A. Appendix: Broader impact

In this work, we treat the membership inference attacks as a potential tool for privacy protection. By applying these attacks to a model, we can highlight the instances where images have been used in training without the appropriate consent. As more individuals and organizations become aware of this possibility, we might see a push towards more stringent data usage policies and ethical guidelines for machine learning practices. The potential of these attacks to expose privacy violations could serve as a catalyst for a broader dialogue on privacy rights and data ownership in the digital age.

However, these benefits come with a significant caveat. The same mechanism that can be used to protect user privacy can also be used maliciously. If a membership inference attack is successful and achieves high accuracy, it could potentially lead to personal data leakage. Thus, the dual implications of membership inference attacks for diffusion models offer both a warning and a beacon. They highlight the need for robust privacy protection measures while simultaneously alerting us to the potential risks of personal data leakage.

B. Appendix: Limitations

LAION-5B dataset, the source of data used both to train the Stable Diffusion model and to create LAION-mi dataset does not contain the images, just URLs to them. Therefore, to prepare a training set, one needs to first download the images using the URLs, and then train the model. Since the content, that a URL points to is out of our control, it might be a subject of edition or deletion, which we are unable to handle. Because of that, we cannot guarantee with 100% certainty that every member in LAION-mi is a real member of the Stable Diffusion model's train set.

In fact, during our experiments, we find out that approximately 10% of links are dead, i.e. we cannot download images from them. This limitation is inherent to the LAION-5B dataset, and therefore cannot be alleviated. Additionally, a URL to an image from LAION-mi has to be alive to use it for the attack evaluation. We argue that it is unlikely that a URL from the LAION-mi members subset that was dead during training of the Stable Diffusion is now alive, however, we cannot rule out such scenario. Fortunately, since it affects only the members set, it makes the membership inference task harder, in effect saving us from the pitfalls described in Section 4.

C. Appendix: Stable Diffusion-v1.4 training

Datasets Datasets used to train this model are as follows

- LAION-2B EN: a subset of the LAION-5B [28] with images' descriptions in English.
- LAION-Aesthetics V2 5+ [27]: a subset of the LAION-2B EN dataset, in which each image is scored using the LAION-Aesthetics_Predictor V2 [26], and only images with the Aesthetic Score above 5 are a part of it.

Stable Diffusion-v1.4 This version of the Stable Diffusion is first trained for 431k steps using samples from LAION-2B EN, then fine-tuned for 515k steps using LAION-Aesthetics V2 5+, and then fine-tuned for 225k steps on LAION-Aesthetics V2 5+ again [23].

D. Appendix: Loss attacks

In this section, we discuss different approaches to perform the diffusion denoising process in the white-box scenario. In each of the following setups, at every timestep, we collect the Model loss, Latent error, and Pixel error described in Section 6.2. These values are calculated based on the output of the UNet SD-v1.4 backbone using the methods described below.

D.1. All attack methods

Here we describe all the methods used in our experiments. For all methods, the classifier-free guidance is applied with scale 7.5 unless otherwise stated.

1. **Partial denoising** Following findings in [4] in this method, we start our denoising process at the timestep 300 for the steps 26 up to the timestep 50. Latent image representation is noised once, with scale α_{300}, and then for each step calculated from the UNet noise prediction.

2. **Partial denoising non-iterative** is similar to the *Partial denoising* method, but at each timestep we pass a newly noised latent representation of the image to the UNet, instead of the output of the previous timestep's inference through UNet. The main intuition behind this is to see whether the information about membership gets lost or accumulates during the iterative denoising process.

3. **Partial denoising latent shift** follows the *Partial denoising* attack method, but at the middle timestep 170 we shift the latent representation of the image by a scaled random noise. The intuition behind this method is that the member samples should return to the correct trajectory after the shift better than the nonmember samples enabling us to better identify the member samples.

4. **Partial denoising text shift** is similar to the *Partial denoising* latent shift, but instead of shifting the latent representation of the image, we shift the text embedding by a random noise $N(0, I) \times 0.1$ before passing it to the UNet. The intuition here is the same as in the partial denoising latent shift method, but we want to see if

the text embedding is more important than the latent representation of the image.

5. **Partial denoising bigger text shift** is identical to the *Partial denoising text shift*, but the noise added to the text embedding is scaled by 0.5 instead of 0.1. We want to test if adding more noise to the text embedding benefits the performance of the attacks.

6. **Partial denoising wrong start** In this method we follow the *Partial denoising*, but the starting latent representation of the image is noised using α_t from the wrong timestep, $t = 100$ instead of $t = 300$. We want to test whether we can extract more information about the membership if we apply lower noise to the latent representations at the beginning of the process than UNet expects.

7. **Full denoising start 100** is similar to the *Partial denoising wrong start* method, but we perform an almost full denoising process, starting on timestep and 900 ending at timestep 0, denoising every 100 and starting from the latent noised with α_{100} noise scale.

8. **Full denoising start 100 text shift** is close to the *Full denoising start 100*, but we shift the text embedding by a random noise $N(0, \mathbf{I}) * 0.1$ before passing it to the UNet.

9. **Full denoising start 50** is similar to the Full denoising start 100, but the starting latent is noised using α_{50} instead of α_{100}.

10. **Full denoising start 300** is like the Full denoising start 100, but the starting latent is noised using α_{300} instead of α_{100}.

11. **Short denoising start 300** In this method, we perform a short denoising process, starting from timestep 200 ending at timestep 0 every 100 steps instead of the full one. The latent representation of the image is noised using α_{300}.

12. **Reversed noising** We perform 10 denoising steps for timesteps from 900 to 0. The image latent representation we get from the VAE encoder is noised using noise scales α_t in reverse order, e.g. at timestep 800 we pass the latent noised with α_{100} to the UNet. Additionally, during inference, we use classifier-free guidance on the guidance scale 100. The text embedding passed to UNet remains unchanged. The intuition behind the attack is that member samples will behave more robustly than nonmember samples under such conditions.

13. **Full denoising start 300 no cfg** is similar to the *Full denoising start 300*, but we do not use the classifier-free guidance. We want to see if the classifier-free guidance is beneficial for the attacks performance.

14. **Full denoising start 100 non-iterative**. This method is identical with the *Full denoising start 100*, but at each timestep we pass newly noised latent representation with scale α_{100} of the image to the UNet, instead of the output of the previous timestep.

15. **Reversed noising regular cfg** resembles the *Full denoising start 100 non-iterative* attack method. The latent representations we pass to the UNet are noised using α_t from the timesteps in the reversed order, i.e. from 0 to 900, so the first latent passed to the UNet is noised using α_0, while UNet receives timestep 900 as input. The classifier-free guidance is applied with scale 7.5.

16. **Reversed denoising** In this method, we reverse the order of timesteps at which we perform denoising using UNet. We start from the timestep 0 and then go up to the timestep 900, for 10 steps in total. Similarly as in the baseline loss threshold method, we apply noise to the latent image representation once, but this time using noise scale α_{100}. At each timestep we measure all losses described in Section 6.2 and use them to evaluate the attack. The intuition here is similar as in the *Reversed noising* method, but here we test if the iterative nature of the diffusion denoising process will magnify this effect.

D.2. Classifier attack

In addition to the *threshold* attack type described in 6.2 we also introduce the *classifier* attack type. *Classifier* attack builds on top of the *threshold* attack. It uses a binary classifier C to predict the membership of an image x based on the losses collected during inference through the diffusion model. The classifier is trained on a set of images with known membership and returns a probability of membership for a given image. We then perform a *threshold* attack on the classifier's output.

Table 3. **Hyperparameters used for the classifiers:** Logistic Regression (LR), Decision Tree Classifier (DTC) and Random Forest Classifier (RFC).

Classifier	Hyperparameter	Values
LR	C	[0.01, 0.1, 1, 10, 100]
DTC	max_depth	[2, 8, 16]
	min_samples_split	[2, 8, 16]
RFC	n_estimators	[10, 100, 1000]
	max_depth	[2, 8, 16]

We train four model classes: Logistic Regression (LR), Decision Tree Classifier (DTC), Random Forest Classifier (RFC), and Neural Network binary classifier (NN). The classifiers are trained in the binary classification task, with the member samples as positive examples and the nonmember samples as negative examples. The input for every classifier consists of the vector of the loss values we collect at every timestep. For the Logistic Regression (LR), Decision Tree Classifier (DTC) and Random Forest Classifier (RFC) we perform Grid Search with k-Cross Validation, $k = 5$ for each fit. The hyperparameters used for the classifiers are described in Table 3. Note that we perform the Grid Search for every fit separately, therefore we do not report the best parameters in our table, since they turn out to be different for different fits.

For the Neural Network classifier (NN) we use the following architecture: 3 fully connected layers, with the input size of $3 * timesteps$ (since for different methods we have a different amount of data), hidden size of 10 and a single output for the binary classification. We use ReLU activation function for the hidden layers and Sigmoid for the output layer. We use Adam optimizer with a learning rate 0.001 and binary cross entropy loss. We train the classifier for 100 epochs with batch size 32 and early stopping. We use the best model from the early stopping for the evaluation.

Following 6.4 we fit our models 100 times on 100 different training sets sampled from the whole attack set and evaluate them on the remaining evaluation sets. We report the mean and standard deviation of the metrics for the 100 fits. The results are presented in Table 4.

D.3. Results

Following our evaluation method described in Section 6.4 we report our results for each method in Table 4 for *threshold* attacks, and in Table 5 for *classifier* attacks.

We conclude that extraction of the membership information from the attacked model can be improved by modifying the diffusion denoising process by altering the timesteps, latent representations, and text embeddings. We also see that the information about membership can be better extracted by using *classifier* attack, but for most methods, a simple *threshold* attack outperforms the *classifier* attack.

We also observe that different methods achieve visibly different performance on different timesteps, which points out that the timing of loss measurement is also really important when performing the *threshold* attack, see Figure 7.

E. Appendix: Experiments randomization

Here we highlight the need to perform the randomization described in Section 6.4. For each of the methods described in Appendix D we show the differences between the best,

Table 4. ***Threshold* attack results for each method.** We see that for each attack method Model loss gives out better performance of *threshold* attacks compared to Latent or Pixel error. It is also more robust for the denoising procedure modification than Latent Error, dropping to at most 2%, while Latent error performance can drop to 1.1%. Pixel error seems also more stable than Latent Error, but we cannot achieve as high results as for the Model loss or Latent error. The high discrepancy between different attack methods points that we can indeed influence the amount of membership information extracted during influence, with some methods performing better on Model loss and some on other losses, Pixel and Latent error.

Loss method	Model Loss	Latent Error	Pixel Error
Partial denoising	2.3%±0.61	**2.4**%±0.62	1.7%±0.68
Partial denoising non-iterative	2.3%±0.68	1.1%±0.46	1.39%±0.59
Partial denoising latent shift	2.3%±0.62	2.24%±0.9	1.6%±0.62
Partial denoising text shift	2.2%±0.57	2.28%±0.57	1.7%±0.64
Partial denoising bigger text shift	2.3%±0.62	2.22%±0.75	1.74%±0.6
Partial denoising wrong start	2.2%±0.69	2.13%±0.85	1.99%±0.56
Full denoising start 100	2.08%±0.64	1.1%±0.41	1.84%±0.6
Full denoising start 100 text shift	2.01%±0.6	1.1%±0.43	1.8%±0.6
Full denoising start 50	2.0%±0.6	1.15%±0.38	1.95%±0.55
Full denoising start 300	1.99%±0.61	1.34%±0.49	1.72%±0.64
Short denoising start 300	2.32%±0.67	2.06%±0.72	1.57%±0.67
Reversed denoising	2.25%±0.64	2.17%±0.64	**2.03**%±0.55
Full denoising 300 no cfg	2.07%±0.62	1.45%±0.52	1.7%±0.6
Full denoising 100 non-iterative	2.2%±0.55	2.18%±0.75	1.91%±0.53
Reversed noising regular cfg	**2.5**%±0.74	2.03%±0.6	1.92%±0.5
Reversed noising	**2.51**%±0.73	1.26%±0.52	1.9%±0.51

Table 5. ***Classifier attack* results for each method.** Same as in Tab. 4 we observe that for different attack methods the same model classes achieve visibly different performance. This again confirms that we are able to obtain more information about the membership of samples by influencing the whole denoising process of the large diffusion model. Unsurprisingly, we are able to outperform the simple *threshold* attacks, because the membership information ends up spread out on different timesteps and losses. Using *classifier* attack we can combine the information from the whole inference procedure and make better predictions. Surprisingly, it seems to be harder to perform such attacks in the TPR@FPR=1% regime. For almost all methods and model classes we are not able to outperform the simple *threshold* attacks, especially on the Model loss. For almost all methods the Random Forest Classifier model outperforms all other model types, with the exception of *Reversed noising regular cfg* method data fitted using Logistic Regression. Decision Tree Classifier fails to be better than a random guess for almost all methods, and the Neural Network classifier also fails to deliver satisfying results.

CLASSIFIER CLASS METHOD	LR	DTC	RFC	NN
PARTIAL DENOISING	1.83%±0.87	0.67%±1.09	2.27%±0.76	1.50%±0.92
PARTIAL DENOISING NON-ITERATIVE	2.24%±0.86	**1.10**%±1.12	2.35%±0.88	1.52%±1.03
PARTIAL DENOISING LATENT SHIFT	1.94%±0.92	0.86%±0.94	2.35%±0.86	1.40%±0.91
PARTIAL DENOISING TEXT SHIFT	1.71%±0.87	0.49%±0.84	2.43%±0.93	1.40%±1.00
PARTIAL DENOISING BIGGER TEXT SHIFT	2.14%±1.01	0.80%±1.13	2.30%±0.73	1.49%±0.90
PARTIAL DENOISING WRONG START	1.50%±0.70	0.74%±1.08	2.26%±0.87	1.51%±0.92
FULL DENOISING START 100	1.09%±0.56	0.54%±0.74	1.89%±0.85	1.30%±0.76
FULL DENOISING START 100 TEXT SHIFT	1.10%±0.62	0.62%±0.76	1.88%±0.80	1.24%±0.69
FULL DENOISING START 50	1.16%±0.68	0.70%±0.68	1.89%±0.78	1.31%±0.78
FULL DENOISING START 300	1.21%±0.58	0.42%±0.85	1.87%±0.87	1.39%±0.80
SHORT DENOISING START 300	1.94%±0.91	0.54%±0.83	2.31%±0.91	1.49%±0.85
REVERSED DENOISING	1.96%±0.96	0.80%±1.17	2.37%±0.98	1.26%±0.78
FULL DENOISING 300 NO CFG	1.31%±0.73	0.31%±0.65	1.78%±0.78	1.37%±0.82
FULL DENOISING 100 NON-ITERATIVE	1.60%±0.77	0.68%±1.05	2.19%±0.85	1.60%±0.73
REVERSED NOISING REGULAR CFG	**2.41**%±1.09	0.47%±1.03	2.17%±0.84	1.40%±0.82
REVERSED NOISING	1.98%±0.97	0.41%±1.02	**2.75**%±1.03	**1.62**%±0.86

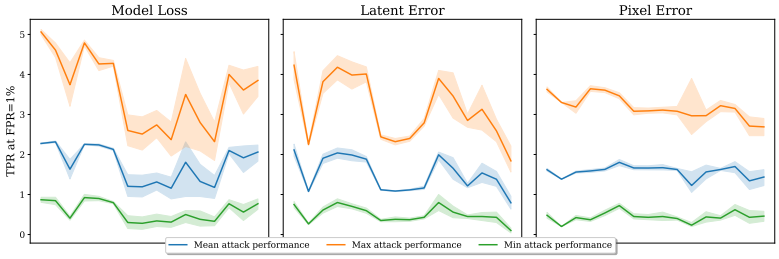


Figure 5. **Min, mean, max TPR@FPR=1% for different methods and losses, *threshold* attack type.** Solid line indicates the mean value of the TPR@FPR=1% metric from 100 experiments, and the shaded area is the 95% confidence interval.

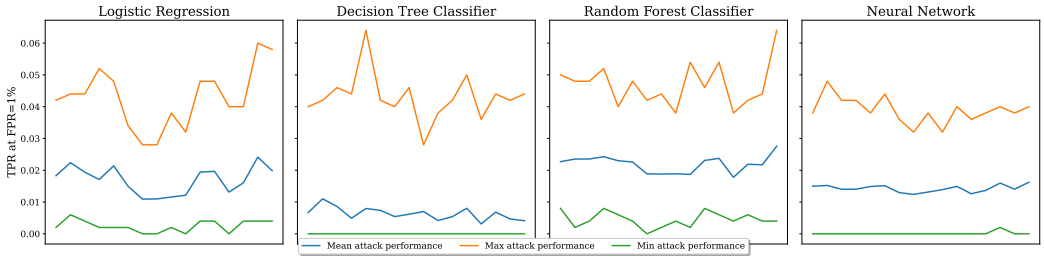


Figure 6. **Min, mean, max TPR@FPR=1% for different methods, classifier classes and losses, *classifier attack* type.**

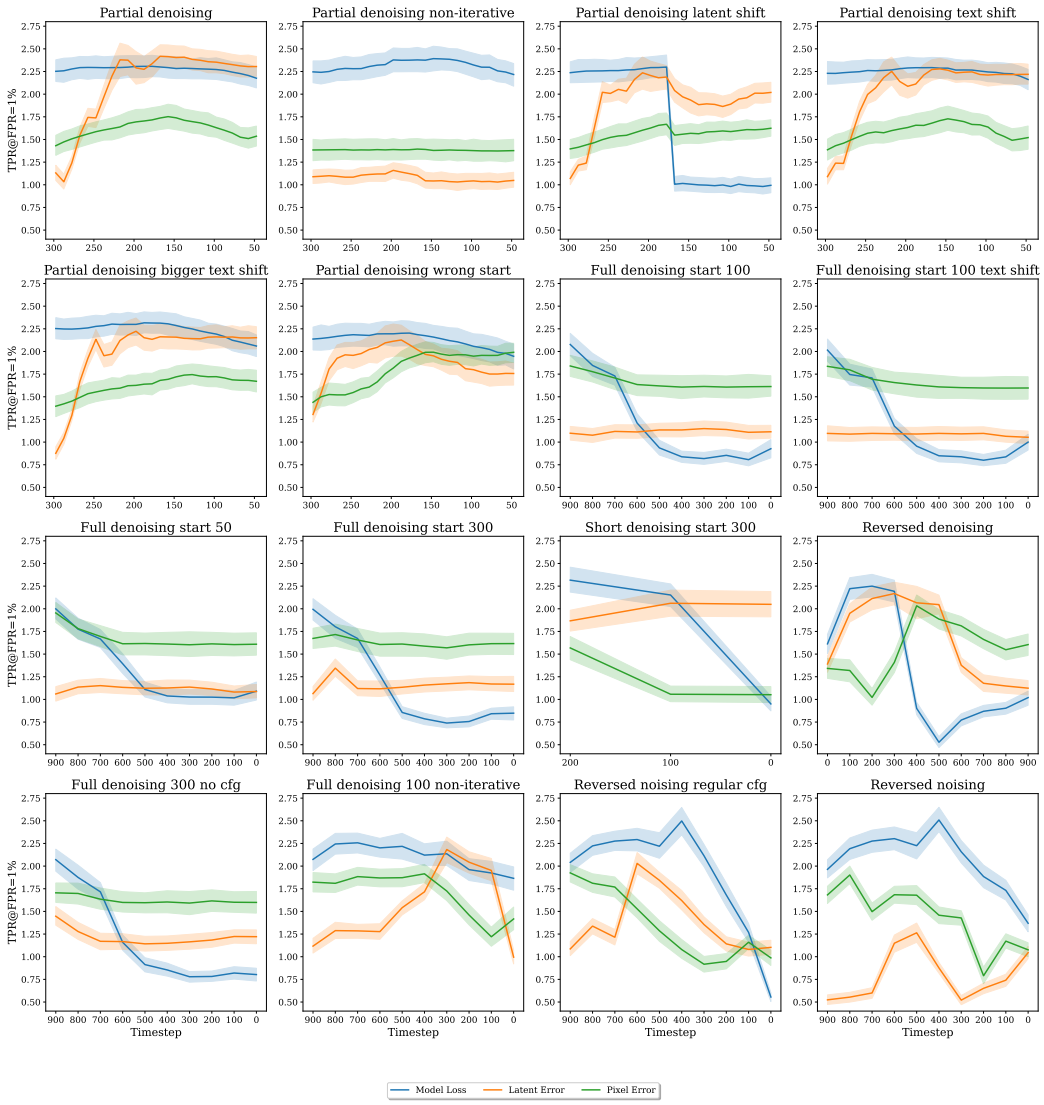


Figure 7. **Attack performances on different losses in different timesteps for each attack method for the *threshold* attack type.** The solid line indicates the mean value of the TPR@FPR=1% metric from 100 experiments, and the shaded area is the 95% confidence interval. We see how important is to collect all three losses on each timestep, with the biggest difference visible for the *Reversed noising* method, with the worst result around 0.5% and the best around 2.5% for Latent error at timestep 0 and Model loss at timestep 500. We derive the following insights from these plots. Applying the denoising process iteratively instead of passing the newly noised latent to the UNet greatly benefits the Latent and Pixel error *threshold* attacks, while slightly reducing the effectiveness of *threshold* attacks on the Model loss. Shifting the latent by a random noise mid-inference affects the results negatively, with the biggest drop in performance for the Model loss. Shifting text embedding by a random noise for different noise scales (0.1 and 0.5) does not make any significant difference in terms of the final results, generally being slightly harmful to the performance of the *threshold* attack. The mismatch between the actual and latent noise timesteps is by far the most impactful modification we implemented. We see improvements on the Pixel error *threshold* attacks for *Partial denoising wrong start* method, where we start from the latent noised with the noise scale of $\alpha_t = 100$ and denoising from the timestep 300 to 50. However, for the *Full denoising* attack methods we see that different noise scales applied to the starting latent representations do not change the performance. Applying the classifier-free guidance seems to be insignificant for the final results (see *Full denoising 300* vs *Full denoising 300 no cfg*), but increasing the guidance scale (from the default 7.5 to 100) reduces the effectiveness of the *threshold* attacks on Pixel and Latent error. The best results are obtained when we reverse the order of noise scales used to apply noising on the latent representation of the input image, while the input timesteps we pass to UNet remain in the normal order. We also see in Tab. 5 that these methods allow the *classifier* attacks to achieve the best results, suggesting that this approach extracts the most data about the membership from the model.

mean and worst results for each attack from the separate runs. We also visualise these differences for the classification attack type. We can observe a mismatch between mean and best results for all of the attacks, some of them being even three times worse than the best ones (*Partial denoising* method). When the available size of the nonmembers set is relatively small, the randomization is crucial to obtain reliable results, especially when proposing attacks based on more sophisticated methods that require training a classifier, which is the case in the *classifier* attack type. In this case, we suggest splitting the whole attack set randomly in the way described in Section 6.4 into training and evaluation sets, then training the classifier, and repeating the whole procedure for at least 100 times. The reported results should be the mean of the results for each attack from each run. In this way, we can mitigate the influence of the potential outliers and obtain more reliable results.

Visualisation of the mismatch between the best, mean, and worst results can be found in Figure 5 for the *threshold* attacks and Figure 6 for the *classifier* attacks.

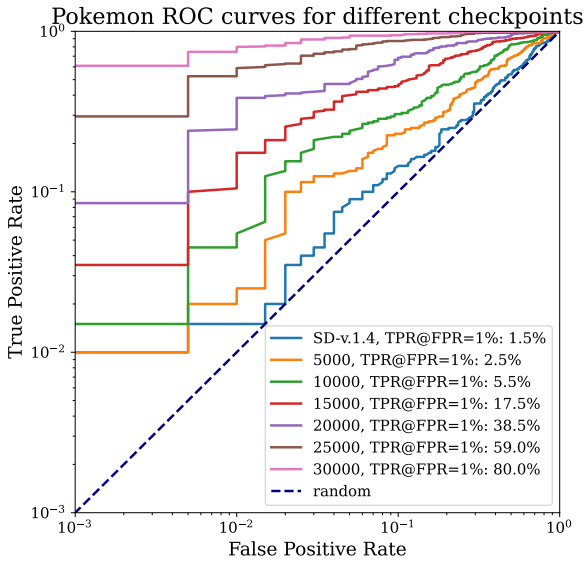
F. Appendix: Overfitting impact on membership inference attacks

In this section, we present the results of our experiments on overfitting on the POKEMON dataset from Section 6.4. We finetune the original StableDiffusion-v1.4 using the default method from [16] for 30000 train steps with the learning rate 0.00001 and gradient accumulation of 4. Every 5000 steps we save the partially finetuned model.

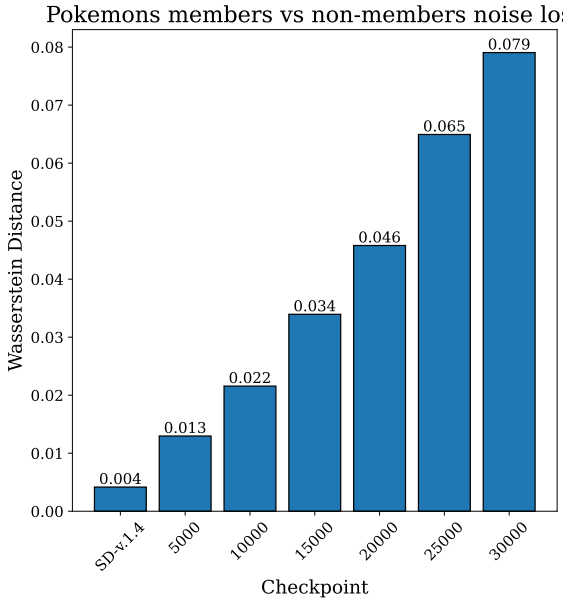
The attack method we use to perform for this section is the Baseline loss threshold method, described in Section 6.3. The ROC curves and TPR@FPR=1% results for different checkpoints can be found in Figure 8a. We can observe that the model is overfitting to the training set, as the TPR@FPR=1% results are significantly better for the checkpoints from the later stages of the training. This is in line with the findings in [3], which also show that model loss based membership inference attacks give better results when the model is overfitted. This issue seems to be not addressed in some of the previous contributions, e.g. in [6] authors claim really good performance of their attacks on the finetuned version of the SD-v1.4 model on the same dataset.

G. Appendix: Fine-tuned Shadow models for Large Diffusion Models

Training a new Stable Diffusion model from scratch multiple times is in practice infeasible. Thus, it is impossible to directly apply the existing shadow model attack in this case. To overcome this limitation we focus our analysis on model fine-tuning, rather than training from the start. Motivated by the state-of-the-art results achieved by shadow model membership inference, we draw inspiration for our approach



(a) **ROC curves and TPR@FPR=1% results for different checkpoints.**



(b) **Wasserstein Distance for different checkpoints.**

Figure 8. Figure 8a shows how overfitting correlates with the attacks performance. Figure 8b shows the Wasserstein Distance calculated between the losses of members and nonmembers sets for each checkpoint.

from the offline *LIRA* attack introduced in [3]. Intuitively, we aim to answer the question "*is there a difference between finetuning the model on member and nonmember samples?*". To overcome the memory requirements for storing multiple versions of the Stable Diffusion models we apply fine-tuning with *LoRA* [11].

First, we sample n nonmember samples from LAION-mi.

(a) LAION-mi nonmember samples.

(b) LAION-mi member samples.

Figure 9. Visualization of the random subset of LAION-mi dataset

For each single sample, we finetune the Stable Diffusion v1.4 model on a single training step at $t = 100$ using *LoRA*. We measure the ratio of the training loss after and before the finetuning. We repeat that procedure for n member samples from LAION-mi. Thus we obtain loss ratio distributions for shadow models finetuned on member (D_{mem}) and nonmember (D_{nonmem}) samples. For inference, we fine-tune the model on the new sample using the same procedure, obtaining the corresponding loss ratio $l\star$. If $\frac{Pr[l\star|D_{mem}]}{Pr[l\star|Dnonmmem]} > \tau$ we classify the sample as a member.

This attack can only be performed in the white-box scenario. The attacker needs to have access to the model weights to perform the fine-tuning.

We evaluate our method on 100 subsets of 1000 member and 1000 nonmember samples randomly selected from 4000 samples from each set. The approach based on shadow models achieves TPR@FPR=1% equal to 2.21% $\pm$ 1.11. The performance of the attack is thus comparable with the attacks described in Section 6.3. Although shifting the focus of shadow models from full training to finetuning using *LORA* enabled us to apply this kind of approach, the resources required to execute this attack are still significantly larger than for the methods described in Sec. 6.3.

H. Appendix: LAION-mi samples

In this section we provide a random sample of the LAION-mi dataset, 16 images from the nonmembers set and 16 images from the members set, see Figure 9.

I. Appendix: GPU cost

To conduct the experiments for this paper we utilized 4000 hours of Nvidia A100 GPU compute.