# a recipe for steganogravy

▶ author's note

with ai scrapers and government agencies roaming the internet and snarfing down every last byte (hoping to profit when you mistakenly publish useful information online), it’s gotten harder to share data without it becoming a future liability.

one wrong step and you find yourself accidentally contributing to automating your own job, having your identity stolen, or offending the kind of person that seems to always be complaining about other people being offended.

what if we could hide data in a place *no one would ever think to look*? what if we could submerge our delicious morsels of knowledge in a flavorless slop so devoid of nutritional value even the most ravenous ai agents would spit it out?

# tbrockman/recipe-blog-encoding

is a vibe-coded (and at least partially plagiarized[1]) python cli that allows you to encode data as completely natural language[2] using neural linguistic steganography.

given a shared prompt and a model, it can hide your secrets where they're least expected: **recipe blog introductions**.

example.sh

```
python main.py encode \
  --message "https://www.nokings.org/" \
  --stego-file stego_output.txt \
  --model "models/Qwen3-32B-Q4_K_M.gguf" \
  --prompt "<|im_start|>user
You are a blog author writing your stereotypical recipe introduction, aiming to maximize the chances of your recipe being seen by gaming your articles SEO, without being too verbose. Output only the recipe introduction and nothing else, using a maximum of 12 paragraphs. Choose a random recipe.<|im_end|>
<|im_start|>assistant
<think>

</think>"
```

which produces something like the following:

example-out.md

Looking for a quick and delicious way to impress your family and friends? This **One-Pan Garlic Butter Chicken with Herbed Potatoes** is the answer. Packed with flavor and easy to make, this dish is perfect for weeknight dinners or weekend feasts. What makes it stand out? It requires only one pan, minimizes cleanup, and allows the rich flavors to infuse perfectly while everything cooks.

The chicken is seasoned with garlic, onion powder, and a touch of paprika, then seared to golden perfection. But the real star here is the buttery herbed potatoes, tender and slightly crispy, soaking up every ounce of aromatic goodness. A quick toss with fresh parsley and oregano adds the warmth that makes this recipe so special.

This recipe is optimized for search engines with keywords like *easy one pan chicken recipe*, *garlic butter chicken*, and *herbed potato side dish*. Whether you're a seasoned chef or a beginner in the kitchen, this dish is a breeze to make and guaranteed to deliver big flavor.

[ ... ]

which any reader, knowing the original prompt and model used, can use to recover the political messaging hidden in your favorite garlic butter chicken recipe:

```
python main.py decode --stego-file stego_output.txt --model
"models/Qwen3-32B-Q4_K_M.gguf" --prompt "..."

# some processing ... ⌛
```

```
============================================================
RECOVERED SECRET MESSAGE:
============================================================
https://www.nokings.org/
============================================================
```

just how grandma would have made it.

# how it works

▶ author's note

the implementation largely follows arithmetic coding steganography. at a high-level, you can imagine the following:

1. we convert our secret into a binary fraction which represents a point somewhere on a number line between [0, 1).
2. we use the model's next token probability distribution to carve out adjacent intervals on the line, where the width of each interval is proportional to the token's probability.
3. we repeatedly choose tokens whose interval contains our point, narrowing the interval further and further, until enough of the leading bits of the start and end points of the interval agree, such that we've encoded our message.

here’s a simple example with a 3-bit secret:

```
secret: 1 0 1
  → binary fraction: 0.101
  → point on [0, 1): 0.625

step 1: interval [0, 1)
  "The"      [0,    0.4)
  "Looking"  [0.4,  0.55)
  "This"     [0.55, 0.8)   ← 0.625
  "A"        [0.8,  1)
  → select "This", narrow to [0.55, 0.8)

step 2: interval [0.55, 0.8)
  " recipe"  [0.55, 0.7)   ← 0.625
  " is"      [0.7,  0.76)
  " One"     [0.76, 0.8)
  → select " recipe", narrow to [0.55, 0.7)

step 3: interval [0.55, 0.7)
  " is"      [0.55, 0.625)
  " uses"    [0.625, 0.67) ← 0.625
  " for"     [0.67,  0.7)
  → select " uses", narrow to [0.625, 0.67)

[0.625,  0.67) in binary:
[0.101..., 0.101...]
 ^^^        ^^^
leading bits agree: 1 0 1 → secret recovered ✓
```

the generated text would then read: “this recipe uses”

decoding is just the reverse: run the same model with the same prompt, reconstruct the probability distribution at each step, and read the secret bits back out by checking which tokens were used. it's important to note that both sides need the **exact same model, quantization, top-k, and prompt** – any mismatch and the distributions diverge, producing garbage.

# limitations

### it's pretty wasteful

you're loading massive models to encode and decode a small amount of information, slowly, at < 2-3 bits/token. it's not a great use of compute.

### bpe tokenization

it turns out that if you pick a token during encoding, decode it to text, and then re-tokenize that text, you don't always get the same token back. for instance, if the text so far tokenizes to `[..., "hel"]` and the model picks the `"lo"` as the next token, the combined text `"hello"` might re-tokenize as a single `"hello"` rather than `"hel" + "lo"`. then, when decoding, the decoder sees a completely different

token at that position and everything after it diverges.

claude's fix : add a filter that, at each step, tests whether a candidate token would survive a round-trip through decoding and tokenization. tokens that wouldn’t are excluded from the cdf before any interval math happens. you lose some encoding capacity, but you can be certain that if your message can be encoded, it can also be decoded.

**model end-of-sequence can be reached before the secret is fully encoded**

question: what do we do if the prompt we’ve chosen doesn’t provide a path to generate sufficient tokens to encode our secret, converging on end-of-sequence before giving us enough bits?

answer: 🤷 choose a better prompt and try again.

**security**

the prompt acts as a shared key, but it’s a leaky one. the generated text is statistically conditioned on the prompt, where the prompt is partially revealed

by its own output (which is generally not seen as an ideal property for encryption methods).

threat model : passing a note to your friend about which girl you like in class, through an untrusted intermediary

**local llm only**

not that it’s not possible to use any remote apis (it should be so long as they provide sufficient determinism and logits), local’s just all that was implemented.

# try it out

available on google collab or from source:

```
git clone https://github.com/tbrockman/recipe-blog-encoding/
cd recipe-blog-encoding
python3 -m venv env
source env/bin/activate
pip install -r requirements.txt
python main.py --help
```

have fun cooking ✌️

**footnotes:**

1. subsequent investigation suggests that `artkpv/arithmetic-coding-steganography/` is likely where claude found inspiration for the implementation ↩

2. as determined by the shared prompt and model ↩