StegoCrypt

Overview

This app hides encrypted secrets (text or any file) inside an image or video so the secret is invisible to casual inspection. It has a small GUI with two primary modes:

- **Embed** — take a message or a file, encrypt it with a password, and hide it inside a chosen image or video file; produces a new stego file (e.g. movie_stego.mkv or image_stego.png).
- **Extract** — read a stego file, provide the same password, and recover the hidden data. If it's text you get a popup (not saved); if it's binary you're prompted to save the file.

Key design goals

- **Stealth**: make modifications visually and statistically hard to notice.
- **Secrecy**: even if someone extracts the raw embedded bytes, they cannot get the secret without the password (strong encryption).
- **Practicality**: works on common image formats and video files, and produces lossless outputs that preserve embedded bits.

Modes & options (what they mean)

- **LSB (Least Significant Bit)**
  We write payload bits into the least significant bit(s) of pixel byte values. Default is **1 LSB per color channel** (B,G,R) which is the least noticeable visually.
- **Spread (pseudo-random spreading / stealth)**
  After a tiny sequential header and salt, the remaining payload bits are scattered pseudo-randomly across pixels/frames using a deterministic PRNG whose seed is derived from the password+salt. This removes obvious clusters and makes detection harder.
- **ECC (Reed–Solomon)** — *optional*
  If enabled, an outer Reed–Solomon code adds parity so small amounts of corruption can be corrected. Tradeoff: larger payload (bigger stego file) and potentially less stealth. Useful only when you expect lossy transmission/encodings; unnecessary if you can preserve a lossless container.

- **Lossless codec choices**
    - **h264rgb (libx264rgb, lossless)** — lossless RGB H.264. Much smaller files than FFV1 but still preserves exact decoded RGB pixels (recommended balance: smaller files + reliable extraction).
    - **FFV1 (Matroska)** — truly lossless and robust, but often produces much larger files. Good when absolute maximum fidelity is desired and size/space isn't a concern.
    - Fallback: write frames with OpenCV (may use MJPEG if ffmpeg not available) — less ideal.
- **Streaming / chunked video mode**
  To avoid loading an entire video into RAM, embedding/extraction operates in chunks of frames. Each chunk uses a per-chunk deterministic seed so permutation is reproducible without full-memory state.

How the hidden data is packaged (high level)

1. **Encryption & payload format**
    - The secret (text or file bytes) is encrypted with AES-GCM using a key derived from the password via PBKDF2 (strong KDF).
    - The encrypted payload layout is: salt || nonce || ciphertext || tag.
    - A small header MAGIC || payload_length (fixed-size) is prepended so the extractor recognizes the stego format and knows how many bytes to read.
2. **Header & salt placement**
    - The header and the salt are embedded **sequentially** at the start of the embedding stream (so the extractor can recover salt and reconstruct the RNG seed).
3. **Payload embedding**
    - The remainder (nonce + ciphertext + tag, or RS-encoded version if ECC is enabled) is turned into bits and those bits are embedded into pixel LSBs.
    - Bits are either written sequentially or — for stealth — **spread pseudo-randomly** across pixels and frames using a deterministic permutation seeded by password +

salt. During extraction the same seed reproduces the permutation so bits are recovered in the correct order.

4. **Image vs Video**
    - **Image**: single-frame; treat pixel bytes as a 1D slot array and embed.
    - **Video**: frames are a 3D stream (frame, y, x, channel). We map global slot indices across frames and may process frames in chunks. For lossless output, we either write frames to PNGs and assemble with ffmpeg (into h264rgb or ffv1), or use a lossless-capable writer if available.

Extraction behavior

- The extractor reads the first header + salt sequentially, uses the salt + password to derive the PRNG seed and decryption key, then reconstructs the permutation to read the rest of the bits.
- After reassembling the encrypted payload bytes, the app attempts AES-GCM decryption. If the tag verification fails you get an error (wrong password or corrupted data).
- If extraction yields UTF-8 text, the app displays it in a popup (not automatically saved). If it's binary, the app prompts you to save it to disk.

Capacity note (rough)

Capacity (bits) ≈ frames * width * height * channels * LSBs_per_channel.
Example: 320×240 @100 frames, 3 channels, 1 LSB → ~23 million bits ≈ 2.9 MB. Use 1 LSB for maximum stealth.

Practical limitations & recommendations

- **Lossy re-encoding (e.g., typical H.264 MP4, social media re-uploads)** will almost always corrupt LSB steganography — unless you use strong ECC (larger payload) and hope some redundancy survives. Best practice: use a lossless final container (h264rgb/FFV1) for transport between sender and receiver.
- **Keep the password secret** — without it extraction yields only ciphertext or garbage.

- **Do not rely on ECC to survive heavy recompression** — it helps with some bit flips, but heavy lossy transforms will still probably defeat recovery.
- **Steganalysis**: LSB techniques are still subject to statistical steganalysis if someone knows what to look for. PRNG spreading + 1 LSB/channel minimizes detectability but doesn't make you invisible to an advanced forensic analyst.

Usage (high level)

```
python stego_cli.py embed  --in INPUT          --out OUTPUT   --password PWD
[options]

python stego_cli.py extract --in STEGO_FILE    --password PWD  [options]
```

Common options

- --in / -i <path>
  Input file (image or video). For embed: the cover image/video. For extract: the stego file.
- --out / -o <path>
  Output file. Required in **embed** mode (where a new stego file is created). Not required in extract mode (extraction prints text or prompts to save binary).
- --password / -p <password>
  Password used to derive the encryption key. **Required** for embed and extract.
- --message / -m "<text>"
  Inline message to embed. If omitted in embed mode, CLI will read bytes from a file specified with --embed-file.
- --embed-file / -f <path>
  Embed this file's raw bytes. Used only in embed mode and mutually exclusive with --message.
- --lsb <1|2|3>
  Number of least-significant bits per channel to use. Default **1** (recommended for stealth).
- --no-spread
  Disable pseudo-random spreading. Default is to **spread**. (Use only for testing.)

- --codec <h264rgb|ffv1>
  Lossless codec to use for video output. Default recommended: h264rgb. (Requires ffmpeg.)
- --use-ecc
  Enable Reed–Solomon ECC (optional). Increases payload size. Requires reedsolo package.
- --rs-nsym <N>
  ECC parity byte count (only if --use-ecc is set). Default common value: 32.
- --chunk-frames <N>
  Number of frames per streaming chunk (video embed/extract). Default: 90 (tunable for memory vs speed).
- --verbose
  Print progress and diagnostic messages to stdout/stderr.
- --help / -h
  Show help text.

Examples

1) Embed a short text message into a video (recommended defaults)

```
python stego_cli.py embed \
  --in clip.mp4 \
  --out clip_stego.mkv \
  --password "My$ecretPwd" \
  --message "Meet at 10" \
  --codec h264rgb \
  --lsb 1
```

- Embeds the text into clip_stego.mkv using 1 LSB per channel and libx264rgb lossless assembly.

2) Embed a binary file (PDF) into an image

```
python stego_cli.py embed \
  --in cover.png \
```

```
  --out cover_stego.png \
  --password hunter2 \
  --embed-file secret.pdf \
  --lsb 1
```

- Hides secret.pdf in cover_stego.png. Output is a PNG (single-frame) stego image.

3) Extract the secret (auto-detects text vs binary)

```
python stego_cli.py extract \
  --in clip_stego.mkv \
  --password "My$ecretPwd"
```

- If the payload is UTF-8 text it prints to stdout (or opens a popup in the GUI). If binary, the CLI will prompt (or accept --out) to write to disk:

python stego_cli.py extract --in cover_stego.png --password hunter2 --out recovered.pdf

4) Embed with ECC (if you expect slight corruption during transport)

```
python stego_cli.py embed \
  --in clip.mp4 --out clip_stego.mkv \
  --password "My$ecretPwd" \
  --embed-file large.zip \
  --use-ecc --rs-nsym 64 \
  --codec h264rgb
```

- Adds Reed–Solomon parity bytes (64 bytes) so the extractor can correct some bit flips. Tradeoff: larger payload & stego file.

5) Low-memory streaming mode for large videos (default)

The CLI uses streaming/chunked embedding by default for videos — you can adjust:

```
python stego_cli.py embed --in long.mp4 --out long_stego.mkv --password pass --embed-file big.bin --chunk-frames 30
```

Notes, best practices & constraints

- **Password**: keep it secret. Without the password decryption fails with a MAC check error.

- **LSB = 1** is the stealthiest setting. Increasing LSBs increases capacity but makes alterations more detectable.
- **Use h264rgb** for smaller lossless outputs if libx264rgb is present in your ffmpeg build (recommended). ffv1 is still supported and sometimes slightly faster but tends to produce larger files.
- **ECC**: only use when you expect lossy transport or corrupting transformations. Otherwise skip it to keep the payload small.
- **Extraction**: CLI extraction will not save the recovered secret unless --out is provided for binary payloads — text is printed to stdout.
- **ffmpeg**: required for assembling frames into lossless video formats. Ensure ffmpeg is in your PATH.