" in the README file in
   * https://git.kernel.org/pub/scm/linux/kernel/git/kay/libabc.git
   *
   * sopv: 1.0
```

```
    */

   struct sop_ctx_st;
   typedef struct sop_ctx_st sop_ctx;

   sop_ctx*
   sop_ctx_new ();
   void
   sop_ctx_free (sop_ctx *sop);

   /* Logging: */

   typedef enum {
     SOP_LOG_NEVER = 0,
     SOP_LOG_ERROR = 1,
     SOP_LOG_WARNING = 2,
     SOP_LOG_INFO = 3,
     SOP_LOG_DEBUG = 4,

     /* ensures a stable size for the enum -- do not use! */
     SOP_LOG_MAX = INT_MAX,
   } sop_log_level;

   static inline const char *
   sop_log_level_name (sop_log_level log_level) {
   #define rep(x) if (log_level == SOP_LOG_ ## x) return #x
     rep(ERROR);
     rep(WARNING);
     rep(INFO);
     rep(DEBUG);
   #undef rep
     return "Unknown";
   }

   /* Handle warnings and other feedback.
    *
    * A SOP implementation that is capable of producing log
    * messages will invoke the requested function with the log
    * level of the message, and a NULL-terminated UTF-8
    * human-readable string with no trailing whitespace.
    *
    * the "passthrough" pointer is supplied by the library user
    * via sop_set_log_level.
    *
    * sopv: 1.0
    */
   typedef void (*sop_log_func) (sop_log_level log_level,
                                 void *passthrough, const char *);
```

```
   sop_err
   sop_set_log_function (sop_ctx *sop, sop_log_func func,
                         void *passthrough);
   /* Set the logging verbosity.
    *
    * Only log warnings up to max_level. (by default, max_level
    * is SOP_LOG_WARNING, meaning SOP_LOG_INFO and
    * SOP_LOG_DEBUG will be suppressed).
    *
    * sopv: 1.0
    */
   sop_err
   sop_set_log_level (sop_ctx *sop, sop_log_level max_level);



   /* Information about the library: */

   /* The name and version of the implementation of the C API
    * (simple NUL-terminated string, no newlines), or NULL if
    * there is an error producing the version.
    *
    * sopv: 1.0
    */
   const char *
   sop_version (sop_ctx *sop);
   /* The name and version of the primary underlying OpenPGP
    * toolkit (or NULL if there is no backend, or if there was
    * an error producing the backend version)
    *
    * sopv: 1.0
    */
   const char *
   sop_version_backend (sop_ctx *sop);
   /* Any arbitrary extended version information other than
    * sop_ctx_version.  Version info should be UTF-8 text,
    * separated by newlines (a NUL-terminated string, no
    * trailing newline).  Can return NULL if there is nothing
    * more to report beyond sop_version.
    *
    * sopv: 1.0
    */
   const char *
   sop_version_extended (sop_ctx *sop);

   /* note: there is nothing comparable to sop version
    * --sop-spec because that should be visible based on the
    * exported symbols in the shared object */
```

```
   /* CLEARTEXT (and other raw data): */

   /* This is a standard buffer for bytestrings produced by
    *  sop.  Users never create this kind of object, but it is
    *  sometimes returned from the library.
    *
    * sopv: 1.0
    */
   struct sop_buf_st;
   typedef struct sop_buf_st sop_buf;

   void
   sop_buf_free (sop_buf *buf);
   size_t
   sop_buf_size (const sop_buf *buf);
   const uint8_t *
   sop_buf_data (const sop_buf *buf);


   /* CERTS objects: This object represents a collection of
    * OpenPGP Certificates (Transferable Public Keys).  It can
    * also hold a copy of a caller-supplied label, which is a
    * NULL-terminated C string.
    *
    * sopv: 1.0
    */
   struct sop_certs_st;
   typedef struct sop_certs_st sop_certs;

   /* data and len indicate the contiguous block of bytes that
    * will be parsed to create the CERTS object.
    *
    * label is a NULL-terminated C string.  It may be NULL.  If
    * label is not NULL, the implementation makes an internal
    * copy of the C string during sop_certs_from_bytes.
    */
   sop_err
   sop_certs_from_bytes (sop_ctx *sop,
                         const uint8_t *data, size_t len,
                         const char *label,
                         sop_certs **out);
   void
   sop_certs_free (sop_certs *certs);


   /* Retreive a const pointer to the internal copy of the
    * CERTS object's label, or NULL if there is no label for
    * this CERTS object.
```

```
    *
    * sopv: 1.1
    */
   sop_err
   sop_certs_get_label (const sop_certs *certs,
                        const char **out);

   /* SIGNATURES objects: This object represents a collection
    * of OpenPGP Signature packets
    *
    * sopv: 1.0
    */
   struct sop_sigs_st;
   typedef struct sop_sigs_st sop_sigs;

   sop_err
   sop_sigs_from_bytes (sop_ctx *sop,
                        const uint8_t *data, size_t len,
                        sop_sigs **out);
   void
   sop_sigs_free (sop_sigs *sigs);



   /* VERIFICATIONS (output only, describes valid, verified
    * signatures):
    *
    * sopv: 1.0
    */
   struct sop_verifications_st;
   typedef struct sop_verifications_st sop_verifications;

   void
   sop_verifications_free (sop_verifications *verifs);
   sop_err
   sop_verifications_count (const sop_verifications *verifs,
                            size_t *out);
   /* textual representations of verifications, in the form
    * described by VERIFICATIONS in the CLI
    *
    * sopv: 1.0
    */
   sop_err
   sop_verifications_to_text (const sop_verifications *verifs,
                              sop_buf **out);
   /* returns SOP_INTERNAL_ERROR if index is out of bounds. */
   sop_err
   sop_verifications_get_time (const sop_verifications *verifs,
```

```
                              size_t index, sop_time *out);
   /* returns SOP_INTERNAL_ERROR if index is out of bounds.  If
    * the signature is neither type 0x00 nor 0x01, this should
    * probably not be considered a valid, verified signature.
    *
    * sopv: 1.1
    */
   sop_err
   sop_verifications_get_mode (const sop_verifications *verifs,
                               size_t index, sop_sign_as *out);

   /* returns SOP_INTERNAL_ERROR if index is out of bounds.
    *
    * sopv: 1.1
    */
   sop_err
   sop_verifications_get_signer_count (const sop_verifications *verifs,
                                       size_t index, size_t *out);

   /* returns SOP_INTERNAL_ERROR if either verif_index or
    * signer_index is out of bounds.  Yields a pointer to the
    * sop_certs object that could have made the signature.
    *
    * sopv: 1.1
    */
   sop_err
   sop_verifications_get_signer (const sop_verifications *verifs,
                                 size_t verif_index,
                                 size_t signer_index,
                                 const sop_certs **out);

   /* FIXME: (do we want to get more detailed info
      programmatically?  each verification should also have an
      issuing key fingerprint, a primary key fingerprint, and a
      trailing text string) */

   /* verify detached signatures:
    *
    *
    * sopv: 1.0
    */
   struct sop_op_verify_st;
   typedef struct sop_op_verify_st sop_op_verify;

   sop_err
   sop_op_verify_new (sop_ctx *sop, sop_op_verify **out);
   void
   sop_op_verify_free (sop_op_verify *verify);
```

```
   sop_err
   sop_op_verify_not_before (sop_op_verify *verify, sop_time when);
   sop_err
   sop_op_verify_not_after (sop_op_verify *verify, sop_time when);
   sop_err
   sop_op_verify_add_signers (sop_op_verify *verify,
                              const sop_certs *signers);

   /* if no verifications are possible with the set of signers,
    * this returns SOP_NO_SIGNATURE, and *out is set to NULL
    *
    * sopv: 1.0
    */
   sop_err
   sop_op_verify_detached_execute (sop_op_verify *verify,
                                   const sop_sigs *sigs,
                                   const uint8_t *msg,
                                   size_t sz,
                                   sop_verifications **out);



   /* INLINESIGNED object:
    *
    *
    * sopv: 1.0
    */
   struct sop_inlinesigned_st;
   typedef struct sop_inlinesigned_st sop_inlinesigned;

   sop_err
   sop_inlinesigned_from_bytes (sop_ctx *sop,
                                const uint8_t *data, size_t len,
                                sop_inlinesigned **out);
   /* if the inlinesigned object uses the Cleartext Signing
    * framework, the armor parameter is ignored.
    *
    * sopv: 1.0
    */
   void
   sop_inlinesigned_free (sop_inlinesigned *inlinesigned);


   /* sop inline-verify
    *
    *
    * sopv: 1.0
    */
```

```
   sop_err
   sop_op_verify_inline_execute (sop_op_verify *verify,
                                 const sop_inlinesigned *msg,
                                 sop_verifications **verifications_out,
                                 sop_buf **msg_out);

   /* sop validate-userid
    *
    * sopv: 1.2
    */
   struct sop_op_validate_userid_st;
   typedef struct sop_op_validate_userid_st sop_op_validate_userid;

   sop_err
   sop_op_validate_userid_new (sop_ctx *sop,
                               sop_op_validate_userid **out);

   void
   sop_op_validate_userid_free (sop_op_validate_userid *validate);

   sop_err
   sop_op_validate_userid_add_authority (sop_op_validate_userid *validate,
                                         const sop_certs *authority);

   sop_err
   sop_op_validate_userid_at (sop_op_validate_userid *validate,
                              sop_time when);

   sop_err
   sop_op_validate_userid_addr_spec_only (sop_op_validate_userid *validate,
                                          bool addr_spec_only);

   sop_err
   sop_op_validate_userid_execute (sop_op_validate_userid *validate,
                                   const char *userid,
                                   bool *out)

   #endif /* __SOPV_H__ */
```

The rest of the stateless OpenPGP functionality is in sop.h, which explicitly includes sopv.h:

```
   /* -*- mode: c; fill-column: 60; -*- */

   #ifndef __SOP_H__
   #define __SOP_H__

   #include <sopv.h>
```

```
   /* C API for Stateless OpenPGP */
   /* Depends on C99 */

   typedef enum {
     SOP_INLINE_SIGN_AS_BINARY = 0,
     SOP_INLINE_SIGN_AS_TEXT = 1,
     SOP_INLINE_SIGN_AS_CLEARSIGNED = 2,

     /* ensures a stable size for the enum -- do not use! */
     SOP_INLINE_SIGN_AS_MAX = INT_MAX,
   } sop_inline_sign_as;

   typedef enum {
     SOP_ENCRYPT_AS_BINARY = 0,
     SOP_ENCRYPT_AS_TEXT = 1,

     /* ensures a stable size for the enum -- do not use! */
     SOP_ENCRYPT_AS_MAX = INT_MAX,
   } sop_encrypt_as;

   /* Serialize objects that sopv can read: */
   sop_err
   sop_certs_to_bytes (const sop_certs *certs,
                       bool armor, sop_buf **out);
   sop_err
   sop_sigs_to_bytes (const sop_sigs *sigs,
                      bool armor, sop_buf **out);
   sop_err
   sop_inlinesigned_to_bytes (const sop_inlinesigned *inlinesigned,
                              bool armor, sop_buf **out);

   /* PROFILE objects: */

   /* These describe a profile (e.g. for generate-key or
    * encrypt).  This use used when the implementation might
    * legitimately want to offer the user some minimal amount
    * of control over what is done.  The profile-listing
    * functions return blocks of four profiles.  A sop_profile
    * value of NULL represents no profile at all.  In a list of
    * sop_profile objects, once a NULL profile appears, no
    * non-NULL profiles may follow.

    */
   struct sop_profile_st;
   typedef struct sop_profile_st sop_profile;
   /* the NUL-terminated string returned by sop_profile_name
      MUST be a UTF-8 encoded string, and MUST NOT include any
      whitespace or colon (':') characters.  It MUST NOT vary
```

```
      depending on locale. */
   const char *
   sop_profile_name (const sop_profile *profile);
   /* The NUL-terminated string returned by
      sop_profile_description cannot contain any newlines, and
      it MAY vary depending on locale(7) if the implementation
      is internationalized. */
   const char *
   sop_profile_description (const sop_profile *profile);


   #define SOP_MAX_PROFILE_COUNT 4

   typedef struct {
     sop_profile *profile[SOP_MAX_PROFILE_COUNT];
   } sop_profiles;

   static inline size_t
   sop_profiles_count(const sop_profiles profiles) {
     for (size_t i = 0; i < SOP_MAX_PROFILE_COUNT; i++)
       if (profiles.profile[i] == NULL)
         return i;
     return SOP_MAX_PROFILE_COUNT;
   }

   /* Return a list of profiles supported by the library for
    * generating keys.
    */
   sop_err
   sop_list_profiles_generate_key (sop_ctx *sop, sop_profiles *out);



   /* KEYS objects: */
   struct sop_keys_st;
   typedef struct sop_keys_st sop_keys;

   sop_err
   sop_keys_from_bytes (sop_ctx *sop,
                        const uint8_t *data, size_t len,
                        sop_keys **out);
   sop_err
   sop_keys_to_bytes (const sop_keys *keys,
                      bool armor, sop_buf **out);
   void
   sop_keys_free (sop_keys *keys);
```

```
   /* Generate a new, minimal OpenPGP Transferable secret key.
      'profile' can be NULL to mean the default profile. */
   sop_err
   sop_generate_key_with_profile (sop_ctx *sop,
                                  sop_profile *profile,
                                  bool sign_only,
                                  sop_keys **out);

   static inline sop_err
   sop_generate_key (sop_ctx *sop, sop_keys **out) {
     return sop_generate_key_with_profile (sop, NULL, false, out);
   }

   /* For each key in the sop_keys object, add the given user
      ID, and return a new sop_keys object containing the
      updated keys.  If the supplied user ID is not valid UTF-8
      text, this call will fail and return SOP_EXPECTED_TEXT.

      If the implementation rejects the user ID string by
      policy for any other reason, this call will fail and
      return SOP_BAD_DATA.
    */
   sop_err
   sop_keys_add_uid (const sop_keys *keys, const char *uid,
                     sop_keys **out);

   /* returns true if any of the secret key material is
      currently locked with a password */
   sop_err
   sop_keys_locked (const sop_keys *keys, bool *out);

   /* return a new sop_keys object with any secret key material
      encrypted with 'password' unlocked, Returns SOP_OK if all
      keys have now been unlocked.

      If any locked key material could not be unlocked, return
      SOP_KEY_IS_PROTECTED, while also unlocking what key
      material can be unlocked.

      This allows the user to try an arbitrary bytestream as a
      password.  Most users will just invoke the inlined
      sop_keys_unlock, below.

      An implementation MUST NOT reject proposed passwords by
      policy during unlock, but rather should try them as
      requested.
   */
   sop_err
```

```
   sop_keys_unlock_raw (const sop_keys *keys,
                        const uint8_t *raw_password, size_t len,
                        sop_keys **out);


   static inline sop_err
   sop_keys_unlock (const sop_keys *keys, const char *password,
                    sop_keys **out) {
     return sop_keys_unlock_raw (keys,
                                 (const uint8_t *)password,
                                 strlen (password),
                                 out);
   }


   /* return a new sop_keys object where all secret key
      material is locked with ‘password‘ where possible.

      During locking, a safety-oriented implementation MAY
      reject the supplied password by policy for any number of
      reasons.  This helps libsop ensure that the proposed
      password can be successfully re-supplied during some
      future unlock attempt.

      If the implementation requires passwords to be UTF-8 text
      and the supplied password is not valid UTF-8, the
      implementation will fail, returning SOP_EXPECTED_TEXT.
      If an implementation rejects a supplied password for some
      other reason (for example, if it contains an NUL,
      unprintable, or otherwise forbidden character), this call
      will fail and return SOP_BAD_DATA.

      If any key material is already locked, it does nothing
      and returns SOP_KEY_IS_PROTECTED.

      Upon a successful locking, the user probably wants to use
      sop_keys_free to free the original keys object.
   */
   sop_err
   sop_keys_lock_raw (const sop_keys *keys,
                      const uint8_t *password, size_t len,
                      sop_keys **out);

   static inline sop_err
   sop_keys_lock (const sop_keys *keys, const char *password,
                  sop_keys **out) {
     return sop_keys_lock_raw (keys,
                               (const uint8_t *)password,
```

```
                              strlen (password),
                              out);
   }



   /* Return the OpenPGP certificates ("Transferable Public
      Keys") that correspond to the OpenPGP Transferable Secret
      Keys. */
   sop_err
   sop_keys_extract_certs (const sop_keys *keys, sop_certs **out);


   /* Return an OpenPGP revocation certificate for each
      Transferable Secret Key found in the input. */
   sop_err
   sop_keys_revoke_keys (const sop_keys *keys, sop_certs **out);



   /* create detached signatures: */
   struct sop_op_sign_st;
   typedef struct sop_op_sign_st sop_op_sign;

   sop_err
   sop_op_sign_new (sop_ctx *sop, sop_op_sign **out);
   void
   sop_op_sign_free (sop_op_sign *sign);

   sop_err
   sop_op_sign_use_keys (sop_op_sign *sign, const sop_keys *keys);

   sop_err
   sop_op_sign_detached_execute (sop_op_sign *sign,
                                 sop_sign_as sign_as,
                                 const uint8_t *msg,
                                 size_t sz,
                                 sop_buf **micalg_out,
                                 sop_sigs **out);

   /* sop inline-sign */
   sop_err
   sop_op_sign_inline_execute (sop_op_sign *sign,
                               sop_inline_sign_as sign_as,
                               const uint8_t *msg,
                               size_t sz,
                               sop_inlinesigned **out);
```

```
/* sop inline-detach */
sop_err
sop_inlinesigned_detach (const sop_inlinesigned *msg,
                         sop_sigs **sigs_out,
                         sop_buf **msg_out);

#endif /* __SOP_H__ */
```

This proposed interface currently deals only with signing. Encryption and decryption will be added in a future revision.

B.1. Design Choices for Library API

The library is deliberately minimal, with data types and functionality corresponding to the SOP CLI. The interface itself should expose no dependencies beyond libc.

All datatypes are opaque structs. Library implementations MUST NOT expose library users to the memory layout of the underlying objects.

The library deals with data that is all in RAM, and produces data in RAM. For simplicity, it does not currently expose a streaming interface.

It should be fairly straightforward to implement the SOP CLI on top of such a library.

B.2. Library Use Patterns

There are two main kinds of data structures: operations (e.g., sop_op_sign and sop_op_verify) and datatypes (e.g., sop_keys and sop_certs).

Operation objects are one-shot objects. They are used in the following pattern:

* create an operations object (sop_op_*_new)
* adjust it to behave in certain ways (e.g., sop_op_sign_use_keys, sop_op_verify_not_before)
* execute it (with some specific sop_op_*_execute function)
* dispose of it (sop_op_*_free)

The library user MUST NOT execute the same operation object more than once. When a single operation object is executed more than once, it should fail with SOP_OPERATION_ALREADY_EXECUTED. FIXME: if a use case arises with a reasonable need to re-execute an already adjusted object, we could extend the API to allow the user to clone an object.

Datatype objects are reusable objects. For example, it is fine for a library user to pass the same sop_certs to multiple sop_op_* operation objects, as long as the sop_certs object is not freed before the execution of all the operation objects it has been passed to.

Datatype objects are also immutable. Any function which modifies a datatype object always creates a new copy of the object, with the specific change applied. This immutability avoids any ambiguity about what should happen when a datatype object is adjusted after it was passed to an operation object but before it was executed.

B.3. libsopv C API Subset

A minimalist library subset that only does OpenPGP signature verification might be called libsopv. This library is useful wherever the use case is just OpenPGP signature verification.

This minimal library interface should be sufficient to implement the sopv subset (see Section 3).

B.3.1. libsopv 1.1 C API Subset

Most functions in sopv.h are marked sopv: 1.0, indicating that they’re necessary to support version 1.0 of the sopv subset.

The following four functions are additionally necessary to implement sopv version 1.1:

* sop_verifications_get_mode
* sop_verifications_get_signer_count
* sop_verifications_get_signer
* sop_certs_get_label

B.3.2. libsopv 1.2 C API Subset

And the following functions are additionally necessary to implement sopv version 1.2:

* sop_op_validate_userid_new
* sop_op_validate_userid_free
* sop_op_validate_userid_add_authority
* sop_op_validate_userid_at
* sop_op_validate_userid_addr_spec_only
* sop_op_validate_userid_execute

Appendix C. Simple CLI Test

The following POSIX-compliant shell script can be pointed to a SOP implementation. It will report which subcommands have basic coverage.

It does not consider all possible combinations of all options.

```
#!/bin/sh

# Simple, positive self-test for Stateless OpenPGP implementations

# https://datatracker.ietf.org/doc/draft-dkg-openpgp-stateless-cli/

# This does not test all possible combinations of options or
# argument structures, it merely confirms that the standard
# subcommands and options are all implemented.

# This code makes many simplifying assumptions (e.g., there is no
# whitespace or metacharacters in filenames; filenames follow a
# strict convention) in order to be simple POSIX-compliant shell.
# The invocations are not necessarily safe shell programming if
# those assumptions are not met.  Please use caution when borrowing
# from this test script.

# Author: Daniel Kahn Gillmor
# License: CC-0

SOP=$1

if [ -z "$SOP" ]; then
    cat >&2 <<EOF
Usage: $0 SOP

SOP should refer (either by \$PATH or by absolute path) to an
implementation of the Stateless OpenPGP command-line interface.
```

```
   See https://datatracker.ietf.org/doc/draft-dkg-openpgp-stateless-cli/
   EOF
       exit 1
   fi

   if ! COMMAND_OUTPUT=$(command -v "$SOP"); then
       printf >&2 "No such command: %s\n" "$SOP"
       exit 1
   fi

   shift

   # We skip commands whose inputs are not available.
   # Return 0 if the test should be skipped, 1 otherwise.
   # missing inputs are printed to stdout.
   skip_test() {
       # do not skip commands that consume no input.
       if [ "$1" = generate-key \
                 -o "$1" = list-profiles \
                 -o  "$1" = version ]; then
           return 1
       fi
       shift
       local arg=""
       local ret=1
       for arg in $SIN "$@"; do
           local noninput='^--\(\(.*out\|as\|profile\|userid\|'
           noninput="${noninput}"'validate-at\|\(verify-\|\)'
           noninput="${noninput}"'not-\(before\|after\)\)=\|[^=]*$\)'
           if printf %s "$arg" | grep -q "$noninput" ; then
               continue
           fi
           arg=$(printf %s "$arg" | sed 's/^--.*=\(.*\)$/\1/')
           if ! [ -r "$arg" ]; then
               ret=0
               printf ' %s' "$arg"
           fi
       done
       return "$ret"
   }


   sop() {
       local suffix=""
       if [ -n "$SIN" ]; then
           suffix=" < $SIN"
       fi
       if [ -n "$SOUT" ]; then
```

```
            suffix="$suffix > $SOUT"
        fi
        local missing=""
        if missing=$(skip_test "$@"); then
            printf " skipped [%s %s%s] due to missing inputs%s\n" \
                   "$SOP" "$*" "$suffix" "$missing"
            SKIPCOUNT=$(( $SKIPCOUNT + 1 ))
            return
        fi
        printf " [%s %s%s]\n" "$SOP" "$*" "$suffix"
        if ! ( if [ -n "$SIN" ]; then exec < "$SIN"; fi;
               if [ -n "$SOUT" ]; then exec > "$SOUT"; fi;
               $SOP "$@") ; then
            printf " Failed: %s%s\n" "$*" "$suffix"
            rm -f "$SOUT"
            ERRORS="$ERRORS
    $*$suffix"
        else
            PASSCOUNT=$(( $PASSCOUNT + 1 ))
        fi
    }

    sop_fail() {
        local suffix=""
        if [ -n "$SIN" ]; then
            suffix=" < $SIN"
        fi
        if [ -n "$SOUT" ]; then
            printf 'ERROR: do not call sop_fail with expected stdout\n'
            exit 1
        fi
        local missing=""
        if missing=$(skip_test "$@"); then
            printf " skipped failing test [%s %s%s] due to %s%s\n" \
                   "$SOP" "$*" "$suffix" "missing input" "$missing"
            SKIPCOUNT=$(( $SKIPCOUNT + 1 ))
            return
        fi
        printf " [%s %s%s]\n" "$SOP" "$*" "$suffix"
        if ( if [ -n "$SIN" ]; then exec < "$SIN"; fi; $SOP "$@"); then
            printf >&2 " succeeded when it should have failed: %s%s\n" \
                   "$*" "$suffix"
            ERRORS="$ERRORS
    ! $*$suffix"
        else
            PASSCOUNT=$(( $PASSCOUNT + 1 ))
        fi
    }
```

```
   compare() {
       local args=""
       if [ "$1" = text -o "$1" = clearsigned ]; then
           args=--ignore-trailing-space
       fi
       comptype="$1"
       shift
       if ! [ -r "$1" -a -r "$2" ]; then
           printf " skipped %s comparison (%s) of %s and %s\n" \
                  "missing inputs" "$comptype" "$1" "$2"
           SKIPCOUNT=$(( $SKIPCOUNT + 1 ))
           return
       fi
       if diff --unified $args "$1" "$2"; then
           printf " %s and %s match!\n" "$1" "$2"
           PASSCOUNT=$(( $PASSCOUNT + 1 ))
       else
           printf "  %s and %s do not match!\n" "$1" "$2"
           ERRORS="$ERRORS
   Mismatch ($*)"
       fi
   }

   show_errs() {
       if [ -z "$1" ]; then
           if [ 0 -ne $SKIPCOUNT ]; then
               printf "No errors, but %d tests skipped somehow\n" \
                      $SKIPCOUNT
           else
               printf "No errors!\n"
           fi
       else
           local SKIPMSG=''
           if [ 0 -ne $SKIPCOUNT ]; then
               SKIPMSG=$(printf "%d tests skipped due to prior errors" \
                                $SKIPCOUNT)
           fi
           cat <<EOF

   $PASSCOUNT tests passed.
   $SKIPMSG

   === ERRORS ===
   $1

   === Error summary ===
   EOF
           E=$(echo "$1" | grep -v '^$')
```

```
        printf "%d Errors:\n" $(echo "$E" | wc -l)
        echo "$E" | sed 's/^! //' | cut -f1 -d\  | sort | uniq -c
    fi
}

DEARMORED=""

dearmor() {
    SIN="$1" SOUT="$1.bin" sop dearmor
    DEARMORED="$DEARMORED $1.bin"
}

ERRORS=""
SKIPCOUNT=0
PASSCOUNT=0
WORKDIR=$(mktemp -d)
printf "Working in: %s\n" "$WORKDIR"
cd "$WORKDIR"

sop version
sop version --extended
sop version --backend
sop version --sop-spec
sop version --sopv

sop list-profiles generate-key
sop list-profiles encrypt

SOUT=test.key sop generate-key "Example User <user@example.net>"
dearmor test.key
SIN=test.key SOUT=test.cert sop extract-cert
dearmor test.cert

SOUT=zeina.key sop generate-key "Zeina <zeina@example.net>"
dearmor zeina.key
SIN=zeina.key SOUT=zeina.cert sop extract-cert
dearmor zeina.cert

for f in cert key; do
    cat zeina.$f.bin test.$f.bin > both.$f.bin
    SIN=both.$f.bin SOUT=both.$f sop armor
done

SIN=test.key SOUT=test-revoked.cert sop revoke-key
dearmor test-revoked.cert

echo b4n4n4s > pw-orig.txt
SIN=test.key SOUT=test-locked.key sop change-key-password \
```

```
                    --new-key-password=pw-orig.txt
dearmor test-locked.key

# ensure that the key password is based on content, not filename
mv pw-orig.txt pw.txt
echo no-bananas > wrong-pw.txt

SIN=test-locked.key sop_fail change-key-password \
                          --old-key-password=wrong-pw.txt

SIN=test-locked.key SOUT=test-unlocked.key sop change-key-password \
                     --old-key-password=pw.txt
dearmor test-unlocked.key
compare binary test.key.bin test-unlocked.key.bin

cat > test.txt <<EOF
This is a test message.

We all  OpenPGP!
EOF

for as in '' binary text; do
    asarg=''
    if [ -n "$as" ]; then
        asarg=--as=$as
    fi
    SIN=test.txt SOUT=test.$as.sig sop sign $asarg test.key
    dearmor test.$as.sig
    # should fail because no password is supplied.
    SIN=test.txt sop_fail sign $asarg test-locked.key

    # should fail because wrong password is supplied.
    SIN=test.txt sop_fail sign $asarg \
                 --with-key-password=wrong-pw.txt test-locked.key
    SIN=test.txt SOUT=test.$as.siglocked sop sign $asarg \
                 --with-key-password=pw.txt test-locked.key
    dearmor test.$as.siglocked

    for sig in test.$as.sig test.$as.sig.bin test.$as.siglocked \
                            test.$as.siglocked.bin; do
        for cert in test.cert test.cert.bin \
                              both.cert both.cert.bin; do
            SIN=test.txt sop verify $sig $cert
        done
        for cert in test-revoked.cert test-revoked.cert.bin; do
            SIN=test.txt sop_fail verify $sig $cert
        done
    done
```

```
   done

   for as in '' binary text clearsigned; do
       asarg=''
       cmparg=binary
       if [ -n "$as" ]; then
           asarg=--as=$as
           cmparg=$as
       fi
       SIN=test.txt SOUT=test.$as.signed sop inline-sign $asarg test.key
       msgs=test.$as.signed
       if [ "$as" != clearsigned ]; then
           dearmor test.$as.signed
           msgs="$msgs test.$as.signed.bin"
       fi
       for msg in $msgs; do
           SIN=$msg SOUT=$msg.body sop inline-detach \
                    --signatures-out=$msg.detached-sigs
           compare $cmparg test.txt $msg.body
           for cert in test.cert test.cert.bin both.cert \
                               both.cert.bin; do
                   SIN=$msg SOUT=$msg.$cert.verified.txt sop \
                            inline-verify $cert
                   compare $cmparg test.txt $msg.$cert.verified.txt
                   SIN=$msg.body sop verify $msg.detached-sigs $cert
           done
           for cert in test-revoked.cert test-revoked.cert.bin; do
               SIN=$msg sop_fail inline-verify $cert
           done
       done
   done

   SIN=test.txt SOUT=test.msg sop encrypt test.cert
   dearmor test.msg
   SIN=test.txt SOUT=test.both.msg sop encrypt both.cert.bin
   dearmor test.both.msg

   for msg in test.msg test.msg.bin test.both.msg test.both.msg.bin; do
       SIN=$msg SOUT=$msg.decrypted.txt sop decrypt test.key
       compare binary test.txt $msg.decrypted.txt

       SIN=$msg sop_fail decrypt test-locked.key
       SIN=$msg sop_fail decrypt --with-key-password=wrong-pw.txt \
                test-locked.key
       SIN=$msg SOUT=$msg.locked-decrypted.txt sop decrypt \
                --with-key-password=pw.txt test-locked.key
       compare binary test.txt $msg.decrypted.txt
   done
```

```
   for x in $DEARMORED ; do
       SIN=$x SOUT=$x.asc sop armor
       SIN=$x.asc SOUT=$x.asc.bin sop dearmor
       compare binary $x $x.asc.bin
   done

   # TODO (subcommands still untested):

   # merge-certs
   # update-key
   # certify-userid
   # validate-userid

   # TODO (sop features still untested):

   # symmetric encryption/decryption (with password)
   # using --no-armor explicitly
   # sop generate-key --signing-only
   # sop generate-key --with-key-password
   # sop revoke-key --with-key-password
   # using the -- delimiter between options and positional args
   # sop sign --micalg-out
   # signing and encrypting at the same time
   # decrypting and verifying at the same time
   # using profiles
   # using session keys
   # using date ranges
   # using special designators (@FD: and @ENV:)
   # using piped input instead of material in the filesystem
   # confirming error codes for expected failures
   # put multiple TSKs in a KEYS object
   # sop_fail when KEYS is offered where CERTS should be
   # sop_fail when CERTS are offered where KEYS should be

   # This script does not test different algorithms or protocol-layer
   # subtleties For more complete testing, see the OpenPGP
   # Interoperability Test Suite, at https://tests.sequoia-pgp.org/

   show_errs "$ERRORS"
   if [ -d "$WORKDIR" ]; then
       rm -rf "$WORKDIR"
   fi
```

Appendix D. Testing the sopv Subset

The following two POSIX-compliant shell scripts can be used (with a signing-capable sop implementation) to exhaustively test a sopv implementation.

First, use setup-sopv-test with a signing-capable sop implementation to generate a set of test vectors in the current working directory. Then, run sopv-test against the sopv implementation:

```
./setup-sopv-test some-sop
./sopv-test my-sopv
```

D.1. setup-sopv-test

```
#!/bin/sh

# Create a-test environment for sopv: Stateless OpenPGP
# implementation Verification-only subset.  This needs a
# signing-capable SOP implementation to work.

# https://datatracker.ietf.org/doc/draft-dkg-openpgp-stateless-cli/

# Author: Daniel Kahn Gillmor
# License: CC-0

set -e

unset USE_DEFAULT_PROFILES
if [ "$1" = "--default-profile" ]; then
    USE_DEFAULT_PROFILES=true
    shift
fi

SOP=$1

if [ -z "$SOP" ]; then
    cat >&2 <<EOF
Usage: $0 [--default-profile] [--clean|SOP]

SOP should refer (either by \$PATH or by absolute path) to an
implementation of the Stateless OpenPGP command-line interface.

https://datatracker.ietf.org/doc/draft-dkg-openpgp-stateless-cli/

This will build a list of files in the current directory which can
be used with ./test-sopv to confirm support for the sopv subset.

If --clean is provided, destroy the list of files for testing sopv.

Normally, the default generate-key profile will be used for Alice,
and the second listed profile will be used for Bob.  If
--default-profile is provided, the default profile will be used for
both Alice and Bob.
```

```
   EOF
       exit 1
   fi

   objs() {
      for s in .bin ''; do
          printf "%s\n" alice.cert$s bob.cert$s both.cert$s
          for m in text binary; do
              for u in alice bob both; do
                  for o in sig inlinesigned; do
                      printf "%s\n" msg.$m.$u.$o$s
                  done
              done
          done
      done
      for u in alice bob both; do
          printf "%s\n" msg.text.$u.csf
      done
      printf "%s\n" msg.text msg.binary alice.key bob.key
      printf "%s\n" expired.cert valid-from-expired.sig
      printf "%s\n" invalid-from-expired.sig timetravel-expired.sig
      printf "%s\n" baseline.cert baseline.sig baseline-revoked.cert
   }

   if [ "$SOP" = --clean ]; then
       rm -f $(objs)
       exit 0
   fi

   sop() {
       "$SOP" "$@"
   }

   # use the first two profiles for the keys, reusing the default
   # if zero or one exists
   if [ "$USE_DEFAULT_PROFILES" = "true" ]; then
       profiles=$(echo default && echo default)
   else
       profiles=$(sop list-profiles generate-key | cut -f1 -d: && \
                      echo default && echo default)
   fi
   profile_line=1

   for uid in alice bob; do
       profile=$(printf "%s\n" "$profiles" | sed -n ${profile_line}p)
       profile_line=$(( $profile_line + 1 ))
       if [ "$profile" = default ]; then
           profile=
```

```
      else
          profile=--profile=$profile
      fi
      sop generate-key --signing-only $profile "$uid" \
          > "$uid.key"
      sop extract-cert < "$uid.key" > "$uid.cert"
      sop dearmor < "$uid.cert" > "$uid.cert.bin"
  done

  cat alice.cert.bin bob.cert.bin > both.cert.bin
  sop armor < both.cert.bin > both.cert

  cat > msg.text <<EOF
  This is the signed message for the sopv test suite.

  It should test the following things:

  - Messages using the cleartext signing framework (CSF)
  - Text-based signatures (armored and non-armored)
  - Binary data signatures (armored and non-armored)
  - Multiple certificates per CERTS or INLINESIGNED
  - Unknown signatures in a CERTS
  - @ENV as CERTS or SIGNATURES input
  - @FD as CERTS or SIGNATURES input
  - @FD as --verifications-out
  - UTF-8 data ()
  - Armored and non-armored OpenPGP certificates

  Please confirm!
  EOF

  base64 -d > msg.binary <<EOF
  AAECAwQFBgcICQoLDA0ODxAREhMUFRYXGBkaGxwdHh8gISIjJCUmJygpKissLS4v
  MDEyMzQ1Njc4OTo7PD0+P0BBQkNERUZHSElKS0xNTk9QUVJTVFVWV1hZWltcXV5f
  YFNPUFYgaXMgdGhlIFN0YXRlbGVzcyBPcGVuUEdQIFZlcmlmaWNhdGlvbiBTdWJz
  ZXTimaVhYmNkZWZnaGlqa2xtbm9wcXJzdHV2d3h5ent8fX5/gIGCg4SFhoeIiYqL
  jI2Oj5CRkpOUlZaXmJmam5ydnp+goaKjpKWmp6ipqqusra6vsLGys7S1tre4ubq7
  vL2+v8DBwsPExcbHyMnKy8zNzs/Q0dLT1NXW19jZ2tvc3d7f4OHi4+Tl5ufo6err
  7O3u7/Dx8vP09fb3+Pn6+/z9/v8=
  EOF

  # generated by ./generate-sopv-test-static-objects
  cat > expired.cert <<EOF
  -----BEGIN PGP PUBLIC KEY BLOCK-----

  xjMEXgvhABYJKwYBBAHaRw8BAQdAsaTsNjmaCYylGOk6y3ZghkilgqGY7Nl7XFl2
  LSRSUizNFHRoaXMgY2VydCBpcyBleHBpcmVkwoEEExYIACkFgl4L4QAFCQWkl6AC
  mwMCHgAWIQT+ZY0dgTiHcdLO/cnxYLD0zfFAiQAKCRDxYLD0zfFAiYtKAQChYiLC
```

```
   snvVV3AZUtDIFfSPG6PqqYuTFy0CzwFgAJR6cAD6AjcVWXsOWA6PlsZ4jmoWRWDW
   /XQNIGrO5KPK4K4FtQ0=
   =uQei
   -----END PGP PUBLIC KEY BLOCK-----
   EOF

   cat > valid-from-expired.sig <<EOF
   -----BEGIN PGP SIGNATURE-----

   wnUEABYIAB0FgmHO8MAWIQT+ZY0dgTiHcdLO/cnxYLD0zfFAiQAKCRDxYLD0zfFA
   iQ+eAP42je5wvtCN4Kfqub1bEhMi1EkioJgAWMcvE0Efa+47fgD9HYe/syBCXQWY
   ARbV0lOfTMDwliWs5wPZupuOsBOryAs=
   =TUXj
   -----END PGP SIGNATURE-----
   EOF

   cat > invalid-from-expired.sig <<EOF
   -----BEGIN PGP SIGNATURE-----

   wnUEABYIAB0FgmWSAIAWIQT+ZY0dgTiHcdLO/cnxYLD0zfFAiQAKCRDxYLD0zfFA
   iT0jAP9ZZt9UqI4PZtIiiYsXiJ55vT/54KFTnJ1xrrRxMLyeSwEAqwFfzXvpfXS5
   eKVJcMG89qIkczZII32Ya+5kbU0JsQg=
   =b9rH
   -----END PGP SIGNATURE-----
   EOF

   cat > timetravel-expired.sig <<EOF
   -----BEGIN PGP SIGNATURE-----

   wnUEABYIAB0FglwqWSAWIQT+ZY0dgTiHcdLO/cnxYLD0zfFAiQAKCRDxYLD0zfFA
   iWAAAP4tTANTYA6b+9m2UDEy7WhY0Q7IlBXz3Ppjuwf4i9LxJAD/epSvE8lVFHQI
   w1vQvheqmIa3JOLhNDen3Wv85SwT4wo=
   =zrLP
   -----END PGP SIGNATURE-----
   EOF

   cat > baseline.cert <<EOF
   -----BEGIN PGP PUBLIC KEY BLOCK-----

   xjMEXgvhABYJKwYBBAHaRw8BAQdArUiK8psA3U3Q8mdO1DZR2UW8XgXBB3VVtHaj
   r1UaUFfNJWJhc2VsaW5lIGNlcnRpZmljYXRlLCB3aWxsIGJlIHJldm9rZWTCewQT
   FggAIwWCXgvhAAKbAwIeABYhBHGqmGkigo5sDdKjgjk1BkEkKnCVAAoJEDk1BkEk
   KnCVUNcBANRfAasPy5lXXFXskfX4yyFx1M1kxX17vxoQEDtB6TrqAQCTyzV4vjUm
   Y3tgljlN8iZZd1aJEDaHYY6thPKqE0UACg==
   =WsIg
   -----END PGP PUBLIC KEY BLOCK-----
   EOF
```

```
cat > baseline.sig <<EOF
-----BEGIN PGP SIGNATURE-----

wnUEABYIAB0FgmfUYiwWIQRxqphpIoKObA3So4I5NQZBJCpwlQAKCRA5NQZBJCpw
lfW7AQD+/Oc8y6Vgij/CSgZm+Bg74ezJ91K6fCSKlgMsXc/g5AEAv7/kLe1Gdu5u
IA26te6ytPfJAUoiI/Tyw5YYV+CgRQo=
=obH5
-----END PGP SIGNATURE-----
EOF

cat > baseline-revoked.cert <<EOF
-----BEGIN PGP PUBLIC KEY BLOCK-----

xjMEXgvhABYJKwYBBAHaRw8BAQdArUiK8psA3U3Q8mdO1DZR2UW8XgXBB3VVtHaj
r1UaUFfCeAQgFggAIAWCYc7wwAIdABYhBHGqmGkigo5sDdKjgjk1BkEkKnCVAAoJ
EDk1BkEkKnCVwgEA/RAFHGWHBjgEvthbeV13/fdrfSC+5RrrvF50a2haXBuEAQD/
Z2U5Eg/ka5LrjdPLyZfE2+j7+boVD8dQq2b0g8+0D80lYmFzZWxpbmUgY2VydGlm
aWNhdGUsIHdpbGwgYmUgcmV2b2tlZMJ7BBMWCAAjBYJeC+EAApsDAh4AFiEEcaqY
aSKCjmwN0qOCOTUGQSQqcJUACgkQOTUGQSQqcJVQ1wEA1F8Bqw/LmVdcVeyR9fjL
IXHUzWTFfXu/GhAQO0HpOuoBAJPLNXi+NSZje2CWOU3yJll3VokQNodhjq2E8qoT
RQAK
=J9fE
-----END PGP PUBLIC KEY BLOCK-----
EOF


for signer in alice bob; do
    for form in text binary; do
        sop sign --as=$form $signer.key < msg.$form \
            > msg.$form.$signer.sig
        sop dearmor < msg.$form.$signer.sig \
            > msg.$form.$signer.sig.bin
        sop inline-sign --as=$form $signer.key < msg.$form \
            > msg.$form.$signer.inlinesigned
        sop dearmor < msg.$form.$signer.inlinesigned \
            > msg.$form.$signer.inlinesigned.bin
    done
    sop inline-sign --as=clearsigned $signer.key < msg.text \
        > msg.text.$signer.csf
done

for form in text binary; do
    sop sign --as=$form alice.key bob.key < msg.$form \
        > msg.$form.both.sig
    sop dearmor < msg.$form.both.sig > msg.$form.both.sig.bin
    sop inline-sign --as=$form alice.key bob.key < msg.$form \
        > msg.$form.both.inlinesigned
    sop dearmor < msg.$form.both.inlinesigned \
```

```
        > msg.$form.both.inlinesigned.bin
done
sop inline-sign --as=clearsigned alice.key bob.key \
    < msg.text > msg.text.both.csf

if ! ls $(objs) > /dev/null; then
    exit 1
fi
```

D.2. sopv-test

```
#!/bin/sh

# Test the Stateless OpenPGP implementation Verification-only subset.

# This needs to be run from within a directory created by the
#  setup-sopv-test script.

# https://datatracker.ietf.org/doc/draft-dkg-openpgp-stateless-cli/

# Author: Daniel Kahn Gillmor
# License: CC-0

SOPV=$1

if [ -z "$SOPV" ]; then
    cat >&2 <<EOF
Usage: $0 SOPV

SOPV should refer (either by \$PATH or by absolute path) to an
implementation of the Stateless OpenPGP Verification-only subset.
See https://datatracker.ietf.org/doc/draft-dkg-openpgp-stateless-cli/
EOF
    exit 1
fi

sopv() {
    local suffix=""
    if [ -n "$SIN" ]; then
        suffix=" < $SIN"
    fi
    if [ -n "$SOUT" ]; then
        suffix="$suffix > $SOUT"
    fi
    if [ -n "$FD_3" ]; then
        suffix="$suffix 3< $FD_3"
    fi
    if [ -n "$FD_4" ]; then
```

```
            suffix="$suffix 4< $FD_4"
        fi
        if [ -n "$FD_5" ]; then
            suffix="$suffix 5< $FD_5"
        fi
        # FD 9 is used for output, not input
        if [ -n "$FD_9" ]; then
            suffix="$suffix 9> $FD_9"
        fi
        local missing=""
        printf " [%s %s%s]\n" "$SOPV" "$*" "$suffix"
        if ! ( if [ -n "$SIN" ]; then exec < "$SIN"; fi;
               if [ -n "$SOUT" ]; then exec > "$SOUT"; fi;
               if [ -n "$FD_3" ]; then exec 3< "$FD_3"; fi;
               if [ -n "$FD_4" ]; then exec 4< "$FD_4"; fi;
               if [ -n "$FD_5" ]; then exec 5< "$FD_5"; fi;
               if [ -n "$FD_9" ]; then exec 9> "$FD_9"; fi;
               $SOPV "$@") ; then
            printf " Failed: %s%s\n" "$*" "$suffix"
            rm -f "$SOUT"
            ERRORS="$ERRORS
    $*$suffix"
            return 1
        else
            PASSCOUNT=$(( $PASSCOUNT + 1 ))
            return 0
        fi
    }

    sopv_fail() {
        local suffix=""
        if [ -n "$SIN" ]; then
            suffix=" < $SIN"
        fi
        if [ -n "$SOUT" ]; then
            printf 'ERROR: do not call sopv_fail and expect stdout\n'
            exit 1
        fi
        if [ -n "$FD_3" ]; then
            suffix="$suffix 3< $FD_3"
        fi
        if [ -n "$FD_4" ]; then
            suffix="$suffix 4< $FD_4"
        fi
        if [ -n "$FD_5" ]; then
            suffix="$suffix 5< $FD_5"
        fi
        # FD 9 is used for output, not input
```

```
      if [ -n "$FD_9" ]; then
          suffix="$suffix 9> $FD_9"
      fi
      local missing=""
      printf " [%s %s%s]\n" "$SOPV" "$*" "$suffix"
      if ( if [ -n "$SIN" ]; then exec < "$SIN"; fi;
           if [ -n "$FD_3" ]; then exec 3< "$FD_3"; fi;
           if [ -n "$FD_4" ]; then exec 4< "$FD_4"; fi;
           if [ -n "$FD_5" ]; then exec 5< "$FD_5"; fi;
           if [ -n "$FD_9" ]; then exec 9> "$FD_9"; fi;
           $SOPV "$@" > fail.out); then
          printf >&2 " succeeded when it should have failed: %s%s\n" \
                 "$*" "$suffix"
          ERRORS="$ERRORS
  ! $*$suffix"
      else
          if [ -s fail.out ]; then
              printf >&2 " produced material to stdout: %s%s\n" \
                     "$*" "$suffix"
              sed 's/^/ > /' < fail.out >&2
              ERRORS="$ERRORS
  ! $*$suffix PRODUCED OUTPUT"
          else
              PASSCOUNT=$(( $PASSCOUNT + 1 ))
          fi
      fi
      rm -f fail.out
  }

  compare() {
      local args=""
      if [ "$1" = text -o "$1" = clearsigned ]; then
          args=--ignore-trailing-space
      fi
      comptype="$1"
      shift
      if ! [ -r "$1" -a -r "$2" ]; then
          printf " skipped %s cmp (missing inputs): %s and %s\n" \
                 "$comptype" "$1" "$2"
          SKIPCOUNT=$(( $SKIPCOUNT + 1 ))
          return
      fi
      if diff --unified $args "$1" "$2"; then
          printf " %s and %s match!\n" "$1" "$2"
          PASSCOUNT=$(( $PASSCOUNT + 1 ))
      else
          printf "  %s and %s do not match!\n" "$1" "$2"
          ERRORS="$ERRORS
```

```
   Mismatch ($*)"
       fi
   }

   reject_output() {
       for f in "$@"; do
           if [ -s "$f" ]; then
               printf " %s should not exist with content!\n" "$f"
               ERRORS="$ERRORS
   Should-not-exist $f"
           else
               PASSCOUNT=$(( $PASSCOUNT + 1 ))
           fi
       done
   }

   confirm_mode() {
       local foundmode=''
       for m in $(cut -f4 -d\  < "$2"); do
           if [ "$m" != "mode:$1" ]; then
               printf " %s should have mentioned mode:%s, was %s!\n" \
                      "$2" "$1" "$m"
               ERRORS="$ERRORS
   VERIFICATIONS-bad-mode $2 (was: $m; wanted mode:$1)"
           else
               foundmode=yes
           fi
       done
       if [ -z "$foundmode" ]; then
               printf " %s had no mode, wanted %s!\n" "$2" "$1"
               ERRORS="$ERRORS
   VERIFICATIONS-no-mode $2 (wanted mode:$1)"
       else
           PASSCOUNT=$(( $PASSCOUNT + 1 ))
       fi
   }

   show_errs() {
       if [ -z "$1" ]; then
           if [ 0 -ne $SKIPCOUNT ]; then
               printf "No errors, %d tests passed.\n"
               printf "but %d tests skipped somehow\n" \
                      $PASSCOUNT $SKIPCOUNT
           else
               printf "No errors! %d tests passed\n" $PASSCOUNT
           fi
       else
           local SKIPMSG=''
```

```
          if [ 0 -ne $SKIPCOUNT ]; then
              SKIPMSG=$(printf "%d tests skipped due to prior errors" \
                               $SKIPCOUNT)
          fi
          cat <<EOF

   $PASSCOUNT tests passed.
   $SKIPMSG

   === ERRORS ===
   $1

   === Error summary ===
   EOF
          E=$(echo "$1" | grep -v '^$')
          printf "%d Errors:\n" $(echo "$E" | wc -l)
          echo "$E" | sed 's/^! //' | cut -f1 -d\  | sort | uniq -c
      fi
   }


   ERRORS=""
   SKIPCOUNT=0
   PASSCOUNT=0

   combine() {
       # runners take: sopv|sopv_fail signer cert [cert...]
       local runner=$1
       shift
       $runner sopv alice alice.$cert
       $runner sopv bob bob.$cert
       $runner sopv_fail bob alice.$cert
       $runner sopv_fail alice bob.$cert
       $runner sopv both alice.$cert
       $runner sopv both bob.$cert
       $runner sopv both both.$cert
       $runner sopv alice both.$cert
       $runner sopv bob both.$cert
       $runner sopv alice alice.$cert bob.$cert
       $runner sopv alice bob.$cert alice.$cert
       $runner sopv bob alice.$cert bob.$cert
       $runner sopv bob bob.$cert alice.$cert
       FD_3=alice.$cert $runner sopv alice @FD:3
       FD_3=bob.$cert FD_4=alice.$cert $runner sopv alice @FD:3 @FD:4
       # don't try to test @ENV on non-armored certs
       if [ "$cert" = "cert" ]; then
           SIGNER_CERT=$(cat alice.$cert) $runner sopv \
                      alice @ENV:SIGNER_CERT
```

```
    fi
}

detached() {
    local sopv=$1
    shift
    local signer=$1
    shift
    SIN=msg.$form $sopv verify $delim msg.$form.$signer.$sig "$@"
    FD_5=msg.$form.$signer.$sig SIN=msg.$form $sopv verify $delim \
                                @FD:5 "$@"
    # don't try to test @ENV on non-armored signatures
    if [ "$sig" = "sig" ]; then
        SIGNATURE=$(cat msg.$form.$signer.$sig) SIN=msg.$form $sopv \
                 verify $delim @ENV:SIGNATURE "$@"
    fi
}

inlinesigned() {
    local sopv=$1
    shift
    local signer=$1
    shift
    local vout=msg.$form.$signer.$inlmsg.verifs
    rm -f "$vout"
    if [ "$sopv" = sopv ]; then
        if SIN=msg.$form.$signer.$inlmsg \
               SOUT=msg.$form.$signer.$inlmsg.out $sopv \
               inline-verify --verifications-out=$vout \
               $delim "$@" ; then
            confirm_mode "$form" "$vout"
        fi
        if FD_9=$vout.fd SIN=msg.$form.$signer.$inlmsg \
                         SOUT=msg.$form.$signer.$inlmsg.out.fd \
                         $sopv inline-verify \
                         --verifications-out=@FD:9 \
                         $delim "$@" ; then
            confirm_mode "$form" "$vout.fd"
        fi
        compare $form msg.$form msg.$form.$signer.$inlmsg.out
        compare binary msg.$form.$signer.$inlmsg.out \
                msg.$form.$signer.$inlmsg.out.fd
        rm -f msg.$form.$signer.$inlmsg.out $vout \
           msg.$form.$signer.$inlmsg.out.fd $vout.fd

        # inlinesigned msgs can't be used as detached signatures:
        SIN=msg.$form sopv_fail verify $delim \
                      msg.$form.$signer.$inlmsg "$@"
```

```
        else
            SIN=msg.$form.$signer.$inlmsg $sopv inline-verify \
                                          --verifications-out=$vout \
                                          $delim "$@"
            FD_9=$vout.fd SIN=msg.$form.$signer.$inlmsg $sopv \
                          inline-verify --verifications-out=@FD:9 \
                          $delim "$@"
            reject_output $vout $vout.fd
        fi
    }

    sopv version --extended
    sopv version --sopv

    for delim in '' --; do
        for cert in cert cert.bin; do
            for form in binary text; do
                # test detached signature
                for sig in sig sig.bin; do
                    combine detached
                done

                # test inline-signed messages
                for inlmsg in inlinesigned inlinesigned.bin; do
                    combine inlinesigned
                done
            done

            # test CSF
            form=text inlmsg=csf combine inlinesigned
        done
    done

    sopv verify valid-from-expired.sig expired.cert < msg.binary
    sopv_fail verify invalid-from-expired.sig expired.cert < msg.binary
    sopv_fail verify timetravel-expired.sig expired.cert < msg.binary

    sopv verify baseline.sig baseline.cert < msg.binary
    sopv_fail verify baseline.sig baseline-revoked.cert < msg.binary

    # FIXME:
    #
    # - --not-before and --not-after
    # - JSON extension to VERIFICATIONS, including "signers" (sopv 1.1)
    # - using --argument=foo vs. --argument foo ?
    # - review equivalence of VERIFICATIONS
    # - confirm failure when --verifications-out already exists
    # - passing CERTS where SIGNATURES are expected MUST fail
```

```
   # - passing KEYS where CERTS are expected MUST fail

   show_errs "$ERRORS"

   # Succeed only if $ERRORS is empty:
   [ -z "$ERRORS" ]
```

Appendix E. 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 Nordhy
* Antoine Beaupr
* Edwin Taylor
* Guillem Jover
* Heiko Schaefer
* Jameson Rollins
* Justus Winter
* Paul Schaub
* Vincent Breitmoser

Appendix F. Future Work

* certificate transformation into popular publication forms:
  - WKD
  - DANE OPENPGPKEY
  - Autocrypt
* sop encrypt -- specify compression? (see Section 13.2)
* 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)?

* 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 11.6)

* do we need to be able to convert a message with a text-based signature to a CSF INLINESIGNED message? I'd rather not, given the additional complications.

* add encryption and decryption to C Library API

Appendix G. Document History

G.1. Substantive Changes between -13 and -14:

* Improve sopv tests in Appendix D

* libsop C API: split out sopv.h from sop.h

* libsop C API: use int64_t for sop_time

* libsop C API: use size_t for counts

* Recommend emitting trailing newline when outputting SESSIONKEY

* sopv 1.0: note that @ENV: and @FD: also need to work for SIGNATURES

* Define sopv 1.2: add validate-userid

G.2. Substantive Changes between -12 and -13:

* Define sopv 1.1 (structured json VERIFICATIONS output)

* Fix misspellings

G.3. Substantive Changes between -11 and -12:

* Improvements in POSIX shell tests (including robust sopv test)
* Define security, performance, and compatibility profile aliases
* Clarify that sop armor ought to be able to armor any output that sop produces
* Intro: acknowledge increased attention to key/cert management
* Add KEY_CANNOT_CERTIFY error code
* Relax guidance on multi-signature operations and --micalg-out
* Define sopv 1.0 (with changelog)
* Document exceptions to statelessness for system-level state

G.4. Substantive Changes between -10 and -11:

* update-key: new key management subcommand
* merge-certs: new certificate management subcommand
* certify-userid: new User ID certification subcommand
* validate-userid: new User ID verification subcommand
* Replace references to RFC 4880 with RFC 9580
* Set aside error 1 as UNSPECIFIED_FAILURE
* Encourage JSON output in tail of VERIFICATIONS lines
* Add universal (ignorable) --debug option
* Add simple (and incomplete) shell-script test in appendix

G.5. Substantive Changes between -09 and -10:

* drop @HARDWARE: special designator in favor of the simple [I-D.dkg-openpgp-hardware-secrets] or other magic
* drop hardware-specific C API function
* define SemVer-versioned sopv subset of CLI

* sop version: add --sopv option
* define libsopv subset of C API
* explicitly require BAD_DATA failure when KEYS are passed as CERTS

G.6. Substantive Changes between -08 and -09:

* enable the use of hardware-backed secret key material via the @HARDWARE: special designator
* C API: clarify design goals and usage patterns
* C API: major overhaul and normalization:
  - allow passthrough "cookie" for logging
  - allow NULL return from sop_version_*
  - explicitly offer SOP_LOG_NEVER
  - use *_from_bytes and *_to_bytes instead of *_import and *_export
  - datatype objects are now immutable
  - operation objects are one-shot
  - always return sop_err, even at a slight cost to C caller ergonomics

G.7. Substantive Changes between -07 and -08:

* revoke-key, change-key-password: add --no-armor option
* generate-key: should fail on non-UTF-8 USERID
* generate-key: acknowledge that implementations MAY reject USERIDs that seem bad
* armor: drop --label option
* encrypt: add --session-key-out option
* ASCII-armored objects should not be concatenated
* signature verification should only work for sigtypes 0x00 (binary) and 0x01 (canonical text)

* sign: Constrain input when --micalg-out is present for alignment with [RFC3156]

* propose simple C API for signing and verification

G.8. Substantive Changes between -06 and -07:

* generate-key: add --signing-only option

* new key management subcommand: change-key-password

* new key management subcommand: revoke-key

G.9. Substantive Changes between -05 and -06:

* version: add --sop-spec argument

* encrypt: add --profile argument

G.10. Substantive Changes between -04 and -05:

* decrypt: change --verify-out to --verifications-out

* encrypt: add missing --with-key-password

* add the concept of "profiles", use with generate-key

* include table of known implementations

* VERIFICATIONS can now indicate the type of the signature (mode:text or mode:binary)

G.11. Substantive Changes between -03 and -04:

* Reinforce that PASSWORD and SESSIONKEY are indirect data types

* encrypt: remove --as=mime option

* Handle password-locked secret key material: add --with-key-password options to generate-key, sign, and decrypt.

* Introduce INLINESIGNED message type (Section 7.5)

* Rename detach-inband-signature-and-message to inline-detach, clarify its possible inputs

* Add inline-verify

* Add inline-sign

G.12. 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

G.13. 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

G.14. 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)

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