Invisible Message Transmission through Image Steganography

Ilyass Naji Abderahmane Chakir

April 17, 2025

What is Steganography?

Definition

Steganography is the art and science of hiding a message within another medium so that its existence is concealed. In digital image steganography, a secret message is embedded into an image (the cover image) without causing noticeable changes.

Overview of the Photocrypt Algorithm

- Convert a secret key to a SHA1 hash (160 bits).
- Embed the SHA1 digest into the first row of the cover image.
- Convert the secret message to a bit stream (UTF-8) and append an 8-bit termination marker.
- Embed the message bits starting from the second row.
- Use key-dependent bit positions for embedding (choosing LSB or second LSB).
- During decryption, verify the key via the embedded hash before extracting the message.

Utility Functions: set_bit and get_bit

set_bit

```
mask = 0xFF ^ (1 << bit_pos) return
(value & mask)| (bit << bit_pos)
```

get_bit

```
return (value >> bit_pos)& 1
```

- **set_bit:** Clears the target bit in an 8-bit value then sets it to the desired value.
- **get_bit:** Shifts the value right by bit_pos and masks with 1 to isolate the bit.

Encryption Process

- **Load Cover Image:** Convert it to RGB and obtain a NumPy array.
- **Key Preparation:** Convert the key to bytes and compute its SHA1 hash.
- **Hash Embedding:** Embed each of the 160 hash bits into pixel channels of the first row. The key byte modulo 2 determines the bit position (LSB or second LSB).
- **Message Conversion:** Convert the message to a bit stream (8 bits per character) and append eight zero bits as a termination marker.
- **Message Embedding:** Embed the message bits into subsequent rows using key-dependent bit positions.

Decryption Process

- **Load Stego Image:** Convert it back to RGB and build a NumPy array.
- **Extract Hash:** Retrieve the 160 embedded bits from the first row using the same key mechanism.
- **Key Verification:** Compare the extracted SHA1 hash with the computed hash from the provided key.
- **Message Extraction:** Starting from the second row, extract bits until the termination marker (eight consecutive zeros) is reached.
- **Message Reconstruction:** Group the extracted bits into bytes and decode them to retrieve the original message.

Encryption: Preprocessing & Hash Embedding (Part 1)

```python
def encrypt(cover_image_path, message, key, stego_image_path):
    # Load the cover image
    cover_img = Image.open(cover_image_path)
    if cover_img.mode != 'RGB':
        cover_img = cover_img.convert('RGB')
    img_array = np.array(cover_img, dtype=np.uint8)
    height, width, _ = img_array.shape
    if width < 54:
        raise ValueError("Cover image width must be at least 54
            pixels")

    # Convert key to bytes
    key_bytes = key.encode('utf-8')
    key_length = len(key_bytes)
    if key_length == 0:
        raise ValueError("Key cannot be empty")

    # Compute SHA1 hash of the key (160 bits)
    sha1 = hashlib.sha1(key_bytes).digest()
    hash_bits = []
    for byte in sha1:
        for i in range(7, -1, -1):
            hash_bits.append((byte >> i) & 1)
    hash_bits = hash_bits[:160]
```

Encryption: Hash Embedding (Part 2)

```
    # Embed hash into the first row
    i_component = 0  # Component index (0 to 3*width-1)
    j_row = 0        # First row
    key_index = 0
    for k in range(160):
        key_byte = key_bytes[key_index % key_length]
        b = key_byte % 2
        pixel_x = i_component // 3
        channel = i_component % 3
        if pixel_x >= width:
            break
        current_val = img_array[j_row, pixel_x, channel]
        new_val = set_bit(current_val, b, hash_bits[k])
        img_array[j_row, pixel_x, channel] = new_val
        i_component += 1
        key_index += 1

    # Convert message to UTF-8 bytes and append eight zeros
    message_bytes = message.encode('utf-8')
    message_bits = []
    for byte in message_bytes:
        for i in range(7, -1, -1):
            message_bits.append((byte >> i) & 1)
    for _ in range(8):
```

Encryption: Message Embedding (Part 1)

```
    # Embed message starting from the second row
    i_component = 0
    j_row = 1
    key_index = 0
    message_index = 0
    total_bits = len(message_bits)
    while message_index < total_bits:
        if j_row >= height:
            raise ValueError("Insufficient space to embed
                message")
        key_byte = key_bytes[key_index % key_length]
        b = key_byte % 2
        pixel_x = i_component // 3
        channel = i_component % 3
        if pixel_x >= width:
            i_component = 0
            j_row += 1
            continue
        current_val = img_array[j_row, pixel_x, channel]
        new_val = set_bit(current_val, b,
            message_bits[message_index])
        img_array[j_row, pixel_x, channel] = new_val
        message_index += 1
        key_index += 1
```

Encryption: Message Embedding (Part 2) & Save

```
        if i_component >= width * 3:
            i_component = 0
            j_row += 1

    # Save the stego image
    stego_img = Image.fromarray(img_array)
    stego_img.save(stego_image_path)
```

Decryption: Loading & Key Preparation

```python
def decrypt(stego_image_path, key):
    # Load the stego image
    stego_img = Image.open(stego_image_path)
    if stego_img.mode != 'RGB':
        stego_img = stego_img.convert('RGB')
    img_array = np.array(stego_img, dtype=np.uint8)
    height, width, _ = img_array.shape

    # Convert key to bytes
    key_bytes = key.encode('utf-8')
    key_length = len(key_bytes)
    if key_length == 0:
        raise ValueError("Key cannot be empty")
```

Decryption: Hash Extraction

```
    # Extract hash from the first row
    extracted_hash_bits = []
    i_component = 0
    j_row = 0
    key_index = 0
    for _ in range(160):
        key_byte = key_bytes[key_index % key_length]
        b = key_byte % 2
        pixel_x = i_component // 3
        channel = i_component % 3
        if pixel_x >= width:
            break
        current_val = img_array[j_row, pixel_x, channel]
        extracted_hash_bits.append(get_bit(current_val, b))
        i_component += 1
        key_index += 1

    # Convert hash bits to bytes
    extracted_hash = bytearray()
    for i in range(0, 160, 8):
        byte = 0
        for bit in extracted_hash_bits[i:i+8]:
            byte = (byte << 1) | bit
        extracted_hash.append(byte)
```

Decryption: Hash Validation & Message Extraction (Part 1)

```
    # Validate the hash
    expected_hash = hashlib.sha1(key_bytes).digest()
    if extracted_hash != expected_hash:
        raise ValueError("Incorrect key or tampered image")

    # Extract message bits
    message_bits = []
    i_component = 0
    j_row = 1
    key_index = 0
    zero_count = 0
    while j_row < height:
        key_byte = key_bytes[key_index % key_length]
        b = key_byte % 2
        pixel_x = i_component // 3
        channel = i_component % 3
        if pixel_x >= width:
            i_component = 0
            j_row += 1
            continue
        current_val = img_array[j_row, pixel_x, channel]
        bit = get_bit(current_val, b)
```

Decryption: Message Extraction (Part 2)

```
            if bit == 0:
                zero_count += 1
                if zero_count == 8:
                    message_bits = message_bits[:-8]  # Remove
                        termination marker
                    break
            else:
                zero_count = 0
            key_index += 1
            i_component += 1
            if i_component >= width * 3:
                i_component = 0
                j_row += 1

    # Convert bits to message (UTF-8)
    message_bytes = bytearray()
    for i in range(0, len(message_bits), 8):
        bits = message_bits[i:i+8]
        if len(bits) < 8:
            break
        byte = 0
        for bit in bits:
            byte = (byte << 1) | bit
        message_bytes.append(byte)
```

General Model of the Photocrypt Algorithm

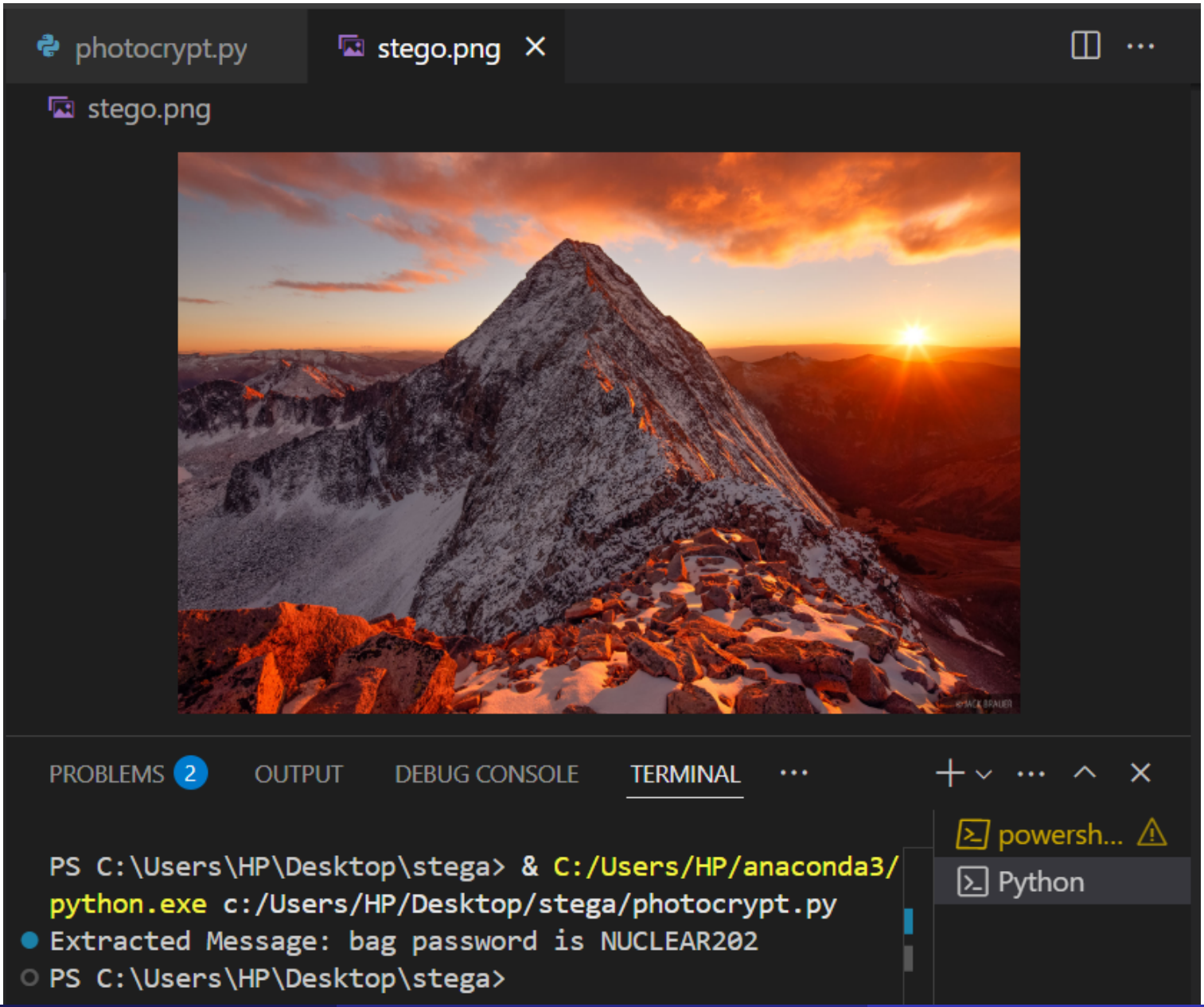
- **Input:**
  - Cover Image (24-bit, minimum 54 pixels wide)
  - Secret Message (text)
  - Secret Key (password)
- **Preprocessing:**
  - Convert the secret key to UTF-8 bytes.
  - Compute SHA1 hash (160 bits) of the key.
  - Convert the message to a bit stream and append an 8-bit termination marker.
- **Embedding Process:**
  - Embed the SHA1 hash bits in the first row using key-dependent bit positions.
  - Embed the message bits in subsequent rows using the same scheme.
- **Decryption Process:**
  - Verify the key by extracting and comparing the embedded SHA1 hash.
  - Extract the message bits until the termination marker is reached.
  - Reconstruct the original message from the bit stream.

LSB vs. Photocrypt Algorithm

- **Classical LSB:**
  - Always replaces the least significant bit of each pixel component.
  - Uses a constant pattern that is predictable.
  - Vulnerable if the embedding method is known.
- **Photocrypt Algorithm (Proposed):**
  - Uses a secret key to compute a SHA1 hash.
  - Embeds the hash in the first row for key verification.
  - Selects the bit position based on the key (choosing LSB or second LSB).
  - Introduces variability in embedding, making detection and extraction more difficult.

Cover Image

Resulting Stego Image



Conclusion

- The Photocrypt algorithm secures image steganography by embedding a SHA1 key hash along with the message.
- Key-dependent bit selection (LSB or second LSB) adds variability and strengthens security.
- The resultant stego image remains visually similar to the cover image, making the hidden message difficult to detect.

References

[1] Munikar, M. (2016). *Image Steganography: Basic Concepts and Proposed Algorithm*.
`https://www.researchgate.net/publication/334637828_Photocrypt`

[2] Morkel, T., Eloff, J., & Olivier, M. (2005). *An Overview of Image Steganography*.
In ISSA 2005 Proceedings.

[3] Cheddad, A., Condell, J., Curran, K., & McKevitt, P. (2010). *Digital Image Steganography: Survey and Analyses of Current Methods*.
Signal Processing, 90(3), 727-752.

[4] Kessler, G.C. (2001). *Steganography: Hiding Data Within Data*.
IEEE Potentials, 20(1), 31-34.