The GPG Guide

Modern OpenPGP for Every Workflow

2026 Edition

TONY GIES

Table of Contents

1. The GPG Guide

OpenPGP has been around since 1991, but the ecosystem in 2026 looks nothing like the one described in the guides most people learned from. The protocol has forked into two competing standards. The cryptographic recommendations have shifted. Keyserver infrastructure has collapsed and been rebuilt. Sequoia PGP has matured into a real alternative to GnuPG. And entirely new tools -- age, Sigstore, FIDO2 -- have taken over jobs that PGP used to do alone.

Most existing PGP guides either predate these changes or cover only one narrow workflow. This guide is different. It covers the **full lifecycle** of a modern PGP identity -- from generating your first key on an air-gapped machine, through hardware token provisioning and daily use, to maintenance, rotation, and emergency recovery years later. Every command has been tested against GnuPG 2.5.x and Sequoia sq 1.3.1. Where PGP is the wrong tool for the job, this guide says so and points you to what works better.

The approach is opinionated: **v4 keys, Ed25519 cryptography, hardware tokens where practical.** Not because there are no other valid choices, but because you have actual work to do and a clear "just do this" path is more useful than a survey of every option. Three reader tracks let you skip what you don't need -- if all you want is Git signing and SSH, you can be done in an afternoon without wading through Debian packaging or the Web of Trust.

How to read this guide

This guide is organized for both **sequential reading** (if you're setting up from scratch) and **reference lookup** (if you need a specific workflow).

Choose your track:

- **Track A -- "I just need Git signing and SSH"**: Parts I, II, V, VI, VII
- **Track B -- "Full identity setup with YubiKey"**: Parts I through VII, then whatever workflows you need
- **Track C -- "Debian Developer / high-assurance identity"**: The whole guide, especially Parts XII–XIV

1.1 What's Inside

PART	TOPIC	WHAT YOU'LL LEARN
I	Philosophy & Foundations	Why Ed25519, the LibrePGP/RFC 9580 split, threat models
II	Key Generation	Air-gapped setup with GnuPG or Sequoia
III	Backup & Recovery	Paperkey, QR codes, encrypted USB, revocation
IV	Hardware Provisioning	YubiKey setup, touch policies, alternatives
V	Daily Machine Setup	GnuPG config, cross-platform, WSL2
VI	SSH Authentication	GPG-agent, FIDO2, PIV -- three paths compared
VII	Git Signing	GPG signing, SSH signing, GitHub/GitLab, CI/CD
VIII	Email Encryption	Thunderbird, Mutt, Autocrypt, ProtonMail
IX	Password Management	pass, gopass, passage
X	File Encryption	GPG encryption, age, encrypted backups
XI	Secrets Management	SOPS, git-crypt, comparison
XII	Key Distribution	Keyservers, WKD, Keyoxide
XIII	Web of Trust	caff, keysigning parties, Debian path
XIV	Package Signing	deb, RPM, releases, containers
XV	Maintenance	Expiry renewal, YubiKey switching, emergencies
XVI	Complementary Tools	Sequoia, age, SOP, when NOT to use PGP

Appendices: Config Reference · Troubleshooting · Cheat Sheet · Glossary · Migration Guides · Legal & Compliance

2. Free Sample

This is a free sample of The GPG Guide

You're reading a preview containing **Parts I–III** (Philosophy & Foundations, Key Generation, Backup & Recovery) plus the **Cheat Sheet** appendix — everything you need to generate a solid key pair and back it up safely.

The complete guide continues with 13 more parts and 5 additional appendices covering hardware tokens, daily machine setup, SSH, Git signing, email encryption, password management, secrets management, key distribution, the Web of Trust, package signing, maintenance, and complementary tools.

Get the full guide at: https://leanpub.com/gpg-guide

3. I: Philosophy & Foundations

3.1 Part I: Philosophy & Foundations

Before generating a single key, it pays to understand the landscape you are stepping into. OpenPGP has been around since 1991, but 2026 looks nothing like 2016 -- the protocol has forked, the cryptographic recommendations have shifted, and the tooling ecosystem has expanded far beyond GnuPG alone.

This part covers the strategic decisions every reader must make **before** touching a terminal:

1. **The State of OpenPGP in 2026** -- The LibrePGP / RFC 9580 split, the ecosystem map, and why v4 keys remain the universal choice.
2. **Cryptographic Choices** -- Why Ed25519 + Cv25519 is the golden path, what EUCLEAK means for your hardware, and when RSA still matters.
3. **Key Architecture & Threat Model** -- The master + subkeys model, threat scenarios, and the three reader tracks (A, B, C).
4. **Toolchain Choice: Two Paths** -- GnuPG-only vs. Hybrid (Sequoia + GnuPG), with a decision matrix.

Which track are you?

Throughout this guide, sections are tagged for three reader tracks:

- **Track A -- Minimal:** "I just need Git signing and SSH." You can skip hardware provisioning entirely if you prefer software keys.
- **Track B -- Standard:** "Full identity setup with YubiKey." The path most readers will follow.
- **Track C -- Advanced:** "Debian Developer / high-assurance identity." Keysigning parties, WoT, package signing.

Every reader should read Part I. After that, follow the track markers to skip sections that do not apply to you.

3.2 1.1 The State of OpenPGP in 2026

Tracks: A, B, C

OpenPGP is the most widely deployed public-key encryption standard outside of TLS. It underpins Git commit signing, encrypted email, software package verification, and password management tools like `pass`. But the ecosystem in 2026 is very different from the one described in older guides -- and understanding the current landscape will save you from making choices that look reasonable on paper but break in practice.

The Split: LibrePGP vs. RFC 9580

In 2023, the OpenPGP working group at the IETF finalized **RFC 9580**, a major update to the OpenPGP standard. RFC 9580 introduces **v6 keys** -- a new packet format with improved cryptographic agility, mandatory AEAD encryption, and cleaner metadata handling.

GnuPG's maintainer, Werner Koch, disagreed with several design decisions in RFC 9580 and published an alternative specification called **LibrePGP** (`draft-koch-librepgp`). GnuPG follows LibrePGP, not RFC 9580.

This means:

FEATURE	GNUPG (LIBREPGP)	SEQUOIA PGP (RFC 9580)
v4 keys	Full support	Full support
v6 keys	**Not supported**	Supported (openpgp crate 2.0+)
AEAD encryption	v5 AEAD (experimental)	RFC 9580 AEAD
Interoperability	Universal for v4	Universal for v4, limited for v6

⚠ **What this means for you**

For 2026, use v4 keys. Both GnuPG and Sequoia fully support v4 keys. v6 keys work only between Sequoia instances (and other RFC 9580 implementations). GnuPG will refuse to import v6 keys entirely.

If you have read older blog posts recommending v6 keys as "the future" -- they are correct about the direction, but premature about the timeline. v6 adoption requires GnuPG support, and that has not happened.

The Ecosystem Map

OpenPGP is not just GnuPG. Multiple implementations exist, each with different strengths:

IMPLEMENTATION	LANGUAGE	STANDARD	USED BY
GnuPG	C	LibrePGP	System GPG on Linux/macOS/Windows, `gpg-agent`, smartcard support
Sequoia PGP	Rust	RFC 9580	`sq` CLI, Fedora tooling, security audits
RNP	C++	RFC 4880 + extensions	**Thunderbird** (built-in OpenPGP)
OpenPGP.js	JavaScript	RFC 4880	ProtonMail web client
GopenPGP	Go	RFC 4880	ProtonMail native apps

Thunderbird uses RNP, not GnuPG

This is one of the most common sources of confusion. Thunderbird's built-in OpenPGP support uses RNP -- a completely separate implementation. RNP does **not** support smartcards or hardware tokens. If you use a YubiKey, you must configure Thunderbird to delegate private-key operations to your system GnuPG installation. See Part VIII: Email Encryption for the setup steps.

GnuPG Versions in 2026

BRANCH	VERSION	STATUS	NOTES
2.5.x	2.5.17	**Stable**	Current upstream recommended version
2.4.x	2.4.7	Oldstable	End-of-life: 2026-06-30
2.2.x	--	EOL	Do not use

Most distributions still ship GnuPG 2.4

Ubuntu 24.04 LTS ships 2.4.4, Debian Trixie and Tails 7.x ship 2.4.7, and Fedora 41 ships 2.4.6. If you install GnuPG from your package manager, you will almost certainly get 2.4.x. **This is fine -- everything in this guide works on GnuPG 2.4.** Where a feature is 2.5-specific, it is called out explicitly.

GnuPG 2.5 brings several improvements over 2.4:

- Experimental **Kyber** (post-quantum) key support
- Improved default cipher preferences (SHA-512, AES-256)
- Better smartcard handling

If your distribution ships GnuPG 2.4, plan to upgrade before the June 2026 end-of-life date -- but there is no urgency. Your distribution will likely provide 2.5.x as a routine update before then.

Sequoia PGP

Sequoia PGP is a modern, Rust-based OpenPGP implementation. Its CLI tool `sq` (current version: 1.3.1) provides a cleaner interface for key management and cryptographic operations.

Key facts about `sq`:

- **Default profile:** `--profile rfc4880` produces v4 keys (correct for interoperability)
- **Alternative profile:** `--profile rfc9580` produces v6 keys (do NOT use for keys that will touch GnuPG or YubiKey)
- **No smartcard support:** `sq` cannot load keys onto hardware tokens. You must use GnuPG's `keytocard` for that.
- **Best for:** Key generation with saner defaults, key inspection, WKD publishing, and as a second opinion on key operations

The Bottom Line

For 2026, the practical strategy is:

1. **Generate v4 keys** -- universally compatible
2. **Use Ed25519/Cv25519** -- modern, fast, secure (see 1.2 Cryptographic Choices)
3. **Use GnuPG at runtime** -- required for smartcard operations, SSH agent, and the widest tool support
4. **Optionally use Sequoia (`sq`) for generation** -- if you prefer its interface (see 1.4 Toolchain Choice)
5. **Ignore v6 keys for now** -- revisit when GnuPG adds support (if ever)

This guide follows this strategy throughout. Every command, configuration file, and workflow has been tested against this v4 + Ed25519 baseline.

3.3 1.2 Cryptographic Choices

Tracks: A, B, C

Choosing your key algorithm is the first decision with lasting consequences -- you will live with it for the lifetime of your key (potentially decades). This section explains why **Ed25519 + Cv25519** is the right choice for almost everyone in 2026, and when exceptions apply.

The Golden Path: v4 + Ed25519 / Cv25519

SLOT	ALGORITHM	CURVE	PURPOSE
Master [C]	EdDSA	Ed25519	Certify subkeys and UIDs
Signing [S]	EdDSA	Ed25519	Sign commits, emails, files
Encryption [E]	ECDH	Cv25519	Decrypt messages and files
Authentication [A]	EdDSA	Ed25519	SSH authentication

This combination gives you:

- **Universal compatibility:** Works with GnuPG (all supported versions), Sequoia, Thunderbird/RNP, OpenKeychain, YubiKey (firmware 5.2.3+), and every tool covered in this guide.
- **Modern cryptography:** 128-bit security level, equivalent to RSA-3072 but with keys and signatures that are a fraction of the size.
- **No known side-channel attacks on Ed25519:** The signing algorithm uses constant-time operations with no secret-dependent branching -- critical for hardware tokens (see EUCLEAK below).
- **Fast operations:** Key generation, signing, and verification are all significantly faster than RSA.

Ed25519 naming in GnuPG

GnuPG uses the name `ed25519` for both EdDSA signing keys and the master (certification) key. The encryption subkey uses `cv25519` (Curve25519 in its Montgomery form for ECDH key exchange). These are the names you will pass to `--quick-generate-key` and `--quick-add-key`.

EUCLEAK: Why Ed25519 Is the Safe Choice

In September 2024, researchers at NinjaLab published **EUCLEAK** (CVE-2024-45678), a side-channel attack against Infineon's ECDSA cryptographic library. This library was used in **all YubiKey 5 series devices with firmware earlier than 5.7**.

What EUCLEAK affects

- **ECDSA signatures using NIST curves** (P-256, P-384): The Infineon library leaks timing information during the modular inversion step of ECDSA signing.
- **Requires physical access:** An attacker needs the YubiKey in their hands plus specialized electromagnetic measurement equipment.
- **CVSS score: 4.9** (medium severity) -- reflects the physical access requirement.

What EUCLEAK does NOT affect

- **Ed25519 / EdDSA:** Uses completely different mathematics (twisted Edwards curves with constant-time scalar multiplication). The vulnerable Infineon code path is never invoked for Ed25519 operations.
- **RSA keys:** Different algorithm entirely, not affected.
- **YubiKey firmware 5.7+:** Yubico replaced the Infineon library with their own cryptographic implementation. ECDSA on firmware 5.7+ is not vulnerable.

Check your firmware version

Run `ykman info` to see your YubiKey's firmware version. Firmware **cannot be upgraded** on YubiKey -- it is burned at manufacture.

- **Firmware < 5.7 + NIST curve keys:** Potentially vulnerable. But if you follow this guide's recommendation of Ed25519, you are not affected.
- **Firmware 5.7+:** Not vulnerable to EUCLEAK regardless of algorithm.
- **Any firmware + Ed25519:** Not vulnerable. This is our golden path.

The bottom line on EUCLEAK

EUCLEAK is a real vulnerability, but it is **not a reason to panic** if you use Ed25519 keys. It is, however, a strong argument *for* choosing Ed25519 over NIST curves -- even on newer hardware. Ed25519's constant-time design provides defense-in-depth against future side-channel discoveries.

When RSA-4096 Still Makes Sense

RSA is not broken, and RSA-4096 remains a perfectly secure choice. However, it has practical disadvantages compared to Ed25519:

PROPERTY	ED25519	RSA-4096
Security level	~128-bit	~128-bit
Public key size	32 bytes	512 bytes
Signature size	64 bytes	512 bytes
Key generation	Milliseconds	Seconds
Signing speed	Fast	Slower
Smartcard operations	Fast	Noticeably slower on YubiKey

Consider RSA-4096 if:

- Your organization mandates RSA keys (some enterprise policies, government systems)
- You need compatibility with very old OpenPGP implementations that predate Ed25519 support
- You are interacting with legacy PGP Universal Server infrastructure

For everyone else, Ed25519 is the better choice.

The v6 Key Format: Not Yet

RFC 9580 introduces **v6 keys** with several improvements over v4:

- Mandatory AEAD for symmetric encryption (no more CFB mode)
- Key creation fingerprints based on SHA-256 instead of SHA-1
- Cleaner separation of packet types
- Better metadata privacy

However, v6 keys are **not recommended for general use in 2026**:

- **GnuPG does not support v6** -- it follows LibrePGP, which uses a different (incompatible) approach
- **YubiKey smartcards may not accept v6 keys** -- the OpenPGP card applet expects v4 packet headers
- **Most ecosystem tools expect v4** -- email clients, key servers, verification tools

Do not generate v6 keys unless you know exactly what you are doing

If you generate a v6 key with `sq --profile rfc9580`, you will not be able to import it into GnuPG, load it onto a YubiKey, or use it with most tools in this guide. v6 keys are for Track C experimenters who work exclusively within Sequoia or other RFC 9580 implementations.

For a discussion of v6 migration when the ecosystem is ready, see Appendix E: Migration Guides.

Post-Quantum: A Preview

GnuPG 2.5 includes **experimental Kyber support** -- a post-quantum key encapsulation mechanism. This is a hybrid approach where Cv25519 and Kyber are combined, so that the encryption remains secure even if only one of the two algorithms holds.

Post-quantum keys are experimental in 2026. They are not covered as a primary path in this guide, but are worth watching as quantum computing advances.

3.4 1.3 Key Architecture & Threat Model

Tracks: A, B, C

Your PGP key is not a single entity -- it is a hierarchy of keys with different roles and different risk profiles. Understanding this architecture is essential before you generate anything, because the structure you choose determines what you can recover from when things go wrong.

The Master + Subkeys Model

Every OpenPGP key consists of a **master key** and one or more **subkeys**. Each key has one or more **capabilities**:

CAPABILITY	FLAG	PURPOSE	WHICH KEY?
Certify	`[C]`	Sign other keys and UIDs (your identity)	Master key only
Sign	`[S]`	Sign commits, emails, files	Subkey
Encrypt	`[E]`	Decrypt messages sent to you	Subkey
Authenticate	`[A]`	SSH authentication	Subkey

The critical insight is the **separation of certification from daily use**:

```
Master Key [C] (offline, air-gapped)
├── Signing Subkey [S] (on YubiKey or daily machine)
├── Encryption Subkey [E] (on YubiKey or daily machine)
└── Authentication Subkey [A] (on YubiKey or daily machine)
```

Your master key is your **identity**. It certifies that the subkeys belong to you, and it signs other people's keys in the Web of Trust. Because it rarely needs to be used (only for adding/revoking subkeys, adding UIDs, or signing others' keys), it can be kept **offline** on an air-gapped machine or encrypted backup.

Your subkeys do the **daily work**. They sign commits, decrypt emails, and authenticate SSH connections. They live on your daily machine (ideally on a hardware token like a YubiKey).

Why This Separation Matters

If a subkey is compromised (stolen laptop, malware), you can:

1. Revoke the compromised subkey using your offline master key
2. Generate a new subkey
3. Distribute the updated public key
4. Continue using the same identity -- your certifications, Web of Trust signatures, and key distribution all remain intact

If you had put all capabilities on a single key and that key were compromised, you would need to start over from scratch: new key, new fingerprint, new WoT signatures, new distribution.

Subkey Expiration

Subkeys should have an **expiration date** -- this guide recommends **2 years**. Expiration is a safety net, not a death sentence:

- When subkeys approach expiry, you extend them from the air-gapped master (see Part XV: Maintenance)
- If you lose access to everything, expired subkeys stop working automatically -- no one can send you encrypted mail you cannot read
- Expiration encourages periodic backup verification ("can I still access my master key?")

> **Expiration is reversible**
>
> Setting an expiration date does not destroy anything. You can always extend the expiration before (or even after) it passes, as long as you have access to the master key. Think of it as a dead man's switch, not a time bomb.

The master key should have **no expiration** (`never`). Its lifetime is managed through revocation, not expiration.

Cross-Certification

When GnuPG creates a subkey, it automatically **cross-certifies** it: the subkey signs the master key, and the master key signs the subkey. This bidirectional signature prevents an attacker from detaching your subkey and attaching it to their own master key (a "subkey theft" attack).

Cross-certification has been automatic in all modern GnuPG versions, but older keys may lack it. The `require-cross-certification` option in `gpg.conf` ensures GnuPG rejects any subkey that is not properly cross-certified.

Threat Scenarios

Understanding what each layer of protection covers:

Stolen laptop (subkeys on YubiKey)

COMPONENT	STATUS	WHY
Master key	Safe	Not on the laptop
Subkeys	Safe	On YubiKey (PIN + touch required)
Public key	Not a secret	Public by design
Passphrase	N/A	YubiKey uses PIN, not passphrase

Recovery: None needed. Your YubiKey is still in your pocket.

Compromised daily machine (malware)

COMPONENT	STATUS	WHY
Master key	Safe	Never on daily machine
Subkeys (YubiKey)	Safe	Malware cannot extract keys from hardware
Subkeys (software)	**Compromised**	Malware can read key files
Recent operations	Exposed	Malware could have intercepted plaintext

Recovery: Revoke compromised subkeys, generate new ones from offline master. This is the primary argument for using a hardware token.

Lost YubiKey (no known compromise)

COMPONENT	STATUS	WHY
Master key	Safe	On air-gapped backup
Subkeys on lost YubiKey	At risk	Finder could attempt PIN brute-force (3 attempts before lockout)

Recovery: Switch to backup YubiKey (see 4.5 Backup YubiKey). Optionally revoke the lost key's subkeys and re-provision from master backup.

Forgotten passphrase

COMPONENT	STATUS	WHY
Everything	**Inaccessible**	Passphrase protects the master key backup

Recovery: If you have no written record of the passphrase -- none. This is why Part III: Backup emphasizes paper records of the passphrase.

The Three Tracks

This guide supports three reader profiles. Each section is tagged with the tracks it applies to:

Track A: Minimal

> *"I just need Git signing and SSH."*

- Generate keys (software or hardware)
- Set up Git commit signing
- Set up SSH authentication
- Skip: email encryption, WoT, keysigning, package signing

Track B: Standard

> *"Full identity setup with YubiKey."*

- Full key generation on air-gapped system
- Hardware provisioning (YubiKey)
- Git signing, SSH, email encryption
- Password management with `pass`
- Basic key distribution (keyservers, WKD)

Track C: Advanced

> *"Debian Developer / high-assurance identity."*

- Everything in Track B, plus:
- Web of Trust participation (keysigning parties, `caff`)
- Package signing (`debsign`, RPM)
- Key distribution to multiple keyservers
- Advanced maintenance procedures
- Legal/compliance considerations

3.5 1.4 Toolchain Choice: Two Paths

Tracks: A, B, C

With the landscape and cryptographic choices established, the final strategic decision is which tools to use for key generation and day-to-day operations. This guide supports two paths -- choose one and follow it consistently.

Path 1: GnuPG-Only (The Universal Path)

Generate and manage keys entirely with `gpg`. This is the path most readers should follow.

Advantages:

- Battle-tested for decades; every tutorial, Stack Overflow answer, and man page assumes GnuPG
- Direct smartcard support (`keytocard`, `gpg-agent` SSH)
- Single toolchain -- no bridging between tools
- Required for hardware token operations regardless of generation path
- Ships with every major Linux distribution

Disadvantages:

- The `gpg` CLI is famously arcane (120+ options for `--edit-key` alone)
- Error messages are often cryptic
- Configuration requires a well-tuned `gpg.conf` to get modern defaults (we provide one in Part II)

Best for: Most users, especially those who want the simplest possible setup with the most community support.

Path 2: Hybrid (Sequoia + GnuPG)

Generate keys with Sequoia's `sq` CLI for its cleaner interface and saner defaults, then use GnuPG at runtime for smartcard support and SSH agent functionality.

Advantages:

- `sq` has better error messages and a more intuitive CLI
- Key generation defaults are secure out of the box
- RFC 9580-aligned (future-proofing, though we use v4 keys for now)
- Useful as a second opinion: `sq inspect` gives clearer key analysis than `gpg --list-keys`

Disadvantages:

- Two tools to install and learn
- Must bridge keys from `sq` format into GnuPG keyring (straightforward but an extra step)
- `sq` cannot perform smartcard operations -- GnuPG is still required for `keytocard`, `gpg-agent`, etc.
- Smaller community; fewer troubleshooting resources online

Best for: Users comfortable with newer tooling who value a better CLI experience for key generation and inspection, and who accept the minor overhead of maintaining two tools.

The Bridge: How Hybrid Works

If you choose Path 2, the workflow is:

```
sq key generate ──→ OpenPGP key file ──→ gpg --import ──→ GnuPG keyring
      ↑                                       ↓
 (v4 keys only!)                    keytocard, gpg-agent, etc.
```

The bridge works because both tools speak **v4 OpenPGP** -- the universal format. The key generated by `sq` is bit-for-bit compatible with what `gpg` expects.

Do NOT bridge v6 keys

If you generate a v6 key with `sq --profile rfc9580`, the bridge fails -- GnuPG will reject the import. Always use `--profile rfc4880` (the default in sq 1.3.1) for keys that will touch GnuPG.

Decision Matrix

FACTOR	GNUPG-ONLY	HYBRID (SQ + GPG)
Setup complexity	Lower	Slightly higher
CLI usability	Arcane but documented everywhere	Modern, clearer errors
Smartcard support	Native	Must use GnuPG
SSH agent	Native (`gpg-agent`)	Must use GnuPG
Key inspection	`gpg --list-keys` (terse)	`sq inspect` (detailed)
Community support	Vast	Growing
Future RFC 9580 readiness	No (LibrePGP fork)	Yes (when ecosystem is ready)
Required tools	`gpg` only	`sq` + `gpg`

Recommendation

If you are unsure, choose Path 1 (GnuPG-Only). It has the fewest moving parts and the most documentation. You can always install `sq` later as a supplementary tool without changing your key setup.

Choose Path 2 if you already use Sequoia tools, you want the better CLI experience during initial key generation, or you plan to participate in the RFC 9580 ecosystem as it matures.

You can switch later without re-generating keys

Because both paths produce the same v4 OpenPGP keys, you can install `sq` alongside `gpg` at any time and use it for inspection, WKD publishing, or key manipulation -- without touching your existing keys. The choice here is about **generation and primary workflow**, not a permanent commitment to a single tool.

What You Will Need Installed

Regardless of path, you will need GnuPG installed. Here is a quick reference for the tools required at each stage:

STAGE	GNUPG-ONLY	HYBRID
Key generation (air-gapped)	`gpg`	`sq` + `gpg`
Hardware provisioning	`gpg`, `ykman`	`gpg`, `ykman`
Daily machine setup	`gpg`, `gpg-agent`	`gpg`, `gpg-agent`, optionally `sq`
SSH authentication	`gpg-agent`	`gpg-agent`
Git signing	`gpg`	`gpg`
Email	`gpg` or Thunderbird	`gpg` or Thunderbird

Installation instructions for each platform are covered in Part II: Environment Preparation.

4. II: Key Generation

4.1 Part II: The Air-Gapped Forge (Key Generation)

Tracks: B, C (Track A: read 2.2 or 2.3 for non-air-gapped generation, then skip to Part V)

Your master key is your cryptographic identity. It will sign your subkeys, certify other people's keys, and anchor your presence in the Web of Trust. Generating it on an air-gapped machine -- one that has never been and will never be connected to a network -- ensures that no malware, keylogger, or remote attacker can observe or steal it during the most sensitive moment of its lifecycle.

This part walks through:

1. **Environment Preparation** -- Setting up an air-gapped system (Tails, Alpine, or NixOS LiveCD).
2. **Key Generation -- GnuPG Path** -- Generating your master key and subkeys with `gpg`.
3. **Key Generation -- Sequoia Path** -- Generating with `sq` and bridging to GnuPG.
4. **Identity & UIDs** -- Adding email addresses, managing multiple identities.
5. **Verification** -- Reading `gpg -K` output and confirming your key structure.

Do not skip air-gapping for Track B/C

If you are following Track B (YubiKey setup) or Track C (high-assurance), generating your master key on your daily machine defeats the purpose of the entire offline master + hardware subkey architecture described in Part I.

Track A readers who do not plan to use a hardware token may generate keys on their daily machine -- but should still use a strong passphrase.

4.2 2.1 Environment Preparation

Tracks: B, C

The goal is a clean, network-isolated system with GnuPG installed and a temporary keyring stored in RAM. Three options, in order of convenience -- choose whichever matches your hardware and comfort level.

Option A: Tails OS (Recommended)

Tails is an amnesic, Tor-based live operating system. When you shut it down, all data in RAM is wiped. This makes it ideal for one-time key generation ceremonies.

Steps

1. **Download and verify Tails** from tails.net. Follow their verification instructions -- Tails provides a browser extension and a GPG-signed SHA256 checksum.
2. **Write to USB** using the Tails Installer, Etcher, or `dd`.
3. **Boot from USB.** At the Tails greeter, set an administration password (needed for installing additional packages if required).
4. **Disable networking.** In the Tails greeter's additional settings, choose "Disable all networking." Alternatively, do not connect to any Wi-Fi after boot.
5. **Verify GnuPG is installed:**

   ```
   $ gpg --version
   ```

   Tails ships GnuPG by default.

⚠ **Sequoia Path users: stage sq package before booting Tails**

Tails does not ship `sq`. If you are following the Sequoia Path (2.3), download the Debian package on a networked machine and transfer it to your Tails USB before booting offline.

Download the sq package (requires Docker):

```
# On your networked machine
$ mkdir sq-deb
$ docker run --rm -v "$(pwd)/sq-deb:/output" debian:trixie bash -c '
  apt-get update -qq && cd /output && apt-get download sq 2>&1 | tail -1
'
# Result: sq-deb/sq_1.3.1-*.deb (~5 MB)
```

Without Docker, download the `.deb` directly from packages.debian.org/trixie/sq -- use the `amd64` download link (or `arm64` for Raspberry Pi).

Transfer to USB and install on Tails:

```
# Copy sq-deb/ to your Tails USB drive or a second USB drive.
# After booting Tails with networking disabled:
$ sudo dpkg -i /media/amnesia/USB_DRIVE/sq-deb/sq_*.deb
$ sq version   # Should print: sq 1.3.1
```

All of `sq`'s library dependencies (libnettle, libgmp, libssl, etc.) are already present on Tails 7+ because GnuPG depends on them.

Option B: Alpine Linux on Raspberry Pi

For the hardware-inclined, a Raspberry Pi running Alpine Linux provides a cheap, physically air-gapped environment free of opaque management coprocessors (Intel ME, AMD PSP). This approach is adapted from the drduh secure environment guide.

Steps

1. **Download Alpine** for aarch64 (or armhf for older Pi models).
2. **Prepare packages on a networked machine.** On a separate machine (or Tails), download the required packages and their dependencies:

```
# On the networked machine
$ apk fetch --recursive gnupg gnupg-scdaemon pcsclite-libs
```

3. **Transfer packages to USB drive.** Copy the `.apk` files to a FAT32 USB drive.

4. **Boot the Pi from SD card** with Alpine. Do NOT connect any network cable.
5. **Install packages from USB:**

```
$ mount /dev/sda1 /mnt
$ apk add --allow-untrusted /mnt/*.apk
```

6. **Verify SHA256 checksums** of the packages against known-good values recorded on the networked machine.

Option C: drduh's NixOS LiveCD

The drduh YubiKey guide provides a NixOS configuration that builds a purpose-specific live ISO with all required tools pre-installed.

Steps

1. **Build the ISO** on a networked NixOS machine:

```
$ nix-build '<nixpkgs/nixos>' -A config.system.build.isoImage \
  -I nixos-config=path/to/yubikey-guide/nix/yubikey-image.nix
```

2. **Write to USB** and boot.
3. **No networking, no persistence** -- everything runs in RAM.

This option is the most convenient if you already use NixOS, as the environment is fully reproducible.

Common Setup: Temporary GNUPGHOME

Regardless of which option you chose, set up a temporary GnuPG home directory in RAM before generating any keys:

```
$ export GNUPGHOME=$(mktemp -d -t gnupg_XXXXXXXXXX)
```

Why a temporary GNUPGHOME?

By default, GnuPG stores keys in `~/.gnupg`. On a live system, this is already in RAM -- but using an explicit temp directory makes it clear that nothing persists, and avoids accidentally mixing with any pre-existing keyring.

Next, create a hardened `gpg.conf` in this directory. The full annotated configuration is in Appendix A: Configuration Reference. For the air-gapped key generation ceremony, copy the Appendix A `gpg.conf` and add these two options that are specific to the air-gapped environment:

```
$ cp /path/to/appendix-a-gpg.conf "$GNUPGHOME/gpg.conf"
```

Or create it manually, adding these air-gapped-specific options:

```
$ cat << 'EOF' >> "$GNUPGHOME/gpg.conf"
# Air-gapped additions (not in daily config)
require-secmem
throw-keyids
EOF
```

OPTION	PURPOSE	WHY AIR-GAPPED ONLY?
`require-secmem`	Refuse to run without secure memory	Desktop environments sometimes interfere
`throw-keyids`	Omit recipient key IDs in encrypted output	Privacy; not needed on daily machine where debugging matters more

See Appendix A for the complete annotated `gpg.conf` with all algorithm preferences, display options, and keyserver settings.

You are now ready to generate your keys. Proceed to either 2.2 GnuPG Path or 2.3 Sequoia Path, depending on your toolchain choice.

4.3 2.2 Key Generation -- GnuPG Path

Tracks: A, B, C -- Path 1 (GnuPG-Only)

This section generates a complete key hierarchy using only `gpg`: a certification-only master key and three purpose-specific subkeys, all using Ed25519/Cv25519.

Prerequisites

- Air-gapped environment set up (2.1)
- Temporary `$GNUPGHOME` with hardened `gpg.conf`
- A strong passphrase prepared (see below)

Choosing a Passphrase

Your passphrase protects the master key's private material. If an attacker obtains your encrypted key backup, the passphrase is the only thing standing between them and your identity.

Use a Diceware passphrase

Roll physical dice (or use `shuf` on the air-gapped machine) to select **6 or more words** from a Diceware word list. Example:

```
correct horse battery staple lunar quantum
```

- Do NOT reuse a passphrase from any other system
- Do NOT rely on a "clever" substitution pattern -- randomness beats cleverness every time
- **Write the passphrase on paper** and store it in a physically secure location (separate from your digital backups)

Step 1: Set Your Identity

Define your identity string. GnuPG uses this as the primary UID on your key:

```
$ IDENTITY="Alice Example <alice@example.com>"
```

Step 2: Generate the Master Key

Create a certification-only Ed25519 master key with no expiration:

```
$ gpg --quick-generate-key "$IDENTITY" ed25519 cert never
```

GnuPG will prompt you for a passphrase. Enter your Diceware passphrase.

What `cert never` means

- `ed25519`: Use the Ed25519 algorithm
- `cert`: Certification capability only -- this key can sign other keys and UIDs but cannot sign data, encrypt, or authenticate
- `never`: No expiration date. The master key's lifetime is managed through revocation, not expiration.

Capture the fingerprint for subsequent commands:

```
$ KEYFP=$(gpg --list-options show-only-fpr-mbox --list-secret-keys | awk '{print $1}')
$ echo "Key fingerprint: $KEYFP"
```

Step 3: Add Subkeys

Add three subkeys, each with a 2-year expiration:

Signing subkey [S] **Encryption subkey [E]** **Authentication subkey [A]**

```
$ gpg --quick-add-key $KEYFP ed25519 sign 2y
```

```
$ gpg --quick-add-key $KEYFP cv25519 encr 2y
```

```
$ gpg --quick-add-key $KEYFP ed25519 auth 2y
```

Or run all three in sequence:

```
$ gpg --quick-add-key $KEYFP ed25519 sign 2y
$ gpg --quick-add-key $KEYFP cv25519 encr 2y
$ gpg --quick-add-key $KEYFP ed25519 auth 2y
```

Why 2-year expiration?

Subkey expiration is a safety net. If you lose access to everything, expired subkeys stop working automatically. You can always extend the expiration before it passes (see Part XV: Maintenance). Two years balances convenience against risk.

Step 4: Verify the Key Structure

List your secret keys:

```
$ gpg -K
```

You should see output like this:

```
sec   ed25519/0x1234567890ABCDEF 2026-02-09 [C]
      Key fingerprint = ABCD 1234 5678 90AB CDEF  1234 5678 90AB CDEF 1234
uid                   [ultimate] Alice Example <alice@example.com>
ssb   ed25519/0x2345678901BCDEF0 2026-02-09 [S] [expires: 2028-02-09]
ssb   cv25519/0x3456789012CDEF01 2026-02-09 [E] [expires: 2028-02-09]
ssb   ed25519/0x456789012ADEF012 2026-02-09 [A] [expires: 2028-02-09]
```

Verify:

- **Master key:** `sec` with `[C]` capability, `ed25519`, no expiration
- **Signing subkey:** `ssb` with `[S]`, `ed25519`, 2-year expiry
- **Encryption subkey:** `ssb` with `[E]`, `cv25519`, 2-year expiry
- **Authentication subkey:** `ssb` with `[A]`, `ed25519`, 2-year expiry
- **UID:** Your identity string with `[ultimate]` trust

If anything looks wrong, delete the key and start over -- you are on an air-gapped machine with nothing to lose.

The gpg.conf Explained

The configuration from 2.1 is based on the full annotated reference in Appendix A: Configuration Reference. See Appendix A for a line-by-line explanation of every option.

The key points for key generation:

- **Algorithm preferences** (`personal-cipher-preferences`, `cert-digest-algo`, `s2k-*`): Ensure SHA-512 and AES-256 are preferred
- **`keyid-format 0xlong`**: Display full 16-character key IDs with `0x` prefix (short IDs are collision-prone)
- **`require-cross-certification`**: Reject subkeys without back-signatures
- **`require-secmem`**: Air-gapped only -- refuse to run without secure memory
- **`throw-keyids`**: Air-gapped only -- omit recipient key IDs for privacy

GnuPG 2.5 defaults are improved

If you are using GnuPG 2.5+, the default cipher and digest preferences are already sane (AES-256, SHA-512). The explicit `gpg.conf` is still valuable for `throw-keyids`, `no-comments`, `no-emit-version`, and other privacy/ operational options that are not defaults.

Next Steps

- **Add additional UIDs:** If you have multiple email addresses, see 2.4 Identity & UIDs
- **Verify your key:** See 2.5 Verification
- **Back up immediately:** Proceed to Part III: Backup before doing anything else with this key

4.4 2.3 Key Generation -- Sequoia Path

Tracks: A, B, C -- Path 2 (Hybrid: Sequoia + GnuPG)

This section generates a complete key hierarchy using Sequoia's `sq` CLI, then bridges it into GnuPG for runtime use. Choose this path if you prefer `sq`'s cleaner interface and error messages (see 1.4 Toolchain Choice).

Prerequisites

- Air-gapped environment set up (2.1)
- `sq` installed on the air-gapped system (**v1.0+ required** -- see warning below)
- `gpg` also installed (needed for the bridge and all subsequent hardware/runtime steps)

⚠ **sq 1.0+ required -- distro packages are often too old**

Ubuntu 24.04 ships sq 0.33, which has a **completely different CLI** from sq 1.0+. The commands in this section will not work with 0.x. Check with `sq version` -- if it reports 0.x, install via `cargo install --locked sequoia-sq` (requires Rust toolchain and build dependencies -- see 16.1 Sequoia).

Tails 7+ is based on Debian Trixie and can install the `sq` 1.3.1 package -- but you must stage the `.deb` on USB before booting offline. See 2.1 Environment Preparation for instructions.

Step 1: Generate the Key

```
$ sq key generate \
  --own-key \
  --name "Alice Example" --email alice@example.com \
  --cipher-suite cv25519 \
  --output secret-key.pgp \
  --rev-cert secret-key.rev
```

This creates a v4 key with:

- Ed25519 master key with certification capability
- Ed25519 signing subkey
- Cv25519 encryption subkey

`sq` will prompt for a passphrase to protect the private key material.

Profile check: v4 only

The default profile in sq 1.3.1 is `--profile rfc4880`, which produces **v4 keys**. This is correct.

Do NOT use `--profile rfc9580` -- it produces v6 keys that cannot be imported into GnuPG and may not load onto YubiKey smartcards.

If a future version of `sq` changes the default, always specify explicitly:

```
$ sq key generate --profile rfc4880 --cipher-suite cv25519 ...
```

Step 2: Add an Authentication Subkey

The default `sq key generate` creates signing and encryption subkeys but not an authentication subkey. Add one (the command updates the file in place):

```
$ sq key subkey add --can-authenticate --cert-file secret-key.pgp --output secret-key.pgp
```

Why authentication separately?

The authentication subkey [A] is used for SSH via `gpg-agent`. Not everyone needs it -- Track A readers who use FIDO2 for SSH (see Part VI) can skip this step.

Step 3: Inspect the Key

Verify the key structure before bridging:

```
$ sq inspect secret-key.pgp
```

You should see output showing:

- A master key with `C` (certify) capability, algorithm `EdDSA`
- A signing subkey with `S` capability
- An encryption subkey with `E` capability
- An authentication subkey with `A` capability (if added)
- Your UID

Sequoia terminology

Sequoia uses the RFC term "primary key" in its output. This guide uses "master key" for the same concept.

Step 4: Bridge to GnuPG

Import the Sequoia-generated key into GnuPG:

```
$ gpg --import secret-key.pgp
```

The bridge only works for v4 keys

This import succeeds because both `sq` and `gpg` speak v4 OpenPGP. If you had generated a v6 key, `gpg` would reject it with an error about unknown packet types.

Set the trust level to ultimate (this is your own key):

```
$ KEYFP=$(gpg --list-options show-only-fpr-mbox --list-secret-keys | awk '{print $1}')
$ echo -e "5\ny\n" | gpg --command-fd 0 --expert --edit-key $KEYFP trust
```

Verify the import:

```
$ gpg -K
```

You should see the same key structure as in 2.2 Verification -- master `[C]` key with three subkeys `[S]`, `[E]`, `[A]`.

Step 5: Secure the Original File

After a successful import and verification, securely delete the original Sequoia key file:

```
$ shred -u secret-key.pgp
```

The key now lives only in the GnuPG keyring (in your temporary `$GNUPGHOME`). From this point forward, all operations use `gpg` -- including backup, hardware provisioning, and daily use.

Differences from the GnuPG Path

ASPECT	GNUPG PATH (2.2)	SEQUOIA PATH (2.3)
Generation tool	`gpg --quick-generate-key`	`sq key generate`
Configuration	Requires hardened `gpg.conf`	Sane defaults built-in
Subkey expiry	Set explicitly (`2y`)	Set by `sq` defaults
Bridge required	No	Yes (`gpg --import`)
End result in GnuPG keyring	Identical	Identical

After the bridge, the keys are indistinguishable. All subsequent sections in this guide work the same regardless of which path you used to generate.

Next Steps

- **Add additional UIDs:** See 2.4 Identity & UIDs
- **Verify your key:** See 2.5 Verification
- **Back up immediately:** Proceed to Part III: Backup

4.5 2.4 Identity & UIDs

Tracks: A, B, C

A User ID (UID) binds a human-readable identity -- typically a name and email address -- to your key. You can have multiple UIDs on the same key, one for each email address you want associated with your cryptographic identity.

Adding UIDs

If you have additional email addresses (work, personal, project-specific), add them as UIDs to your key:

```
$ gpg --quick-add-uid $KEYFP "Alice Example <alice@work.example.com>"
```

Repeat for each additional email address.

Setting the Primary UID

The primary UID is the one displayed by default when others view your key. To set a specific UID as primary:

```
$ gpg --edit-key $KEYFP
```

Then in the interactive prompt:

```
gpg> uid 2           # Select the UID you want as primary
gpg> primary         # Set it as the primary UID
gpg> save            # Save and exit
```

When primary UID matters

The primary UID affects which name and email appear by default in:

- `gpg --list-keys` output
- Git commit signatures
- Keyserver listings
- Email client key selection

Choose the identity you use most frequently as your primary UID.

What NOT to Put in UIDs

Comment fields

The OpenPGP UID format allows a comment in parentheses:

```
Alice Example (Personal) <alice@example.com>
```

Do not use comments. They are unnecessary metadata that:

- Cannot be changed without revoking the UID and adding a new one
- Leak information about your key's intended purpose
- Clutter keyserver listings

If you need to distinguish between personal and work identities, use separate UIDs with the appropriate email address -- the email itself provides context.

Photos

GnuPG supports embedding a photo in your key via `addphoto` in `--edit-key`. **Do not do this** -- photos:

- Bloat your public key (JPEG adds kilobytes to a key that should be kilobytes)
- Cause problems with keyserver uploads
- Provide no cryptographic benefit

Separate Keys for Separate Identities

If you need a pseudonymous identity (online handle, project persona) that must be **completely unlinkable** to your real identity, do not add it as a UID on your main key. Instead:

1. **Generate a separate master key** with its own subkeys
2. **Use a different passphrase** (prevents correlation via passphrase reuse)
3. **Never cross-sign** between the two keys
4. **Use different keyservers or distribution channels** if possible
5. **Generate and manage in separate sessions** on the air-gapped machine

UIDs on the same key are inherently linked

Anyone who retrieves your public key from a keyserver will see ALL UIDs on that key. If you add both `alice@example.com` and `darkwolf1337@mail.com` as UIDs on the same key, those identities are permanently and publicly linked.

For true separation, you need separate keys.

Revoking a UID

If you need to remove a UID (changed jobs, decommissioned email):

```
$ gpg --edit-key $KEYFP
```

```
gpg> uid 2          # Select the UID to revoke
gpg> revuid         # Revoke it
gpg> save
```

Revocation is permanent and public

Revoking a UID does not delete it -- it adds a revocation signature that marks the UID as invalid. The old UID and email address remain visible on the key when you distribute the updated public key; they are simply marked as revoked.

Next Steps

After adding all desired UIDs, proceed to 2.5 Verification to confirm your complete key structure, then immediately to Part III: Backup.

4.6 2.5 Verification

Tracks: A, B, C

Before you leave the air-gapped environment, verify that your key was generated correctly. This is the last checkpoint before backup -- mistakes caught here are free; mistakes caught later are expensive.

Reading `gpg -K` Output

Run the secret key listing:

```
$ gpg -K
```

Here is an annotated example of correct output:

```
sec   ed25519/0x1234567890ABCDEF 2026-02-09 [C]
      Key fingerprint = ABCD 1234 5678 90AB CDEF  1234 5678 90AB CDEF 1234
uid                   [ultimate] Alice Example <alice@example.com>
uid                   [ultimate] Alice Example <alice@work.example.com>
ssb   ed25519/0x2345678901BCDEF0 2026-02-09 [S] [expires: 2028-02-09]
ssb   cv25519/0x3456789012CDEF01 2026-02-09 [E] [expires: 2028-02-09]
ssb   ed25519/0x456789012ADEF012 2026-02-09 [A] [expires: 2028-02-09]
```

Key prefixes

PREFIX	MEANING
`sec`	Secret (private) master key is present locally
`sec#`	Master key is NOT present locally (stub only -- seen on daily machine after backup)
`ssb`	Secret subkey is present locally
`ssb>`	Subkey is on a smartcard (seen after `keytocard`)

Capability flags

FLAG	CAPABILITY
`[C]`	Certify -- can sign other keys and UIDs
`[S]`	Sign -- can sign data (commits, emails, files)
`[E]`	Encrypt -- can decrypt data encrypted to this key
`[A]`	Authenticate -- can be used for SSH

Trust levels

TRUST	MEANING
`[ultimate]`	You own this key (set it yourself)
`[full]`	You fully trust this key's owner to verify identities
`[marginal]`	You somewhat trust this key's owner
`[unknown]`	No trust assigned

Verification Checklist

Go through each item:

- ✓ **Master key algorithm:** `ed25519` (not RSA, not NIST P-256)
- ✓ **Master key capability:** `[C]` only (not `[SC]` or `[SCE]`)
- ✓ **Master key expiration:** None (no `[expires:]` shown)
- ✓ **Signing subkey:** `ed25519`, `[S]`, 2-year expiry
- ✓ **Encryption subkey:** `cv25519`, `[E]`, 2-year expiry
- ✓ **Authentication subkey:** `ed25519`, `[A]`, 2-year expiry
- ✓ **UID(s):** Correct name and email address(es)
- ✓ **Trust:** `[ultimate]` for all UIDs
- ✓ **Key count:** Exactly 1 master + 3 subkeys (4 total)

Common mistakes

- **Master key has `[SC]`:** You generated a combined certify+sign master. This works but means your master key participates in daily signing operations, reducing the benefit of offline storage. Regenerate with `cert` only if you are following Track B/C.
- **Wrong algorithm:** If you see `rsa4096` or `nistp256`, you did not specify `ed25519` during generation. Start over.
- **Missing subkey:** If you only see 2 subkeys, you may have forgotten the authentication subkey. Add it now with `gpg --quick-add-key $KEYFP ed25519 auth 2y`.

Detailed Key Inspection

For more detail, use the `--with-keygrip` flag:

```
$ gpg -K --with-keygrip
```

This shows the **keygrip** -- a hash of the key's public parameters that `gpg-agent` uses internally. Keygrips become relevant when configuring SSH authentication (see Part VI).

Example additional output:

```
      Keygrip = A1B2C3D4E5F6A1B2C3D4E5F6A1B2C3D4E5F6A1B2
```

Record Your Fingerprint

Write down or print your full key fingerprint:

```
$ gpg --fingerprint $KEYFP
```

You will need this fingerprint for:

- Distributing to others for verification
- Configuring Git signing (`user.signingkey`)
- Hardware token provisioning
- Key server uploads

Next Steps

Your key is generated and verified. **Do not shut down the air-gapped machine yet.** You must first complete:

1. **Part III: Backup & Disaster Recovery** -- Export and back up everything
2. **Part IV: Hardware Provisioning** -- Load subkeys onto YubiKey (Track B/C)

Only after backups are verified and hardware is provisioned should you shut down the air-gapped system.

5. III: Backup & Recovery

5.1 Part III: Backup & Disaster Recovery

Tracks: A, B, C

You have just generated the most sensitive cryptographic material you own. Before you do anything else -- before loading keys onto a YubiKey, before leaving the air-gapped machine -- you must create verified backups.

The goal is **defense in depth**: multiple backup formats, stored in multiple physical locations, each independently sufficient to restore your identity.

1. **Export Everything** -- Public key, secret key, subkeys-only, ownertrust.
2. **Paperkey** -- Printable, minimal secret-key backup on paper.
3. **QR Code Backup** -- Machine-readable paper backup via QR codes.
4. **Encrypted USB Backup** -- Full GNUPGHOME on LUKS-encrypted USB drives.
5. **Revocation Certificates** -- The nuclear option for emergency identity invalidation.

Backup BEFORE keytocard

If you are following Track B/C (YubiKey), you will use `keytocard` in Part IV to load subkeys onto hardware. `keytocard` is **destructive** -- it replaces the on-disk private key with a stub. If you have not backed up before that step, your private key material exists only on the YubiKey and cannot be extracted.

Complete ALL of Part III before proceeding to Part IV.

5.2 3.1 Export Everything

Tracks: A, B, C

GnuPG stores key material in an internal database. To create portable backups, you must export it to standard files. Four exports are needed, each serving a different recovery scenario.

The Four Exports

1. Public Key

```
$ gpg --export --armor $KEYFP > $GNUPGHOME/public.asc
```

Contains: All public key material, UIDs, self-signatures, and subkey public portions. This is what you distribute to others and upload to keyservers.

When you need it: Always. Required by `paperkey` for restoration, required for import on daily machines, required for keyserver uploads.

2. Full Secret Key

```
$ gpg --export-secret-keys --armor $KEYFP > $GNUPGHOME/secret.asc
```

Contains: Everything in the public key, plus the private key material for the master key AND all subkeys. Encrypted with your passphrase.

When you need it: Complete disaster recovery. This single file, plus your passphrase, can reconstruct everything.

3. Subkeys Only

```
$ gpg --export-secret-subkeys --armor $KEYFP > $GNUPGHOME/subkeys.asc
```

Contains: Public key material plus private key material for subkeys only. The master key's private portion is **stripped** -- replaced with a dummy packet.

When you need it: Importing onto your daily machine (Track A with software keys). This gives your daily machine signing, encryption, and authentication capabilities without exposing the master key.

4. Ownertrust

```
$ gpg --export-ownertrust > $GNUPGHOME/ownertrust.txt
```

Contains: Your trust database -- which keys you trust and at what level.

When you need it: Restoring your trust decisions after importing keys on a new machine. Without this, all keys default to "unknown" trust.

Summary Table

FILE	CONTAINS	SENSITIVE?	RECOVERY SCENARIO
`public.asc`	Public key + UIDs	No	Paperkey restoration, daily machine import
`secret.asc`	Full private key	**Yes**	Complete disaster recovery
`subkeys.asc`	Subkeys private only	**Yes**	Daily machine import (software keys)
`ownertrust.txt`	Trust database	Low	Restoring trust decisions

Verify the Exports

Before proceeding, verify that each file was created and is non-empty:

```
$ ls -la $GNUPGHOME/*.asc $GNUPGHOME/ownertrust.txt
```

You can also verify the public key parses correctly:

```
$ gpg --import-options show-only --import $GNUPGHOME/public.asc
```

This displays the key without importing it -- useful for verifying the export contains what you expect.

Why export to $GNUPGHOME?

Keeping all exports in the temporary GNUPGHOME directory ensures they are in one place for copying to backup media (USB drives, paper). On a live system, this directory is in RAM and will be wiped on shutdown.

Treat secret exports like the keys themselves

`secret.asc` and `subkeys.asc` contain your private key material (encrypted with your passphrase). Handle them with the same care as the keyring itself: never store them on unencrypted media, never transmit them over a network, and securely delete any intermediate copies after transferring to backup media.

Optional: SSH Public Key

If you use GPG for SSH authentication (see Part VI), also export the SSH public key for convenient setup on remote hosts:

```
$ gpg --export-ssh-key $KEYFP > $GNUPGHOME/gpg-ssh.pub
```

This file is safe to distribute -- it is equivalent to an `id_ed25519.pub` and can be added to `~/.ssh/authorized_keys` on any server.

Next Steps

With raw exports in hand, proceed to create multiple backup formats:

- 3.2 Paperkey -- printable paper backup
- 3.3 QR Code Backup -- machine-scannable paper backup
- 3.4 Encrypted USB Backup -- digital backup on encrypted drives

5.3 3.2 Paperkey -- The Gold Standard

Tracks: B, C (Track A: optional but recommended)

Paperkey extracts only the secret portion of your key, producing a compact output that fits on a single printed page for Ed25519 keys. Paper survives USB drive failures, filesystem corruption, and format obsolescence.

How Paperkey Works

An OpenPGP secret key file contains two things:

1. The **public key** (large: certificates, UIDs, signatures)
2. The **secret key material** (small: just the private numbers)

Paperkey strips away the public portion, leaving only the secret bytes plus metadata needed for reconstruction. For Ed25519, the private key is 32 bytes -- the paperkey output is trivially small.

ALGORITHM	FULL SECRET EXPORT	PAPERKEY OUTPUT	REDUCTION
Ed25519	~3 KB	~300 bytes	~90%
RSA-4096	~6 KB	~3 KB	~50%
DSA	~2 KB	~1.8 KB	~10%

Creating the Paperkey Backup

```
$ gpg --export-secret-keys $KEYFP | paperkey --output $GNUPGHOME/paperkey.txt
```

View the output:

```
$ cat $GNUPGHOME/paperkey.txt
```

You will see output like:

```
# Secret portions of key 0x1234567890ABCDEF
# Base16 data extracted Sat Feb  9 12:00:00 2026
# Created with paperkey 1.6 by David Shaw
#
# Symmetric cipher: AES256
# S2K mode: iterated and salted
...
1: 01 FE 2A B3 ... (hex bytes)
```

Paperkey is USELESS without your public key

This is the most common misunderstanding about paperkey. The output contains **only** the secret bytes. To restore a working key, you need **both**:

1. The paperkey output (the secret bytes)
2. Your **public key** (the structure, UIDs, signatures)

Always back up `public.asc` alongside your paperkey printout. While you *may* be able to retrieve it from a keyserver, keyservers can purge keys (GDPR requests) or go offline. **Do not rely solely on keyserver retrieval for disaster recovery.** Paperkey alone = unrecoverable key.

Printing

Print the paperkey output on a laser printer (laser toner is more durable than inkjet):

```
$ lpr $GNUPGHOME/paperkey.txt
```

Or copy the file to a USB drive and print from a separate machine. Store the printed copy in a physically secure location -- a safe, a bank deposit box, or with a trusted family member.

Print the public key alongside paperkey

Even if you upload your public key to a keyserver, print `public.asc` on a separate sheet so your paper backup is **self-contained**. Keyservers can purge keys (e.g., GDPR requests on keys.openpgp.org), and digital-only backups of the public key can degrade. For a machine-scannable alternative, include the public key in your QR Code Backup. A paper backup that depends on an external service for restoration is not a complete backup.

Restoration

To restore your key from a paperkey backup, you need both files:

```
$ paperkey --pubring public.asc --secrets paperkey.txt | gpg --import
```

If your public key is on a keyserver:

```
$ gpg --recv-keys $KEYFP
$ paperkey --pubring <(gpg --export $KEYFP) --secrets paperkey.txt | gpg --import
```

The Restoration Drill

Test your backup before trusting it

Do this now, before leaving the air-gapped machine. Create a second temporary GNUPGHOME and attempt a full restoration:

```
$ export TEST_HOME=$(mktemp -d -t gnupg_test_XXXXXXXXXX)
$ GNUPGHOME=$TEST_HOME gpg --import $GNUPGHOME/public.asc
$ paperkey --pubring $GNUPGHOME/public.asc \
           --secrets $GNUPGHOME/paperkey.txt | \
           GNUPGHOME=$TEST_HOME gpg --import
$ GNUPGHOME=$TEST_HOME gpg -K
```

You should see the same key structure as your original. If the restoration fails, your paperkey backup is defective -- regenerate it before proceeding.

```
$ rm -rf $TEST_HOME
```

Next Steps

- 3.3 QR Code Backup -- Machine-scannable version of the paperkey output
- 3.4 Encrypted USB Backup -- Full digital backup on encrypted drives

5.4 3.3 QR Code Backup

Tracks: B, C (Track A: optional)

QR codes provide a machine-readable paper backup that avoids the error-prone process of manual OCR or retyping hex digits. For Ed25519 keys, the paperkey output is small enough to fit in a single QR code. For larger keys, we split across multiple codes.

Prerequisites

Install the QR code tools on your air-gapped system (pre-download if needed):

Linux (Debian/Ubuntu) **Linux (Alpine)**

```
$ sudo apt install qrencode zbar-tools imagemagick
```

```
$ apk add qrencode zbar imagemagick
```

Generating QR Codes

Single QR (Ed25519 keys)

If your paperkey output is small (under ~1 KB, typical for Ed25519):

```
$ cat $GNUPGHOME/paperkey.txt | qrencode -l H -o $GNUPGHOME/qr-backup.png
```

The `-l H` flag sets **high error correction** (30% of the code can be damaged and still scan successfully).

Split QR (RSA or large keys)

For larger keys, split the paperkey output into chunks:

```
$ split -b 1500 $GNUPGHOME/paperkey.txt $GNUPGHOME/qr-chunk-

$ for chunk in $GNUPGHOME/qr-chunk-*; do
    qrencode -l H -o "${chunk}.png" < "$chunk"
done
```

Creating a Printable Sheet

Combine QR codes into a single printable page using ImageMagick:

```
$ montage $GNUPGHOME/qr-*.png \
  -geometry +10+10 \
  -tile 2x \
  -title "GPG Key Backup - $(date +%Y-%m-%d)" \
  $GNUPGHOME/qr-sheet.png
```

Print the sheet:

```
$ lpr $GNUPGHOME/qr-sheet.png
```

Verification: The Round-Trip Test

Always verify before trusting

Scan the QR code(s) back and compare with the original paperkey output. A QR code that does not scan is worse than no backup -- it gives false confidence.

```
$ zbarimg -q --raw $GNUPGHOME/qr-backup.png > $GNUPGHOME/qr-decoded.txt
$ diff $GNUPGHOME/paperkey.txt $GNUPGHOME/qr-decoded.txt
```

The `-q` flag suppresses extraneous output that would otherwise corrupt the decoded data. If `diff` shows no output, the round-trip is perfect. If there are differences, regenerate the QR code.

For split QR codes, decode each and concatenate:

```
$ for png in $GNUPGHOME/qr-chunk-*.png; do
    zbarimg -q --raw "$png"
done > $GNUPGHOME/qr-decoded.txt

$ diff $GNUPGHOME/paperkey.txt $GNUPGHOME/qr-decoded.txt
```

Including the Public Key

Paperkey output is **useless without the public key** (see 3.2 Paperkey). To make your paper backup self-contained, also encode your public key as a QR code:

```
$ gpg --export --armor $KEYFP > $GNUPGHOME/public.asc
$ cat $GNUPGHOME/public.asc | qrencode -l H -o $GNUPGHOME/qr-public.png
```

For Ed25519 keys with a single UID and three subkeys, the ASCII-armored public key is typically ~800 bytes -- within a single QR code's capacity at `-l H` (~1,273 bytes). For keys with many UIDs, use `-l M` for more capacity or the split approach above. For larger keys (RSA-4096), always use the split approach.

Add the public key QR to your printable sheet:

```
$ montage $GNUPGHOME/qr-backup.png $GNUPGHOME/qr-public.png \
  -geometry +10+10 \
  -tile 2x \
  -title "GPG Key Backup - $(date +%Y-%m-%d)" \
  $GNUPGHOME/qr-sheet.png
```

Verify the public key QR too

Run the same round-trip test on `qr-public.png`:

```
$ zbarimg -q --raw $GNUPGHOME/qr-public.png > $GNUPGHOME/qr-public-decoded.asc
$ diff $GNUPGHOME/public.asc $GNUPGHOME/qr-public-decoded.asc
```

Storage Recommendations

- Print on **laser printer** (toner resists moisture better than inkjet)

- Store in a **waterproof bag** (ziplock or lamination)
- Keep **separate from** your encrypted USB backups (different threat model: paper survives filesystem corruption; USB survives physical damage)
- Label clearly: "GPG Key Backup -- requires public key for restoration"

Next Steps

- 3.4 Encrypted USB Backup -- Full digital backup
- 3.5 Revocation Certificates -- Emergency identity invalidation

5.5 3.4 Encrypted USB Backup

Tracks: A, B, C

Paper backups protect against digital failure. Encrypted USB drives protect against paper loss. Together, they form a robust disaster recovery strategy.

Why LUKS?

LUKS (Linux Unified Key Setup) is the standard full-disk encryption for Linux. It provides:

- AES-256 encryption of the entire partition
- A passphrase-protected key slot (your same Diceware passphrase works here)
- Standard tooling available on every Linux distribution

Creating an Encrypted USB Drive

This will erase all data on the target USB drive

Double-check the device name. `lsblk` will help you identify the correct drive.

Step 1: Identify the USB drive

```
$ lsblk
```

Look for your USB drive (e.g., `/dev/sdb`). It will not have your system partitions on it.

Step 2: Create the LUKS volume

```
$ sudo cryptsetup luksFormat /dev/sdX
```

You will be prompted for a passphrase. Use the same Diceware passphrase as your key, or a different one if you prefer (but then you have two passphrases to remember/store).

Step 3: Open and format

```
$ sudo cryptsetup luksOpen /dev/sdX gpg-backup
$ sudo mkfs.ext4 /dev/mapper/gpg-backup
```

Step 4: Mount and copy

```
$ sudo mkdir -p /mnt/gpg-backup
$ sudo mount /dev/mapper/gpg-backup /mnt/gpg-backup
$ sudo cp -r $GNUPGHOME/* /mnt/gpg-backup/
```

Step 5: Verify the copy

```
$ ls -la /mnt/gpg-backup/
$ diff $GNUPGHOME/public.asc /mnt/gpg-backup/public.asc
$ diff $GNUPGHOME/secret.asc /mnt/gpg-backup/secret.asc
```

Step 6: Unmount and close

```
$ sudo umount /mnt/gpg-backup
$ sudo cryptsetup luksClose gpg-backup
```

How Many Drives?

At least two, in different locations

USB drives fail. Locations flood, burn, or get burgled. Create at least two encrypted USB backups and store them in **different physical locations**:

- One at home (safe, locked drawer)
- One offsite (bank deposit box, trusted family member's home, secure office)

The cost of a USB drive is trivial compared to the cost of losing your cryptographic identity.

What to Copy

The USB backup should contain your **entire** `$GNUPGHOME`, which at this point includes:

FILE	PURPOSE
`public.asc`	Public key export
`secret.asc`	Full secret key export
`subkeys.asc`	Subkeys-only export
`ownertrust.txt`	Trust database
`paperkey.txt`	Paperkey output
`gpg.conf`	Hardened configuration
`openpgp-revocs.d/`	Auto-generated revocation certificate
`private-keys-v1.d/`	GnuPG internal private key storage
`pubring.kbx`	GnuPG public keyring
`trustdb.gpg`	Trust database (binary)

Restoration from USB

On any Linux machine:

```
$ sudo cryptsetup luksOpen /dev/sdX gpg-backup
$ sudo mount /dev/mapper/gpg-backup /mnt/gpg-backup

$ export GNUPGHOME=$(mktemp -d -t gnupg_restore_XXXXXXXXXX)
$ cp -r /mnt/gpg-backup/* $GNUPGHOME/
$ gpg -K  # Verify the key is accessible
```

Media Health and Refresh

USB flash drives degrade over time -- stored data can suffer bit rot after several years, especially on cheap NAND flash. Mitigate this:

- **Verify every 6 months.** Mount the drive, diff key files against a known good copy. This is part of the maintenance calendar.
- **Refresh every 2-3 years.** Copy to a new drive and retire the old one. Flash cells wear out even without writes (charge leakage).
- **Use quality drives.** Enterprise-grade or industrial USB drives have better NAND and error correction than promotional giveaways.

macOS (non-LUKS alternative)

macOS cannot natively open LUKS volumes. Use an encrypted APFS disk image instead:

```
# Create the encrypted image (you will be prompted for a passphrase)
$ hdiutil create -size 64m -encryption AES-256 -fs APFS \
  -volname "gpg-backup" gpg-backup.dmg

# Mount, copy, and eject
$ hdiutil attach gpg-backup.dmg
$ cp -r $GNUPGHOME/* /Volumes/gpg-backup/
$ hdiutil detach /Volumes/gpg-backup
```

Next Steps

- 3.5 Revocation Certificates -- Generate and store the emergency invalidation certificate

5.6 3.5 Revocation Certificates

Tracks: A, B, C

A revocation certificate is a pre-signed statement that permanently invalidates your key. Once published to keyservers, it tells the world: "This key should no longer be trusted." It cannot be undone.

Why You Need One Before You Need It

If your master key is compromised, you need to revoke it immediately. But if the attacker also has your backups, you might not have access to the master key to generate a revocation certificate at that point. By creating one now -- before any disaster -- you have an emergency exit.

A revocation certificate is a weapon

Anyone who obtains your revocation certificate and uploads it to a keyserver can permanently destroy your public key. Store it **separately** from your key material and with extreme care.

Auto-Generated Revocation Certificate

GnuPG automatically creates a revocation certificate when you generate a key. It is stored in:

```
$ ls $GNUPGHOME/openpgp-revocs.d/
```

The filename is your full fingerprint with a `.rev` extension. This certificate uses reason code `0` (no reason specified) and has no comment.

Generating Custom Revocation Certificates

For more control, generate certificates with specific reason codes:

```
$ gpg --gen-revoke $KEYFP > $GNUPGHOME/revoke-compromised.asc
```

GnuPG will prompt you for:

1. **Reason for revocation:**
   - `0` -- No reason specified
   - `1` -- Key has been compromised
   - `2` -- Key is superseded (you have generated a replacement)
   - `3` -- Key is no longer used
2. **Optional description** -- A short text explaining the revocation.
3. **Confirmation** -- Type `y` to confirm.

Create one for each scenario

Consider generating two revocation certificates:

- One with reason `1` (compromised) -- for emergencies
- One with reason `2` (superseded) -- for planned key rotation

Label them clearly so you use the correct one under stress.

Storage

Revocation certificates must be stored **separately** from your key backups:

STORAGE	KEY BACKUPS	REVOCATION CERTIFICATES
Home safe	Yes	**No**
Bank deposit box	Yes	**Yes** (different location from key backup)
Paper in sealed envelope	No	**Yes** (with trusted person)

The logic: if someone breaks into your safe and steals your key backups, they should NOT also find the tool to revoke your key (which would add insult to injury but also alert you via keyserver). Conversely, if you lose access to your key material, you should still be able to reach a revocation certificate from a different location.

Using a Revocation Certificate

In an emergency, import the revocation certificate into your keyring and upload to keyservers:

```
$ gpg --import revoke-compromised.asc
$ gpg --keyserver hkps://keyserver.ubuntu.com --send-keys $KEYFP
$ gpg --keyserver hkps://keys.openpgp.org --send-keys $KEYFP
```

After this:

- Your key appears as **revoked** on keyservers
- Anyone who refreshes your key will see the revocation
- Encrypted messages to your key will warn the sender
- Signatures from your key will show as "revoked signer"

Revocation is permanent

There is no "un-revoke" operation. Once a revocation certificate is published and propagated to keyservers, your key is permanently invalid. You will need to generate an entirely new key and rebuild your identity from scratch.

Only use this when you are certain the key must be invalidated.

The Backup Summary

At this point, you should have:

BACKUP TYPE	CONTENTS	LOCATION
Paper (paperkey)	Secret bytes only	Safe or deposit box
Paper (QR code)	Machine-readable secret bytes	With paperkey
Paper (public key)	Public key (if not on keyserver)	With paperkey
USB drive #1	Full GNUPGHOME (LUKS encrypted)	Home safe
USB drive #2	Full GNUPGHOME (LUKS encrypted)	Offsite (deposit box, trusted person)
Paper (revocation cert)	Pre-signed revocation	**Different** location from keys
Paper (passphrase)	Diceware passphrase	**Different** location from keys

With these backups verified, you are ready to proceed to Part IV: Hardware Provisioning to load your subkeys onto a YubiKey, or to Part V: Daily Machine Setup if you are using software keys (Track A).

5.7 3.6 Deterministic Keys (gpg-hd)

For the curious. This is NOT a recommended backup strategy.

Standard GPG key generation uses random entropy from `/dev/urandom`. If you lose the key material, it is gone -- there is no way to regenerate it. Every backup method in this guide (paperkey, QR codes, encrypted USB drives) exists to protect against that loss.

Deterministic key generation takes a different approach: derive the key material from a **seed phrase** (like BIP-39 mnemonics used in cryptocurrency wallets). Given the same seed, you get the same key every time -- no backup media required.

How gpg-hd Works

gpg-hd is a Python tool that takes a BIP-39 seed phrase and deterministically generates a complete GPG keychain: one master key and three subkeys (signing, encryption, authentication). It can optionally write the subkeys directly to a YubiKey with `--card`.

```
$ ./gpg-hd --name="Alice Example" --email="alice@example.com" \
    "fetch december jazz hood pact owner cloth apart impact then person actual"
```

The tool uses the seed to derive key material through a reproducible process. By default it sets the key creation timestamp to the Unix epoch (1970-01-01) as a signal that the key is deterministic, and sets a two-year expiration.

Why This Is Experimental

Not mainstream -- use at your own risk

Deterministic key generation carries serious risks that standard backup methods do not:

- **No independent audit.** gpg-hd has not undergone a formal security review. If its derivation algorithm has a subtle bug, your "backup" is actually a different key -- and you will only discover this when you need it most.
- **Seed phrase security.** Your entire cryptographic identity reduces to a memorized phrase. If someone learns the phrase, they have your key. If you forget or mis-remember a single word, the key is unrecoverable.
- **Algorithm lock-in.** If the tool's derivation logic changes between versions (or the project is abandoned), you must keep the exact version that generated your key. A dependency update could silently alter output.
- **Non-standard.** No OpenPGP specification covers deterministic key generation. Other tools cannot reproduce gpg-hd's output.

Comparison with Standard Backup Methods

FACTOR	DETERMINISTIC (GPG-HD)	PAPERKEY / USB BACKUP
Recovery needs	Seed phrase + exact tool version	Backup media + public key
Audit status	Unaudited	Paperkey is widely reviewed
Single point of failure	Seed phrase	Backup copies (redundancy)
Portability	Memorizable (in theory)	Physical media
Community adoption	Niche / experimental	Standard practice

The Bottom Line

If the idea of a memorizable key backup appeals to you, gpg-hd is worth understanding -- but it is not a substitute for the proven methods in this chapter. The safe approach is to generate keys normally and back them up with paperkey, QR codes, and encrypted USB drives.

If you do experiment with gpg-hd, treat it as an **additional** recovery path, not your only one. And always verify that re-running the tool with your seed phrase produces the same key fingerprint before relying on it.

6. Appendices

6.1 Appendix C: Cheat Sheet

The commands you actually use day-to-day, in quick-reference format. Both GnuPG (`gpg`) and Sequoia (`sq`) are shown where applicable.

Key Management

OPERATION	GNUPG	SEQUOIA
List public keys	`gpg --list-keys`	`sq cert list`
List secret keys	`gpg --list-secret-keys`	`sq key list`
Show fingerprint	`gpg --fingerprint <FPR>`	`sq inspect --cert <FPR>`
Import a key	`gpg --import key.asc`	`sq cert import key.pgp`
Export public key	`gpg --armor --export <FPR>`	`sq cert export --cert <FPR>`
Delete public key	`gpg --delete-keys <FPR>`	--
Delete secret key	`gpg --delete-secret-keys <FPR>`	--
Edit key	`gpg --edit-key <FPR>`	--

Encryption & Decryption

OPERATION	GNUPG	SEQUOIA
Encrypt (signed)	`gpg -se -r <FPR> file`	`sq encrypt --for <FPR> --signer-self file`
Encrypt (unsigned)	`gpg -e -r <FPR> file`	`sq encrypt --for <FPR> --without-signature file`
Decrypt	`gpg -d file.gpg`	`sq decrypt file.pgp`
Symmetric encrypt	`gpg -c file`	--

Signing & Verification

OPERATION	GNUPG	SEQUOIA
Detached sign	`gpg --detach-sign file`	`sq sign --signer-self --signature-file file.sig file`
Clearsign	`gpg --clearsign file`	`sq sign --signer-self --cleartext file`
Verify detached	`gpg --verify file.sig file`	`sq verify --signer-file cert.pgp --signature-file file.sig file`

Keyservers

OPERATION	GNUPG	SEQUOIA
Upload key	`gpg --keyserver hkps://keys.openpgp.org --send-keys <FPR>` `gpg --keyserver hkps://keyserver.ubuntu.com --send-keys <FPR>`	`sq network keyserver publish --cert <FPR>`
Download key	`gpg --recv-keys <FPR>`	`sq network search <FPR>`
Search by email	`gpg --locate-keys user@example.org`	`sq network search user@example.org`
Refresh all keys	`gpg --refresh-keys` (use parcimonie instead!)	--

Smartcard / YubiKey

OPERATION	COMMAND
Show card status	`gpg --card-status`
Switch YubiKey	`gpg-connect-agent "scd serialno" "learn --force" /bye`
Edit card	`gpg --card-edit`
Move key to card	`gpg --edit-key <FPR>` then `keytocard`
Kill agent	`gpgconf --kill gpg-agent`
Kill scdaemon	`gpgconf --kill scdaemon`

SSH with GPG

OPERATION	COMMAND
List SSH keys	`ssh-add -L`
Export SSH public key	`gpg --export-ssh-key <FPR>`
Set GPG_TTY	`export GPG_TTY="$(tty)"`
Update terminal	`gpg-connect-agent updatestartuptty /bye`

Trust & Certification

OPERATION	GNUPG	SEQUOIA
Sign a key	`gpg --sign-key <FPR>`	`sq pki vouch add --certifier <YOUR_FPR> --cert <FPR> --email user@example.org`
Local sign	`gpg --lsign-key <FPR>`	--
Set trust	`gpg --edit-key <FPR>` then `trust`	--

SOP (for scripting)

For automation and CI/CD, prefer SOP over parsing `gpg` output.

OPERATION	SQOP
Generate key	`sqop generate-key "Name <email>" > key.pgp`
Extract cert	`sqop extract-cert < key.pgp > cert.pgp`
Sign	`sqop sign key.pgp < data > sig`
Verify	`sqop verify sig cert.pgp < data`
Encrypt	`sqop encrypt cert.pgp < data > encrypted`
Decrypt	`sqop decrypt key.pgp < encrypted > data`

Quick Recipes

Encrypt a file for someone

```
$ gpg --armor --encrypt --recipient alice@example.org secret.txt
```

Sign and encrypt an email attachment

```
$ gpg --armor --sign --encrypt --recipient alice@example.org message.txt
```

Verify a downloaded release

```
$ gpg --recv-keys <PROJECT_FPR>
$ gpg --verify release.tar.gz.asc release.tar.gz
```

Back up your secret keys

```
$ gpg --armor --export-secret-keys <YOUR_FPR> > master-backup.asc
$ gpg --armor --export <YOUR_FPR> > public-key.asc
$ gpg --export-ownertrust > trustdb.txt
```

Extend subkey expiration

```
$ gpg --edit-key <YOUR_FPR>
gpg> key 1    # select subkey
gpg> expire
gpg> key 1    # deselect
gpg> key 2    # select next
gpg> expire
gpg> save
$ gpg --keyserver hkps://keys.openpgp.org --send-keys <YOUR_FPR>
$ gpg --keyserver hkps://keyserver.ubuntu.com --send-keys <YOUR_FPR>
```

7. End of Free Sample

You've reached the end of this free sample. Thanks for reading!

If you found Parts I–III useful, the full guide picks up exactly where this sample leaves off — starting with loading your keys onto a YubiKey and configuring your daily machine.

7.1 What's in the Full Guide

PART	TOPIC	WHAT YOU'LL LEARN
IV	Hardware Provisioning	YubiKey setup, touch policies, backup tokens, Nitrokey
V	Daily Machine Setup	GnuPG configuration, cross-platform notes, WSL2
VI	SSH Authentication	GPG-agent, FIDO2 resident keys, PIV — three paths compared
VII	Git Signing	GPG signing, SSH signing, GitHub/GitLab/Codeberg, CI/CD
VIII	Email Encryption	Thunderbird, Mutt/NeoMutt, Autocrypt, ProtonMail
IX	Password Management	pass, gopass, passage
X	File Encryption	GPG file encryption, age, encrypted backups
XI	Secrets Management	SOPS, git-crypt, comparison matrix
XII	Key Distribution	Keyservers, WKD, Keyoxide, privacy-preserving refresh
XIII	Web of Trust	caff, keysigning parties, the Debian Developer path
XIV	Package Signing	Debian/Ubuntu, RPM, software releases, container images
XV	Maintenance & Rotation	Expiry renewal, YubiKey switching, emergency procedures
XVI	Complementary Tools	Sequoia, age, SOP, when NOT to use PGP

Plus 5 more appendices: Configuration Reference · Troubleshooting · Glossary · Migration Guides · Legal & Compliance

Get the full guide at: https://leanpub.com/gpg-guide