ChatGPT

Improving Payload Handling and Training for Audio Watermarking

Handling Variable Payload Length (Up to 64 Bytes)

The system is designed to handle a maximum payload of 64 bytes per 1-second audio segment, but it should also accept smaller payloads. To achieve this, **any payload shorter than 64 bytes is padded up to 64 bytes** before encoding. This ensures the encoder/decoder always processes a fixed 64-byte block. In practice, if a payload is, say, 40 bytes, the remaining 24 bytes can be filled with padding (e.g. zeros or random dummy data) so that the total reaches 64 bytes. The padded payload is then fed into the error-correction and embedding pipeline. On decoding, the system always attempts to retrieve a full 64-byte block and can strip away padding to recover the original message length. In the code, the `decode_eul` function explicitly truncates the recovered data to the expected payload length (using `expected_bytes=64` for a 64-byte payload) [1] [2]. This modification guarantees that regardless of input size, the decoder outputs a 64-byte sequence (with padding if needed), simplifying downstream processing.

Why pad to 64 bytes? Standardizing to 64 bytes simplifies the design of the neural watermarking model. The model can be trained with a consistent bit-length (64 bytes = 512 bits) target, avoiding dynamic output sizes. Internally, the training code fixes `cfg.rs_payload_bytes = 64` [3], and always builds a 64-byte "ground truth" payload by padding shorter data up to this length [4]. This way, the **payload bit tensor is always 512 bits** long, making the neural network's job consistent. The STFT configuration (44.1 kHz audio, `n_fft=882`, hop=441) provides ~100 time frames per second [5], which is sufficient to embed 512+ bits when spread across frequency bins. By padding and standardizing payload length, we can later generalize to variable payloads easily: the script can accept any payload $\leq$64 bytes, pad it to 64, then proceed with encoding/decoding normally. This adds flexibility for future use cases without changing the core model.

Redundancy and Error Correction (Repetition *r*=3 and RS Coding)

To ensure the hidden message survives audio processing (like mastering or compression), the system uses **redundant embedding and error-correction coding**. Currently, the repetition factor is **r = 3**, meaning **each symbol/bit is embedded in three distinct time-frequency locations** for redundancy [3]. In other words, the watermark bits are repeated across the spectrogram such that if one location is degraded (e.g. by noise or EQ changes), the other placements can still carry the information. This strategy is backed by research: repeating the same watermark bits in multiple locations improves robustness [6], albeit at the cost of more embedding capacity and the need to later "locate" and combine these bits. Our implementation plans to use techniques like CRC-based validation and majority voting to reconcile the triple-embedded bits during decoding (ensuring all three copies of a bit agree) [7] [8]. By defining *r* clearly: *r = 3* indicates triple redundancy per payload bit in the time-frequency map, which significantly boosts the likelihood of recovery under distortions.

In addition to repetition, we apply **Reed–Solomon (RS) forward error correction** to the payload bytes before embedding. Reed–Solomon coding adds parity bytes that allow the decoder to correct certain errors in the recovered bitstream. The original pipeline uses an RS(167,125) code: 125 data bytes are

expanded to 167 bytes by adding 42 parity bytes [9]. This code can correct up to 21 byte errors per 167-byte block. However, if our actual payload is only 64 bytes, using the full RS(167,125) is inefficient – it would pad the message from 64 up to 125 bytes internally, producing a 167-byte codeword (1336 bits) [10], many of which correspond to padding. We will adopt an **RS code better suited for 64 bytes**. For example, by shortening the code to RS(106,64) (still with 42 parity bytes), we get a 106-byte codeword. This retains the same strong error-correction capability (42 parity bytes able to fix 21 errors) but only produces 106 bytes of output, which is ~848 bits – much closer to our available embedding capacity. In practice, the RSCodec library can handle shortened codes by treating the omitted bytes as known zeros [9]. The encoding function will pad the 64-byte payload to 64 bytes (no change, since 64 is the target) and then generate 42 parity bytes, outputting a 106-byte encoded message. The **interleaving** step (depth = 4 by default) then shuffles these bytes across time so that burst errors are spread out [11], further protecting against contiguous bit drops. On the decoding side, the RS decoder will reconstruct the original 64 bytes from any subset of the 106 bytes that survive (tolerating up to 21 random byte errors). We will update the training script to use this optimized RS coding for 64 bytes, which means also adjusting the `per_eul_bits_target` to the full code length (848 bits) so that **all encoded bits are embedded** (avoiding the previous issue where not all code bits were utilized). This comprehensive redundancy—**triple embedding plus RS(106,64) with interleaving**—prioritizes payload recovery, giving the model multiple ways to retrieve the data even after the audio undergoes transformations.

Audio Processing and Dataset Considerations

Our model operates on 44.1 kHz audio with a short-time Fourier transform window of 20 ms (`n_fft=882`) and 50% overlap (`hop_length=441`) [12]. This high-resolution spectrogram yields about 100 frames per second and 441 frequency bins, providing a dense grid of ~44,100 time-frequency slots per second (though not all are usable for embedding). We leverage a psychoacoustic model to choose where to embed bits: the Moore–Glasberg analyzer computes the masking threshold per critical band [13], and an adaptive bit allocator assigns more bits to frequency bands with higher "masking headroom" (where the host audio is quiet relative to human hearing thresholds) [14]. This ensures that the watermark is placed in perceptually insignificant components of the audio, preserving sound quality. Because the training set spans **many genres (classical, rock, hip-hop, etc.)**, the watermarking strategy must be robust to different spectral profiles. The adaptive allocation helps achieve this by dynamically adjusting which frequencies/time segments get payload bits for each input track. For loud or spectrally busy music, the model will use fewer or lower-amplitude watermark signals, whereas for sparse audio it can utilize more. The **sync marker** (a faint known bit pattern each second) is also embedded to help the decoder align and detect where each 1-second payload starts [15]. This is crucial in real-world usage when dealing with streaming audio or when the watermark needs frame synchronization.

With a diverse genre dataset, training the model on all sorts of music ensures it learns to distribute and retrieve bits under a wide range of conditions (dense mixes, solo pieces, heavy bass, etc.). It essentially learns a **content-aware hiding strategy**. During training, we monitor the perceptual loss (e.g. MFCC cosine distance) to confirm that the embedding remains inaudible [16]. The goal is to make the watermark **transparent to listeners yet resilient**. The multi-genre training data will force the model to generalize, finding a balance that works across different audio characteristics.

Training Strategy: Prioritizing Payload Recovery and Robustness

Currently, the training has plateaued – the model isn't reliably recovering bits even without any data augmentation or simulated attacks. To address this, we will re-focus the training objective to **prioritize**

payload bit recovery above all else, even if it means easing off on the audio fidelity constraints initially. Several adjustments will help:

- **Loss Function Weighting:** We will increase the weight on the bit-recovery loss and temporarily reduce or disable the perceptual loss. In the config, the bit loss has weight `w_bits=1.0` while the perceptual loss is small (`w_perc=0.01`) [17]. We can go further by setting `w_perc=0` (no perceptual penalty) for the first phase of training. This lets the model find an embedding that works for recovery without being penalized for audible changes. Once the BER (bit error rate) drops and the model starts successfully extracting the 64-byte payload, we can gradually reintroduce perceptual loss to fine-tune audio quality. This curriculum (easy task first, then making it harder) is reflected by the "warmup" settings already present (e.g. `warmup_perc_epochs=2` to ignore perceptual loss for 2 epochs) [18]. We might extend that warmup if needed until we see near-zero BER on training data. The **bit-wise binary cross-entropy loss** drives the model to output the correct payload bits; by emphasizing it, we signal that **"reconstructing the hidden message is the top priority."**

- **Verify Gradient Flow in Decoder:** We need to ensure the decoder network's gradients truly reflect bit recovery performance. The current implementation converts decoded bits to 0/1 and uses `BCEWithLogits` on them [19], which might not fully leverage the continuous logits from the invertible network. We will adjust this to use the raw decoder output (e.g. the real-valued spectrogram coefficients at the chosen slots) before thresholding, so the loss is differentiable and can properly guide the model. In other words, instead of thresholding the decoder output to hard bits inside the forward pass, we compute the bit loss on soft values (logit form) so that small changes in the model output produce a gradient when the bit is incorrect. This fix should help the model overcome the plateau, as it will get clearer error signals when a bit is wrong (rather than a flat zero gradient from a discretized prediction).

- **Gradual Complexity Increase:** After the model starts recovering bits on clean audio, we will incrementally introduce **data augmentation that simulates mastering and other distortions**. The ultimate goal is to make the watermark survive real-world audio processing, so we do plan to add an "attack layer" during training as done in robust neural watermarking research [20]. For example, we can apply random equalization, dynamic range compression, slight reverb or noise, and lossy compression to the watermarked audio before decoding (only during training). This will train the network to handle such transformations. However, adding these too early can confuse training (making it much harder to learn the embedding from scratch). So we will hold off on heavy augmentations until the base model reliably transmits the payload in unaltered audio. Once we reach that point, we can gradually mix in simulated mastering effects (e.g., a dynamic range compressor with typical mastering settings, or an EQ change) to **teach the model to be invariant to those**. Research suggests that including a variety of common attacks in training greatly improves robustness [20] [21]. By following a staged approach – first achieve capacity, then add robustness – we ensure the model doesn't get stuck.

- **Adjust Network Capacity if Needed:** If the above steps still don't yield progress, we may consider increasing the model's capacity (for instance, using more invertible blocks or parameters in the INNWatermarker) or training for more epochs. The current setup uses 8 invertible blocks and trains for 20 epochs [12] [22], which might be insufficient for the complexity of the task (hiding 512 bits while remaining inaudible is quite challenging, especially since typical audio watermarking schemes often embed as low as ~20–100 bits per second [23] [24]). We will monitor the training loss and bit error rate; if it plateaus early, a learning rate schedule or extended training time (along with the other fixes) can help the model escape local minima. The training script already has a learning rate warmup and plateau-based bumping strategy [25]. We

might tweak those hyperparameters (for example, allow a lower minimum LR or use a smaller initial LR to stabilize training).

By implementing these changes, our training will initially **"overfit" to making the payload recovery perfect on clean audio**, effectively learning the simplest way to hide/recover bits. Then, we will **gradually steer it towards more imperceptible and robust solutions**. The redundancy measures (r=3 repetition and strong RS code) give the model some breathing room – it can afford some bit errors as long as they're correctable. The combination of triple embedding and RS means the system can tolerate many individual bit flips and still recover the full 64-byte message [26] [6]. This should be emphasized to the model: even if not every embedded instance of a bit comes out clean, as long as the majority or the RS-decoded result is correct, the payload will decode successfully. In training, we'll track **payload success rate** (whether all 64 bytes are recovered exactly) in addition to raw bit accuracy [27]. Our aim is to quickly push that metric toward 100% on training data by prioritizing it in the loss function.

Preserving Music Mastering and Audio Quality

While payload recovery is the top priority in early development, we must also ensure that the watermarking process **preserves the audio quality, even through mastering**. "Music mastering" typically involves dynamic range compression, EQ, and normalization applied to the final mix – processes that could potentially alter or remove the embedded watermark. Our strategy to handle this is two-fold: **robust embedding** and **post-training augmentation tests**. Robustness is already addressed via redundancy and error correction. For mastering-specific resilience, we will likely train the model with some mastering-like augmentations (as noted above) once it can handle the basic scenario. For example, we can simulate a mastering chain in training (apply a mild compressor and a tonal balance EQ to the watermarked audio before decoding). This will encourage the model to place bits in parts of the audio less affected by these operations (for instance, embedding more in mid-frequency ranges that are stable, and not relying solely on very dynamic transients that a compressor would squash). Our use of a psychoacoustic mask margin ("amp_safety" and masking threshold margin) can be tightened to ensure the watermark is well below audible thresholds [28] – this inherently helps survive mastering because if the watermark is buried under the audio's noise floor or mask, even aggressive mastering won't suddenly make it audible or remove it entirely.

Finally, after training, we will validate the system on real mastering scenarios. We'll take watermarked audio and run it through actual mastering plug-ins or processes, then attempt decoding. The expectation is that with **the planned training improvements, the payload will still decode correctly**. If not, we'll identify failure modes and iterate. For instance, if we find the watermark tends to fail after heavy bass boost EQ, we might adjust the bit allocator to be more conservative in low-frequency bands. The **multi-band adaptive allocation** already leans towards embedding in less audible bands [29] [30], and we will refine this as needed. By continually testing with mastering adjustments, we ensure the model truly *"preserves music mastering"* in the sense that the final mastered audio still contains the hidden payload transparently.

In summary, our approach is to make the system highly flexible and robust: accept variable payload sizes (pad to 64 bytes), use strong redundancy (r=3) and appropriate RS coding for error correction, and modify the training process to overcome current plateaus by emphasizing bit recovery. As we progress, we will incrementally reintroduce perceptual constraints and add mastering-like augmentations to teach the model to handle real-world audio processing. This balanced focus should yield a watermarking model that *prioritizes payload recovery* while still maintaining audio fidelity through all stages of production.

Sources: The implementation details and parameters are based on the provided code and design specs [31] [5], and our strategy aligns with recent research on neural audio watermarking which emphasizes redundancy and attack simulation for robustness [20] [6], as well as industry guidelines on watermark payload capacity and imperceptibility [23].

---

[1] [9] [10] [11] [13] [14] [26] [29] ingest_and_chunk.py
file://file_00000000c4e0622f8fcb4e8c253a3a95

[2] [3] [4] [5] [12] [15] [16] [17] [18] [19] [22] [25] [27] [28] [31] training_new.py
file://file_000000003dd8622fa34b585bf895cb7b

[6] [20] [21] aclanthology.org
https://aclanthology.org/2024.emnlp-main.258.pdf

[7] [8] error_correction.py
file://file_0000000056b8622fb533ef2496094ecc

[23] Audio Watermarking: A Comprehensive Review
https://thesai.org/Downloads/Volume15No5/Paper_141-Audio_Watermarking_A_Comprehensive_Review.pdf

[24] An adaptive and large payload audio watermarking against jittering ...
https://www.sciencedirect.com/science/article/abs/pii/S0045790625002642

[30] SoundSafe.ai_Data Ingestion Plan & Chunking Strategy.pdf
file://file_00000000f288622fb955207beb371ed9