## Pulsar: Secure Steganography through Diffusion Models

Tushar M. Jois  
City College of New York  
tjois@ccny.cuny.edu

Gabrielle Beck  
Johns Hopkins University  
becgabri@cs.jhu.edu

Gabriel Kaptchuk  
Boston University  
kaptchuk@bu.edu

### Abstract

Widespread efforts to subvert access to strong cryptography has renewed interest in steganography, the practice of embedding sensitive messages in mundane cover messages. Recent efforts at provably secure steganography have only focused on text-based generative models and cannot support other types of models, such as diffusion models, which are used for high-quality image synthesis. In this work, we initiate the study of securely embedding steganographic messages into the output of image diffusion models. We identify that the use of variance noise during image generation provides a suitable steganographic channel. We develop our construction, Pulsar, by building optimizations to make this channel practical for communication. Our implementation of Pulsar is capable of embedding $\approx 275\text{–}542$ bytes (on average) into a single image without altering the distribution of the generated image, all in the span of $\approx 3$ seconds of online time on a laptop. In addition, we discuss how the results of Pulsar can inform future research into diffusion models. Pulsar shows that diffusion models are a promising medium for steganography and censorship resistance.

### 1 Introduction

Years of sustained effort by the security and cryptography communities has led to widespread deployment of strong, secure communication technologies, including end-to-end encrypted messaging [PM, Wha17] and TLS [Res18]. While these technologies prevent high-resource attackers from viewing or manipulating the contents of communications, they do nothing to hide that the communication itself is occurring. In areas where encrypted communication technologies are blocked or cause for suspicion, leaking this metadata can have deadly consequences. As governments’ suspicion of encrypted communication continues to grow around the world, it is important to proactively develop new techniques that can complement our existing secure communication technologies and provide security and privacy to individuals at increased risk; if this development is not done proactively, the technology will be too immature for deployment when it is needed.

Steganography [Sim83] allows a sender to embed a sensitive message into a mundane context such that only the intended receiver is able to detect and extract the message, making it the ideal tool for communicating in areas where encryption cannot be safely used. Steganography can be used to transform a conversation that might be seen as “subversive” into one that would not arouse suspicion—even when the content of the conversation itself is directly monitored by authorities. Importantly, this prevents censors from selectively blocking content (or selectively blocking encrypted communications). Steganography can also enable digital “dead drop” deployments, in which encoded messages are left on the public internet and the intended recipient can recover the message without leaving evidence of direct communication.

Although the value of efficient, secure steganography tools has been evident for decades, concrete constructions have long been lacking. Formal techniques, stemming from the cryptographic literature, provide provable hiding guarantees but cannot be concretely deployed, either because they are inefficient or make unrealistic assumptions. Heuristic steganographic techniques, on the other hand, have been deployed in practice, but are often vulnerable to simple statistical attacks. For example, JPEG steganography (in which bits of the sensitive message are embedded into the low order bits of pixels’ values) is often considered a viable steganographic choice, but the manipulation of low order bits is detectable [FPK07, SCC07].
Steganography for generative models. A new line of research attempts to marry the formal techniques with the heuristic approaches by steganographically encoding messages into the output of generative, machine-learning models. This provides a systematic divide between the provable guarantees and the heuristic assumptions such that they can be analyzed separately. These steganographic schemes can ensure that an adversary cannot distinguish between typical model output and model output carrying a sensitive message, but make no formal claims about an adversary attempting to detect if a message is generated with the help of a model (as opposed to created manually by the sender). While this approach falls short of providing end-to-end security, this is the best one could hope for when embedding into contexts where explicit descriptions of the statistical nature of the communication channel cannot be described (e.g., natural human language, art, or photographs). In these cases, encoded messages can only hope to achieve indistinguishably with the best known approximation we have of the communication channel, and machine learning models are the best approximation of natural human language, art, or photographs currently available.

Existing steganographic proposals [KJGR21, DCW+23] are designed to work with popular model structures, like neural networks and general adversarial networks. These models iteratively generate output by producing an explicit probability distribution of different tokens (e.g., words, pixels) that could follow the given prompt. During typical operation, a single token is sampled from this distribution as output, appended to the prompt, and the updated prompt is then fed back into the model. This process continues until the output is the desired length. The randomness used to sample from the distribution is pulled from an arbitrary high entropy source.

When steganographically embedding a sensitive message into the output, the sampling is done as a function of the sensitive message such that the receiver can infer the bits of the sensitive message. For example, the sender might encode the leading bits of the message into a token using an arithmetic encoding scheme or a Huffman code [KJGR21]. To ensure that this steganographic encoding process does not change the statistical profile of the model output, the sensitive message is generally encrypted using a pseudorandom cipher,1 making it, in effect, a high entropy source.

Steganography for diffusion models. Although text is a natural medium to consider for steganographic communication, embedding into text produces stegotext (the steganography equivalent of ciphertext) that is significantly longer than the initial message. This is because natural language text is actually quite low entropy and steganographic encoding rates are bound by the entropy in the communication channel. As such, sending even relatively short sensitive messages steganographically might require sending paragraphs or pages of model-generated text. While transmitting this much text might occasionally be appropriate, in many cases this will break steganography’s illusion. This limitation makes existing, text-based approaches insufficient, motivating the need for encoding techniques that work with other media.

After text, images are the next media into which we might want to embed steganographic messages. Importantly, images do not share the same limitations as text: images have significantly higher entropy than text and it is commonplace to exchange or post large image files. These properties mean that it should (in principle) be possible to steganographically send large amounts of information using machine learning-generated images without arousing suspicion. Building on this intuition, Ding et al. [DCW+23] generated steganographic images using ImageGPT, a neural network designed to produce images based on a prompt. While transformer networks remain the most effective generative models in the natural language processing domain, a new model architecture, diffusion models [SDWMG15, SME20, HJA20, DN21, RBL+22], have proven to produce higher quality output for images. Diffusion models have quickly captured the public’s imagination and have become the de facto option for machine-learning image generation. Unlike neural networks, diffusion models generate all the pixels in the output image at the same time. This significant departure from typical transformer network architecture means that existing steganographic approaches cannot be adapted to efficiently work for diffusion models.

In this work, we initiate the study of steganographic embedding mechanisms that can work with diffusion models. Our techniques allow for the production of steganographic communication tools that can send large

---

1 Some proposals [GGA+05, SSSS07, YHC+09, CC10, CC14, FJA17, VNBB17, YJH+18, Xia18, YGC+19, HH19, DC19, ZDR19] fail to properly encrypt the message before encoding, leading to an insecure construction.
amounts of data without arousing suspicion. Additionally, we find that encoding with diffusion models can be faster than encoding with other networks, because encoding using existing techniques requires querying a transformer model many times while diffusion models are one shot. Additionally, the justification for using machine learning models in steganography relies on the assumption that the models are the best available approximation of human creation. Using other models in steganographic image generation undermines this assumption, as diffusion models have surpassed their quality.

Our contributions. We initiate the study of steganographic encoding schemes for images generated by diffusion models. In doing so, we have several concrete contributions:

- A principled study of steganography for diffusion models. As we are initiating the study of steganographic for diffusion models, we begin our work by providing a principled study of the various opportunities for hiding data that diffusion models afford. We begin by noting that cryptographically secure steganography is fundamentally about randomness recovery. As such, we can systematically iterate through the entropy sources consumed by a machine learning model and identify those that are most promising for hiding data. This study ensures that our construction, Pulsar, is optimized to encode as much data as possible.

- Pulsar, a novel steganographic encoding scheme for diffusion models. We design a novel, symmetric key steganographic encoding scheme that encodes data into the output of diffusion models that operate in the pixel space. Pulsar embeds data into images by mapping pixels in the image to bits in the message; when sampling Gaussian noise for each pixel, the encoder uses one of two pseudorandom functions, one for pixels mapped to a zero bit and one for pixels mapped to a one bit. This results in a noisy, randomness recovery scheme, which can be improved using error correcting codes. We implement Pulsar and integrate it with existing diffusion models, showing that it can encode hundreds of bytes of plaintext information in a 256 × 256 pixel generated image. Our implementation encodes faster than state-of-the-art neural network steganographic systems.

- Recommendations for steganography-friendly machine learning models. We note that the empirical nature of machine learning research can lead to some model architectures incorporating semi-arbitrary design choices, while also significantly impairing the ability to steganographically embed messages into model output. Reflecting on our study of steganography for diffusion models, we highlight several ways in which model architecture can be improved to better support steganographic encoding. We hope that this can inspire designers of future model architectures to test if models can be made steganography friendly without reducing output quality.

Deployment scenario and threat model. In this work we assume a similar deployment scenario and threat model as recent work on symmetric key, model-based steganography [KJGR21, DCW+23]. Namely, a sender and receiver generate shared key material out-of-band and select a diffusion model to use as a covert channel. We assume that their communications are monitored by a computationally powerful adversary (e.g., a state actor) who also has access to the selected diffusion model. The sender and receiver wish to disguise the contents of their communication such that the adversary cannot determine if their exchanged messages contain typical model output or steganographically encoded messages. As this work focuses on laying out a feasibility result, we make the simplifying assumption that the adversary does not launch active attacks and must distinguish based on observing encoded messages.

We do not attempt to formalize the ability of the adversary to distinguish between model output and “normal” human communication. For full-scale deployments of steganography, deployment designers must carefully consider how well steganographically generated messages fit in with existing communication channels (in addition to incorporating other best practices from cryptographic messaging like forward security). While this is certainly a limitation of our work, we note that humans exchanging the outputs of generative

\(^2\)Pulsar is designed for models that operate in the pixel space, as opposed to a compressed latent space. We discuss this difference in Section 3.1.
models has become increasingly common. For example, professional communications may be generated with the help of text models like ChatGPT [Kor23] and social media was flooded with “AI art” [Vin22] after the release of Stable Diffusion [RBL+22] and similar models. These developments mitigate the risks associated with encoding information in model output.

2 Related Work

Steganography maps a message into a stegotext, a set of elements from a chosen target distribution. Typically, this distribution is mundane such that a censor would not find it suspicious. This strengthens encryption, in which a ciphertext can have an arbitrary distribution and does not aim to hide the fact that it is a ciphertext.

Steganography was first formalized by Simmons [Sim83] and has since been the subject of a tremendous amount of theoretical research. The theoretical literature has focused on establishing the universal feasibility of steganography for arbitrary stegotext distributions, provided the distribution has some amount of entropy. For example, prior work has shown that universal steganography can be realized with information-theoretic security [AP98, ZFK+98, Mit99, Cac00], computational security, [HLv02, vH04, BC05] and statistical security [SSM+06a, SSM07, SSM+06b]. There are also symmetric-key [Cac00, HLv02, RR03] and public-key constructions [vH04, BC05, Le03, LK03] discussed in the literature. More recently, techniques deeply related to steganography have been studied as a tool to achieve security when an adversary is able to compel a receiver to decrypt ciphertexts [HPRV19, PPY22].

A complementary line of research has studied the feasibility of concretely efficient, deployable steganography. Generally, to achieve this result, these steganographic constructions either rely on heuristic security analyses, e.g., protocol obfuscators like obfs4/ScrambleSuit [WPF13] and domain fronting [FLH+15], or can only embed messages into very specific covertext distributions, e.g., pseudorandom bit streams. For example, SkypeMorph [MLDG12], CensorProofer [WGN+12], and FreeWave [HRBS13] all tunnel Tor [RSG98, DMS04, Tor] traffic through Voice-Over-IP (VoIP) traffic, which is usually encrypted with a pseudorandom cipher. Other examples include Format Transforming Encryption [LDJ+14, DCRS13, DCS15, OYZ+20], which requires implementers to explicitly describe the statistical properties of the target distribution.

Recently, there has also been work attempting to leverage generative neural networks to instantiate concretely efficient universal steganography by embedding messages into the output of the model. By using machine learning models, these works cleanly separate their heuristic guarantees from their formal ones. Specifically, these protocols offer no formal guarantees about how easy it is to detect that content has been produced by a machine learning model, but can make formal arguments about the statistical shifts induced by the steganographic embedding. This line of work started in the machine learning community with constructions that modified the output distribution of the model, thus failing short of any notion of provable security [Bal17, HWJ+18, Har18, SAZ+18, Cha19, WYL18]. Building on these works, Kaptchuk et al. [KJGR21] and Ding et al. [DCW+23] showed how to encode steganography messages into neural network output without modifying the output distribution. Kaptchuk et al. [KJGR21] accomplish this by repeatedly re-encrypting the message using a stream cipher, using the resulting pseudorandom ciphertext bits to sample tokens from the neural network’s probability distribution, and leveraging an arithmetic encoding scheme to enable a receiver to recover the message bits. Ding et al. [DCW+23] are able to achieve a better encoding rate by using the message bits to “rotate” the neural network’s probability distribution before sampling.

As we discuss in the next section, the techniques presented in these works cannot be adapted to work with diffusion models, as they make implicit assumptions about the model architecture of generative models that do not hold for diffusion models.

3 Steganography for Diffusion Models

We start by systematically exploring the opportunities that diffusion models provide for steganographic encoding.
3.1 Diffusion Models

Diffusion models [SDWMG15, SME20, HJA20, DN21, RBL+22] are a novel generative model architecture tailored to produce visual media. Rather than generate output one token at a time, as is common in the neural networks used in language models, diffusion models take in a random seed (and possibly a prompt) and produce the entire image at once. The model predicts the changes that would need to be applied to this seed to make the image closer to the desired distribution, e.g., photo-realistic images or fantasy art. We provide a visual representation of the generation process for a typical diffusion model in Figure 1.

Diffusion models are trained on a large corpus of training examples. Each example is a pair of images $(\text{img}, \text{img} + \mathcal{N}(0, I))$, where img is an image from the desired output distribution and img + $\mathcal{N}(0, I)$ is a modified version of img with Gaussian noise added. Many examples are created from each img, with multiple values of $I$; the most extreme examples appear to be pure noise while others are quite close to the output image. Given these examples (along with the value of $I$), the model is trained to predict a value from $\mathcal{N}(0, I)$ such that it can be subtracted from the example to recover img.

During image generation (Algorithm 1), the pipeline is reversed: a seed $s_0$ is sampled from a Gaussian distribution, and the model predicts a noise residual $\text{pred}$ such that $s_0 - \text{pred}$ is in the target distribution. Rather than remove $\text{pred}$ all at once, the model takes $\text{pred}$ as an indication of the direction in which it must modify the $s_0$ to get it to the desired distribution. As such, the model subtracts a function of $\text{pred}$ rather than $\text{pred}$ itself. For example, the models that we work with compute $s_1 = s_0 - \epsilon \cdot \text{pred}$, for some $0 < \epsilon < 1$. This process is repeated a fixed number of times to produce the values $s_2, \ldots, s_{\text{final}}$; recall that the model
Algorithm 1: Typical Diffusion Model Operation

Output: Generated Image $\text{img}$

Set $s_0 \sim N^{n \times n}$

for $1 \leq i < t$ do

\[ r_i \sim N^{n \times n} \]

\[ \text{pred}_i \leftarrow \text{Model}(s_{i-1}) \]

\[ s_i \leftarrow \text{Scheduler}_i(s_{i-1}, \text{pred}_i; r_i) \]  
// Deterministic Final Schedule

\[ \text{pred}_t \leftarrow \text{Model}(s_{t-1}) \]

\[ \text{img} \leftarrow \text{Scheduler}_{\text{final}}(s_{t-1}, \text{pred}_t) \]

Output $\text{img}$

Figure 2: Typical diffusion model image generation

was trained on many levels of noised images and can predict the noise that should be removed from $s_1, s_2, \ldots$ even though they are “less noisy” than $s_0$.

This iterative process is called *denoising* [SME20, HJA20] and is organized according to a *schedule*, which we denote with a set of functions $\{\text{Scheduler}_i\}_{i \in [t]}$. Each function determines how the prediction from the model should be applied to the state $s_{i-1}$ to produce an updated state $s_i$ (e.g. applying the $\epsilon$ scaling factor). Different concrete instantiations of diffusion models also might incorporate different modifications into each step of the schedule. One common modification is to apply additional Gaussian noise, called *variance noise*, to $s_i$ in each step $i < t$, ensuring that image generation is *non-deterministic*. The final image is then generated deterministically by $\text{Scheduler}_t$.

In the above, we have described the diffusion process as though the seed $s_0$ and intermediary states $s_1, s_2, \ldots$ are all elements in the image space (i.e., if the model is trained to generate $256 \times 256$ color images, then $s_i$ is also a $256 \times 256$ color image). In practice, the space of $s_i$ can be different than the image space; some diffusion models (e.g. Stable Diffusion [RBL+22]) operate in a latent space, a compressed space that is more succinctly able to represent and capture the complexity of images. When operating in the latent space, the model adds a final mapping step that maps $s_{\text{final}}$ into a final image using a variational autoencoder (VAE) [KW13].

### 3.2 Integrating Steganography

Given the structure of diffusion models, we can now turn to the task of studying the steganographic opportunities that structure affords. We first discuss why existing steganographic techniques cannot be directly adapted to work with this model structure. Next, we review classical steganographic techniques before turning to the opportunities themselves.

**Why existing techniques fail short.** A natural approach to steganography for diffusion models would be to extract the intuition behind steganographic approaches designed for other machine learning model architectures and adapt it for this new architecture. For example, Meteor [KJGR21] and Discop [DCW+23] are steganographic approaches for transformer-like architectures, which are the leading models for text generation. Both follow the same template: given some initial prompt (representing the context in which the encoded message will be sent) the sender uses the machine learning model to produce an explicit probability distribution over the token (e.g., a word) to be appended to the prompt. During typical generation, neural-network-like architectures would select a token from this probability distribution at random. When encoding a message steganographically, both constructions instead sample the token as a function of the message.\(^3\) Importantly, the receiver can use knowledge of the probability distribution and the selected token to efficiently

\[^3\text{Meteor encrypts the message first, to ensure the message is uniformly distributed. Discop instead uses an information theoretically secure approach to selecting the token, and thus does not require encryption.}\]
recover a few bits of the message (in both cases, the receiver can recover a variable number of bits). The sender can repeat this process, appending samples to the prompt until the entire message is encoded.

Critical to the approaches of Meteor and Discop is the assumption that the receiver gets access to the results of many (conditionally linked) sampling events. Although it is theoretically possible for a token to encode a large number of bits, the chances of this happening become vanishingly small as the number of bits increases. The throughput of the steganographic encoding scheme is tightly linked to the entropy in the channel: the expected number of bits to encode cannot exceed the instantaneous entropy in the model output distribution. For example, if a message is 128 bits long, it could only be encoded into a single token if the chances of choosing that token were $2^{-128}$ (assuming the sender and receiver share no prior information about the message distribution).

The structure required by Meteor and Discop is not present in diffusion models. Specifically, diffusion models do not produce explicit probability distributions from which the image is sampled. Moreover, the entire output that is accessible to the receiver is produced in a single shot, making subdivision (e.g., encoding into each pixel independently) impossible. Even adapting more classical steganographic schemes to diffusion models seems challenging. For example, foundational steganographic constructions (e.g. [Cac00, HLv02, vH04]) rely on the use of universal hash functions to subdivide the output space into segments that correspond to different bit sequences and then use rejection sampling to find a sample output that hashes to the desired message. While it is possible to use such a technique with diffusion models, the entire message would need to be encoded into the singular output image. Even though an image might be high entropy enough to encode a large number of bits, using rejection sampling to find an image that hashes to the desired bits is computationally infeasible.

**Opportunities for steganography.** While there are numerous techniques that can be used to construct steganography, all of them rely on embedding the message into the entropy in the channel. Because the space of messages that can be sent over a particular communication channel is fixed by the distribution within which the encoded message is supposed to hide, the choice of which message to send is the only subliminal channel available. Therefore, when evaluating the opportunities for steganographically embedding into the output of diffusion models, we begin by examining the sources of entropy.

When generating images, diffusion models use two sources of entropy: (1) the initialization seed, and (2) the noise added during each step of the schedule. To understand the viability of using these sources, we must understand how using different randomness for each changes the image seen by the receiver. We study this question empirically, as randomness is largely a means to an end within these models.

1. **Initialization seed:** It is tempting to try and embed the message into the initialization seed used by the model. Because this seed is both large and high entropy, it is natural to assume that we could steganographically embed a large amount of information into the seed. Unfortunately the image generation process is not invertible. In our experiments with real-world diffusion models, we found that each step of the scheduler was injective, but not surjective. Even if only some information was lost in each step of the scheduler, most concrete instantiations of diffusion models have as many as 50 steps, each one of which is unpredictably lossy. As such, it is unclear how to steganographically embed information into the seed while allowing it to be recoverable.

2. **Variance noise added by scheduler:** Each step of the schedule adds a small amount of Gaussian noise to the current state $s_i$. In our experiments, we found that modifying the noise added to a single pixel in state $s_i$ can result in a small, local change in the equivalent pixel in $s_{i+1}$. When operating in the latent space, the decompression results in changes not only in the equivalent point in the latent space, but also to surrounding points; this effect is amplified when the resulting latent space is mapped into the image space.

The localized nature of (2) provides an opportunity for an efficient steganographic channel, which we investigate below.
Figure 3: Sample output from the celebahq model. Note that the differences shown in (c) are not changes introduced to encode a message, but a reflection of the entropy in the generative model that we exploit to embed steganographically.

4 Pulsar: A Symmetric Key Steganographic Scheme for Diffusion Models

We now describe Pulsar, a symmetric-key steganographic encoding scheme for pixel-space diffusion models.

4.1 Intuition

Pulsar leverages the steganographic channel enabled by the addition of Gaussian noise in the schedule. As discussed above, resampling the noise for a pixel in the final round of the schedule can result in localized change to the output image. When this resampling is a function of the message, the receiver can observe these changes and recover the message.

In more detail, the sender and receiver share key material that allows them to derandomize their model operation such that they can keep their models perfectly synchronized. This allows the sender and receiver to have their models generate the same exact image. We divide this key material into three concrete PRG keys: a seed key $k_s$, and then a pair of reference image keys $k_0$ and $k_1$. The seed key is used by both the sender and receiver to synchronize their models until the final step of the scheduler, i.e., they use $k_s$ to sample the model’s initial seed and sample the variance noise in the first $t-1$ scheduler iterations. The sender and receiver can then generate two reference images $\text{img}_0$, $\text{img}_1$ by sampling the variance noise for the $t^{th}$ iteration of the sampler with $k_0$ and $k_1$ respectively. To illustrate this process, we include concrete reference images $\text{img}_0$ and $\text{img}_1$ in Figures 3a and 3b.

In order to build intuition, we make the following simplifying assumptions: (1) the model’s state is simply the pixels of the image itself, and (2) the final iteration of the scheduler operates on each element of the state independently, i.e., each element of the final state is a function of exactly the corresponding element in the previous state and neighboring elements have no effect on one another. If these assumptions were true, a clear steganographic channel would emerge. The sender would divide the message $m$ they want to send into bits $m_0, m_1, \ldots$ and then sample variance noise for each element of the state from $k_0$ or $k_1$ depending on the bit value of the message. For example, if message bit $m_i$ were $b$, then the sender samples the variance noise for the $i^{th}$ element using $k_b$. The receiver can then guess about the value of $m_i$ by comparing the final output of the model to $\text{img}_0$ and $\text{img}_1$; whichever reference image the final output more closely resembles in the $i^{th}$ pixel determine the receiver’s guess about $m_i$. This would allow the sender and receiver to encode a single bit per pixel.

These simplifying assumptions serve a unified purpose: minimizing the error rate in the channel. Indeed, even without these assumptions, there may still be errors, as the reference images may be identical in particular pixels. For example, see Figure 3c, in which we give a heatmap of the difference between the
Algorithm 2: Pulsar Encode

**Input:** Plaintext Message $m$, Key $k$

**Output:** Stegotext Image $img$

```
Parse $(k_s, k_0, k_1) \leftarrow k$ // Offline Phase

Set $s_0 \overset{\$}{\leftarrow} N_{k_s}^{n \times n}$

for $1 \leq i < t - 1$ do

$r_i \overset{\$}{\leftarrow} N_{k_s}^{n \times n}$

$pred_i \leftarrow \text{Model}(s_{i-1})$

$s_i \leftarrow \text{Scheduler}_i(s_{i-1}; r_i)$

Compute rate $\leftarrow \text{EstimateRate}(s_{t-2})$

$m_{ECC} \leftarrow \text{ECC}.\text{Encode}(m, \text{rate})$ // Online Phase

for $0 \leq j < |m_{ECC}|$ do

if $m_{ECC}[j] = 0$ then

$r_{t-1}[j] \overset{\$}{\leftarrow} N_{k_0}$

else

$r_{t-1}[j] \overset{\$}{\leftarrow} N_{k_1}$

pred$_{t-1} \leftarrow \text{Model}(s_{t-2})$

$s_{t-1} \leftarrow \text{Scheduler}_{t-1}(s_{t-2}, \text{pred}_{t-1}; r_{t-1})$

// Deterministic Final Schedule

pred$_t \leftarrow \text{Model}(s_{t-1})$

img $\leftarrow \text{Scheduler}_t(s_{t-1}, \text{pred}_t)$

Output $img$
```

Figure 4: Pulsar Encode

two example reference images. Black pixels in this heatmap show that resampling the Gaussian noise has negligible effect on the final pixel value. Without assumption (2), sampling variance noise from different sources might have unpredictable effects, as the changes to one particular pixel might “contaminate” the signal in neighboring pixels. Finally, if the state does not have a clean correspondence to the image, it may not be clear where the receiver should look to recover information about how variance noise is sampled.

To recover from these errors, we introduce the use of a binary error correcting code. A binary error correcting code introduces redundancy into a message such that when the message is transmitted over a noisy channel (i.e., a channel that introduces bit flips), the receiver can run some recovery algorithm and output the initial message. Careful use of this error correcting code allows us to continue using the same intuition above, but ensures that the message can be recovered from the receiver, even without our simplifying assumptions.

We begin by removing assumption (2) and accept that changes in the way variance of noise is sampled for a particular pixel may impact neighboring pixels—or, indeed, any pixel; we simply treat this as additional error to be corrected. The sender and receiver synchronize their models as before, reaching the final iteration of the scheduler. Before attempting to encode the message, the sender first encodes the message with a binary error correcting code, and then proceeds as before. Only one question remains: what rate should the sender use for their error correcting code, i.e., how much redundancy should the sender add into the message. If the error rate is high, the sender needs to increase the redundancy, as more information will be erased by the channel. On the other hand, adding unnecessary redundancy is wasteful, as fewer message bits will fit in the fixed capacity of the image.

To fix the rate of the error correcting code, the sender estimates the error rate. While it might be possible to use a fixed error rate, our experiments found that different images have very different error rates that are
Algorithm 3: Pulsar Decode

**Input:** Stegotext Image img, Key k

**Output:** Plaintext Message m

| Parse \((k_s, k_0, k_1) \leftarrow k\) | // Offline Phase |
|---|
| Set \(s_0 \leftarrow N_{k_s}^{n \times n}\) |
| for \(1 \leq i < t - 1\) do |
| \(r_i \leftarrow N_{k_s}^{n \times n}\) |
| \(\text{pred}_i \leftarrow \text{Model}(s_{i-1})\) |
| \(s_i \leftarrow \text{Scheduler}_i(s_{i-1}; r_i)\) |
| Compute rate \(\leftarrow \text{EstimateRate}(s_{t-2})\) |
| \(\text{pred}_{t-1} \leftarrow \text{Model}(s_{t-2})\) |
| // Generate reference image 0 |
| Set \(r_{0_{t-1}} \leftarrow N_{k_0}^{n \times n}\) |
| Set \(s^{0}_{0_{t-1}} \leftarrow \text{Scheduler}_{t-1}(s_{t-2}, \text{pred}_{t-1}; r_{0_{t-1}})\) |
| \(\text{pred}^0_{t} \leftarrow \text{Model}(s^{0}_{t-1})\) |
| \(\text{img}^0_{0} \leftarrow \text{Scheduler}_t(s^{0}_{t-1}, \text{pred}^0_{t})\) |
| // Generate reference image 1 |
| Set \(r_{1_{t-1}} \leftarrow N_{k_1}^{n \times n}\) |
| Set \(s^{1}_{1_{t-1}} \leftarrow \text{Scheduler}_{t-1}(s_{t-2}, \text{pred}_{t-1}; r_{1_{t-1}})\) |
| \(\text{pred}^1_{t} \leftarrow \text{Model}(s^{1}_{t-1})\) |
| \(\text{img}^1_{0} \leftarrow \text{Scheduler}_t(s^{1}_{t-1}, \text{pred}^1_{t})\) |
| for \(0 \leq j < |m_{\text{ECC}}|\) do | // Online Phase |
| if \(|\text{img}[j] - \text{img}^0_{0}[j]| < |\text{img}[j] - \text{img}^1_{0}[j]|\) then |
| \([m_{\text{ECC}}[j]] \leftarrow 0\) |
| else |
| \([m_{\text{ECC}}[j]] \leftarrow 1\) |
| Output \(m \leftarrow \text{ECC} \cdot \text{Recover}(m_{\text{ECC}}, \text{rate})\) |

Figure 5: Pulsar Decode

dependent on the structure of the image (discussed more in Section 5). As such, we instead have the sender estimate the error rate for each generated image by encoding random messages into the last iteration of the scheduler and measuring the number of errors by attempting to recover the encoded message. This process allows the sender to get a good estimate on the error rate, and then use conservative parameters on the error correcting code to ensure recovering the message is possible with high probability.

Relaxing assumption (1) appears to be more challenging. As discussed above, many concrete instantiations of diffusion models (e.g., Stable Diffusion) operate on a latent space, a compressed representation of the pixel space, in order to be more efficient. This design choice necessitates the use of an expanding mapping between the latent space and the pixel space. While we find that the approach outlined above can function for such models, the result is highly inefficient. As such, we restrict attention to diffusion models that operate directly in the pixel space. We included a more detailed discussion of diffusion models in the latent space in Section 7.
4.2 Pulsar Description

**Notation.** We assume that the sender and receiver (and adversary) both have access to the same diffusion model that operates in the pixel space and generates $n \times n$ pixel images with 3 color channels, resulting in an image space of $[0, 255]^{n \times n \times 3}$. Let this diffusion model have an associated scheduler $\text{Scheduler} : [0, 255]^{n \times n \times 3} \times \mathbb{Z} \rightarrow [0, 255]^{n \times n \times 3}$. Let $\mathcal{N}(0, I)$ be a Gaussian, and $\mathcal{N}^{n \times n}(0, I)$ be $n \times n$ copies of the underlying Gaussian. We denote a keyed Gaussian as $\mathcal{N}_k(0, I)$, in that samples drawn from the Gaussian are drawn using entropy generated by a pseudorandom generator $[BM82, Ruh17]$ with key $k$. As the norm and standard deviation of the distribution are fixed model parameters, we omit them. We abuse notation and write $s + \mathcal{N}$ to denote sampling a value from $\mathcal{N}$ before adding it to $s$.

We index into images and messages using square bracket notation, i.e., $x[i]$ is the $i$th element $x$. For images, we assume some arbitrary pixel ordering to facilitate single dimension indexing. Let $\text{ECC}$ be an error correcting code made up of two algorithms: encoding algorithm $\text{ECC}.\text{Encode}$ and a recovery algorithm $\text{ECC}.\text{Recover}$. Both take in a message and an error rate. We expand on error correcting codes and the properties we require of them in Section 5.

**Pulsar encoding and decoding.** The encode and decode algorithms for Pulsar are described in Algorithm 2 and Algorithm 3 respectively. When encoding, the sender runs the typical image generation algorithm for the first $t-2$ iterations. The sender then estimates the error rate for encoding and adds redundancy to the message using an error correcting code $\text{ECC}$. Then, based on the bits of the expanded message $m_{\text{ECC}}$, the sender samples variance noise either using $k_0$ or $k_1$. Then, the final image is generated using $\text{Scheduler}$. When decoding, the receiver estimates the error rate in the image and generates reference images $\text{img}_0$ and $\text{img}_1$. In order to recover $m_{\text{ECC}}$, the receiver compares each pixel to the reference images, and guesses the corresponding bit of $m_{\text{ECC}}$ accordingly. $m$ can then be recovered by applying $\text{ECC}.\text{Recover}$.

**Offline-online paradigm.** We note that much of the logic of Pulsar can be run before the sender knows the message. Specifically, the first $t-2$ iterations of the schedule are completely independent of the message. As such, this computation can be run offline, and the results can be stored until the sender has a message they want to encode. At that time, the sender can run the online parts of Pulsar, which include $\text{ECC}$, sampling the variance noise $r_{t-1}$, and final image generation. We highlight the offline and online phases of encoding and decoding in Algorithm 2 and Algorithm 3.

**Proof sketch.** To show security, we consider a real-or-random notion of chosen hidden-text security, similar to in Meteor [KJGR21], in which the adversary must distinguish if it gets access to a typical model generation oracle or a steganographic encoding oracle. The security of Pulsar is a straightforward reduction to the security of the pseudorandom generator used to sample Gaussian noise. Informally, a computationally bounded adversary cannot not distinguish between real randomness and the pseudorandomness output by the PRG. As such, the adversary cannot distinguish between variance noise that is honestly sampled and that is sampled from different PRGs. The proof is simply a set of hybrids in which each sampling event when embedding $m_{\text{ECC}}$ is swapped from true randomness to the appropriate PRG. The final hybrid derandomizes the first $t-2$ iterations of the schedule, swapping honest randomness for pseudorandomness generated using $k_s$.

5 Error Correction for Pulsar

We now investigate how to build $\text{EstimateRate}$ and select an $\text{ECC}$ for Pulsar to maximize throughput.

5.1 Channel Error Structure

A naive approach to computing $\text{EstimateRate}$ would simply have the sender generate a random message, encode it into an image, locally attempt to recover it and measure the decoding error rate. Based on this

---

4 We assume square images for simplicity, but other dimensions are trivially supported.
error rate, they would select an ECC that can successfully encode at that error rate. For instance, an estimate for the error rate of the image in Figure 3d is high (29.1%) but we can build a code for this high error rate.

To improve on this baseline, we investigate the sources of error a little more closely. An error occurs in Pulsar when, for a bit \( b \), the difference in the corresponding pixel between the encoded \( \text{img} \) and \( \text{img}_{b'} \) is closer than that of \( \text{img} \) and \( \text{img}_b \), where \( b \neq b' \). There is, therefore, an inverse relationship between the absolute magnitude of differences between the references images \( \text{img}_0 \) and \( \text{img}_1 \) and the error rate: roughly speaking, the greater the difference for a pixel, the less likely decoding that pixel will result in an error. The naive approach above assumes that the distribution of errors is uniform throughout the image.

A closer look at the heatmap in Figure 3c reveals that the differences between \( \text{img}_0 \) and \( \text{img}_1 \) are not uniformly distributed in the image—there is structure. The heatmap is dark for the regions of the image that make up the background in Figure 3d, creating a semi-visible silhouette of the face. In this significant background region, the differences between \( \text{img}_0 \) and \( \text{img}_1 \) are consistently low. Because Pulsar leverages differences in order to steganographically embed, the background will contribute more to the error rate than the rest of the image. Put another way, any single bit encoded in this part of the image will have a much higher error rate.

This notion of “difference regions” with varying magnitudes is more obvious when looking at Figure 6a and associated heatmap in Figure 6b. In addition to the background, parts of the generated face are also less discernible, and the estimated overall error rate is 34.1%: over one in three bits is incorrectly decoded. Not only are there distinct difference regions, but they also differ from image to image.

### 5.2 Variable Error Correction

The naive approach in Section 5.1 fails to fully utilize the high entropy present in some regions of the image. The model is more detailed in some areas, leading to high entropy (and low error). In other areas, the model is less detailed, and so the available entropy is lower. These regions are essentially guaranteed to introduce high amounts of error, diluting the overall error rate. Unlike most applications of error correcting codes, however, in Pulsar we actually can estimate where the errors will happen; we can divide the communication channel into smaller channels, each of which has a predictable error rate.

An obvious solution would be to leverage the sender’s and receiver’s shared knowledge of the image structure to only encode into those regions that contain low error rates. This would entail selecting a code for low error regimes and using that on these regions. This approach would work, but is also inefficient. The number of low error regions is image-dependent; an image like Figure 3d has many more than an image like Figure 6a. Additionally, this approach excludes “medium” error rate regions, i.e., where the error rate is between low and high. These regions will have relatively higher error rates, but could still encode some amount of data.

So, what we need is a library of error correcting codes, selecting the best code for a region based on its error rate. Intuitively, if we can define these difference regions, we can estimate the error rates within each region separately, and then select error correcting codes per-region instead of over the image overall. This approach helps us more efficiently utilize the low error rate regions of the image to encode bytes.

Our region-based error rate estimation for EstimateRate proceeds as follows. First, we use the naive error estimate strategy on each difference region of the image individually. To do so, we compute \( l \) trail encodings at the current model state on random messages. These trial encodings are then subsequently decoded to generate difference heatmaps (like those seen in Figures 3c and 6b). To systematically define the difference regions from these heatmaps, we divide the pixels in the image into \( u \) buckets based on the magnitude of the difference shown in the heatmap. Each bucket represents one unique difference region of the image—all of the pixels in a bucket have a similar magnitude of their differences and therefore (roughly) similar error rates during Pulsar encoding. We chose to bucketize the differences rather than apply image analysis (e.g., convolution to find region boundaries) because the latter would involve more expensive techniques and may miss difference regions that are non-contiguous.

We calculate the error rate for each difference regions, as defined by its bucket, based on the messages from the trial encoding. rate is now a list of regions with associated error rates, we select the appropriate ECC for each region. To do so, we start with the lowest error regions, and use codes suitable for those
regions. We then work our way up the difference regions, selecting a code from our library that works at each region’s calculated error rate. Eventually, we reach a region that cannot be encoded using our codes due to its significant noise, and we consider encoding finished.

Using this approach, Figure 3d can encode 620 bytes of information, and Figure 6a can encode 200 bytes of information. We also note that EstimateRate is still a fully offline operation (Section 4.2), as the difference regions for an image can be estimated without any information about the message to encode. Both the sender and receiver can perform multiple runs of EstimateRate in preparation for a communication.

Note that our EstimateRate has two parameters: the number of buckets for each estimate $u$, and the number of estimates $l$. We experimentally determine these parameters, and present our results in Section 6.1. A more formal description of EstimateRate can be found in Appendix A.

5.3 Identifying Candidate Codes

Our variable error correction approach requires building a library of error correcting codes that support differing error rates, such that more efficient codes can be used in difference regions with low error. We approach the development of this library heuristically, as developing optimal codes for our setting is a difficult problem beyond the scope of this work. We note, however, that there may be significant room for Pulsar performance improvement by identifying superior codes.

Error correcting codes. In this work we will consider linear $[N,k,d]_q$ codes, which are codes defined over a field $\mathbb{F}_q$ of size $q$. The size of the codeword $N$ is called the block length, $k$ is the dimension and $d$ is the distance of the code. The rate of a code is given by $\frac{k}{N}$ and the unique decoding radius (the number of errors a code can tolerate while still uniquely recovering the original codeword) is $\lfloor \frac{d}{2} \rfloor$. The channel we have identified can be characterized as a binary symmetric channel ($BSC_p$), which is parameterized by the probability $p$ that an error is introduced in each bit. More formally, if $Y$ is the output of the channel and $X$ is the input, $\Pr[Y = b|X = 1-b] = p$ for $b \in \{0,1\}$. The capacity for $BSC_p$—which is the highest possible rate we could hope to achieve—is $1 - H_2(p)$ where $H_2(x) = -x \cdot \log_2(x) + (1-x) \cdot \log_2(1-x))$.

Error correcting for binary channels. When we quantize the image into $u$ different difference regions, the result is that our communication channel is actually the concatenation of $u$ channels $Ch_1, \ldots, Ch_u$. We assume that each channel $Ch_i$ functions as a binary symmetric channel $BSC_{p_i}$, for some probability $p_i \in [0,1]$. Our goal is to come up with an algorithm so that, given block lengths $N_1, \ldots, N_u$, we identify $u$ codes $ECC_1, \ldots, ECC_u$ such that $\forall j \in [u]$ (1) $ECC_j$ can recover from errors induced by a $BSC_{p_j}$ channel with all but low probability, (2) $ECC_j$ has as high a rate as possible, (3) $ECC_j$ has a practically efficient encoder.
and decoder, (4) ECC$_j$ has an an alphabet size of $2^8$. Requirement (4) is not actually a pure necessity, but ease implementation.

A good approach to binary codes is using a linear concatenated code [For65]. A linear concatenated code $\text{ECC}_{\text{out}} \circ \text{ECC}_{\text{in}}$ uses two codes in tandem, where messages are first using $\text{ECC}_{\text{out}}$ and then each resulting symbol is encoded using $\text{ECC}_{\text{in}}$. To be precise, let the outer code $\text{ECC}_{\text{out}}$ be an $[N, k, d]_q$ where $Q = q^{k'}$ for some $k' \in \mathbb{Z}^+$ and the inner code $\text{ECC}_{\text{in}}$ an $[N', k', d']_q$ code. To encode a message $\vec{m} \in \mathbb{F}^k$, first produce $\vec{c} = (\vec{c}_1, \ldots, \vec{c}_N) = \text{ECC}_{\text{out}}.\text{Encode}(\vec{m})$; the final codeword is $\text{ECC}_{\text{in}}.\text{Encode}(\vec{c}_1), \ldots, \text{ECC}_{\text{in}}.\text{Encode}(\vec{c}_N)$. The resulting concatenated code is an $[NN', kk', z]$ linear code where $z \geq dd'$. For binary concatenated codes, we set $q = 2$.

One approach to finding a binary concatenated codes with rate approaching capacity, involves fixing an outer code achieving the singleton bound (i.e., Reed-Solomon) and searching (via brute-force) to find a smaller inner code meeting capacity on the BSC$_p$ channel [GRS12]. The resulting concatenated code decoder uses a maximum likelihood decoder for $\text{ECC}_{\text{in}}.\text{Recover}$ and then applies an efficient unique decoding algorithm for $\text{ECC}_{\text{out}}.\text{Recover}$. While this would produce high quality codes, it is not computationally feasible to find such an inner code probabilistically and finding “optimal” codes is beyond the scope of establishing the feasibility of our approach$^5$. As such, we take a more computationally realistic approach in this work: for a given channel BSC$_p$, we instead do manual search over two different families of inner codes—Reed-Muller and Hamming codes—meeting our requirements from the preceding paragraph$^6$. For each code, we empirically test its performance on the BSC$_p$ for 1,000 inputs to get an accurate error rate for decoding. Consider the random variable $X_j$ to be the number of errors that occur when $m_j$ are decoded after being put through the channel BSC$_p$. For a given outer code block length $N$, we then find the minimum $z \in \{0, \ldots, N\}$ so that $\Pr(X_N \leq z) \geq 0.99$. We then set the outer code’s distance and dimension to allow recovery of up to $z$ errors and consider only those valid codes achieving the best possible rate$^7$. The full code for doing this was implemented in SageMath. See Appendix A for the list of codes we identified for Pulsar.

6 Implementation and Evaluation

We implemented Pulsar using PyTorch and the diffusers library [Hug] using the DDIM scheduler [SME20]. We use SageMath [The23] to provide error correcting codes. To ensure that both the sender and receiver have synchronized states, we use HMAC-DRBG [BK15] to seed our random number generators. We will release our code and benchmarks as artifacts for the community upon publication.

Our implementation uses the following diffusion models (which generate the following images): church (churches and other places of worship), celebahq (celebrity faces), bedroom (bedrooms), cat (cats), all of which were published by Google on Hugging Face [Good, Gooc, Gooa, Goob]. Example images generated by each model can be found in Appendix B.

We run our implementation on (1) “Desktop”, a desktop tower with an AMD Ryzen 9 3900X CPU, NVIDIA GeForce RTX 4070 Ti GPU, and 32 GiB of RAM running Arch Linux, and (2) “Laptop”, a MacBook Pro with an M1 Pro system-on-chip with 16 GiB of RAM running macOS Ventura. We believe the Laptop benchmarks are more representative of consumer hardware.

6.1 Parameter Selection

In Section 5.2, we discussed our optimized EstimateRate approach that variably encodes using different error correcting codes based on the available entropy in the image, and in Section 5.3, we discussed how we determined the library of codes to support this approach. There are two parameters in EstimateRate that we

$^5$As a direction for future work, we could feasibly use Forney codes. This is theoretically more optimum except practical decoding time could drastically increase with these codes as one approaches capacity.

$^6$Note, the inner code does not necessarily need a decoder that is efficient in its block length, just in the block length of the concatenated code.

$^7$This means setting $d \approx 2z$ and calculating $k$ based on $d$ and $N$. 
have yet to determine: the number of buckets to generate for our difference regions $u$, and the total number of estimates to generate $l$. We experimentally determine these parameters for our Pulsar implementation.

**Number of buckets in each estimate $u$.** We first want to find the appropriate number of buckets to define difference regions in `EstimateRate`. We can increase the number of buckets to get more granular regions, but this increases execution time and may result in over-fitting. So, we fixed $l = 1$ and ran 100 iterations of `EstimateRate` on Desktop for multiple candidate $u$ values on the `church` model. Our results can be found in Table 1a. $u = 100$ appears to be the best balance between mean message length, variability of the message length, and estimation time, so we choose that as the parameter.

**Number of estimates $l$.** The other parameter is the number of estimates generated during `EstimateRate`. We want to know if increasing offline estimates improves the online success rate – the percentage of encoded images that can be successfully decoded. Each estimate involves an iteration of a model, so we want to ensure that these additional estimates are worth the longer computation. In each trial, we fix $u = 100$ (based on our evaluation for $u$ above), and run `EstimateRate`, varying $l$. We then encode a message using the estimated regions and see if the generated image decodes back into the same message. This shows if more estimates lead to a better chance of a successful encoding. We run 100 of these trials on Desktop and the `church` model, and show our results in Table 1b. Interestingly, our results demonstrate that additional estimates do not meaningfully improve the success probability of encoding. So, we use $l = 1$ for the remainder of this evaluation.

### 6.2 End-to-End Benchmarks

We run 100 trials of our implemented Pulsar system from end to end for each of the models we consider. In each trial, we run the offline step of `EstimateRate`, encode a random message at the sender into a 16-bit PNG image, and decode this image back into the original message. Note that we implement and provide results for the offline step of Algorithm 3; since Algorithm 2’s offline step is a subset of that of Algorithm 3, we can use the same offline state for both encoding and decoding. We run our experiments on both Desktop and Laptop.

**Throughput.** We first seek to understand the throughput: the number of bytes per image that can be communicated steganographically with Pulsar. Table 1c summarizes this information, with information about the mean $\bar{x}$ and standard deviation $s$ of the bytes encoded in each image, along with the rate at which the receiver can successfully decode a message. As mentioned above, decoding can fail if the estimate is not representative of the actual message encoding. The expected throughput is therefore this decoding success rate times the mean bytes per image $\bar{x}$. While the success rate is not 100%, we do note that it is high enough

![Image](image.png)

<table>
<thead>
<tr>
<th>Buckets</th>
<th>Estimation Time (sec)</th>
<th>Message Length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
<td>$\bar{x} = 2.84, s = 0.04$</td>
<td>$\bar{x} = 443.09, s = 165.58$</td>
</tr>
<tr>
<td>50</td>
<td>$\bar{x} = 3.08, s = 0.04$</td>
<td>$\bar{x} = 502.77, s = 174.44$</td>
</tr>
<tr>
<td>100</td>
<td>$\bar{x} = 3.10, s = 0.04$</td>
<td>$\bar{x} = 542.13, s = 192.75$</td>
</tr>
<tr>
<td>125</td>
<td>$\bar{x} = 3.10, s = 0.04$</td>
<td>$\bar{x} = 542.15, s = 196.67$</td>
</tr>
<tr>
<td>150</td>
<td>$\bar{x} = 3.11, s = 0.04$</td>
<td>$\bar{x} = 533.41, s = 192.61$</td>
</tr>
</tbody>
</table>

(a) Averages from 100 trials of our $u$ experiment.

<table>
<thead>
<tr>
<th>Estimates</th>
<th>Estimation Time (sec)</th>
<th>Success Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$\bar{x} = 2.86, s = 0.00$</td>
<td>96.0%</td>
</tr>
<tr>
<td>3</td>
<td>$\bar{x} = 3.79, s = 0.01$</td>
<td>93.0%</td>
</tr>
<tr>
<td>5</td>
<td>$\bar{x} = 4.49, s = 0.01$</td>
<td>93.0%</td>
</tr>
<tr>
<td>10</td>
<td>$\bar{x} = 6.22, s = 0.02$</td>
<td>87.0%</td>
</tr>
<tr>
<td>30</td>
<td>$\bar{x} = 13.17, s = 0.06$</td>
<td>95.0%</td>
</tr>
</tbody>
</table>

(b) Averages from 100 trials of our $l$ experiment.

<table>
<thead>
<tr>
<th>Model</th>
<th>$\bar{x}$</th>
<th>$s$</th>
<th>Rate</th>
<th>Throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>church</td>
<td>541.70</td>
<td>190.82</td>
<td>94.0%</td>
<td>$E[X] = 509.20$</td>
</tr>
<tr>
<td>celebahq</td>
<td>351.02</td>
<td>89.04</td>
<td>98.0%</td>
<td>$E[X] = 344.00$</td>
</tr>
<tr>
<td>bedroom</td>
<td>275.00</td>
<td>112.70</td>
<td>95.0%</td>
<td>$E[X] = 261.25$</td>
</tr>
<tr>
<td>cat</td>
<td>386.32</td>
<td>288.52</td>
<td>98.0%</td>
<td>$E[X] = 378.59$</td>
</tr>
</tbody>
</table>

(c) Bytes encoded per-image for all models for 100 runs.

Table 1: Experimental results showing impact of $u$ and $l$ and the performance of different models.
to support throughputs in the hundreds of bytes. Moreover, because Pulsar is a symmetric key scheme, the sender will know when decoding will fail and can abort.

We also seek to understand how the number of bytes per image varies between the models. We chart the distributions for this value for each model in Figure 7a. Each model has a different distribution, which is reasonable considering that each model generates substantially different images. For example, note that celebahq is mostly tightly centered, likely because human faces have a core structure that does not deviate significantly. On the other hand, cat has high variance, which we attribute to the model’s lower quality (see Appendix B).

**Time.** Table 2 contains runtime for each phase of Pulsar for each model on both Desktop and Laptop. The offline phase is about the same for every model, which is in line with how diffusion models work. Even though each model generates different images, all models have the same number of steps (and therefore PyTorch calls to their weights), and should have the same runtimes. We see that decoding is generally slower than encoding. We can also see that the runtime of Laptop is about three times that of Desktop, and that encoding and decoding times vary between the models. In particular, the encoding and decoding times are proportional to the number of bytes the model generates (see Table 1c).

To investigate these results more, we separate each runtime by the constituent parts of Pulsar, and chart it in Figure 7b. “Model” is time spent generating an image using diffusion in PyTorch, “Sage” is time spent in SageMath’s error correcting code routines, and “Other” is time spent outside of the prior two tasks. The offline phase is dominated by the model, as it is generating the estimate that will be used for the online phases. Laptop’s slower runtime can thus be attributed to its lack of model processing ability relative to Desktop.

Encoding and decoding are instead dominated by SageMath. So, models that require more calls to SageMath are slower in encoding and decoding; the models that call SageMath more are those with a higher capacity for encoding bytes. For example, the church model takes the most time to encode/decode because it needs to encode/decode more bytes. Thus, our results are consistent. As an aside, our implementation performs a call to SageMath as a subprocess, which means our runtimes include a relatively lengthy startup for the SageMath interpreter. A more efficient implementation (i.e., a direct implementation of the underlying ECCs without SageMath) can help reduce this runtime further.

**Comparison to text model methods.** We now compare our results to that of prior work on text-based steganography. We specifically compare Pulsar to [DCW+23], where the authors provide seconds per bit results for their construction, as well as those of other modern baselines such as Meteor [KJGR21]. Their hardware configuration is similar to our Desktop one.

Based on the evaluation in Table II of [DCW+23], the Pulsar construction represents a performance
Another consequence of having a deterministic final scheduler step is that we are only able to use one
iteration of the diffusion model being applied, which perturbs the encoded information and introduces errors into the encoding. In
Pulsar, we solve this through the use of error correcting codes. If the final scheduler step were randomized,
we would no additional model iteration to apply after encoding, the decoding process would be less error
prone. This lack of errors would in turn allow for smaller (or even no) error correcting codes, resulting in higher utilization of the image.

Another consequence of having a deterministic final scheduler step is that we are only able to use one
color channel in the generated image. Applications of the model propagates changes from one color channel
to the others. In our experiments, we found that any information embedded into the variance noise of a
second channel is effectively wiped out when the model is applied. As such, two thirds of the image is
Reversible model iterations. Instead of introducing randomness to the final scheduler iteration, it would also be sufficient to design model iterations that are invertible. That is, given access to the final image, it would be possible to recover intermediary states of the model or even the initial Gaussian noise that form the seed for the diffusion model. As discussed in Section 3.2, this seed is an excellent source of entropy that could be leveraged for steganographic embedding, but current model structures prevent the receiver from recovering the seed efficiently. Each model iteration applied to the seed modifies the image and each step is not efficiently reversible due to the nature of the neural network model.

Recovering the initialization seed would require non-trivial changes to the diffusion model structure. The internal model structures would have to change to be reversible. We note that non-linearity within the model will always make inversion more difficult, and it may be that highly non-linear models are inherent in designing performant models. Alternatively, model designers could train pairs of models that are able to both generate images (forwards) and recover seeds (backwards). If designing models with this property were possible, we could apply Pulsar’s embedding strategy to the seed directly. In this case we would have (ideally) $n \times n \times 3$ bits of information for encoding per image, exceeding our current results (Table 1c).

Detailed and contextually appropriate models. Pulsar is best able to encode information images that are highly detailed. For example, both Figures 3d and 6a are realistic images of human faces. As the respective difference heatmaps in Figures 3c and 6b show, however, Figure 3d is more detailed (e.g., at the top of the head), which means it can encode more information in Pulsar, and is therefore a better target for encoding than Figure 6a. As such, having access to models that produced highly detailed images is desirable.

We note that there is a distinction between “detailed” and “realistic”, and, of course, the quality of the generated images is critical. For instance, cat is a detailed model, but its relatively low quality images (see Appendix B) may make censors more suspicious. Moreover, the number of bytes it can encode has high variance, as seen in Figure 7a. Thus, future models need to prioritize detail, but not at the lack of quality.

Having models that produce images that are contextually appropriate is also critical. Our running examples in this work come from celebahq as they have the highest level of realism, but the model’s outputs are heavily biased towards generating white, conventionally attractive faces, reflecting the biases of its training set. As a result, the images have limited contexts in which they would be appropriate. Future diffusion models should be trained on more diverse data sets. The problem of bias in model outputs is not unique to our use case, but only having a few types of outputs would be detrimental to Pulsar’s real-world deployment. A censor monitoring a channel may expect only certain types of images, and perhaps images like celebahq or church would be considered strange at best, or subversive at worst. More diverse models would allow users to generate the correct types of images needed for steganography without arousing a censor’s suspicion.

Support latent diffusion architectures. The models for our Pulsar evaluation are all pixel diffusion models, in which each model iteration performs operations on all pixels of the output image at once. Our models operate over tensors of shape $256 \times 256 \times 3$ to generate an image of dimension $256 \times 256$ with 3 color channels. As mentioned in Section 3.1, an alternative to this approach are latent diffusion models like Stable Diffusion [RBL+22]. Operations are performed on latents, which are a smaller representation of the image, and upscaled at the end using a VAE. Latent diffusion architectures have good results, even on relatively limited hardware.

While the core embedding strategy of Pulsar can be applied to latent diffusion models, there are barriers to making this approach productive. One is the space for encoding. The latent space used by these models typically have shape $64 \times 64 \times 3$, much smaller than that of the pixel diffusion models from our Pulsar implementation and potentially reducing throughput.

The more concerning issue is the VAE itself. Our experiments found that the VAE made it extraordinarily difficult for a receiver to determine anything about how noise was injected during model generation. In particular, because the VAE “decompresses” the latent space, applying noise to even a single element of the latent would result in noticeable changes in many pixels. As such, a receiver has a hard time differentiating between the effects of two neighboring latents (i.e., the effects contaminate each other). We found that
encoding was possible if we spread out the locations in the latents into which we steganographically encoded, but the result extraordinarily low performance, e.g., 16 or 32 bits per image.

Existing approaches to VAE inversion involve estimation, and the result is not guaranteed to be the original input. Our preliminary experiments showed that attempts to reverse the VAE step resulted in too many errors to accurately reconstruct that original latents. If it were possible to design a VAE that is deterministically invertible, it would be possible to encode with very high rate into the latent space. Future work in this space could identify ways to recover latents from an image.

8 Conclusion

We present Pulsar, the first provably secure steganography scheme with support for diffusion models. Pulsar is practical, encoding hundreds of bytes per image at speeds exceeding that of prior, text-based solutions. Future work in both steganography and diffusion models can use the lessons from the design of Pulsar to create even more efficient systems.

Acknowledgments

This work was funded, in part, by the National Science Foundation’s Convergence Accelerator Program, Track G under contract number 49100422C0024. The first author would like to acknowledge support from the National Science Foundation under Grant #1955172. The second author was supported by DARPA under Contract No. HR001120C0084. The third author is supported by the NSF under Grant #2030859 to the Computing Research Association for the CFellows Project and is supported by DARPA under Agreement No. HR00112020021. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government, NSF, or DARPA.

References


A Additional Details on Error Correction in Pulsar

Pulsar uses error correcting codes to improve the performance of the steganographic channel inside of diffusion models. A more formal treatment description of the \texttt{EstimateRate} used in Pulsar can be found in Algorithm 4. Note that our optimized \texttt{EstimateRate} requires the estimation of the errors in the current model state by performing trial encoding and decoding of random messages. We define a subroutine in Algorithm 5, \texttt{CalcErrors}, that performs this estimation for \texttt{EstimateRate}. Note that the parameters \texttt{EstimateRate} is parameterized by the number of buckets in each estimate $u$ and $l$, which we discuss in Section 6.1 in the main body of this paper.

Table 3 contains information on the concrete codes library that we built for Pulsar. We provide the exact SageMath function calls that generate our outer and inner codes. This library forms the Library\textsubscript{ECC} in Algorithm 4.

As we discovered these codes experimentally, we make no claims about optimality. In particular, note that the code for error rate 0.15 does not meet the traditional definition of a concatenated code, since there is a mismatch between the codeword size of the outer code (8 bits) and the message size of the inner code (6 bits). In our testing, however, the code performed well, and we included it in our library. We leave a more systematic design for the concrete error correcting codes used in Pulsar to future work.

### Algorithm 4: Pulsar \texttt{EstimateRate}

\begin{verbatim}
Input: Model state $s_{t-2}$
Output: Difference region-based ECC rates $\text{rate}$
// Get error rates at each pixel
$\text{err} \leftarrow \text{CalcErrors}(l)$
// Bucket regions based on error rates
$\text{regions} \leftarrow \text{Bucketize}(\text{err}, u)$
for $\text{region} \in \text{regions}$ do
    $\text{regionErr} \leftarrow \text{mean}(\text{err}[\text{region}])$
    // Select appropriate ECC parameters
    $\text{params} \leftarrow \text{LibraryECC}(\text{regionErr})$
    $\text{rate}[\text{region}] \leftarrow \text{params}$
Output: $\text{rate}$
\end{verbatim}

Figure 8: The optimized \texttt{EstimateRate} used in Pulsar.
Algorithm 5: CalcErrors

**Input:** Model state $s_{t-2}$, Number of estimates $l$

**Output:** Mean error rates at each pixel location $err$

for $1 \leq g < l$ do
  
  // Similar to the online phase of encoding
  $m \leftarrow [0,1]^{n \times n}$
  for $0 \leq j < |m|$ do
    if $m[j] = 0$ then
      $r_{t-1}[j] \leftarrow N_{k_0}$
    else
      $r_{t-1}[j] \leftarrow N_{k_1}$

  pred$_{t-1} \leftarrow \text{Model}(s_{t-2})$
  $s_{t-1} \leftarrow \text{Scheduler}_{t-1}(s_{t-2}, \text{pred}_{t-1}; r_{t-1})$

  // Deterministic Final Schedule
  pred$_{t} \leftarrow \text{Model}(s_{t-1})$
  $\text{img} \leftarrow \text{Scheduler}_{t}(s_{t-1}, \text{pred}_{t})$

  // Similar to the offline phase of decoding
  // Generate reference image 0
  Set $r_{t-1}^0 \leftarrow N_{k_0}^{n \times n}$
  Set $s_{t-1}^0 \leftarrow \text{Scheduler}_{t-1}(s_{t-2}, \text{pred}_{t-1}; r_{t-1}^0)$
  pred$_{t}^0 \leftarrow \text{Model}(s_{t-1}^0)$
  $\text{img}^0 \leftarrow \text{Scheduler}_{t}(s_{t-1}^0, \text{pred}_{t}^0)$

  // Generate reference image 1
  $r_{t-1}^1 \leftarrow N_{k_1}^{n \times n}$
  Set $s_{t-1}^1 \leftarrow \text{Scheduler}_{t-1}(s_{t-2}, \text{pred}_{t-1}; r_{t-1}^1)$
  pred$_{t}^1 \leftarrow \text{Model}(s_{t-1}^1)$
  $\text{img}^1 \leftarrow \text{Scheduler}_{t}(s_{t-1}^1, \text{pred}_{t}^1)$

  // Similar to the online phase of decoding
  for $0 \leq j < |m|$ do
    if $|\text{img}[j] - \text{img}^0[j]| < |\text{img}[j] - \text{img}^1[j]|$ then
      $m'[j] \leftarrow 0$
    else
      $m'[j] \leftarrow 1$

  // Save the n x n matrix of magnitudes to a list
  errs$_{g} \leftarrow \text{abs}(m - m')$

  // Mean of all estimated magnitudes
  Output $err \leftarrow \text{mean}(errs)$

Figure 9: The CalcErrors subroutine of EstimateRate.
B Sample Pulsar Images

Figure 10 contains additional sample images from our Pulsar evaluation. In each figure, the first image represents the image generated by model on which Pulsar achieved the best encoding length, and the second image represents the one on which Pulsar achieved the worst encoding length.

(a) A Pulsar image from the church model which encodes 1120 bytes.

(b) A Pulsar image from the church model which encodes 100 bytes.

(c) A Pulsar image from the bedroom model which encodes 720 bytes.

(d) A Pulsar image from the bedroom model which encodes 100 bytes.

(e) A Pulsar image from the cat model which encodes 1420 bytes.

(f) A Pulsar image from the cat model which encodes 100 bytes.

Figure 10: Sample Pulsar outputs from the church, bedroom, and cat models.