Steganography in .osu Files

Muhammad Rasheed Qais Tandjung
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
13522158@std.stei.itb.ac.id, mrqaist@gmail.com

***Abstract*— This paper proposes and evaluates a robust steganographic algorithm for the .osu plaintext file format used in the rhythm game osu!. While previous attempts at data concealment in this format, specifically through plaintext comments, failed due to server-side post-processing and lack of subtlety, the method introduced here utilizes Least Significant Bit (LSB) modification of the "time" parameter within the hit object section. By shifting the timing of hit objects by a subliminal one millisecond, data can be embedded with a capacity of 1 bit per hit object. Experimental results demonstrate a 100% success rate for data integrity and extraction after submission to official osu! servers. Analysis of the player experience through the lens of sensorimotor synchronization (SMS) and the just noticeable difference (JND) threshold indicates that these modifications are imperceptible to human players. Despite a minor 5% reduction in the effective hit window at the highest difficulty levels, the algorithm proves to be a viable and covert channel for transmitting short-form data, such as URLs or cryptographic keys, within the routine network traffic of the osu! ecosystem.**

Keywords — Data concealment, Least Significant Bit (LSB), osu!, Steganography, .osu File Format

I. Introduction

osu! is a free-to-play rhythm game developed by Dean Herbert. Its core gameplay mechanic involves the player responding to a series of visual *hit objects* (e.g. circles, sliders, and spinners) that appear on the playfield in synchronization with a chosen audio track. Success is measured by the player's ability to activate these objects within temporally bounded windows, with subsequent accuracy calculations determining a numerical performance rating.

In osu!, A beatmap (sometimes called beatmapset) is a set of game levels (referred to as *difficulties* in osu! terminology), each of which comprise various hit objects and almost always represent a single song. Each level's specification, such as the hit objects, timings, and metadata, is defined by a plaintext file with the `.osu` extension. These levels, along with other components such as the audio track and decorative images, are packed into an archive with the `.osz` extension.

Beatmap distribution in osu! occurs through a large network of official and unofficial channels, handling massive volumes of routine downloads. This creates substantial background network traffic, which could serve as effective cover for covert data channels. The commonplace transfer of large game files within this ecosystem reduces scrutiny compared to general-purpose platforms, establishing a potentially viable setting for steganographic operations.

The `.osu` file extension is of particular interest. The format of these files adhere to a rigid, sectioned structure, enumerating metadata (e.g., song title, creator), difficulty parameters (e.g., object timing, timing lenience), and the precise temporal and spatial coordinates of every hit object. As a human-readable INI-like format, it is parsed linearly by the osu! client software to reconstruct the gameplay experience.

Now, in particular, this file format allows multiple different steganographic schema, one of which is to simply append information / bytes into each line as comments. However, an algorithm based on this has already been proposed in 2024 [1], and it resulted in failure after the file has been submitted to official osu! servers due to the server applying `.osu` file post-processing. Thus, if this file format is to be used for steganography, a more sophisticated algorithm should be used that is immune to the file post-processing done by the osu! server.

An important note to make now, is that there exists a previous work of steganography related to osu! [1]. However, I would like to highlight that this previous work explores steganography of the `.osz` file format, not the `.osu` file format. Further details will be explained in section II, but in general, the paper explores standard steganography in the other file formats contained in the `.osz` file such as the audio track, decorative images, and even data concealment in zip metadata. Only the audio steganography succeeded (data concealment survived official osu! server submissions). The steganography algorithm proposed for the .osu configuration file is neither robust nor successful. So while this work is definitely a good reference point and provides a good understanding of steganography mediums related to osu! and its file formats, it doesn't provide a robust algorithm for data concealment in the `.osu` file.

The aim of this paper is to propose and implement a steganographic algorithm for concealment and extraction of data to the `.osu` file format, as well as to evaluate said algorithm on different parameters, such as data capacity, undetectability, and immunity to post-processing by official osu! servers. This paper also aims to improve the previous work by Alamsyah (2025), expanding on the knowledge and providing a more robust steganographic algorithm.

II. Related Work

This section is dedicated to summarizing Alamsyah's work [1], as well as assessing areas of improvement.

To summarize, this study investigates the implementation of steganographic techniques within osu! OSZ beatmap files, exploring multiple methods for covert data transmission through gaming networks. Four primary approaches were examined:

- Manipulation of the underlying ZIP archive structure
- Least significant bit (LSB) modification in audio files
- LSB embedding in background images
- Injection of hidden data into .osu file comments.

The study tested these methods across local, peer-to-peer, and official submission scenarios, ultimately identifying audio LSB steganography as the most effective approach. It achieved 100% success across all test cases, reliably embedding multi-megabyte payloads without degrading audio quality or raising suspicion within the osu! ecosystem. In contrast, ZIP, image, and configuration-based methods *failed* during official server-side processing, including image optimization, size restrictions, and archive reconstruction. These constraints significantly reduce the practical viability of the image and ZIP methods in environments where automated processing is involved.

In addition, this study explores a basic form of steganography within .osu (referred to as "configuration files" by the study) by embedding concealed data as plaintext comments, marked with prefixes such as `//STEGO_START`. However, this method was unable to pass official osu! beatmap submissions, as the server's comment sanitization process removes or standardizes such entries. Beyond its incompatibility with submission pipelines, this approach *lacks subtlety*. A core principle of steganography is that hidden data **should not** be noticeable even upon direct inspection of the file. Plain comment lines containing structured markers are easily detectable to anyone viewing the file, and this obviously does not preserve the stealth aspect of steganography, which makes this method impractical for secure covert communication.

III. Methodology

This section will contain a detailed analysis for steganography in the **`.osu`** file format, the proposed algorithm for steganography, as well as a preliminary analysis of the detectability and the capacity of this algorithm.

*A. Analysis of **`.osu`** File Format*

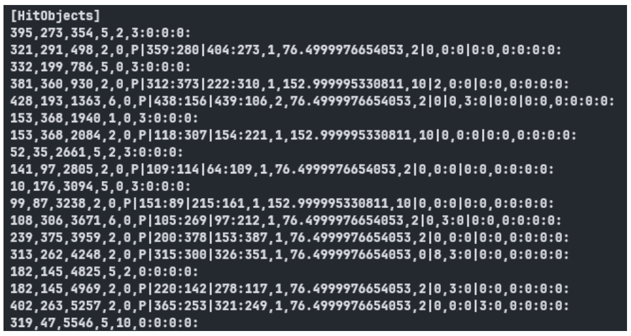
The **`.osu`** file is a plaintext file with several sections [4]. Each section is denoted by a header-like line that is enclosed with square brackets (`[like so]`). Fig. 1 depicts a typical example of the first two sections in an **`.osu`** file.

```
osu file format v14

[General]
AudioFilename: audio.mp3
AudioLeadIn: 0
PreviewTime: 57974
Countdown: 0
SampleSet: Soft
StackLeniency: 0.9
Mode: 0
LetterboxInBreaks: 0
EpilepsyWarning: 1
WidescreenStoryboard: 1

[Editor]
DistanceSpacing: 2.3
BeatDivisor: 4
GridSize: 32
TimelineZoom: 2.3
```

Figure 1. *Typical First Two Sections of an .osu File*

There are 8 sections in an **`.osu`** file. Each of these contain different useful information regarding the level. However, all but the last section contain information that is integral to the "correctness" of the level. If any these were to be changed, there is a large chance that noticable and significant changes to the level will clearly be seen. Besides, the data in the first 7 sections are key-pair values (as in Fig. 1), and it would be *immediately* obvious to anyone opening this file in a text editor if this section were to be tampered to conceal data in any significant size.

However, the last section, the **`[HitObject]`** section, contains all the hit object information for the level. A single level regularly contains hundreds of hit objects, and even thousands for difficult levels. In this section, each line describes a single hit object, and there is an abundance of numerical information for a single hit object, as can be seen from fig. 2.



```
[HitObjects]
395,273,354,5,2,3:0:0:0:
321,291,498,2,0,P|359:280|404:273,1,76.4999976654053,2|0,0:0|0:0,0:0:0:0:
332,199,786,5,0,3:0:0:0:
381,360,930,2,0,P|312:373|222:310,1,152.999995330811,10|2,0:0|0:0,0:0:0:0:
428,193,1363,6,0,P|438:156|439:106,2,76.4999976654053,2|0|0,3:0|0:0|0:0,0:0:0:0:
153,368,1940,1,0,3:0:0:0:
153,368,2084,2,0,P|118:307|154:221,1,152.999995330811,10|0,0:0|0:0,0:0:0:0:
52,35,2661,5,2,3:0:0:0:
141,97,2805,2,0,P|109:114|64:109,1,76.4999976654053,2|0,0:0|0:0,0:0:0:0:
10,176,3094,5,0,3:0:0:0:
99,87,3238,2,0,P|151:89|215:161,1,152.999995330811,10|0,0:0|0:0,0:0:0:0:
108,306,3671,6,0,P|105:269|97:212,1,76.4999976654053,2|0,3:0|0:0,0:0:0:0:
239,375,3959,2,0,P|200:378|153:387,1,76.4999976654053,2|0,0:0|0:0,0:0:0:0:
313,262,4248,2,0,P|315:300|326:351,1,76.4999976654053,0|8,3:0|0:0,0:0:0:0:
182,145,4825,5,2,0:0:0:0:
182,145,4969,2,0,P|220:142|278:117,1,76.4999976654053,2|0,3:0|0:0,0:0:0:0:
402,263,5257,2,0,P|365:253|321:249,1,76.4999976654053,2|0,0:0|3:0,0:0:0:0:
319,47,5546,5,10,0:0:0:0:
```

Figure 2. *Typical Hit Object Information in .osu File*

This is unlike the first seven sections where there are only a few lines, and each line is only a key-value pair. The hit object syntax is as follows:

```
x,y,time,type,hitSound,objectParams,hitSample
```

And here are the meanings of each.

- `x` (Integer) and `y` (Integer): Position in osu! pixels of the object.
- `time` (Integer): Time when the object is to be hit, in **milliseconds** from the beginning of the beatmap's audio.
- `type` (Integer): an 8-bit integer where each bit is a flag with special meaning, indicating the type of the hit object.
- `hitSound` (Integer): Bit flags indicating the hitsound applied to the object. See the hitsound section.
- `objectParams` (Comma-separated list): Extra parameters specific to the object's type.
- `hitSample` (Colon-separated list): Information about which samples are played when the object is hit. It is closely related to hitSound. If it is not written, it defaults to `0:0:0:0:`.

The `x` and `y` parameters should not be modified, since changing a single osu! pixel can cause the gameplay flow to be vastly different. (In competitive play, moderators of the game often disallow or criticize level designs that are off by even a few pixels.) `hitSample` and `hitSound` is not an option for modification becaue it messes with the artistic integrity of the level; the level designer uses these to create custom sound effects and apply them to the hit objects throughout the level. Finally, `type` and `objectParams` are complicated because multiple different modes of gameplay exist. They can take on various values depending on what game mode the level was designed for. As such, for simplicity, this paper's main focus will be to conceal data within *just* the `time` parameter.

As will be explained further in subsection C, the `time` parameter is chosen for data concealment due to the fact that this data is modifiable with extremely negligble, even undetectable side effects. But the main takeaway of this section is that the other parameters are either complicated, or do not permit the modification without significant changes to the level gameplay.

B. The Proposed Algorithm

Given a string of bytes, we want to embed this data in the `time` parameter of the hit objects in the HitObject section of the .osu file. The proposed method is to use the LSB of each hit object's `time` parameter. Algorithm 1 shows the concealment process (which modifies the .osu files in place) and Algorithm 2 shows the extraction process.

```
Algorithm 1 Embed Data into HitObjects
procedure EmbedData($osuFile, message$)
    $lines \leftarrow readlines(osuFile)$
    Convert $message$ to bit sequence:
        For each character, take 7-bit ASCII (reversed, padded)
    $bitStream \leftarrow$ concatenated bit sequence
    $bitIndex \leftarrow 0$
    Skip all lines until $[HitObjects]$ section is found
    for each line in $[HitObjects]$ section do
        Parse time parameter from comma-separated line
        Clear least significant bit of time parameter
        if more bits remain in $bitStream$ then
            Set LSB to current bit from $bitStream$
            Advance $bitIndex$
        end if
        Update line with modified time parameter
    end for
    if not all bits were embedded then
        error "Insufficient HitObject lines for message"
    end if
end procedure
```

Algorithm 1. *Embed Data into Hit Objects*

```
Algorithm 2 Extract Hidden Data from HitObjects
function ExtractData($osuFile$)
    $lines \leftarrow readlines(osuFile)$
    $bitBuffer \leftarrow$ empty
    $result \leftarrow$ empty
    Skip all lines until $[HitObjects]$ section is found
    for each line in HitObjects section do
        Extract $time$ field from comma-separated line
        $bit \leftarrow$ least significant bit of $time$ field
        Append $bit$ to $bitBuffer$
        if $bitBuffer$ contains 7 bits then
            Convert 7 bits (reversed) to character
            if character is null terminator then
                break
            end if
            Append character to $result$
            Clear $bitBuffer$
        end if
    end for
    return $result$
end function
```

Algorithm 2. *Extract Data into Hit Objects*

C. Analysis of Detectability

First we will analyse this method from the perspective of the player's experience. The critical reader should question the validity of the proposed method, since changing the time parameter of the hit objects would obviously change the timing, i.e. when the hit objects appear on the screen. However, research in the area of psychoacoustics show that humans are not able to perceive such a negligible (± 1ms) change in timing.

The human ability to maintain timing is governed by a concept called Sensorimotor Synchronization (SMS) [5], which is the coordination of rhythmic movement to an external stimulus. In their research review, Repp and Su distinguish between two primary error-correction mechanisms: phase correction and period correction. Phase correction is a reactive, automatic process that adjusts the timing of the next motor action based on the asynchrony (the temporal gap between a tap and the target) of the current one. Crucially, research shows that phase correction occurs even when timing perturbations are *subliminal*, meaning they are too small to be consciously detected by the player. Because the human Just Noticeable Difference (JND) for timing (the minimum change required for a person to perceive a shift or change) is typically estimated to be at least 5–10% of the interval (often 10ms or more in gaming contexts), a ±1ms change falls far below the perceptual threshold. The motor system may compensate for such tiny deviations automatically, but the player remains entirely unaware of any change in the game's "feel".

Furthermore, the argument for the negligibility of a 1ms shift is strengthened by the inherent variability in human performance, often referred to as motor noise. Even the most skilled rhythm game players exhibit a natural "jitter" in their tap timing that exceeds 1ms, meaning any microscopic adjustment to hit objects is essentially "drowned out" by the biological variance of the finger's movement. Additionally, humans exhibit a phenomenon known as Negative Mean Asynchrony (NMA), a consistent tendency to tap several milliseconds ahead of a beat rather than exactly on it [7]. This systematic anticipatory error demonstrates that our internal "timekeeper" is not a precision-perfect clock but a predictive system operating within a much larger window of tolerance. Since the NMA alone can range from 20ms to 50ms, a 1ms alteration is an insanely small fraction of the natural temporal fluctuations already present in the human-computer interaction, justifying that such changes do not validly disrupt the gameplay experience.

Now, we should also analyse this from the perspective of the game's core judgement system. That is, does a change of 1ms affect the game's interpretation of the player's "correct"-ness.

A *judgement*, or hit result, is the outcome of interacting with a hit object during its hit window [3]. Score and accuracy are calculated based on which judgements are received, and the more offbeat or far from the hit window the player hits, the less score they will get. In osu!, the highest difficulty. The hit window depends on the beatmap's *overall difficulty* (OD) parameter (configured in `.osu` file). The important observation is that on the game's highest OD, the hit window is 20ms. So in fact, a shift of 1ms in hit objects would shift the hit window (with respect to the game's audio) causing the player's interpretation of the hit window to be different. This does reduce the effective hit window, making it now 19ms (see Fig. 3). Unfortunately this is a reduction of 5% in gameplay correctness, and it is the main trade-off of this algorithm.

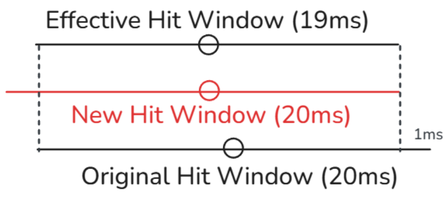


Figure 3. *Effective hit window*

D. Analysis of Capacity

The capacity of this algorithm is precisely 1 bit for each hit object. The capacity is calculated as the ratio of the hidden bytes to the total size of the file. However, it is difficult to calculate this analitically because the other sections in the file are variable in size, as well as the amount of characters per line in the HitObject section heavily depend on the level and object types. Therefore, a numeric approach is taken, and the capacity is calculated over the averages of 2000 files. We will need to determine two things: the average amount of hit objects in a file and the average file size. Equation (1) shows the calculation of the capacity.

$$capacity = \frac{hit\ objects/8}{file\ size} \tag{1}$$

A diagnostic script was run across 2000 .osu files, and it was found that on average, each file had a size of 36268 bytes and had 1007 hit objects. Thus, the capacity of the average file is roughly $(1007/8)/36268 = 0.0034$ or 0.34%. For perspective, a 10kb .osu file would thus be able to hold around 34 bytes with this method. And for the average file (~36KB), it is approximately 125 bytes.

Of course, this is not very much data at all. However, 125 bytes is enough for a typical HTTP URL, or a typical sentence in english. Thus this method is still useful for general purpose communication, just *not* capable of embedding complex data such as audio and images.

IV. Experiment Setup and Results

In this section we will go over how experimentation was done and also the results over several of the expereiments.

A. Experiment Setup

The primary objective of the experiments is to empirically validate the robustness of the proposed LSB steganographic algorithm applied to the time parameter in `.osu` files. Specifically, we aim to verify:

1. Data Integrity: Whether the hidden message survives the official osu! beatmap submission and server-side post-processing.
2. Practical Viability: The empirical capacity of the method across different `.osu` files of varying sizes and hit object counts.
3. Imperceptibility: Whether modifications to time values remain undetectable during actual gameplay.

To achieve this, the following experiment process was designed and executed:

1) Preparation of Test Files

Three `.osu` files were selected to represent a range of typical beatmap sizes and complexities:

1. File A: A small, simple beatmap with relatively few

hit objects (249 hit objects, file size ~12 KB).
2. File B: A medium-length beatmap of average complexity (1166 objects, file size ~36 KB).
3. File C: A long, difficult beatmap with a high hit object count (2563 objects, file size ~85 KB).

Each file was backed up in its original state to serve as a control.

2) Embedding Test Messages

Using the proposed embedding algorithm (Algorithm 1), a unique test message was concealed in each .osu file. The messages were designed to be representative of plausible covert communications:

1. Message for File A: `https://example.com/a1b2c3` (a short URL).
2. Message for File B: `The quick brown fox jumps over the lazy dog` (a plain-text sentence).
3. Message for File C: `U3RlZzIwMjU=` (a base64-encoded short string).

The bitstream was embedded sequentially into the least significant bit of the time parameter of each hit object, starting from the first hit object in the **`[HitObjects]`** section.

3) Submission to Official osu! Beatmap Submission System

Each modified **`.osu`** file was packaged into its corresponding **`.osz`** archive (along with the required audio and image assets) and submitted to the official osu! beatmap submission system via a test account. The submission process includes automated server-side validation, optimization, and potential sanitization of beatmap files.

4) Retrieval and Extraction

After submission, the processed beatmap was downloaded from the official osu! website. The **`.osu`** file was extracted from the downloaded **`.osz`** archive. The extraction algorithm (Algorithm 2) was then applied to retrieve the hidden bitstream from the time parameters, which was subsequently decoded back into the original message format.

5) Evaluation Metrics

The success of each trial was measured using the following criteria:

1. Submission Success: Whether the beatmap containing the modified .osu file successfully pass submission into the official osu! servers.
2. Extraction Success: Whether the extracted message matched the original message exactly.
3. Capacity Extracted: The number of bytes successfully hidden and recovered. **Note** that this includes zero-bytes (i.e. we add zero-byte padding to the rest of the unused hit objects, and extract from them as well), as we would like to examine whether or not a message that *fully* uses the capacity would be sucessfully extracted or not.

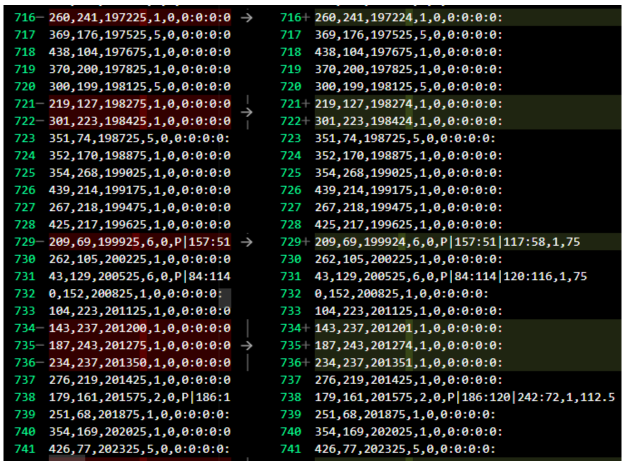
B. Results

Table 1 shows the results of the experiment.

Info	**File A**	**File B**	**File C**
Original size	~12KB	~36KB	~85KB
Amount of Hit Objects	249	1166	2563
Message size (bytes)	24	44	16
Submission Success	Yes	Yes	Yes
Extraction Success	Yes	Yes	Yes
Capacity Extracted	100%	100%	100%

Table 1. *Experiment Results*

Fig. 4 shows a direct comparison of file A in the **`HitObject`** section *before* and *after* the algorithm is applied (respectively).



```
- 260,241,197225,1,0,0:0:0:0
  369,176,197525,5,0,0:0:0:0
  438,104,197675,1,0,0:0:0:0
  370,200,197825,1,0,0:0:0:0
  300,199,198125,5,0,0:0:0:0
- 219,127,198275,1,0,0:0:0:0
- 301,223,198425,1,0,0:0:0:0
  351,74,198725,5,0,0:0:0:0:
  352,170,198875,1,0,0:0:0:0
  354,268,199025,1,0,0:0:0:0
  439,214,199175,1,0,0:0:0:0
  267,218,199475,1,0,0:0:0:0
  425,217,199625,1,0,0:0:0:0
- 209,69,199925,6,0,P|157:51
  262,105,200225,1,0,0:0:0:0
  43,129,200525,6,0,P|84:114
  0,152,200825,1,0,0:0:0:0:
  104,223,201125,1,0,0:0:0:0
- 143,237,201200,1,0,0:0:0:0
- 187,243,201275,1,0,0:0:0:0
- 234,237,201350,1,0,0:0:0:0
  276,219,201425,1,0,0:0:0:0
  179,161,201575,2,0,P|186:1
  251,68,201875,1,0,0:0:0:0:
  354,169,202025,1,0,0:0:0:0
  426,77,202325,5,0,0:0:0:0:
```

```
+ 260,241,197224,1,0,0:0:0:0:
  369,176,197525,5,0,0:0:0:0:
  438,104,197675,1,0,0:0:0:0:
  370,200,197825,1,0,0:0:0:0:
  300,199,198125,5,0,0:0:0:0:
+ 219,127,198274,1,0,0:0:0:0:
+ 301,223,198424,1,0,0:0:0:0:
  351,74,198725,5,0,0:0:0:0:
  352,170,198875,1,0,0:0:0:0:
  354,268,199025,1,0,0:0:0:0:
  439,214,199175,1,0,0:0:0:0:
  267,218,199475,1,0,0:0:0:0:
  425,217,199625,1,0,0:0:0:0:
+ 209,69,199924,6,0,P|157:51|117:58,1,75
  262,105,200225,1,0,0:0:0:0:
  43,129,200525,6,0,P|84:114|120:116,1,75
  0,152,200825,1,0,0:0:0:0:
  104,223,201125,1,0,0:0:0:0:
+ 143,237,201201,1,0,0:0:0:0:
+ 187,243,201274,1,0,0:0:0:0:
+ 234,237,201351,1,0,0:0:0:0:
  276,219,201425,1,0,0:0:0:0:
  179,161,201575,2,0,P|186:120|242:72,1,112.5
  251,68,201875,1,0,0:0:0:0:
  354,169,202025,1,0,0:0:0:0:
  426,77,202325,5,0,0:0:0:0:
```

Figure 4. *File A before the algorithm (Left) and after the algorithm (Right)*

The key observations are as follows:

1. In all three cases, the hidden data survived the official osu! submission pipeline. The LSB modifications to the time parameter were preserved without being normalized or stripped by server-side processing.
2. The amount of data successfully concealed matched the theoretical capacity of 1 bit per hit object. For example, File B with 1166 hit objects allowed embedding a total of 145 bytes, which accommodated the 44-byte test message with room to spare.

3. A playthrough of the level does not show any obvious signs of tampering, and no obvious anomalies were introduced into the **`.osu`** file structure. The modifications are not detectable by manual inspection of the file in a text editor, which satisfies the basic stealth requirement.

V. Conclusion

The proposed steganographic algorithm successfully addresses the limitations of previous work by providing a method for data concealment within **`.osu`** files that is robust against official server-side post-processing. By leveraging the Least Significant Bit (LSB) of the hit object's **`time`** parameter, the algorithm achieves a reliable, though modest, capacity of 1 bit per hit object. For an average beatmap, this translates to a capacity of approximately 0.34% or 125 bytes, which is sufficient for concealing short text strings or URLs.

Additionally, the 1ms shift in object timing falls significantly below the human JND for temporal intervals, which is estimated at 10ms or more in gaming contexts. While the modification results in a 5% reduction of the effective hit window at the highest difficulty levels (reducing it from 20ms to 19ms), this trade-off is negligible for the vast majority of players and does not disrupt the core gameplay experience.

Future work will be to improve the capacity by making use of other fields in the **`HitObject`** syntax, as well as to make use of the other sections in the **`.osu`** file format. Further research in the affects of this algorithm to the gameplay experience should also be conducted by surveying the opinion of skilled players. (After all, this paper provides no such insight and only an analysis based on psychoacoustic literature is given.)

References

[1] Alamsyah, J. I. (2025). *Steganographic implementation in osu! osz beatmap files for data concealment.* Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung.

[2] Osu! Development Team. (2025), *The osu! Wiki: Beatmap* https://osu.ppy.sh/wiki/en/Beatmap, [Accessed: December 26th 2025]

[3] Osu! Development Team. (2025), *The osu! Wiki: osu! Judgement System* https://osu.ppy.sh/wiki/en/Gameplay/Judgement/osu%21, [Accessed: December 26th 2025]

[4] Osu! Development Team. (2025), *The osu! Wiki: .osu (File Format)* https://osu.ppy.sh/wiki/en/Client/File_formats/osu_%28file_format%29 [Accessed: December 26th 2025]

[5] Repp, B. H., & Su, Y. H. (2013). Sensorimotor synchronization: a review of recent research (2006–2012). Psychonomic bulletin & review, 20(3), 403-452.

[6] Reddy, V. L., Subramanyam, A., & Reddy, P. C. (2011). Implementation of LSB steganography and its evaluation for various file formats. Int. J. Advanced Networking and Applications, 2(05), 868-872.

[7] Fisher, N. I. (1993). Statistical analysis of circular data. Cambridge, UK: Cambridge University Press.

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, 26 Desember 2025

[illegible]

Muhammad Rasheed Qais Tandjung
13522158