VigilantEye - CNN Model & GIF Embedding Documentation

Generated: 2026-01-03

1) Executive Summary

VigilantEye analyzes GIF files using a hybrid approach:
1) A lightweight 3D-CNN that looks for learned visual/temporal artifacts across frames.
2) A rules/signature extractor that looks for suspicious embedded payload patterns.
3) (Optional) LLM-based analysis for higher-level reasoning and reporting.
This document explains how the CNN is structured and how the embedding demo hides payload data inside GIFs.

2) CNN Model Architecture (MultiHeadCNN)

Source of truth: test_model.py

The model is a 3D-CNN (Conv3d) over a fixed-length clip of GIF frames. It uses one shared convolutional backbone and two output heads:
- Head A: binary classification (clean vs infected)
- Head B: embedding method classification (which embedding technique is most likely present)

2.1 Input Preprocessing

- GIF is decoded into frames; up to FRAMES=25 frames are used.
- Each frame is converted to RGB, resized to IMAGE_SIZE=(100, 100), and converted to a tensor.
- If the GIF has fewer than 25 frames, the last frame is repeated to reach 25.
- Final tensor shape (per sample): [C=3, T=25, H=100, W=100].
- Device: CUDA if available, otherwise CPU.

2.2 Layer-by-Layer Structure (as implemented)

The backbone is:

```
Conv3d(3 -> 32, kernel=3, padding=1)
BatchNorm3d(32)
ReLU
MaxPool3d(kernel=(1,2,2))
Conv3d(32 -> 64, kernel=3, padding=1)
BatchNorm3d(64)
ReLU
MaxPool3d(kernel=(1,2,2))
Flatten
Linear(64 * 25 * 25 * 25 -> 256)
Head(class): Linear(256 -> 2)
Head(method): Linear(256 -> N_methods)
```

2.3 Tensor Shapes (conceptual)

```
Input:        [B,  3, 25, 100, 100]
Conv3d:       [B, 32, 25, 100, 100]
Pool (1,2,2): [B, 32, 25,  50,  50]
Conv3d:       [B, 64, 25,  50,  50]
Pool (1,2,2): [B, 64, 25,  25,  25]
Flatten:      [B, 1,000,000]
Shared FC:    [B, 256]
Class head:   [B, 2]
Method head:  [B, N_methods]
```

2.4 How the CNN Detects Malware (high-level)

The CNN does not execute the GIF and does not run any embedded code. Instead, it learns statistical/visual/temporal cues from training data.

Examples of cues the model may learn (depending on the dataset):

- Unnatural pixel-level noise patterns (common in steganography/LSB-style embeddings).
- Frame-to-frame inconsistencies in animated GIFs.
- Artifacts introduced by re-encoding/optimization after tampering.
- Distribution shifts correlated with specific embedding techniques.

Inference logic (simplified):
- infected/clean = argmax(class_head)
- method = argmax(method_head)

Note: The model output is a prediction, not proof. Pair it with signature/rule checks and safe file handling.

3) Embedding Methods (Demo) - What We Embed and Where It Lives

Source of truth: embedding_engine.py

The embedding demo shows how payload data can be hidden inside a GIF. Important security note: a GIF is an image format and is not an executable program. Embedded payload data remains dormant unless an external trigger extracts and executes it.

3.1 Method 1 - Append After Terminator

- What it does: writes extra bytes after the GIF terminator.
- Where it lives: after the end-of-file marker (the viewer typically ignores it).
- Why it can work: many decoders stop parsing at the terminator and ignore trailing bytes.
- Detection: check for data after terminator / abnormal size / trailing readable strings.

3.2 Method 2 - GIF Comment Extension

- What it does: inserts a Comment Extension block into the GIF structure.
- Where it lives: a standard extension area intended for comments/metadata.
- Why it can work: comments are valid per spec and usually not displayed.
- Detection: parse extension blocks and inspect comment contents for suspicious patterns.

3.3 Method 3 - Base64 Encoded Append

- What it does: Base64-encodes the payload text and appends it after the GIF terminator with a marker.
- Where it lives: trailing bytes, but text looks less obvious due to Base64 encoding.
- Detection: search for markers (e.g., base64:) and attempt safe decode + inspection.

3.4 Method 4 - LSB Steganography

- What it does: hides payload bits in the least-significant bits of pixel channels across frames.
- Where it lives: distributed throughout pixel data (harder to spot by casual inspection).
- Trade-offs: stealthy but limited by image capacity and requires careful extraction logic.
- Detection: statistical steganalysis, LSB plane analysis, anomaly tests (e.g., chi-square).

4) How the App Explains Embedding (Token-Efficient)

To avoid wasting tokens, the app generates a short, structured explanation with exactly 4 lines:
Embedding / Placement / Detection / Risk.

This keeps outputs consistent, professional, and small while still being useful.

5) Operational Notes / Safety

- Do not execute any extracted payload. Treat it as untrusted data.
- Use least privilege for file processing services and isolate uploads.
- The demo is educational; in production, implement stricter validation and sandboxing.