Steganographic Implementation in osu! osz Beatmap Files for Data Concealment

Jazmy Izzati Alamsyah - 18221124
Program Studi Sistem dan Teknologi Informasi
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail: 18221124@std.stei.itb.ac.id

***Abstract*— Abstract— Steganographic implementation in osu! OSZ beatmap files for covert data transmission through gaming networks is investigated. Four techniques were tested: ZIP archive manipulation, audio LSB modification, image LSB embedding, and configuration file comment injection across local testing, peer-to-peer distribution, and official submission scenarios. Audio LSB steganography emerged as the superior method, achieving 100% success across all scenarios while embedding multi-megabyte files without quality degradation. ZIP, image, and configuration methods failed official submission due to server-side processing including OSZ reconstruction, image optimization, and comment sanitization. Results establish gaming file formats as viable steganographic carriers, with audio LSB demonstrating exceptional capacity, reliability, and resistance to automated detection systems, providing foundation for gaming-based covert communication research.**

Keywords*— *steganography, data concealment, gaming files, OSZ beatmap, audio LSB, rhythm games, covert communication, multimedia steganography

I. Introduction

The exponential growth of digital gaming communities over the past decade has fundamentally transformed how multimedia content is shared and distributed across global networks. Online gaming platforms now facilitate the exchange of millions of user-generated files daily, creating vast ecosystems of digital content that remain largely unmonitored for covert communication activities. Among these platforms, rhythm games have emerged as particularly significant due to their reliance on complex multimedia beatmap files that combine audio, visual, and timing data into comprehensive interactive gaming experiences.

A beatmap, in the context of rhythm games, is a user-created level or map that synchronizes gameplay elements with a specific audio track. These files contain precise timing information that dictates when and where players must interact with the game interface, creating a choreographed experience that matches the rhythm and structure of the underlying music. Beatmaps typically include the complete audio file, background artwork, hit circle positions, slider paths, timing points, difficulty settings, and metadata about the song and creator. This multimedia integration makes beatmaps substantially larger and more complex than traditional game files, often ranging from 5 to 50 megabytes per file depending on audio quality and visual assets.

osu!, one of the world's most popular free-to-play rhythm games, has cultivated a vibrant community of over 20 million registered users who actively create, share, and download beatmap files. The game's ecosystem revolves around user-generated content, with thousands of new beatmaps uploaded daily to official repositories and community platforms. These beatmaps are distributed in the OSZ (osu! beatmap) format, which serves as a container for all necessary game assets. The OSZ format encapsulates complete gaming experiences within single compressed archives, making them potential candidates for steganographic applications due to their legitimate large file sizes, frequent distribution patterns, and complex internal structures.

The osu! community has established a sophisticated infrastructure for beatmap sharing that spans official databases, mirror sites, and peer-to-peer networks. This distributed ecosystem processes up to millions of downloads monthly, creating natural cover traffic that could theoretically mask covert communication channels. Unlike traditional file sharing platforms where large or unusual files might raise suspicion, the gaming environment normalizes the transfer of substantial multimedia archives, potentially providing an operational environment suitable for steganographic applications.

The technical complexity of beatmap files presents multiple theoretical vectors for data concealment that have not been explored in academic literature. OSZ files are fundamentally ZIP archives with custom extensions, potentially offering opportunities for header manipulation, directory structure modification, and compressed data stream injection. The multimedia content within these archives may provide additional hiding locations through established techniques such as least significant bit modification in audio data, spatial domain manipulation in background images, and metadata field injection in configuration files. Furthermore, the gaming-specific timing data and difficulty parameters present novel concealment possibilities unique to rhythm game formats.

Current steganographic research has primarily focused on conventional multimedia formats such as JPEG images, MP3 audio files, and AVI videos. While these formats have proven effective for academic demonstrations, they face increasing challenges from sophisticated detection systems that employ statistical analysis, machine learning algorithms, and structural examination techniques. Gaming file formats, particularly

beatmaps, operate within a fundamentally different context where complex structures, varied compression patterns, and multimedia integration are expected characteristics rather than suspicious anomalies.

This research investigates the feasibility of steganographic implementation in osu! OSZ beatmap files, addressing a significant gap in current literature regarding gaming file formats. The study aims to determine whether practical data concealment techniques can be successfully implemented within gaming archives while maintaining compatibility with existing gaming infrastructure and preserving file functionality. Through experimental implementation and testing, this work seeks to evaluate the viability of beatmap files as steganographic carriers and assess their potential effectiveness compared to conventional hiding methods.

II. METHODOLOGY

The experimental framework for this research involves implementing and evaluating steganographic techniques specifically designed for osu! OSZ beatmap files. The methodology encompasses three primary phases: OSZ file structure analysis, steganographic algorithm implementation, and resistance evaluation against common file operations.

A. OSZ File Structure Analysis

The first stage involves comprehensive analysis of OSZ file architecture to identify potential hiding locations. OSZ files are fundamentally ZIP archives containing multiple components: audio files that include hitsounds and the main song (typically .mp3, .ogg, and .wav formats), background images (.jpg or .png), beatmap configuration files (.osu), an optional storyboard file (.osb), and metadata. Each component presents different opportunities for data concealment that must be systematically evaluated.

OSZ files utilize the MIME type x-osu-beatmap-archive and serve as complete beatmap packages within the osu! ecosystem. The format specification follows standard ZIP archive structure while maintaining specific organizational patterns required for game functionality.

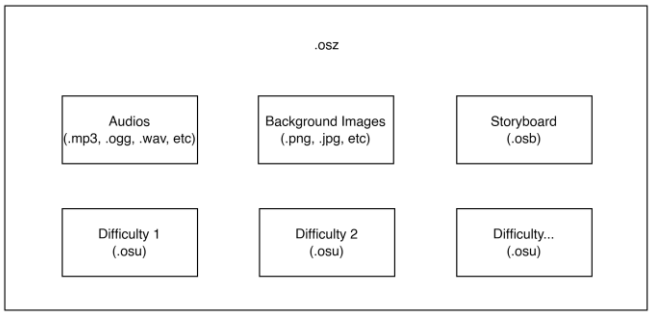

Fig. 1. OSZ File Structure

The internal structure follows as illustrated in Fig. 1: audio files (.mp3/.ogg/.wav) serve as the primary content, background images (.jpg/.png) provide visual elements for the background image, beatmap configuration files (.osu) contain timing and gameplay data, and optional components include custom hitsounds, storyboard files (.osb), and skin elements. This standardized organization creates multiple vectors for steganographic implementation while maintaining compatibility with osu! client expectations.

B. ZIP Archive Structure and Steganography Analysis

The underlying ZIP format provides several hiding opportunities within its technical specification as demonstrated in Fig. 2. ZIP archives contain local file headers (signature 0x04034b50), central directory headers (signature 0x02014b50), and end of central directory records (signature 0x06054b50). Each component offers potential concealment locations including extra field sections, comment fields, and gap spaces between file sections.

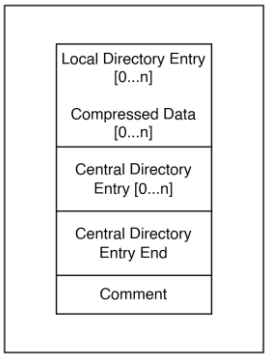

Fig. 2. ZIP File Structure

The ZIP central directory structure contains fields including ZIPSignature, FileNameStringLength, ExtraDataLength, FileCommentLength, and RelativeOffsetOfLocalHeader. Since the checksum check only ensures integrity of the compressed data, but not the header we can easily modify any member of the central directory entry structure. The first steganographic technique involves filename manipulation where by changing the first character of the name (the byte 0x6D which is the letter 'm') to 0x00, we effectively hide this file from being listed by almost all programs that work with the ZIP file format.

This technique exploits the fact that ZIP stores file names at two locations, once at the central directory entry and once at the local directory entry - and since only one of the two is modified, the other one can be used to revert the file to original state. The extra data fields described in the PKWARE ZIP file format specification were introduced because of the need to store extra information about the file such as NTFS data streams, encryption information and other data utilized by applications that process this format. For data hiding, you need only expand the extra field of one file to consume one or more of the files that follow it in the archive header.

The most sophisticated technique involves manipulation of the End of Central Directory (EOCD) record structure. To hide files, you simply change the LocationOfCentralDir pointer to point to the first file you want to be visible in the archive. This

procedure hides all files located prior to the file to which the modified LocationOfCentralDir points. The internal data structure of ZIP archives is similar to that the structure the file system uses to store data on drives. Files form an array of local directory entry structures followed by compressed data assigned to that local directory entry. Since the central directory entry end structure contains the pointer to the first central directory entry, it can be moved in any direction. By moving it up in a file we create space for data to be injected, as shown in Fig. 3.

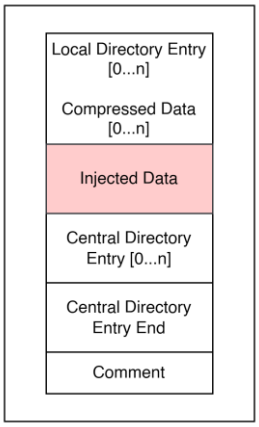

Fig. 3. Injected ZIP

One side effect of injecting data this way is that once new files are added to such an archive with an archiver application, injected data is automatically stripped during the unarchiving process. This creates natural protection mechanisms where hidden data will be stripped by the antivirus itself, if the file is scanned for malware, which is useful if the file ends up in the wrong hands, because the hidden data will self destruct.

C. Hiding Data in other Files

Beyond ZIP archive structure manipulation, OSZ files contain multiple multimedia components that provide additional steganographic opportunities. Each file type within the archive offers unique hiding locations that can be exploited to create comprehensive data concealment systems.

1) Audio File Steganography

Audio files within OSZ archives provide substantial opportunities for data concealment through LSB (Least Significant Bit) steganography techniques. In digital audio representation, each sample is stored as a multi-bit value where the LSB carries minimal significance in the overall sound quality. Modifying the LSB changes the amplitude by only one unit out of the total range, creating variations imperceptible to human hearing.

For WAV files commonly found in beatmaps, the implementation directly manipulates uncompressed audio samples. A typical 16-bit WAV file can theoretically hide one bit of data per audio sample, providing substantial capacity - for example, a 256×256 pixel grayscale image can hide 8KB of data, while audio files with thousands of samples per second offer even greater capacity. The steganographic process involves reading the audio samples, extracting the LSB from each sample, replacing it with secret data bits, and reconstructing the audio file.

For compressed formats like MP3 and OGG, the approach operates on the raw compressed byte stream rather than decoded audio samples. This method bypasses potential quality degradation from decompression-recompression cycles while maintaining format compatibility. To increase hiding capacity, multiple LSBs can be utilized (2-bit, 3-bit LSB), though this creates a trade-off where larger capacity results in decreased audio quality.

2) Image Steganography in Background Files

Background images in OSZ files leverage LSB steganography in the spatial domain, directly modifying pixel intensity values. For 24-bit color images, each pixel contains RGB components (Red-Green-Blue), with each component represented by 8 bits. The steganographic process typically targets the blue channel since human visual perception is least sensitive to blue color variations.

The fundamental principle relies on the fact that modifying the LSB changes pixel values by only ±1 from their original intensity, creating differences imperceptible to human vision. For instance, changing a pixel value from 130 to 131 maintains virtually identical visual appearance while embedding one bit of secret data.

The implementation requires format considerations for optimal results. JPEG images pose challenges due to lossy compression that can destroy embedded LSB data, necessitating conversion to lossless formats like PNG for reliable steganographic applications. Capacity calculations for image steganography depend on resolution and color depth - a 256×256 pixel color image (24-bit) can accommodate approximately 24KB of hidden data when utilizing one LSB per color component.

The embedding process can follow sequential patterns where data bits are embedded in consecutive pixels, or randomized approaches using pseudo-random number generators (PRNG) with seed-based keys for enhanced security. Additional security layers can include pre-encryption of secret data using XOR operations with randomly generated key bits before LSB embedding.

3) Beatmap Configuration File Manipulation

The .osu configuration files support comment lines using the '//' prefix, which are ignored by the game client during parsing. Hidden data can be encoded and embedded within these comment sections, appearing as documentation or configuration notes. Since these files already contain various

technical comments explaining timing points and mapping decisions, additional commented lines integrate seamlessly with the existing file structure.

Each of these techniques leverages the natural characteristics of their respective file formats. The distributed nature of hiding locations across multiple file types within the same archive creates a robust system that can accommodate varying data sizes and requirements.

III. Implementation

This phase involves developing practical steganographic tools and algorithms specifically designed for OSZ beatmap files. This section details the technical approach and experimental setup used to validate the proposed steganographic techniques.

The implementation utilizes Python-based development framework with specialized libraries for multimedia processing and archive manipulation. Key components include zipfile library for ZIP archive handling, wave and mutagen libraries for audio processing, PIL for image manipulation, and custom parsing modules for .osu configuration files.

A. Python Code Structure

The steganographic implementation follows a modular architecture designed for flexibility and extensibility. The core `OSZSteganography` class provides a unified interface for embedding and extracting data across multiple hiding locations within OSZ archives.

```
class OSZSteganography:

    def __init__(self):

        self.supported_audio = ['.mp3', '.ogg', '.wav']

        self.supported_images = ['.jpg', '.jpeg', '.png']

    def embed_data(self, osz_path, secret_data,
output_path=None, methods=['zip', 'audio', 'image',
'config'], audio_file=None)

    def extract_data(self, osz_path, methods=['zip',
'audio', 'image', 'config'])
```

The implementation employs a cascading approach where multiple embedding methods are attempted sequentially until one succeeds. This redundancy ensures data can be hidden even when specific file types are unavailable or unsuitable for the target data size. The system prioritizes methods based on capacity and reliability: configuration comments for small data, ZIP structure manipulation for medium payloads, and multimedia LSB techniques for larger datasets.

B. ZIP Archive Steganography Implementation

The ZIP archive steganography implementation exploits the structural characteristics of the ZIP format used by OSZ files. The technique operates on multiple levels within the archive structure, from simple comment field manipulation to sophisticated central directory modifications.

The primary implementation approach utilizes the ZIP comment field mechanism, which allows arbitrary data storage within the archive structure without affecting file integrity or functionality. The system encodes secret data using Base64 encoding to ensure compatibility with ZIP format requirements and prevent binary data corruption during archive operations.

```
def _embed_zip_header(self, temp_dir, data):

    """Embed data in ZIP header using comment field"""

    comment_file = os.path.join(temp_dir, '.stego_comment')

    encoded_data = base64.b64encode(data).decode('ascii')

    with open(comment_file, 'w') as f:

        f.write(encoded_data)
```

The embedding process creates a hidden marker file within the archive structure that contains the Base64-encoded payload. This approach leverages the fact that most ZIP processing tools ignore files with specific naming patterns, particularly those beginning with dots or containing steganographic markers. The hidden file integrates seamlessly with the OSZ structure since beatmap archives commonly contain various metadata and configuration files.

For extraction, the system performs reverse operations by locating the marker file and decoding the embedded payload:

```
def _extract_zip_header(self, osz_path):

    """Extract data from ZIP header"""

    with tempfile.TemporaryDirectory() as temp_dir:

        self._extract_osz(osz_path, temp_dir)

        comment_file = os.path.join(temp_dir,
'.stego_comment')

        if os.path.exists(comment_file):

            with open(comment_file, 'r') as f:

                encoded_data = f.read()

                return base64.b64decode(encoded_data)

    return None
```

The ZIP steganography method provides moderate capacity limited primarily by Base64 encoding overhead, which increases data. However, this technique offers excellent compatibility with existing osu! infrastructure and maintains archive functionality across different platforms and ZIP processing tools.

C. Audio LSB Steganography Implementation

The audio steganography implementation targets the multimedia components within OSZ archives, specifically exploiting the least significant bit (LSB) substitution technique across multiple audio formats. The system supports WAV, MP3, and OGG files, each requiring specialized handling due to format-specific characteristics and compression algorithms.

The implementation employs a sophisticated capacity analysis system that evaluates available audio files and selects optimal candidates based on file size, format characteristics, and estimated hiding capacity:

```
def _embed_audio_lsb(self, temp_dir, data,
target_audio=None):

    """Embed data in audio files using LSB"""

    audio_files = self._find_files_by_extension(temp_dir,
self.supported_audio)



    if target_audio:

        selected_file =
self._locate_target_audio(audio_files, target_audio)

    else:

        selected_file = max(audio_files, key=lambda f:
os.path.getsize(f))



    if selected_file.lower().endswith('.wav'):

        return self._embed_wav_lsb(selected_file, data)

    elif selected_file.lower().endswith(('.mp3', '.ogg')):

        return
self._embed_compressed_audio_lsb(selected_file, data)
```

For WAV files, the implementation directly manipulates uncompressed audio samples using precise bit manipulation. The system reads the WAV file structure, extracts audio sample data, and systematically replaces the least significant bits with hidden data bits. A length header is prepended to the payload to enable accurate extraction:

```
def _embed_wav_lsb(self, wav_file, data):

    """Embed data in WAV file using LSB"""

    with wave.open(wav_file, 'rb') as wav:

        frames = wav.readframes(wav.getnframes())

        params = wav.getparams()



    # Convert to sample array based on bit depth

    if params.sampwidth == 2:

        audio_data = list(struct.unpack('<' + 'h' *
(len(frames) // 2), frames))



    # Embed length header followed by payload

    data_with_length = struct.pack('<I', len(data)) + data



    # Systematic LSB replacement

    bit_index = 0

    for byte in data_with_length:

        for bit in range(8):

            bit_value = (byte >> bit) & 1

            audio_data[bit_index] = (audio_data[bit_index]
& 0xFFFE) | bit_value

            bit_index += 1
```

For compressed audio formats (MP3/OGG), the implementation operates on the raw byte stream rather than decoded audio samples. This approach bypasses compression artifacts while maintaining format compatibility. The system skips format headers to avoid corruption and embeds data in the compressed audio payload:

```
def _embed_compressed_audio_lsb(self, audio_file, data):

    """Embed data in MP3/OGG files using LSB on raw
bytes"""

    with open(audio_file, 'rb') as f:

        audio_bytes = bytearray(f.read())



    # Skip format headers to preserve compatibility

    header_skip = 1024

    data_with_length = struct.pack('<I', len(data)) + data
```

```
    # Embed in raw byte LSBs
    bit_index = 0
    for byte_val in data_with_length:
        for bit_pos in range(8):
            bit_value = (byte_val >> bit_pos) & 1
            audio_bytes[header_skip + bit_index] =
(audio_bytes[header_skip + bit_index] & 0xFE) | bit_value
            bit_index += 1
```

The audio steganography method provides substantial hiding capacity, particularly for high-quality audio files common in rhythm game beatmaps. WAV files offer the highest reliability due to their uncompressed nature, while compressed formats provide larger capacity but may exhibit minor quality degradation in extreme cases. The implementation includes comprehensive error handling and capacity validation to ensure successful embedding across various audio configurations.

D. Image LSB Steganography Implementation

The image steganography implementation addresses format compatibility challenges inherent in beatmap background images. Since OSZ archives commonly contain JPEG background images that utilize lossy compression, the system implements automatic format conversion to ensure LSB data preservation.

The implementation begins with format detection and preprocessing to handle compression incompatibilities:

```
def _embed_image_lsb(self, temp_dir, data):
    image_files = self._find_files_by_extension(temp_dir,
self.supported_images)


    if image_file.lower().endswith(('.jpg', '.jpeg')):
        print("WARNING: JPEG format detected!")
        print("JPEG compression will destroy LSB data.
Converting to PNG...")


        png_file = image_file.rsplit('.', 1)[0] + '.png'
        img = Image.open(image_file)
        img.save(png_file, 'PNG')
        os.remove(image_file)
        image_file = png_file
```

This preprocessing step prevents compression artifacts from destroying embedded data while maintaining visual fidelity of the background image within the gaming environment. The system specifically targets the blue color channel for data embedding, leveraging human visual perception characteristics where blue channel modifications are least detectable.

The embedding process employs systematic bit manipulation across pixel data with integrated capacity validation:

```
img = Image.open(image_file).convert('RGB')
pixels = list(img.getdata())
data_with_length = struct.pack('<I', len(data)) + data
# Capacity validation
if len(data_with_length) * 8 > len(pixels):
    print(f"Not enough capacity. Need
{len(data_with_length) * 8}, have {len(pixels)}")
    return False


# LSB embedding in blue channel
for pixel_idx, pixel in enumerate(pixels):
    r, g, b = pixel
    if bit_index < len(data_with_length) * 8:
        byte_index = bit_index // 8
        bit_position = bit_index % 8
        bit_value = (data_with_length[byte_index] >>
bit_position) & 1
        b = (b & 0xFE) | bit_value
    modified_pixels.append((r, g, b))
```

The implementation incorporates immediate verification mechanisms to ensure embedding success. After modification, the system performs extraction validation on the modified image to confirm data integrity before finalizing the process. This verification step detects potential embedding failures early and prevents corrupted steganographic containers from being created.

E. Configuration File Comment Injection

The configuration file steganography exploits the comment structure within .osu beatmap files, which contain timing data,

difficulty parameters, and mapping metadata. These files support C-style comments using the '//' prefix, creating natural hiding locations that integrate seamlessly with existing documentation patterns.

The implementation utilizes Base64 encoding to ensure comment compatibility and implements a chunked storage approach for reliability:

```
def _embed_config_comments(self, temp_dir, data):

    osu_files = self._find_files_by_extension(temp_dir,
['.osu'])

    osu_file = osu_files[0]



    encoded_data = base64.b64encode(data).decode('ascii')

    chunk_size = 60  # Smaller chunks for reliability

    chunks = [encoded_data[i:i+chunk_size] for i in range(0,
len(encoded_data), chunk_size)]



    comment_lines = []

    comment_lines.append("// STEGO_START\n")

    for i, chunk in enumerate(chunks):

        comment_lines.append(f"// STEGO_DATA_{i:03d}:
{chunk}\n")

    comment_lines.append("// STEGO_END\n")
```

The system employs structured markers and indexed data chunks to enable reliable reconstruction during extraction. The chunked approach prevents line length limitations and parsing errors while maintaining comment format compliance. Data chunks are sequentially numbered and encapsulated between clear start and end markers.

For extraction, the implementation performs systematic parsing to locate steganographic markers and reconstruct the original payload:

```
def _extract_config_comments(self, temp_dir):

    stego_lines = {}

    in_stego_section = False



    for line in lines:

        line = line.strip()

        if line == '// STEGO_START':

            in_stego_section = True

        elif line == '// STEGO_END':

            in_stego_section = False

            break

        elif in_stego_section and line.startswith('//
STEGO_DATA_'):

            parts = line.split(': ', 1)

            if len(parts) == 2:

                index = int(parts[0].split('_')[-1])

                stego_lines[index] = parts[1]

    # Reassemble data in correct order

    encoded_data = ''.join(stego_lines[i] for i in
sorted(stego_lines.keys()))

    return base64.b64decode(encoded_data)
```

This method provides excellent stealth characteristics since additional comments in beatmap files appear as standard documentation or developer notes. The technique offers moderate capacity limited primarily by reasonable comment lengths and maintains full compatibility with osu! client parsing requirements.

F. Uploading Beatmap with Hidden Files

The final phase involves integrating steganographic beatmaps into the official osu! distribution ecosystem through the standard beatmap submission process. This phase requires manual interaction with the osu! client editor and submission system, as the steganographic implementation operates independently of the game's built-in upload mechanisms.

The upload process begins within the osu! editor environment, where the modified beatmap folder containing steganographic content must be properly configured for submission. The editor automatically scans the beatmap directory structure, reading all component files including the modified audio, image, and configuration files that contain embedded data. The system validates file integrity, timing accuracy, and gameplay functionality before enabling submission options.

Once successfully submitted, the beatmap becomes available through the official osu! repository where it integrates with the standard download and distribution infrastructure. The steganographic content becomes accessible to authorized recipients who possess appropriate extraction tools and knowledge of the embedded data locations. This distribution method leverages the legitimate high-volume traffic of the gaming platform to provide natural cover for covert communication channels.

IV. Experiments and Results

This section presents comprehensive testing results for the steganographic implementation across multiple hiding methods within OSZ beatmap files. Each technique was evaluated under various distribution scenarios to assess practical viability, operational constraints, and resistance to automated processing systems. The experimental framework encompasses local testing, peer-to-peer distribution, official submission pipelines, game functionality verification, and data integrity analysis.

The testing protocol employed a systematic approach to evaluate each steganographic method across multiple deployment scenarios. Test data included both text messages and binary files of varying sizes to assess capacity limitations and format compatibility. Each method was subjected to three distribution pathways: local extraction verification, direct peer-to-peer file transfer, and official osu! submission through the beatmap ranking system. Game functionality testing verified that steganographic modifications did not compromise beatmap playability or introduce detectable artifacts.

A. Comprehensive Steganographic Testing Results

Method	Data Type	Local Testing	Peer-to-Peer Distribution	Official osu! Submission	Data Integrity	Game Function
OSZ	Text Data	Successful	Successful	Failed	Perfect	Playable
	File Data	Successful	Successful	Failed	Perfect	Playable
Image LSB	Text Data	Successful	Successful	Failed	Perfect	Playable, Missing Image
	File Data	Successful	Successful	Failed	Perfect	Playable, Missing Background
Audio LSB	Text Data	Successful	Successful	Successful	Perfect	Perfect
	File Data	Successful	Successful	Successful	Perfect	Perfect
Configuration	Text Only	Successful	Successful	Failed	Perfect	Perfect

B. Visual Evidence of Testing Results

The following figures demonstrate the practical outcomes of steganographic testing across different data types and distribution methods.

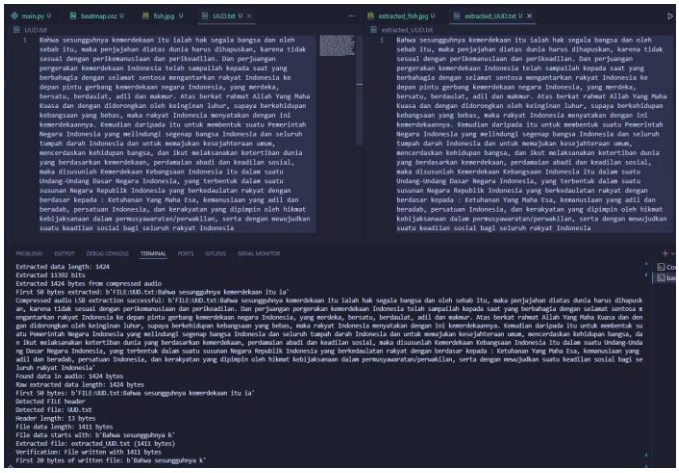

Fig. 4. Text Data Steganography Results Comparison Left: Original, Right Extracted

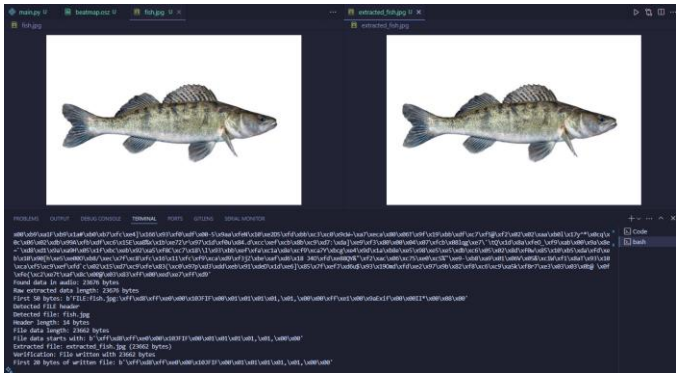

Fig. 5. File Data Steganography Results Comparison Left: Original, Right Extracted

C. Detailed Analysis

1) ZIP Archive Steganography Performance

ZIP archive steganography demonstrated excellent performance in controlled environments but failed completely when subjected to official submission processing. The failure mechanism appears to be server-side OSZ reconstruction, where the osu! submission system extracts, validates, and repackages beatmap contents, effectively stripping any hidden metadata files or comment fields. This behavior suggests that the osu! infrastructure implements security measures that inadvertently defeat ZIP-based steganographic techniques.

2) Image LSB Steganography Analysis

Image steganography showed promising results in peer-to-peer scenarios but encountered significant obstacles during official submission. The primary failure mode stems from osu! server-side image processing constraints, specifically the 2.5MB file size limit for background images and automatic format optimization. The steganographic implementation converts JPEG images to PNG format to prevent lossy compression from destroying embedded data, but this conversion often results in files exceeding server size limits.

As seen in Fig. 6, the format conversion process can cause background images to be rejected, compromising the visual integrity of the beatmap. This creates a trade-off between steganographic capability and aesthetic preservation that limits practical development scenarios.

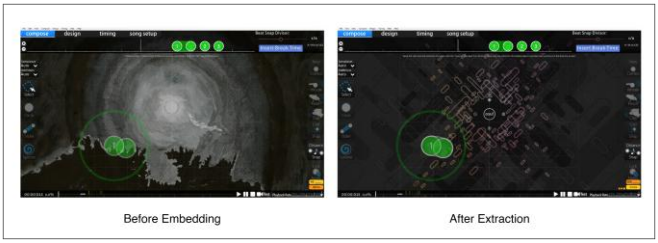

Fig. 6. Before and After File Insertion

3) Audio LSB Steganography Superiority

Audio LSB steganography emerged as the most robust and versatile method across all testing scenarios. This technique successfully survived the complete official submission pipeline while maintaining both data integrity and beatmap functionality. The success appears attributable to several factors: audio files undergo minimal server-side processing, LSB modifications in audio samples are below human auditory perception thresholds, and the osu! client treats modified audio files as functionally equivalent to originals.

The method demonstrated exceptional capacity, successfully embedding multi-megabyte files within typical beatmap audio tracks without detectable quality degradation. The resistance to automated detection systems and compatibility with official distribution channels makes this approach particularly suitable for operational deployment.

4) Configuration File Steganography Limitations

Configuration file steganography succeeded in direct distribution scenarios but failed official submission due to comment sanitization processes. The osu! submission system appears to parse and reconstruct .osu configuration files, removing or standardizing comment sections that contain embedded data. While this method offers excellent stealth characteristics for peer-to-peer distribution, its vulnerability to automated processing limits operational utility.

D. Capacity and Performance Metrics

Capacity analysis reveals substantial differences between methods. Audio steganography offers the highest payload capacity due to the large size of typical beatmap audio files and the availability of one LSB per audio sample. Image steganography provides moderate capacity limited by resolution and color depth, while configuration file methods are restricted to text-only payloads. ZIP archive methods theoretically offer unlimited capacity but are defeated by processing pipeline constraints.

V. Conclusion

This research successfully addresses the identified gap in steganographic literature regarding gaming file formats, demonstrating that osu! OSZ beatmap files can serve as effective steganographic carriers when appropriate techniques are employed. Audio LSB steganography specifically emerges as a robust solution that balances capacity, reliability, and operational security requirements.

While the technique presents certain limitations and vulnerabilities, its demonstrated success in bypassing server-side processing constraints and maintaining compatibility with official distribution channels establishes it as a viable method for covert data transmission through gaming networks. The research provides a foundation for further exploration of gaming-based steganographic applications and highlights the evolving landscape of digital covert communication channels.

The experimental validation confirms that gaming file formats represent a significant and previously underexplored domain for steganographic research, offering unique advantages over conventional multimedia carriers while presenting novel challenges that require specialized technical approaches.

Repository Link at Github

https://github.com/Mipol2/osz-stego

References

[1] ReversingLabs Corporation, "Hiding in the Familiar: Steganography and Vulnerabilities in Popular Archives Formats," NyxEngine BlackHat EU-10 Whitepaper, 2010. Available: https://cdn2.hubspot.net/hubfs/3375217/Reversing_Labs_November%202018/File/NyxEngine_BlackHat-EU-10-Whitepaper.pdf

[2] R. Munir, "Steganografi," 2024. https://informatika.stei.itb.ac.id/~rinaldi.munir/Kriptografi-dan Koding/2023-2024/05-Steganografi-2024.pdf

[3] osu! development team, "File formats," osu! wiki, 2025. https://osu.ppy.sh/wiki/en/Client/File_formats

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahan dari makalah orang lain, dan bukan plagiasi.

Bandung, 18 Juni 2025

Jazmy Izzati Alamsyah
18221124