WHITE PAPER

Adapting Tensorlock to Large Language Models and Generative Models

White paper · October 2026

Summary

Tensorlock encrypts training, test, and inference data so that a fully connected artificial neural network (ANN) can train on the ciphertext directly. This paper describes how the method can extend to transformers, large language models (LLMs), and generative models.

The extension rests on one idea. Party A, the data owner, converts plaintext into a machine-learning representation with a private encoder, and Tensorlock encrypts that representation. The remote party trains in the resulting **encrypted representation space**.

Three uses follow, in order of increasing departure from the initial Tensorlock manuscript.

Text classification over privately encoded text encrypts one vector per document. It stays within the threat model of the initial Tensorlock manuscript.

Generative models over encrypted latent representations train remotely without interaction. Generation requires a decryption step, which Party A must perform inside its own inference oracle.

Language-model training on encrypted token embeddings is feasible in principle. Its protection is limited, because a remote trainer can group the repeated ciphertexts of each token. The corpus is then exposed to the statistical attacks that apply to a substitution cipher.

This paper is conceptual. It reports no experiments, and it makes no security claim beyond those in the initial Tensorlock manuscript.

1. Background

1.1 The Tensorlock transformation

Tensorlock applies four steps to each example: padding, perturbation, fixed index shuffling, and random orthogonal matrix transformation. Let $h \in \mathbb{R}^d$ be a plaintext vector. Padding and perturbation produce a vector of higher dimension $D > d$:

$$u = \mathrm{Pert}(\mathrm{Pad}_{S,r}(h)) \in \mathbb{R}^D.$$

Here, S is the hidden set of coordinates that hold the plaintext signal, and r is a padding vector drawn separately for each example. A fixed permutation matrix P and a Haar-random orthogonal matrix Q then give the ciphertext:

$$z = uA, \qquad A = PQ, \qquad A^\top A = I.$$

This paper writes the aggregate transformation as $z = T(h)$. The key comprises S, A, and the settings that generate padding and perturbation. Party A holds the key and discloses none of it.

A key holder can reverse the orthogonal and padding steps:

$$\hat{h} = \left(zA^\top\right)_S = h + \eta.$$

The residual η is the perturbation noise. The key holder recovers h only up to this noise.

The orthogonal step preserves inner products and distances among the padded vectors. This preserved geometry lets a network learn from ciphertext.

1.2 Parties and threat model

This paper uses the threat model of the initial Tensorlock manuscript. Party A owns the plaintext dataset and the key. Party B receives the encrypted dataset and labels, and trains the network. Party C is an attacker who holds the encrypted dataset and labels, and who may be part of Party B.

That manuscript also defines an **inference oracle** controlled by Party A. After training, Party B passes the trained network to Party A. The inference oracle accepts a plaintext example, encrypts it privately, runs the trained network, and returns only the result. It never discloses ciphertext for a submitted plaintext.

Every design in this paper keeps inference inside the inference oracle. Party B never observes ciphertext derived from a user's input.

1.3 Encrypted representation spaces

In the initial Tensorlock manuscript, the raw input vector is the encrypted object. The designs here insert a private encoding step first:

$$\boxed{\text{plaintext}} \xrightarrow{\text{private encoder}} \boxed{\text{representation } h} \xrightarrow{T} \boxed{\text{encrypted representation } z}.$$

A **private encoder** is a fixed function that Party A applies to plaintext. Examples include a token embedding table, a frozen text encoder, and the encoder half of an autoencoder. Party A keeps its parameters secret, as part of the encryption details.

The private encoder is fixed before encrypted training begins. Party B cannot update it, because Party A computes the encrypted dataset once.

1.4 Discriminative and generative use

A discriminative model maps an encrypted example to a label:

$$x \xrightarrow{T} z \xrightarrow{\text{model}} y.$$

Preserved geometry suffices for this task. The model never returns to the plaintext domain.

A generative model produces a new example, which must return to the plaintext domain. The pipeline gains a return path:

$$x \xrightarrow{\text{private encoder}} h \xrightarrow{T} z \xrightarrow{\text{generative model}} z' \xrightarrow{\text{decryption}} h' \xrightarrow{\text{private decoder}} x'.$$

A **private decoder** is a fixed function at Party A that maps a representation back to plaintext. The return path introduces a decryption step that the initial Tensorlock manuscript does not contain. Section 5.5 treats its consequences.

2. Transformers on encrypted vector sequences

2.1 Input pipeline

A transformer converts a sequence of tokens $(t_1, \ldots, t_L)$ into embedding vectors through an embedding table E:

$$e_i = E(t_i), \qquad e_i \in \mathbb{R}^d.$$

Party A holds E as a private encoder and encrypts each embedding with the same key:

$$z_i = T(e_i), \qquad Z = (z_1, \ldots, z_L) \in \mathbb{R}^{L \times D}.$$

Each vector receives its own padding and perturbation values. Party B receives Z and never receives the tokens or the embeddings.

2.2 Learning in ciphertext coordinates

A transformer whose first operation is a learned linear projection can, in principle, match a transformer trained on plaintext. Let Π_S be the $D \times d$ matrix that selects the signal coordinates, so that $\hat{h} = zA^\top \Pi_S$. Let W be the first projection of a plaintext transformer. The projection

$$W_{\text{in}} = A^\top \Pi_S \, W$$

then gives

$$zW_{\text{in}} = \hat{h}W = (h + \eta)W.$$

A network with this input projection reproduces the plaintext network, up to the perturbation noise. Party B does not know A or S, so training must find a suitable projection from data. The fully connected results in the initial Tensorlock manuscript suggest that training can do so.

The argument is exact only with a learned input projection. A transformer that operates directly at width D applies layer normalization in ciphertext coordinates. Layer normalization depends on the coordinate basis, so the equivalence is then approximate.

2.3 Sequence order and position

Party B receives the vectors in sequence order. Tensorlock therefore protects the content of each vector, and it does not conceal sequence length or order.

Party B can add learned positional embeddings in ciphertext space, because it already knows each position. Party A should not encrypt publicly known positional vectors with the data key. Doing so would give an attacker known pairs of plaintext and ciphertext.

3. Language-model training on a private corpus

3.1 The training target

Causal language-model training uses the sequence $(t_1, t_2, t_3, \ldots)$ as input and the shifted sequence $(t_2, t_3, t_4, \ldots)$ as the target. Plaintext targets would disclose the corpus to Party B. Encryption of the inputs alone is therefore insufficient, and the targets also need protection. Two designs address this.

3.2 Secret vocabulary relabeling

Party A draws a secret permutation π of the vocabulary $\mathcal{V}$ and transmits $\pi(t_{i+1})$ as each target. The network learns the mapping

$$(z_1, \ldots, z_i) \to \pi(t_{i+1}).$$

Party B sees that the network predicts category 17, for example, and does not see which token that category represents. Training needs no interaction with Party A.

3.3 Private output head

Party A keeps the final layer of the language model, the output head, and computes the loss itself. Party B trains the transformer body and sends its final hidden states s_i to Party A:

$$\boxed{\text{Party B: transformer body}} \rightarrow s_i \rightarrow \boxed{\text{Party A: output head and loss}}.$$

Party A computes the loss against the true targets, updates the output head, and returns the gradient $\nabla_{s_i} L$. Party B continues backpropagation from that gradient. Party A therefore takes part in every training step, and Party B never receives a target in any form.

3.4 Security considerations

This use extends the setting of the initial Tensorlock manuscript in a way that weakens protection. In that manuscript, each example is a distinct continuous vector. Here, each token maps to one fixed vector that recurs throughout the corpus.

Repeated tokens can be grouped. Two occurrences of a token share the same signal component and differ only in padding and perturbation. Their ciphertexts are more similar to each other than to those of other tokens, and the preserved geometry exposes this similarity. Padding or perturbation strong enough to hide it would also prevent the network from recognizing tokens.

The corpus then resembles a substitution cipher. Party B can assign an anonymous class to each group of ciphertexts. It then holds the corpus as an ordered sequence of token classes. Token frequencies and co-occurrence patterns remain visible, and these statistics are the classical means of attack on a substitution cipher.

Vocabulary relabeling adds no protection beyond this. The target at position i is the input at position $i + 1$. Party B can therefore match each relabeled target to its input class.

The private output head leaks the same classes. The gradient that Party A returns depends on the target token. Gradients for the same target are similar, so Party B can group them as well.

The strength of the resulting protection depends on what the attacker knows about the language and the tokenizer. An attacker who knows both has strong statistical leverage. Language-model training on a private corpus therefore requires a dedicated security analysis before deployment with an adversarial Party B.

3.5 Contextual private encoders

Party A can extend the private encoder beyond the embedding table. A frozen stack of transformer layers at Party A produces contextual states, which differ at each occurrence of a token:

$$(t_1, \ldots, t_L) \xrightarrow{\text{private contextual encoder}} (h_1, \ldots, h_L) \xrightarrow{T} (z_1, \ldots, z_L).$$

Exact repetition no longer occurs on the input side, which makes grouping harder. Contextual states of one token remain related, and the extent of the improvement is untested. The targets of a language model remain discrete tokens, so the target-side leakage in Section 3.4 persists.

3.6 Inference

Party B passes the trained transformer to Party A. The inference oracle tokenizes the prompt, applies the private encoder and T, and runs the transformer. It maps each output to a token through π^{-1} or the private output head. It repeats these steps for each generated token and returns only the generated text.

No ciphertext leaves the inference oracle, and generation needs no exchange with Party B. A user who submits chosen prompts obtains text, and obtains no pairs of plaintext and ciphertext. Party A bears the computing cost of inference.

4. Text classification over privately encoded text

4.1 Method

Party A applies a frozen private text encoder to each document and obtains one vector:

$$\text{document} \xrightarrow{\text{private text encoder}} h \in \mathbb{R}^d \xrightarrow{T} z \xrightarrow{\text{dense layers}} \hat{y}.$$

The private text encoder performs tokenization, embedding, contextual encoding, and pooling. Party B receives one ciphertext vector and one label for each document, and it trains a fully connected classifier. Training needs no interaction with Party A.

Inference repeats the pipeline inside the inference oracle. The oracle encodes and encrypts the submitted document, runs the trained classifier, and returns only the label.

4.2 Relation to the initial Tensorlock manuscript

This design matches the setting of the initial Tensorlock manuscript, with the document vector h in the role of the plaintext example. Each example is a distinct continuous vector. Party B sees no tokens, no sequence order, and no sequence length. The grouping attack in Section 3.4 does not apply.

Document vectors are small. A vector of 1,024 dimensions needs an orthogonal matrix of about 8 MB at 64-bit precision.

4.3 Security considerations

This design stays within the threat model of the initial Tensorlock manuscript. Three points require attention.

Encoder secrecy. The initial Tensorlock manuscript finds that meaningful recovery requires exact or near-exact knowledge of plaintext covariance. A publicly available text encoder weakens this protection. An attacker can encode

similar public text with it and estimate the covariance of the document vectors. Party A should keep the encoder weights private.

Vector inversion. An attacker who recovers a document vector has not recovered the document. Document vectors can nevertheless disclose much of the text they encode, to a party that also holds the encoder.

Fixed encoder. The private text encoder is not trained on the encrypted task. Accuracy depends on how well the frozen encoder suits that task.

5. Generative models

5.1 The return path

Generation ends with decryption. For a generated ciphertext z', Party A computes

$$h' = \left(z' A^{\top}\right)_S ,$$

and passes h' to the private decoder. The orthogonal transformation and the index shuffle reverse exactly, and the key identifies the padding coordinates to discard.

Perturbation is not reversed, and no reversal is needed. The generative model learns the distribution of perturbed vectors, and its samples carry noise of the same size. The perturbation setting therefore trades security against sample fidelity. A private decoder that tolerates small noise in h' reduces the cost of this trade.

The generative model must also produce values in the padding coordinates. Those values are discarded at decryption, and they consume model capacity.

5.2 Generative models over encrypted latent representations

An autoencoder supplies a matched private encoder and private decoder. Party A trains the autoencoder on plaintext beforehand and then fixes it. The encoder maps each example x to a latent vector h, and Party A transmits $z = T(h)$.

Party B trains a generative model of the encrypted latent vectors. The model can be a generative adversarial network (GAN), a diffusion model, or an autoregressive transformer. Training needs no interaction with Party A. This placement also applies to a variational autoencoder (VAE), whose encoder and decoder remain with Party A.

5.3 Generative adversarial networks

A GAN pairs a generator G with a discriminator D. In ciphertext space, the generator maps a random vector r to a generated ciphertext, and the discriminator compares it with real ciphertexts:

$$G(r) \to z_{\text{generated}}, \qquad D(z) \to \{\text{real}, \text{generated}\}, \qquad z_{\text{real}} = T(h_{\text{real}}).$$

Neither network needs plaintext. The generator learns the distribution of ciphertext vectors, and its final linear layer can produce any coordinate basis.

Both networks must be fully connected, because ciphertext coordinates carry no spatial locality. For images, this favors a compact latent vector over raw pixels.

5.4 Diffusion models

A diffusion model learns to reverse a process that adds Gaussian noise in steps. In ciphertext space, the noised vector at step t is

$$z_t = \sqrt{\bar{\alpha}_t}\, z_0 + \sqrt{1 - \bar{\alpha}_t}\, \epsilon, \qquad \epsilon \sim \mathcal{N}(0, I_D).$$

Isotropic Gaussian noise keeps its distribution under orthogonal transformation. Adding noise to the ciphertext therefore equals adding noise to the padded plaintext vector and then encrypting it. The squared-error training loss is also unchanged by orthogonal transformation. Diffusion training in ciphertext space is thus the same problem as diffusion training on the padded representation, in a rotated basis.

The denoising network is the constraint. Image diffusion models commonly use convolutional denoisers, which need spatial locality that the ciphertext lacks.

Two alternatives avoid convolution. A fully connected denoiser can operate on a single latent vector. A transformer denoiser can operate on a sequence of latent patches, each encrypted with the same key.

Latent patches are continuous vectors that do not repeat, so the grouping attack in Section 3.4 does not apply. Party B does see the order of the patches.

5.5 Inference and security considerations

Party B passes the trained generative model to Party A. The inference oracle draws the random input, runs the model in ciphertext space, decrypts the result, and applies the private decoder. It returns only the generated plaintext.

Generation extends the threat model of the initial Tensorlock manuscript, which contains no decryption step. Four points require attention.

Decryption must stay closed. The inference oracle should decrypt only ciphertext that it generated itself. If it decrypted ciphertext supplied by another party, an attacker could submit training ciphertext and receive the corresponding plaintext.

Generated outputs describe the plaintext distribution. A user who collects generated samples can estimate the statistics of the training data. If Tensorlock encrypted raw plaintext, those samples would supply an estimate of plaintext covariance. This is the auxiliary knowledge that the initial Tensorlock manuscript identifies as the route to recovery. A private encoder between plaintext and encryption removes this route, because the attacker cannot compute the latent vectors of the samples.

Memorization. A generative model can reproduce individual training examples. Decrypted outputs can then disclose private data to their recipients, whatever encryption protected the training set.

Latent structure. Autoencoder latent spaces differ statistically from raw data. A latent space with nearly equal variance in every direction gives a covariance-based attack little structure to align. A latent space with statistically independent

coordinates may instead suit other methods of recovering the basis. Neither effect has been analyzed.

6. Comparison

Use	Encrypted representation	Party A components	Interaction during training	Relation to initial threat model
Fully connected classifier	Raw input vector	Encryption	None	Defines it
Text classification	One vector per document	Private text encoder	None	Within it; see Section 4.3
Language-model training, vocabulary relabeling	Token embeddings	Private embedding table, vocabulary permutation	None	Extends it; token classes visible
Language-model training, private output head	Token embeddings	Private embedding table, output head	At every training step	Extends it; token classes visible
Latent GAN or latent diffusion	Latent vectors or latent patches	Private encoder, private decoder	None	Extends it; decryption at inference

In every row, inference runs inside Party A's inference oracle, and no ciphertext leaves it.

7. Limitations

This paper describes designs and reports no measurements. The accuracy of transformers, GANs, and diffusion models trained on Tensorlock ciphertext remains untested. The equivalence arguments in Sections 2.2 and 5.4 show that suitable networks exist, but they do not show that training finds them.

The security considerations in Sections 3.4, 4.3, and 5.5 identify exposures. They do not bound them. Attacks on other data-encoding methods have shown that such transformations can contain structure that initial analysis misses.

The initial Tensorlock manuscript does not claim information-theoretic privacy or general ciphertext indistinguishability. That limitation applies to the potential designs described here.

8. Conclusion

Tensorlock extends beyond encrypted raw inputs to encrypted representation spaces. A private encoder at Party A produces the representation, and the remote party trains on its ciphertext.

The strength of each use depends on the representation. Document vectors and latent vectors are distinct and continuous, as the examples in the initial Tensorlock manuscript are. Text classification over privately encoded text stays within the initial threat model. Latent generative models add one requirement: a closed decryption step inside the inference oracle.

Token embeddings repeat, and repetition exposes token classes to the remote trainer. Language-model training on a private corpus is therefore the weakest of the uses described here, and it requires a security analysis of its own.