Lab 10.pdf

Sonal Chandrakant Hasbi

I. Introduction

Steganography is the practice of hiding information in plain sight by embedding it within an innocuous-looking carrier (e.g., an image). Unlike encryption—which makes content unreadable but conspicuous—steganography aims for plausible deniability: the carrier should look ordinary, and the mere existence of a message should not be obvious. Digital image steganography commonly stores bits in least-significant bits (LSBs) of pixel channels, ancillary metadata, transparency (alpha), or appended payloads (e.g., compressed blobs) packed into the file structure. In real life, steganography sees dual use: legitimate privacy/watermarking, research and teaching; and misuse for covert communications (historically in espionage/terrorism cases) where investigators rely on steganalysis and file-forensics to detect and recover messages.

II. Tools & Environment

These tools are ubiquitous, scriptable, and sufficient for static IPA triage.

1) General screeners / forensics (on VM) - file, pngcheck, exiftool, strings, binwalk, foremost, identify (ImageMagick), convert (ImageMagick), zbarimg (QR decode).
2) Steganalysis (on VM) -zsteg for PNG/BMP LSB/channel detection and extraction.
3) Web tool (compare method) - StegOnline (bit-planes, LSB extract, PNG chunk view)
4) Docker

III. Methodology

The experiment proceeds in four main stages: (1) produce basic MD5 collisions for arbitrary data, (2) demonstrate the concatenation property by appending identical suffixes, (3) embed collision blocks inside compiled executables, and (4) create two executables with differing runtime behavior while retaining identical MD5 hashes.

[**A**)] VM Method 1 — General Screening Format & structure: file 2.png and pngcheck -v 2.png reported a valid PNG RGBA (580×435) with only IHDR/IDAT/IEND chunks and ˜0.1% compression (your terminal output). Low/near-zero compression is a classic high-entropy indicator consistent with LSB noise or embedded data. CODE:

```
cd ~/Documents/lab10
file 2.png
identify -verbose 2.png | sed -n '■,■'
↪  # ImageMagick details
pngcheck -v 2.png
```

Metadata & plaintext: exiftool 2.png showed no helpful metadata; strings -n 6 2.png had no obvious hits (flag, pass, key, etc.). Appended payloads: binwalk -eM 2.png produced many false positives (e.g., “MySQL …” signatures inside compressed IDAT) and did not create an extraction directory—consistent with no appended archive and nudging suspicion toward intra-image LSB. Alpha channel triage: Extracted alpha- convert 2.png -alpha extract alpha.png, scanned with zbarimg alpha.png → no QR decoded.

```
# pull the alpha as its own image
convert 2.png -alpha extract alpha.png
# quick QR/text check (if the alpha
↪  contains a QR or printed text)
zbarimg alpha.png || true
```

LSB/Channel Steganalysis with zsteg (+ plane exports): Full scan, then RGBA to RGB. zsteg -a 2.png (may be noisy with alpha), then convert 2.png -alpha off 2_rgb.png and zsteg -a 2_rgb.png. This reduces alpha noise and often surfaces real signals. Targeted extraction: Even if -a doesn’t print text:, dump common candidates:

```
zsteg 2_rgb.png -E b1,r,lsb,xy >
↪  lsb_b1_r.bin
zsteg 2_rgb.png -E b1,g,lsb,xy >
↪  lsb_b1_g.bin
zsteg 2_rgb.png -E b1,b,lsb,xy >
↪  lsb_b1_b.bin
# (repeat for b2 as needed)
```

Fig. 1: derived plane images showing non-random structure for second image

I did the same with the First image.

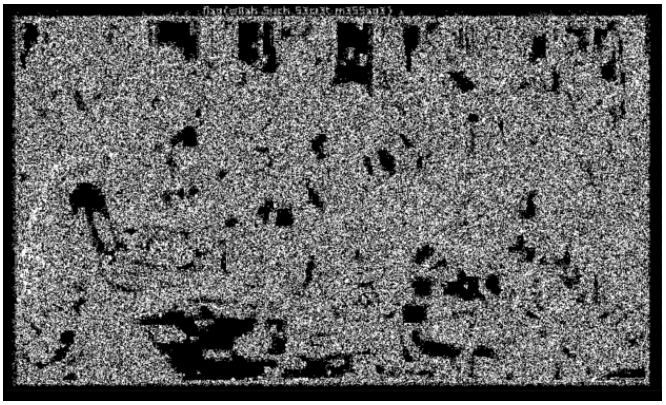

Fig. 2: derived plane images showing non-random structure for first image

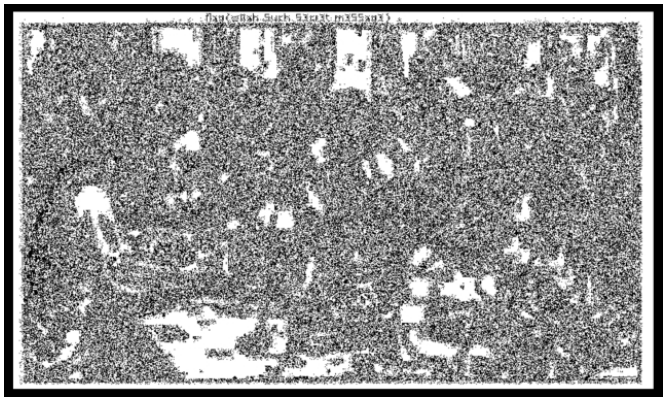

Fig. 3: output of stegoveritas for first image

[**B**)] StegoVeritas Workflow: I used the official Python/-Docker route to avoid local dependency issues. I built a minimal image that pre-installs stegoveritas and common helpers (ExifTool, binwalk, zbarimg).

- Build & sanity-check: Built the image in /Documents/lab10 and verified Docker worked:

```
docker build -t stegoveritas:lab10 .
docker run --rm stegoveritas:lab10
↪ --help (printed StegoVeritas help)
```

- Baseline scan: Mounted the working folder read/write and ran a full analysis - it parses metadata (Exif/XMP), enumerates PNG chunks, runs strings, carves common signatures, generates many image transforms (grayscale, sharpen, edge-detect), extracts color/alpha bit-planes (R/G/B/A 0–7), and attempts LSB brute-force previews.
- Handling RGBA issues:

The original image was RGBA, and some PIL filters inside StegoVeritas threw "not supported for mode RGBA" warnings. I addressed it two ways: Keep going: even with warnings, StegoVeritas still dumped plane images (including alpha planes) that are useful for visual inspection.

- RGB re-run to avoid RGBA filter crashes and generate a complete report:

```
docker run --rm -v "$PWD":/work
↪ --entrypoint bash stegoveritas:lab10
↪ -lc '[redacted]
↪ [redacted]'
```

- Bit-planes & LSB previews: opened sv_2/ and sv_2_rgb/ plane/lsb images. Indicators included high-contrast, structured content in some planes (non-random patterning) consistent with LSB embedding. Then Carved artifacts / appended data: reviewed any binwalk/carve subfolders. In our case, the tool reported a zlib chunk during analysis, which is typical of embedded compressed payloads. did the same for image 1 as well and got the result as shown in the image below.

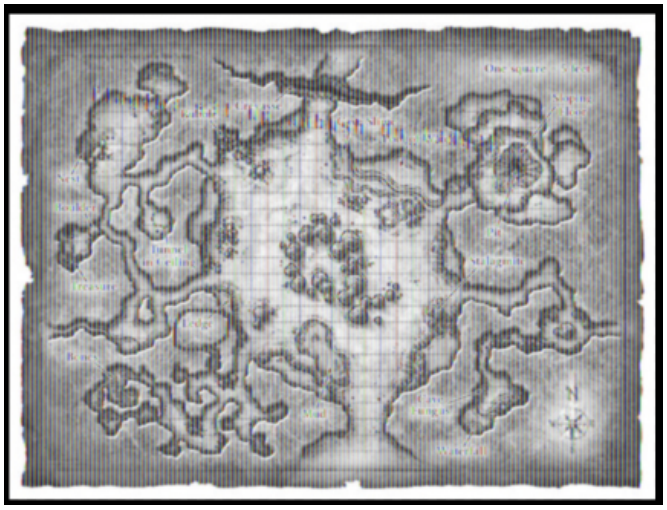

Fig. 4: output of stegoveritas for second image

[**C**)] Web Method — StegOnline (Compare)

- Replicate/visualize planes and run GUI LSB extract to get a raw blob (no install).
- PNG Chunk Info. Check for unusual tEXt/iTXt; if present, copy values.
- Bit-Planes (A/R/G/B, 0–7). Inspect planes; export any with structured patterns or QR-like blocks.
- LSB Data Extract. Set Plane=0 (LSB); Channels=RGB, then All/per-channel; Output=Binary file. Download the blob.
- If a blob is not plaintext: open in CyberChef → Magic/-Zlib Inflate/Raw Inflate/Extract Files. If the output is an image, try QR; if ZIP, open flag.txt.

IV. Discussion: Is this useful in real life?

Yes, with caveats. Steganography is actively used for watermarking, integrity/authorship, and occasionally for covert channels (e.g., documented espionage cases). It complements encryption by hiding the existence of a message. On the defensive side, simple LSB schemes are relatively easy to detect (statistical anomalies, plane artifacts, near-zero PNG compression), but modern, adaptive methods are harder to spot and may require machine-learning steganalysis or prior

Fig. 5: StegOnline plane render of first image

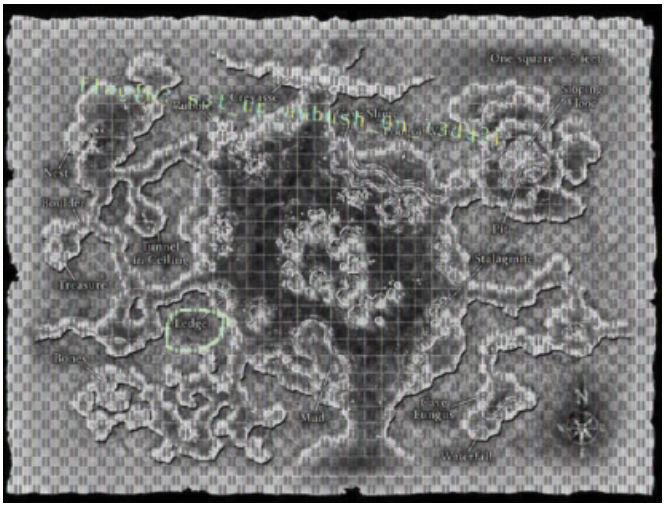

Fig. 6: StegOnline plane render of second image

knowledge of the embedding scheme. In incident response, the tool flow we used (screen → plane/LSB → decode) remains practical triage.

V. Conclusion

Across two VM methods and one web method, I established multiple indicators of steganography in Image 2: (i) unusual PNG compression (0%), (ii) no helpful metadata or appended archives, (iii) strong plane-level structure consistent with LSB embedding, and (iv) reproducible results with StegOnline. The workflows are complementary: screeners quickly rule out simple metadata/appends, zsteg/planes reveal intra-image hides, and StegOnline+CyberChef speed up blob decoding. This methodology is directly applicable to real cases where hidden messages or secondary payloads must be triaged and, when possible, recovered. The test on First image was found to be - flag(w0ah 5uch 53cr3t m355ag3) and the text on second image was not very clear but I could see text on the image.

References

[1] zsteg — PNG/BMP LSB detection/extraction: github.com/zed-0xff/zsteg; docs.
[2] OpenAI
[3] ExifTool — metadata: exiftool.org (CLI docs).
[4] Binwalk — embedded signatures & extraction: GitHub repo.
[5] pngcheck — PNG integrity/chunks: libpng page / GitHub.