DLBox: New Model Training Framework for Protecting Training Data

Jaewon Hur
Seoul National University
hurjaewon@snu.ac.kr

Juheon Yi
Nokia Bell Labs
juheon.yi@nokia-bell-labs.com

Cheolwoo Myung
Seoul National University
cwmyung@snu.ac.kr

Sangyun Kim
Seoul National University
sangyun.kim@snu.ac.kr

Youngki Lee
Seoul National University
youngkilee@snu.ac.kr

Byoungyoung Lee†
Seoul National University
byoungyoung@snu.ac.kr

***Abstract*—Sharing training data for deep learning raises critical concerns about data leakage, as third-party AI developers take full control over the data once it is handed over to them. The problem becomes even worse if the model trained using the data should be returned to the third-party AI developers—e.g., healthcare startup training its own model using the medical data rented from a hospital. In this case, the malicious developers can easily leak the training data through the model as he can construct an arbitrary data flow between them—e.g., directly encoding raw training data into the model, or stealthily biasing the model to resemble the training data. However, current model training frameworks do not provide any protection to prevent such training data leakage, allowing the untrusted AI developers to leak the data without any restriction.**

This paper proposes DLBox, a new model training framework to minimize the attack vectors raised by untrusted AI developers. Since it is infeasible to completely prevent data leakage through the model, the goal of DLBox is to allow only a benign model training such that the data leakage through invalid paths are minimized. The key insight of DLBox is that the model training is a statistical process of learning common patterns from a dataset. Based on it, DLBox defines `DGM-Rules`, which determine whether a model training code from a developer is benign or not. Then, DLBox leverages confidential computing to redesign current model training framework, enforcing only `DGM-Rules`-based training. Therefore, untrusted AI developers are strictly limited to obtain only the benignly trained model, prohibited from intentionally leaking the data. We implemented the prototype of DLBox on `PyTorch` with `AMD SEV-SNP`, and demonstrated that DLBox eliminates large attack vectors by preventing previous attacks (e.g., data encoding, and gradient inversion) while imposing minimal performance overhead.

I. Introduction

Sharing training data for deep learning has been widely studied in literature [1], [2], [3], as it allows AI developers to improve their models using the dataset shared by others. Meanwhile, data owners can also benefit from sharing their proprietary data by receiving monetary rewards from the developers. For instance, it is common for healthcare startups to rent (or contract) medical data from hospitals to train their own models [1], [2]. As another example, startups aiming to prevent financial fraud often require large volumes of transaction data from the banks to train their fraud detection models [4], [5].

However, sharing training data raises concerns about data leakage, as the data is a valuable asset of the data owners (e.g., labeled MRI images collected over a long period). While the data owners wish to restrict the data to be used only for training an AI model, untrusted AI developers can easily leak the data once it is handed over to them, causing financial and reputational losses [6] to the data owners. Suppose an AI developer is allowed to access a dataset of a data owner, and he runs an arbitrary model training code (e.g., a Python [7] code) that reads the data and then trains a model. Once the data is read, it is remarkably easy for him to leak the data—e.g., sending it over network or saving it into his external disk.

For these reasons, data owners often restrict the computing environment for the AI developers. Specifically, the model training has to be performed in the physically and network-isolated environment, so called an air-gapped environment [8]. In this air-gapped environment, the developers' output is verified, and only the final trained model is allowed to be exported outside the environment. However, a malicious developer can still leak the data through the model, since he has a full control over how the data is computed into the model. For example, he may embed the encoded-data into the model, decoding it later to reconstruct the original data [9]. In the extreme case, every bit of the model can be abused to leak the training data, indicating whether a specific sample is used or not [10]. Due to this reason, data owners are still hesitant to share their data, preferring not to take the risk of data leakage [11], [12].

Despite the practical impact of this problem, to the best of our knowledge, there has been no systematic protection for the data in this problem setting. While several approaches have been proposed in the domain of privacy preserving machine

† Corresponding author

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.243001
www.ndss-symposium.org

learning [13], [14], [15], [16], we found they are not applicable to our scenario, where the untrusted AI developers have full control over the data. For example, federated learning [13] assumes the adversary does not have control over the data, limiting them to obtain only benignly computed gradients. However, the attackers in our scenario can perform arbitrary computations with the data, such as encoding it to the model to be leaked later [9]. Multi-party machine learning [16] cannot be applied either, as it assumes the model training code itself is benign—i.e., data leakage through the trained model is out-of-scope. However, we assume the untrusted AI developers can implement a malicious model training code to leak the data.

Unfortunately, we found that it is infeasible to completely prevent the malicious AI developers from leaking the data. This challenge is largely due to the nature of deep learning—i.e., the data (of the data owner) must be fed into the model, which eventually returns to the AI developers. As long as the model shows a reasonable performance, it should remember some information of the data, meaning that the model has to be computationally dependent to the data. Thus, the malicious developers would be able to reconstruct the data, once enough information is collected from the models.

Nevertheless, current model training frameworks [17], [18] offer no data protection at all, allowing malicious AI developers to easily leak the data, even without conducting sophisticated attacks (e.g., model inversion [19]). Thus, we design DLBox, a new model training framework to minimize the attack vectors raised by untrusted AI developers. Instead of trying to completely prevent data leakage through a model, the goal of DLBox is to allow only genuinely training an AI model such that the data leaks through invalid paths are minimized. The key insight of DLBox is that the model training is a statistical process of learning common patterns from a dataset. In other words, a benign model training program treats each data sample equally, so that each sample makes a fair contribution to train a model. As such, DLBox enforces the AI developers to perform only the benign model training, but prevents any other attempts not satisfying this observation, thereby preventing intentional data leakage.

To this end, we define `DGM-Rules`, which determine whether a model training from a developer is benign or not. To be specific, `DGM-Rules` concretize our key observation, checking whether each sample fairly contributes to the model throughout the entire training procedure. `DGM-Rules` consist of three rules, $\mathbf{R_D}$, $\mathbf{R_G}$, and $\mathbf{R_M}$, where each rule checks the procedure of **D**ata augmentation, **G**radients computation, and **M**odel update. In particular, $\mathbf{R_D}$ monitors whether the same augmentation is applied to all samples. $\mathbf{R_G}$ requires all gradients to be back-propagated from equally calculated loss values, and $\mathbf{R_M}$ enforces the model to be updated only by aggregating the gradients from $\mathbf{R_G}$. Then, we demonstrate that enforcing `DGM-Rules` can eliminate large attack vectors by preventing most of the previous attacks to leak the data [9], [20].

Based on the definition of `DGM-Rules`, DLBox redesigns current model training framework to enforce `DGM-Rules`-based training. In particular, DLBox consists of two components:

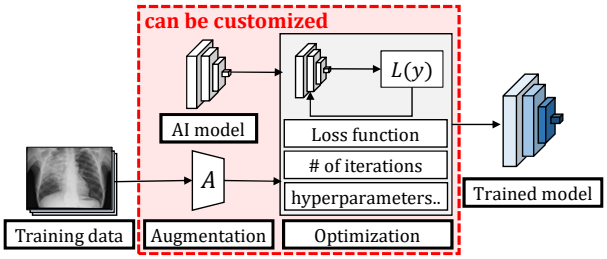


Figure 1: Model training procedure and customizable components.

i) `DL-Isolate`, which isolates the address space in the model training procedure to prevent untrusted code from leaking the data, and ii) `DL-Oracle`, which keeps track of the computations performed on the data to check whether the resulting model is trained under `DGM-Rules` or not. Thus, the developers obtain only the models trained under `DGM-Rules`, strictly limited from leaking the data. Meanwhile, all the components are protected by an enclave (e.g., AMD confidential VM [21]) such that both the data owners and AI developers can guarantee the confidentiality of their data and model training code.

We implemented DLBox on `PyTorch` [17], [22], which is widely used for deep learning, and we employed AMD SEV-SNP [21] to ensure its security guarantees. Thus, DLBox can readily train the state-of-the-art models with real-world datasets, while requiring minimal modification to the model training codes. We evaluated DLBox on image and language models across various tasks: i) image classification and segmentation tasks using fully-connected, convolution, `ResNet18` [23], `MobileNet-v2` [24], and `FCN` [25] models, and ii) sentiment analysis, translation and language modeling tasks using `Bert` [26], `T5` [27], and `Gpt2` [28] models. The evaluation results demonstrate that DLBox eliminates the large attack vectors by preventing `bit-encoding`, `memorizing`, and `gradient-inversion` attacks (i.e., categorized in §II-B) while imposing 4% overhead of learning time on average.

II. Background

In this section, we provide a brief background on deep learning (§II-A) and the attacks to leak shared data while training a model (§II-B). Then, we introduce confidential computing, which ensures confidentiality and integrity of DLBox (§II-C).

A. Deep Learning

A deep learning model is a function with a set of weights, taking input data and returning a corresponding output. For example, an image classification model takes an image as input and outputs the corresponding label of the image (e.g., dog or cat). To be specific, the deep learning model is composed of the model architecture and the weights. The model architecture determines how the input (e.g., an image) is computed into the output (e.g., a likelihood of each label). The weights determine the proportion of the feature value propagation in the architecture.

In order to maximize the model performance (e.g., classification accuracy), AI developers train their models using the data, so-called model training. The goal of model training is to find the best set of weights that maximizes the performance. As such, the model training optimizes a loss value, which penalizes the model when it returns a wrong output (e.g., an incorrect label) on a given input (e.g., image). The model training iteratively updates the weights by computing gradients from the loss values and subtracting it from the weights. The procedure terminates when the weights yield a sufficiently low loss value or high testing accuracy.

Model training can be done with a vast number of different ways as there are many customizable components as shown in Figure 1. To be specific, AI developers can customize i) the model architecture, ii) how the loss is computed, iii) how the weights are updated, and iv) hyperparameters such as the maximum number of epochs. Furthermore, the training data can be augmented through arbitrary operations (such as random flipping) as it is shown that such data augmentation outperforms the baseline [29], [30]. For these reasons, deep learning libraries such as `PyTorch` [17] do not restrict how the implementation of model training should be carried out. Instead, those leave the implementation details entirely up to the developers.

B. Attacks to Leak Shared Training Data

On the current model training frameworks, malicious AI developers have an unlimited number of ways to leak the training data through the model. This is because they can construct purely arbitrary data flows (i.e., model training code) from the data to the model, and the current model training frameworks do not impose any restriction on it. We enumerate a number of already known attacks in the following, but we want to note that the attackers are not limited as they can do whatever they want.

`bit-encoding-attacks`. First of all, malicious developers can simply leak training data by just encoding the raw data directly into the model. The encoded bits can be the data samples itself, but also the set of binary values conveying any information of the data. For example, a malicious AI developer may abuse each bit of the model to indicate the existence of specific samples, reconstructing those samples from the model later. To be specific, he would construct a codebook beforehand that indicates which bit (of a model) corresponds to which sample—e.g., 3rd bit corresponds to the sample `x`. Then, while training, he can set each bit of the model depending on the existence of each sample. Finally, he would be able to reconstruct the data from the model using the codebook—e.g., if 3rd bit is set in the model, reconstruct the sample `x`.

`memorizing-attacks`. More skillful attackers would perform memorizing attacks [9], with which the model stealthily memorizes the data without compromising its performance. With this attack, the resulting model looks benignly trained (e.g., correctly classifying images). However, the model is actually used as a covert channel to leak the training data. For instance, Song et al. [9] demonstrated that using the training data as a regularization term can bias the model to resemble the data (while also working well), thus eventually leaking the data.

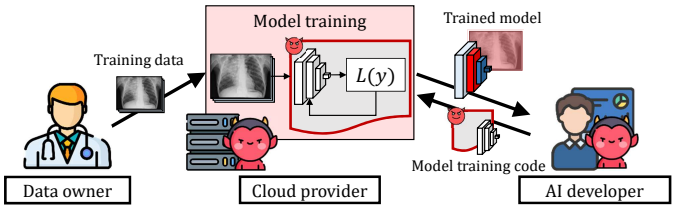


Figure 2: System model of DLBox.

`gradient-inversion-attacks`. Gradient inversion attacks [31], [20], [32] can also be retrofitted to leak the training data by reconstructing it from the gradients of a model. To be specific, malicious developers may obtain two versions of model weights which differ by only a gradient of a certain sample, and retrieve the gradient. Then, they can perform already studied gradient inversion attacks [31], [20], [32] to reconstruct the sample from the retrieved gradient. However, it is known that the performance of gradient inversion attacks sharply decreases if the target gradients are aggregated from multiple samples [32].

`model-inversion-attacks`. Finally, malicious AI developers can perform model inversion attacks [33], [19], [34], which directly reconstruct the training data from the model. These attacks are known to be not effective without auxiliary datasets [19], but the malicious developers may try a variant of these attacks. For example, they may train a model with only a single sample (ignoring all others) in order to bias the model to be more vulnerable to the inversion attacks.

C. Confidential Computing

Confidential computing is a security service provided by the trusted execution environments (TEE) of CPUs (e.g., Intel SGX [35], TDX [36], and AMD SEV [21]) and GPUs (e.g., NVIDIA H100 [37]), so called enclave. Enclave is protected against all privileged components including OSes, hypervisors, and even SoC, thus guaranteeing the integrity and confidentiality of a program running on it. Furthermore, the enclaves provide a remote attestation, by which the owner of an enclave can verify a correct program is loaded before sending their data to it [35], [21]. Thus, emerging cloud applications are using (or expected to use) confidential computing to provide a higher level of security in cloud computing environment (e.g., `Spark` [38], and `Hadoop` [39]).

III. Motivation

In this section, we introduce the system model and the problem setting that DLBox focuses on (§III-A). Then, we formally describe the problem, and discuss the infeasibility of preventing data leaks in such situation (§III-B). Finally, we briefly introduce our solution to the problem (§III-C).

A. System Model of DLBox

DLBox focuses on the problem arising while sharing the data for training a deep learning model—e.g., healthcare

startups training a cancer diagnosis AI on MRI images shared from a hospital [1], [2]. As shown in Figure 2, DLBox assumes three parties involved in this situation: a data owner, an AI developer, and a cloud provider.

- **Data owner** shares his training data to the AI developers for training their models. While allowing the developers to train their models with his data, he does not want them to leak or abuse the data. This is because the data itself is a valuable asset of the data owner, which often holds a significant monetary value for trading (e.g., Reddit's sale of user data for AI training [40]).
- **AI developer** trains his own model using the shared data by his own model training code. However, he does not want to reveal his model training code and the resulting trained models, as they are also the proprietary assets. For instance, deep learning algorithm has become more important than ever, as it has been shown that a novel model architecture with trained weights can outperform the human (e.g., GPT [41]). This trend is expected to be even stronger as the companies and nations recognize the devastating power of AI [42], [4], [41].
- **Cloud provider** runs the training code on behalf of the AI developer. Thus, the cloud provider receives both the training data (provided by the data owner) and the model training code (provided by the AI developer), and returns the trained model (to the developer).

Problem Setting. In this situation, the training data (from the data owner) is not faithfully protected from malicious AI developers and cloud providers. The data can be easily leaked and abused, since it is under the full control of model training code (of the AI developer), running on a cloud platform (of the cloud provider). In the current model training framework, the malicious developers can easily leak the data over the network or through a model, and the cloud provider can easily access and leak the data.

However, merely opening (and verifying) the model training code and protecting its execution environment cannot be the solution as it harms the requirements of AI developers. Furthermore, *it is not universally defined what a benign model training is, which does not incur (or at least minimizes) the data leakage*. Given there are vast range of different model training algorithms (e.g., different data augmentations [29], [30], and model architectures [43], [25], [41]), it is even challenging to manually determine whether a given model training code is benign.

Limitations of Previous Approaches. In this respect, we found that previous approaches in privacy preserving machine learning [13], [14], [15], [16] cannot be applied to our problem either. Federated learning [13] assumes the attacker does not have control over the data, limiting them to obtain only benignly computed gradients. However, we assume the attacker controls the entire data flow, thus easily embedding the data to the model or maliciously biasing the gradients [9]. Differential privacy [15] does not apply to general model training algorithms [29], [17], limited in its usability and scalability. Multi-party machine learning [16] assumes the model training code itself is benign, but only ensure that the code is jointly computed without exposing private data. However, we assume the code itself can be malicious to leak the data (e.g., embedding the data to the model). We leave further discussion on the related works in §X.

B. Infeasibility of Preventing Data Leaks through the Model

Given this problem scope, our first observation is that it is impossible to completely prevent data leaks in the model training. To be specific, as long as the model is trained on the data, it should eventually remember the information of data, and the AI developers finally obtain the model. Thus, the training data should also be leaked through the model.

In order to clarify this point, we formally describe the problem setting and discuss its unsolvable nature in the following.

Problem Setting: Model Training on Shared Data. Based on the explanation in §III-A, we formally describe the problem setting as follows:

> Given training data samples $\{x_i \mid x_i \in S \wedge 1 \leq i \leq N\}$ from the data owner, where S is the sample space and N is the number of samples, an AI developer obtains a trained model $\theta_{\texttt{model}} = f(x_1, ..., x_N)$, where f is an arbitrary model training function (i.e., code) developed by the developer.

With this definition, it is trivial for a malicious AI developer to leak the training data. As an extreme yet straightforward example, the malicious developer can steal the data sample itself by providing an identity function $f(x_1, ..., x_N) = x_c$ (where c is any constant), obtaining the output $\theta_{\texttt{model}} = f(x_1, ..., x_N) = x_c$. Even a boolean function $f(x_1, ..., x_N) = bool(x_c = \alpha)$ would leak the information of x_c, telling whether it is α or not. As a real world example, it is possible to just encode the raw data samples into the model.

Observation: Infeasibility of Preventing Data Leaks through the Model. In order to prevent such data leaks, we should find a set of ideal model training functions $F_{\texttt{ideal}}$, which makes it impossible to reconstruct any data sample from the output trained model. Thus, we can prevent the data leaks by enforcing the AI developers to compute only such ideal model training functions. To be specific, $F_{\texttt{ideal}}$ should satisfy following properties:

> $F_{\texttt{ideal}}$ is a set of functions, where $f_{\texttt{ideal}} \in F_{\texttt{ideal}}$ has following two properties:
> 1) $f_{\texttt{ideal}}$ is able to train a deep learning model.
> 2) Given $f_{\texttt{ideal}}$, it is impossible to reconstruct any sample x_i from the trained model $\theta_{\texttt{model}} = f_{\texttt{ideal}}(x_1, ..., x_N)$.

However, such $F_{\texttt{ideal}}$ does not exist as it is studied before in the differential privacy literature [44], and thus, it is infeasible to prevent data leaks in model training. While we do not provide the proof here, the reasoning is fairly straightforward. As long as the AI developer obtains a reasonably working model, which

is trained on the shared data, the model necessarily contains the information of the data, and the developer obtains it. A malicious AI developer would be able to reconstruct the training data once the enough amount of information is collected from the trained models.

C. Our Solution: Enforcing a Benign Model Training

Despite the practical impact of this problem, current model training frameworks [17], [18] provide no data protection at all, allowing the malicious AI developers to easily leak all the samples. On the other hand, the unsolvable nature of the problem hinders the community from proceeding to a more viable (but possibly incomplete) solution. Thus, sharing the training data for deep learning is still largely prohibited due to the aforementioned security reasons.

To this end, we aim at designing a baseline protection for the data, which is still incomplete, but effectively eliminates large attack vectors in the model training. Instead of trying to completely prevent data leaks through a model, we try to enforce only a benign model training, thereby preventing the data leakage through invalid paths. In particular, our key observation is that *the model training is a statistical process of learning common patterns from a dataset.* That is, the benign model training should treat each data sample equally, thus making a fair contribution to the model.

Based on this observation, we introduce `DGM-Rules`, which determine whether a model training code from an AI developer is benign or not (i.e., §IV). Specifically, `DGM-Rules` check whether the model training procedure satisfies our key observation (i.e., equal treatment of the samples) so that the model is benignly trained. Then, we justify that the malicious attempts to intentionally leak the data do not satisfy `DGM-Rules`, convincing its security properties (i.e., §VII).

Then, we design DLBox to enforce `DGM-Rules`-based model training (i.e., §V). In particular, DLBox sandboxes the untrusted model training code, and monitors the entire training procedure to check whether the model is trained under `DGM-Rules`. Thus, the AI developers can obtain only the models trained under `DGM-Rules`, strictly limited from intentionally leaking the data. In addition, DLBox is further protected by confidential computing to meet the security requirements of both the data owner and developers.

IV. DGM-Rules: Determining a Benign Model Training

`DGM-Rules` determine whether a given model training code from an AI developer is benign (i.e., genuinely training a model) or not. Our key observation behind `DGM-Rules` is that the model training is a statistical process of learning common patterns from a dataset. In other words, the benign model training should treat each data sample equally, and the samples should have fair contributions to the output model. Thus, we define `DGM-Rules` to concretize this key observation in the context of model training code.

Algorithm 1 Typical model training procedure.

Input:
1: $X = \{x_1, ..., x_N\}$: training data samples
Output:
2: θ_T: trained model weights after T epochs
Body:
1: **Set** augmentation $A(x_n)$,
2: loss function $L(\theta_t, \hat{x}_n)$,
3: learning rate η
4: **Initialize** model weights θ_0
5: **Augment samples**
6: $\hat{X} \leftarrow \{\hat{x}_n | \hat{x}_n = A(x_n) \wedge x_n \in X\}$
7: **Split samples to mini-batch**
8: $\{B_1, ..., B_M\} \leftarrow \hat{X}$
9: **for** each epoch $t \in [T]$ **do**
10: **for** each mini-batch $B_m \in \{B_1, ..., B_M\}$ **do**
11: **Compute gradients**
12: For each $\hat{x}_n \in B_m$, compute $g_m(\hat{x}_n) \leftarrow \nabla_{\theta_t} L(\theta_t, \hat{x}_n)$
13: **Average gradients**
14: $\bar{g}_m \leftarrow \frac{1}{|B_m|} \Sigma g_m(\hat{x}_n)$
15: **Descent**
16: $\theta_t \leftarrow \theta_t - \eta \cdot \bar{g}_m$
17: **Update** model weights $\theta_{t+1} \leftarrow \theta_t$
18: **return** θ_T

A. Definition of DGM-Rules

In order to define `DGM-Rules`, we start from a typical model training procedure as shown in Algorithm 1. The model training procedure can be characterized into three phases: i) data augmentation (i.e., lines 5-6), ii) gradients computation (i.e., lines 11-12), and iii) model update (i.e., lines 13-17). Specifically, the data augmentation applies a variation (e.g., random flipping [30]) to each sample in order to improve the robustness of the model [29], [30]. Then, gradients are computed from each (augmented) sample through equally calculating a loss value and back-propagating against the model weights. Finally, the model is updated by adjusting the gradients, where all the gradients (from each sample) are used for each epoch.

The key take away from this observation is that each sample goes through the same computation (except the different model weights) to be the gradients, and all the samples are used exclusively and exhaustively to update the model. This is because the goal of deep learning is to learn common patterns from the samples, and there is no need to prioritize or discriminate a specific sample. While data augmentation techniques such as a resampling [30] may oversample (or undersample) the samples under a certain label, still there is no need to discriminate a specific sample among those of the same label. We discuss the generality of `DGM-Rules` in §IX.

Based on this observation, we define `DGM-Rules` $\mathbf{R_D}$, $\mathbf{R_G}$, and $\mathbf{R_M}$ for each **D**ata augmentation, **G**radients computation, and **M**odel update phase as shown in Table I. Again, `DGM-Rules` capture the common nature of deep learning, which treats each sample equally to contribute to the model. To be specific, the rule for **D**ata augmentation (i.e., $\mathbf{R_D}$) requires the samples to

Table I: Definition of DGM-Rules.

Phase	DGM-Rules
Data augmentation	$\mathbf{R_D}$: All samples must go through the same augmentation.
Gradients computation	$\mathbf{R_G}$: All gradients must be computed by back-propagating a loss value, which is equally computed from each sample.
Model update	$\mathbf{R_M}$: Model must only be updated by exhaustively adjusting the gradients from each sample, which are computed under $\mathbf{R_G}$.

go through the same augmentation operations. The rule for **G**radients computation (i.e., $\mathbf{R_G}$) enforces the gradients to be back-propagated from equally computed loss values, and the rule for **M**odel update (i.e., $\mathbf{R_M}$) enforces the model to be updated by adjusting all the gradients from $\mathbf{R_G}$, exhaustively and exclusively. Thus, any benign model training would satisfy DGM-Rules regardless of the augmentation operations [29], [30], loss functions [45], and model update strategies [46], as long as it treats each sample equally. Based on this definition, we justify in §VII that DGM-Rules successfully prevent the previous attacks to leak the training data (i.e., explained in §II-B).

V. DLBox: Enforcing DGM-Rules while Model Training

Then, we design DLBox based on confidential computing to enforce DGM-Rules while training a model on shared data. DLBox is a secure model training framework with which AI developers can train their models without revealing their code, while the data owners can be assured the security of their data. In addition, the data is further protected against malicious cloud providers leveraging the inherent security guarantees of confidential computing [36], [21].

Threat Model. DLBox employs CPU and GPU enclaves [35], [36], [21] in the untrusted cloud, ensuring its confidentiality and integrity to the other parties (i.e., data owners and AI developers). Thus, it does not trust any other entity outside the enclave which includes untrusted OSes, hypervisors, and peripheral devices. We assume the implementation of DLBox is correct such that a malicious AI developers and cloud providers cannot compromise its internal logic. On the other hand, we do not consider the general security limitations of confidential computing, such as micro-architectural side channels [47], [48], [49] and Iago attacks [50].

Overview. The overall design of DLBox is shown in Figure 3. DLBox takes the training data and model training code as input, and outputs only the benignly trained models. Thus, the data owner can be assured that DLBox prevents any unauthorized data leakage (not satisfying DGM-Rules), while the developers can train their models without revealing their code and models.

We describe the overall workflow of DLBox as follows: 1) a data owner uploads his data into the enclave after attesting DLBox is correctly loaded (① in Figure 3), 2) then, an AI developer runs his model training code (e.g., Python script using PyTorch [17]) on DLBox to train his own model (②). 3) Meanwhile, the code and data are located in different address spaces so that the developer's code cannot directly access the data (③). Instead, operations on the data are only performed through sanitized APIs which are invoked by the

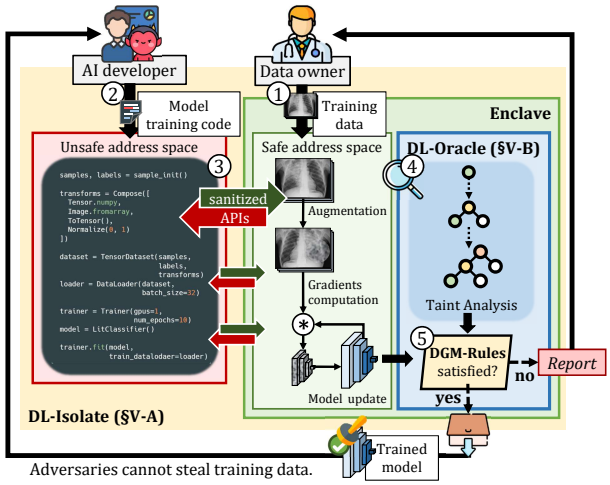


Figure 3: Overall framework of DLBox

code. 4) During the code execution, DLBox continuously traces the operations performed on the data while training a model (④). These traces are used later to check DGM-Rules. 5) Finally, when the model training ends, DLBox checks if DGM-Rules are satisfied, and returns the model to the developer only when it is true (⑤).

To this end, DLBox designs two key components: 1) DL-Isolate, which isolates the address space of the code and data to prevent unauthorized accesses (§V-A), and 2) DL-Oracle, which monitors end-to-end data flows from the data to the model, and determines whether the model is trained under DGM-Rules (§V-B). We describe the designs of these key components in the following.

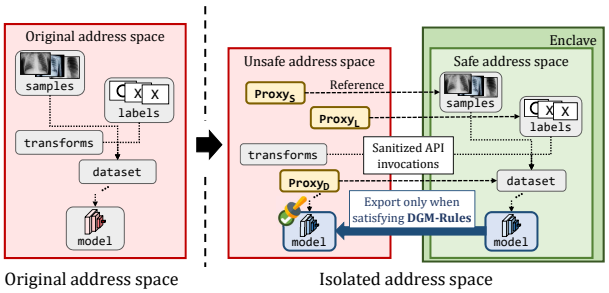
A. Preventing Unauthorized Accesses with DL-Isolate

In order to prevent data leaks, DLBox needs to guarantee that i) the AI developers cannot directly access the data, and ii) they cannot tamper with DLBox's logic (e.g., DGM-Rules checking mechanism). However, current model training frameworks [17], [18] do not provide such protections as they do not consider the developers themselves can be malicious, and thus, the data is under full control of the developers. Suppose an example model training code as shown in Figure 4-(a), which is a Python script using PyTorch, for MNIST classification [51]. Upon running the script, the code and data are loaded into the same address space (as naturally) as shown in the original address space of Figure 4-(b). Within the same address space, the code confronts no restriction to access, modify, and even leak the data (through I/O devices or models).

Thus, DL-Isolate isolates the unified address space into i) unsafe address space, where the model training code runs, and

```
1  def main():
2      samples, labels = sample_init()
3
4      transforms = Compose([ToTensor(), Normalize(0, 1)])
5      samples = [transforms(s) for s in samples]
6
7      dataset = TensorDataset(samples, labels) # Dataset
8      loader = DataLoader(dataset, batch_size=32)
9
10     model = plModule() # Training-Template
11
12     trainer = Trainer(gpus=1, num_epochs=10)
13     trainer.fit(model, train_dataloaders=loader)
```

(a) Example model training code.



(b) Isolated address space by `DL-Isolate`.

Figure 4: Design of `DL-Isolate`.

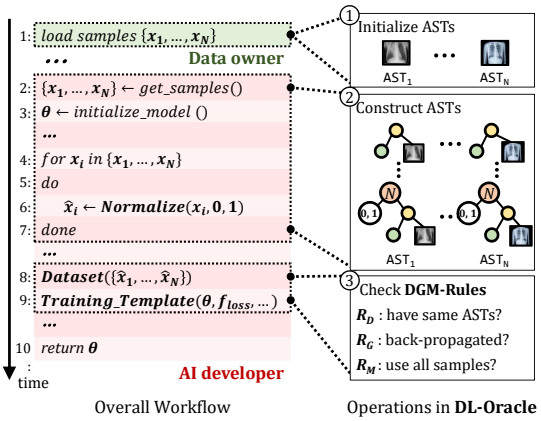


Figure 5: Design of `DL-Oracle`.

ii) safe address space, where the training data (and its following data flows) are loaded (i.e., isolated address space in Figure 4-(b)). In particular, the safe address space is populated in an enclave so that the model training code, as well as the untrusted OSes and hypervisors, cannot directly access the data. Furthermore, all data flows derived from the training data (e.g., computed gradients) are confined in the safe address space such that the model training code cannot get any (partial) information of the data not satisfying `DGM-Rules`.

Meanwhile, `DL-Isolate` provides sanitized APIs for the model training code to perform operations on the data. Especially, the safe address space loads trusted libraries for deep learning (e.g., `PyTorch` [17]), and allows the code from unsafe address space to invoke (trusted) functions on the data. More specifically, `DL-Isolate` introduces a `Proxy` variable inside the unsafe address space (e.g., `Proxy`$_S$), which references each object in the safe address space (e.g., `samples`). Thus, the code can invoke a function on the referenced object by requesting it through the corresponding `Proxy` variable. However, all resulting objects of the functions do not leave the safe address space, and the untrusted code only receives the references to those new objects (e.g., `Proxy`$_D$ referencing `dataset`). This prevents the code from getting any information of the data. Finally, only a benignly trained model, which satisfies `DGM-Rules`, is exported to the unsafe address space for the developers to use it.

On the other hand, this strict address space isolation may hinder the model training procedure in terms of i) debugging and hyper-parameters tuning, and ii) exception handling. For instance, a developer would not be able to visualize a data augmentation procedure or analyze a specific sample, as such data flows cannot leave the safe address space—i.e., they do not satisfy `DGM-Rules`. Furthermore, exception information while processing the data would not be exposed, leaving the developers in the dark. In order to alleviate such cases, `DL-Isolate` allows the data owner to configure a limited number of data flows to be exposed to the developers, even if they do not satisfy `DGM-Rules`. We discuss these trade-offs between security and usability in §IX.

B. Checking Model Training Procedure with `DL-Oracle`

With the help of `DL-Isolate`, `DL-Oracle` securely monitors the model training procedure performed by the AI developers. Especially, `DL-Oracle` traces end-to-end data flows from the samples to the model so that it can check whether the model is trained under `DGM-Rules`. As explained in §IV, `DL-Oracle` checks whether i) all data samples go through the same augmentation (i.e., $\mathbf{R_D}$), ii) all gradients are computed by back-propagating a loss value, which is equally computed from each (augmented) sample (i.e., $\mathbf{R_G}$), and iii) the model is updated only by adjusting all the gradients from each sample, which is computed under $\mathbf{R_G}$ (i.e., $\mathbf{R_M}$).

To this end, `DL-Oracle` designs a taint tracking mechanism that i) traces the operations on the data (e.g., line 5 of Figure 4-(a)) to remember the augmentation history, and ii) sanitizes the taints by hooking the model training APIs (e.g., line 7 and 10 of Figure 4-(a)) to confirm `DGM-Rules` are satisfied. Thus, `DL-Oracle` returns the trained model only when the untrusted code satisfies `DGM-Rules` by invoking the model training APIs with equally tainted samples (i.e., equally augmented samples).

In order to trace the augmentation history of the samples, `DL-Oracle` constructs an abstract syntax tree (AST) for each sample, inserting a node to the AST whenever an operation is applied. For example as shown in Figure 5, upon loading the samples into the safe address space, `DL-Oracle` initializes the ASTs with a single (leaf) node denoting each sample (i.e., ①). Then, for each augmentation operation (e.g., *Normalize*$(x_i, 0, 1)$ in line 6), `DL-Oracle` inserts a root node denoting the operation, with leaf nodes denoting the arguments passed together, if any (i.e., ②). Thus, the ASTs are constructed throughout the entire model training procedure, and the samples

with the same augmentation history construct the same ASTs (except the sample itself).

Then, DL-Oracle checks if DGM-Rules are satisfied by hooking the boilerplate model training APIs, which are commonly used in deep learning to standardize the training procedures [22], [17], [18] (i.e., line 8, 9). Specifically, DL-Oracle leverages i) Dataset API [52], [53], which is used to pipeline the given (augmented) samples to the model training procedure, and ii) Training-Template API [54], [55], [56], which provides a unified interface to implement the loss function, and model update strategies. Thus, DL-Oracle requires the developers to use these boilerplate APIs. DL-Oracle then checks whether they correctly employ those APIs following DGM-Rules.

Upon the invocation of model training APIs, we check the rules as follows (i.e., ③). For the data augmentation rule $\mathbf{R_D}$, we check that all samples used in initializing the Dataset have the same AST, indicating the same augmentation history. Then, for the gradients computation rule $\mathbf{R_G}$, we analyze that the code for loss function implemented in Training-Template is self-confined—i.e., the code uses only the sample given as an argument, and not referencing any external variables. Since the Training-Template uses the same loss function code for every sample [54], [55], [56], and back-propagates the loss values to be the gradients, we can guarantee the rule $\mathbf{R_G}$ is satisfied. Finally, for the model update rule $\mathbf{R_M}$, we check that all the samples (initially loaded by the data owner) are used without any duplication in initializing the Dataset. As the Dataset uniformly pipelines the given samples (by default) for gradients computation and model update [52], [53], we can guarantee that the rule $\mathbf{R_M}$ is satisfied.

When all the rules are satisfied, DL-Oracle exports the trained model so that the developers can freely utilize it. While the current design of DL-Oracle limits the model training code to use only the boilerplate APIs, we want to note that using those APIs is becoming more popular in the AI industry [57]. It is our future work to extend DL-Oracle to support arbitrary model training code.

VI. Implementation

We implemented DLBox on Python [7] to support PyTorch [17], which is a widely used deep learning library. Especially, we packaged DLBox as a library, thus the AI developers use DLBox by importing it in a Python script, or Jupyter Notebook [58]. In addition, we employed AMD SEV-SNP [21] to isolate and protect the safe address space inside a confidential VM.

Implementation of DL-Isolate. We implemented DL-Isolate by compartmentalizing the original Python process into i) a client process, which populates the unsafe address space, and ii) a server process, which populates the safe address space in a confidential VM. In particular, each process populates its own Python context, which loads libraries and runs a code as usual. However, the AI developer's code executes on the client process only, while the data is initially loaded into the server process, which does not directly run the developer's code. As the client process and server process are isolated at

```
1  def main():
2  ①   samples = Proxy('_SAMPLES')
3      labels = Proxy('_LABELS')
4
5      transform = transforms.Compose([
6          transforms.Normalize(0, 1)
7      ])
8
9      samples = [transform(s) for s in samples]
10     samples = torch.cat(samples)
11 ②   dataset = TensorDataset(samples, labels)
12     loader = DataLoader(dataset, batch_size=32)
13 ③   model = plModule()
14
15     trainer = Trainer(gpus=1, max_epochs=10)
16     trainer.fit(model, train_dataloaders=loader)
17 class plModule(pytorch_lightning.LightningModule):
18     ...
19 ③   def training_step(self, batch, batch_idx):
20         imgs, labels = batch
21         preds = self.model(imgs)
22         loss = self.loss_fun(preds, labels)
23
24         return loss
```

Figure 6: Example code for training ResNet18 [23] model on Cifar10 [61] dataset in DLBox.

virtualization layer (and even confidential VM), the malicious developers cannot affect the server process even if they control the entire software stack (e.g., building and running a malicious Python C extension module [59]).

Meanwhile, the server process loads trusted libraries, and we implemented the client process to invoke the functions in server process through remote procedure calls (i.e., gRPC [60]). We assumed open-source libraries such as PyTorch [17] are trusted as they are rigorously managed by maintainers, but the decision for trusted libraries depends on the data owners. We implemented the Proxy variables in the client (as explained in §V-A) to seamlessly convert a function call to a gRPC, which invokes the corresponding function on the referenced object in the server. For example, suppose a Tensor [17] variable S in server and its Proxy variable P in client. A model training code would invoke the mean method of P through P.mean() as usual (assuming it is a Tensor), but P would internally forward the method to S, returning a reference to the resulting object of S.mean(). Thus, actual results of the functions remain in the safe server process, not leaking the data.

We implemented Proxy with 1,700 Lines of Code (LoC) in Python and the gRPC communication with 150 LoC for the client and server each. In addition, we implemented a wrapper library that helps a model training code to seamlessly use DLBox package in 300 LoC. We discuss the design considerations for implementing DL-Isolate regarding the usability and performance, in §IX.

Implementation of DL-Oracle. We implemented DL-Oracle to trace every function invocation (through gRPC) on the samples and its descendant objects to build ASTs. In particular, we applied a minor optimization that remembers the AST in a hash representation as like a merkle tree [62]—i.e., the core taint algorithm is the same as explained in §V-B. For each gRPC invocation, we computed a unique hash from the requested function and arguments if any, and hashed it again with the

AST summary (i.e., hash value summarizing the AST) of the referenced object to get the summary for the resulting object. Thus, the same AST summary represents the same AST.

For checking `DGM-Rules`, we used `TensorDataset` in `PyTorch` [17], and `LightningModule` in `PyTorchLightning` [22] as the boilerplate model training APIs. Whenever a variable of such classes are declared, we checked whether the declaration follows `DGM-Rules` as explained in §V-B. To be specific, when declaring `TensorDataset`, we checked the argument, `tensors`, follows the rules $\mathbf{R_D}$ and $\mathbf{R_M}$, and when declaring `LightningModule`, we checked the implementation of `training_step`, which contains the loss calculation, follows the rule $\mathbf{R_G}$. Finally, when a `Trainer.fit` method (i.e., actual model training) is invoked with the verified `TensorDataset` and `LightningModule`, `DL-Oracle` returns the benignly trained model.

We implemented dynamic taint analysis in-house, and static taint analysis based on `PyT` [63]. Dynamic taint analysis is used to trace the runtime operations on data samples, and the static analysis is used to check the code used in declaring the `LightningModule` variable. We implemented the taint analysis with 1000 LoC in total.

Case Study. An example code for training a model on DLBox is illustrated in Figure 6. The code runs as usual, but DLBox seamlessly converts the function calls to RPC towards the objects in safe address space. During the augmentation, ASTs for the `samples` are continuously constructed to remember the histories (i.e., ①). Then, the rules $\mathbf{R_D}$ and $\mathbf{R_M}$ are checked when the `dataset` (i.e., `TensorDataset`) is declared using the `samples` (i.e., ②). After that, when the `model` (i.e., `LightningModule`) is declared, the rule $\mathbf{R_G}$ is checked by analyzing the implementation of `training_step` (i.e., ③). Finally, invoking `trainer.fit` method (i.e., line 15) successfully trains and returns the actual model only when the rules are satisfied.

VII. Security Analysis

DLBox can effectively eliminate large attack vectors in model training by enforcing `DGM-Rules`. In this section, we explain how enforcing each `DGM-Rule` (i.e., $\mathbf{R_D}$, $\mathbf{R_G}$, and $\mathbf{R_M}$) can prevent the attacks explained in §II-B.

Preventing `bit-encoding-attacks`. `bit-encoding-attacks` do not work under the gradients computation rule $\mathbf{R_G}$ and model update rule $\mathbf{R_M}$. The main assumption exploited by `bit-encoding-attacks` is that the attacker can construct an invertible data flow from the data to the model—e.g., encoding samples into the model is an identity function which is invertible. Enforcing $\mathbf{R_G}$ and $\mathbf{R_M}$ breaks this assumption. To be specific, $\mathbf{R_G}$ enforces the gradients to be computed through back-propagation, which is a non-invertible computation [20], and $\mathbf{R_M}$ further makes it impossible to invert by aggregating all the non-invertible gradients to update the model.

Preventing `memorizing-attacks`. `memorizing-attacks` are prevented by the data augmentation rule $\mathbf{R_D}$ and gradients computation rule $\mathbf{R_G}$. In order to launch the `memorizing-attacks`, specific target samples should be used extra-ordinarily to sneakily encode their information to the model. For example, an attack that biases a model to resemble specific samples exclusively uses them to obtain the loss value, thereby penalizing the model for diverging from the samples [9]. This attack fails under $\mathbf{R_D}$ and $\mathbf{R_G}$, since they ensure the loss value is computed equally from each sample, not prioritizing or discriminating any specific sample.

Preventing `gradient-inversion-attacks`. We can prevent `gradient-inversion-attacks` under the model update rule $\mathbf{R_M}$ unless the size of dataset is remarkably small. Since $\mathbf{R_M}$ enforces the model to be updated by aggregating all the gradients from each sample, the attacker cannot get a model which differs by only a single gradient of a sample. Thus, he cannot retrieve the single gradient also. While the attacker may obtain the gradients aggregated over all samples even under $\mathbf{R_M}$ (by comparing two models that differ by a single epoch), previous works have demonstrated that it is almost impossible to invert gradients averaged over 10 samples [20].

However, prior knowledge of a sample (e.g., partial features that are already leaked) can lead to an entire leakage of that sample. To be specific, the attacker may implement a filter function on those known features such that only the (partially known) sample is aggregated to the model while the others are zeroed out. This strategy still satisfies the model update rule $\mathbf{R_M}$ (as well as the data augmentation rule $\mathbf{R_D}$), because all the samples are aggregated (after applying the same function). Nevertheless, such a leakage would have limited security impacts only to the samples with known features.

Preventing `model-inversion-attacks`. Enforcing all $\mathbf{R_D}$, $\mathbf{R_G}$, and $\mathbf{R_M}$ still cannot prevent `model-inversion-attacks` as the attacks are designed to directly retrieve the data from the (benignly trained) model. However, we want to note that `model-inversion-attacks` are known to be not very effective without auxiliary datasets [19], and enforcing `DGM-Rules` is still meaningful to minimize possible attack surfaces. For example, it would not be possible under `DGM-Rules` to build a vulnerable model by training it with only a single sample (and ignoring all others).

VIII. Evaluation

In order to clearly demonstrate the practical impact of DLBox, we design an experiment to measure the security enhancement and performance overhead of DLBox. To this end, we answer following research questions:

1) How much can DLBox prevent malicious AI developers from leaking the training data? (§VIII-C)
2) How much does DLBox affect the performance of model training? (§VIII-D, §VIII-E)

A. Evaluation Setup

We evaluated DLBox with the scenarios of training the models across two domains: i) image processing, and ii) natural language processing. Especially, we performed an in-depth analysis of security enhancement and performance overhead of DLBox on the image processing models. Then, we

Table II: Image processing tasks and datasets for each task.

Task	Image classification			Image segmentation
Dataset	`Cifar10` [61]	`UTKFace` [66]	`ChestXray` [67], [68]	`SpinalCordMRI` [69]
# of samples	60,000	20,000	15,000	40
Sample spec.	3×32×32	3×200×200	3×512×512	1×512×512

further evaluated DLBox on language models to show its applicability.

We ran all the experiments on the AMD EPYC 7313 CPU with Ubuntu 22.04 server, which supports SEV-SNP [21]. DLBox runs on a confidential virtual machine (VM) protected by the SEV-SNP. We assume the GPU is trusted, and used NVidia GeForce RTX 6000 which is dedicated to the VM using PCI passthrough [64]. While we currently assume the normal GPUs as trusted, we note that employing TEE enabled GPUs (e.g., NVidia H100 [37], Graviton [65]) does not change the design of DLBox.

B. Datasets for Image Processing

We trained image processing models on two different tasks: i) image classification, and ii) image segmentation. For each task, we used following datasets.

Image Classification. As shown in Table II, we used three datasets for image classification as follows: i) **`Cifar10`**, which consists of the images in 10 classes (e.g., airplane, bird, cat, etc.) [61], ii) **`UTKFace`**, which contains the images labeled with the age, the gender, the race of the person and the date&time the picture was taken [66]. iii) **`ChestXray`**, which contains chest X-ray images of the people who have been infected with Covid-19 or not [67], [68].

Image Segmentation. For image segmentation, we used **`SpinalCordMRI`** dataset, which has 40 MRI images containing spinal cord, where each image is paired with the segmentation result marking the precise spinal cord region [69].

C. Security Enhancement

We first evaluate how much DLBox can prevent the malicious AI developers from leaking the training data. To be specific, without DLBox (i.e., baseline), malicious developers can easily leak all the training data by runing arbitrary model training codes. DLBox limits such data leaks by enforcing `DGM-Rules`. Thus, we measure how much DLBox can reduce the data leaks compared to the baseline.

Evaluation Metric. In order to clearly demonstrate the amount of data leaks, we define a leakage bandwidth, which measures how many training data samples can be leaked from the output of an untrusted model training code.

$$\texttt{BW}_{\texttt{leakage}} = \text{the number of training data samples that can be leaked by one transfer of the trained model.}$$

The lower $\texttt{BW}_{\texttt{leakage}}$ indicates the applied security measure is more effective in protecting the data as the malicious developer brings fewer samples at once.

We determine a sample is leaked when a reconstructed image from the trained model satisfies following two criteria, which are widely used in academia [19], [70], [20]: 1) an evaluation model on the same task should infer the correct output (e.g., the label of the original image) from the reconstructed image. The evaluation model should be different and more advanced than the target model [19]; and 2) Peak Signal-to-Noise Ratio (PSNR) between the original image and the reconstructed image should be higher than a configurable parameter γ. PSNR means the pixel-wise similarity between the original image and the reconstructed image [19]. We used the PSNR threshold of 15 to ensure minimal similarity between two images.

Evaluation Scenarios. We compare $\texttt{BW}_{\texttt{leakage}}$ with and without applying the security measures. To this end, we design three adhoc security measures, i.e., $\texttt{Format}_{\texttt{Enforce}}$, $\texttt{Accuracy}_{\texttt{Enforce}}$, and $\texttt{Behavior}_{\texttt{Enforce}}$, where the following one is more advanced than the preceding one—i.e., following one can prevent the attack that the preceding one cannot prevent. DLBox (i.e., enforcing `DGM-Rules`) is the most advanced measure as it can prevent all the attacks the aforementioned measures cannot prevent. In the following, we enumerate the scenarios paired with the possible attacks, in the ascending order of the security.

1) **`Baseline (leak-as-itself)`** is the scenario in which no security measure is applied. Thus, the malicious AI developer can bring any training data of any size as it is. Other sort of attacks (e.g., `gradient-inversion-attacks` [20], [32], [31]) may also be possible, but it is not the interest of the attacker as he can already leak any data without any constraint.
2) **$\texttt{Format}_{\texttt{Enforce}}$ `(bit-encoding-attacks)`** enforces the developers to bring only the data of model format (e.g., files with `pth` extension in case of a `PyTorch` model [17]). Thus, the malicious developer cannot bring arbitrary data which does not have the format of model. However, he can still leak the training data by encoding it into the model and decoding it after bypassing the security measure.
3) **$\texttt{Accuracy}_{\texttt{Enforce}}$ `(memorizing-attacks)`** enforces the developers to obtain the model only when it shows a moderate performance. Thus, the simple `bit-encoding-attacks` cannot succeed unless the model, which encodes the data, actually performs well on the given task. However, the adversary can stealthily memorize the data into the model without degrading the performance as explained in §II-B.
4) **$\texttt{Behavior}_{\texttt{Enforce}}$ `(gradient-inversion-attacks)`** even scrutinizes what the developers do so that they cannot intentionally memorize the data while training the model—i.e., such behavior is totally not related to the model training [9]. However, the adversary can still launch a `gradient-inversion-attack` by utilizing the gradients obtained while training a model.
5) **DLBox `(model-inversion-attacks)`** prevents all the above attacks by enforcing `DGM-Rules` (i.e., explained in §VII). Nevertheless, the developers can still perform `model-inversion-attacks` to leak the data.

Given the attacks available for each security measure, we found the maximum possible $\texttt{BW}_{\texttt{leakage}}$ by adjusting the following settings: i) the model used to leak the training data (i.e., target model), and ii) hyperparameters of the attacks. For instance,

Table III: Specification of the trained target models and the evaluation model for image classification.

Measurement		Target model				Evaluation model
		SimpleFC	SimpleCNN	ResNet18	MobileNet-v2	EfficientNet-b3
Size (MB)		0.12 ∼ 1.5	4.6 ∼ 42	42.6	8.5	40.8
Test accuracy per dataset (%)	Cifar10	40.2	78.5	93.2	92.5	97.3
	UTKFace	80.1	89.3	88.3	87.3	90.1
	ChestXray	87.0	97.7	99.0	98.8	99.4

Table IV: $\texttt{BW}_{\texttt{leakage}}$ achieved in each scenario of image classification task. $\texttt{BW}_{\texttt{leakage}}$ of the attacks not possible in each scenario is set 0. All attacks were performed with the best set of parameters to maximize $\texttt{BW}_{\texttt{leakage}}$.

Security measure	Attack method	Training dataset		
		Cifar10	UTKFace	ChestXray
Baseline	**leak-as-itself**	**60000**	**20000**	**15000**
	bit-encoding	17060	5685	1853
	memorizing [9]	2470	113	53
	gradient-inversion [20]	3.32	2.12	1
	model-inversion [33]	˜0	˜0	˜0
$\texttt{Format}_{\texttt{Enforce}}$	leak-as-itself	0	0	0
	bit-encoding	**17060**	**5685**	**1853**
	memorizing [9]	2470	113	53
	gradient-inversion [20]	3.32	2.12	1
	model-inversion [33]	˜0	˜0	˜0
$\texttt{Accuracy}_{\texttt{Enforce}}$	leak-as-itself	0	0	0
	bit-encoding	0	0	0
	memorizing [9]	**2470**	**113**	**53**
	gradient-inversion [20]	3.32	2.12	1
	model-inversion [33]	˜0	˜0	˜0
$\texttt{Behavior}_{\texttt{Enforce}}$	leak-as-itself	0	0	0
	bit-encoding	0	0	0
	memorizing [9]	0	0	0
	gradient-inversion [20]	**3.32**	**2.12**	**1**
	model-inversion [33]	˜0	˜0	˜0
DLBox	leak-as-itself	0	0	0
	bit-encoding	0	0	0
	memorizing [9]	0	0	0
	gradient-inversion [20]	0	0	0
	model-inversion [33]	**˜0**	**˜0**	**˜0**

When launching the `gradient-inversion-attacks` [20], [32], we found the optimal number of samples that can be averaged to be leaked at once, while also showing a reasonable reconstruction accuracy.

Image Classification. For the image classification, we conducted attacks using four different target models as illustrated in Table III, ranging from a simple fully connected model (i.e., `SimpleFC`) to a fairly complex model (i.e., `ResNet18` [23], and `MobileNet-v2` [24]). We summarized the test accuracy of the correctly trained target models and the evaluation model.

$\texttt{BW}_{\texttt{leakage}}$ in each scenario is summarized in Table IV. In `Baseline`, the adversary can leak the entire data samples (e.g., 60000 samples for `Cifar10` dataset) with only one trial (i.e., `leak-as-itself`) as he can bring arbitrary size of data without any constraint. Under $\texttt{Format}_{\texttt{Enforce}}$, $\texttt{BW}_{\texttt{leakage}}$ depends on the maximum size of the model that can be transferred, which is `ResNet18` in our scenario. The adversary can leak almost 20% of the samples within one trial by encoding the data (i.e., `bit-encoding-attacks`). $\texttt{Accuracy}_{\texttt{Enforce}}$ still cannot protect against `memorizing-attacks`, leaking tens to thousands of samples at only one trial. Even under $\texttt{Behavior}_{\texttt{Enforce}}$, the adversary is able to leak a few samples by partially training the model and performing the `gradient-inversion-attacks`. `SimpleFC` achieves the maximum $\texttt{BW}_{\texttt{leakage}}$ in this attack thanks to its simple model architecture.

DLBox achieves the lowest $\texttt{BW}_{\texttt{leakage}}$ as it prevents all the aforementioned attacks except `model-inversion-attacks` [33]. However, the `model-inversion-attacks` [33] (without auxiliary dataset) were not able to reconstruct any sample satisfying our leakage criteria even after 100 trials. Given that current model training framework allows the developers to leak entire samples without any restriction (i.e., `Baseline`), we believe DLBox significantly improves the security of shared training data.

Table V: Security evaluation on image segmentation models. Accuracy on each sample is computed as the number of correctly marked pixels over the number of entire pixels.

Measurement	Target model FCN	Evaluation model U-Net
Size (MB)	85.3	18.5
Test accuracy (%)	97.5	97.7

(a) Model specifications

Attack method	$\texttt{BW}_{\texttt{leakage}}$
leak-as-itself	40
bit-encoding	40
memorizing [9]	40
gradient-inversion [20]	˜0
model-inversion [33]	**˜0**

(b) Achieved $\texttt{BW}_{\texttt{leakage}}$

Image Segmentation. For the image segmentation on `SpinalCordMRI` dataset [69], we used `FCN` [25] as a target model and `U-Net` [71] as an evaluation model. The size and the test accuracy of the models are summarized in Table V-(a). For the leakage criteria, we used 90% threshold accuracy and 15 for PSNR threshold.

As in the case of image classification, DLBox achieves the lowest $\texttt{BW}_{\texttt{leakage}}$ by enforcing `DGM-Rules` (i.e., shown in Table V-(b)). The first three attacks (i.e., `leak-as-itself`, `bit-encoding-attacks`, and `memorizing-attacks`) leak the entire samples within one trial due to the small size of dataset. `gradient-inversion-attacks`, on the other hand, were not able to reconstruct any sample as the image segmentation has a large gradient dimension. Nonetheless, we note that DLBox is more advanced security measure than just defending against these attacks as it provides well-defined security guarantees.

D. Performance Overhead

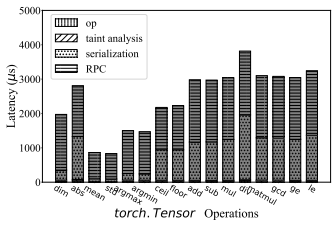
We evaluate the performance overhead of DLBox while training a model. In particular, we evaluate its overhead in two aspects: i) while interactively debugging the code and tuning the hyper-parameters, and ii) while actually training the model.

Overhead in Interactive Debugging and Hyper-parameters Tuning. Most of the model training procedures include debugging and hyper-parameters tuning, which run a small piece of code, visualize the samples, and monitor the metrics [29], [72]. DLBox supports such tasks as it seamlessly runs the `Python` scripts given the developers. However, it incurs runtime overheads due to the underlying operations of `DL-Isolate` and `DL-Oracle`. To be specific, for a single operation requested on a `Proxy` variable, it adds the latency for an RPC (which implicitly includes the effect of compartmentalization), serialization and deserialization of the data, and taint analysis.

Operation by DLBox	Latency (μs)
RPC	1478 (215)
Serialization	816 (138)
Taint analysis	515 (123)

(*) denotes a standard deviation

(a) Latency of single operation



(b) Latency breakdown of `torch.Tensor` operations

Figure 7: Measurements of latencies induced by DLBox.

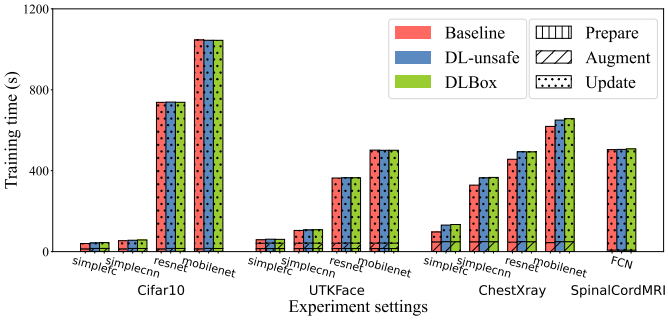


Figure 8: Comparison of learning time for each image processing task and model on `Baseline`, `DL-unsafe`, and DLBox. `Prepare` denotes the time used for loading the model and initializing the samples. `Augment` denotes the time for augmenting each data sample. `Update` denotes the time for computing gradients and updating the model.

To this end, we summarize the latencies induced by each factor as shown in Figure 7. DLBox incurs about 3ms of latency in total for an operation which includes RPC, serialization and taint analysis (i.e., Figure 7-(a)). The latencies and overheads on example `torch.Tensor` operations have shown the consistent results as illustrated in Figure 7-(b). However, we want to note that these overheads are imposed only while interactive debugging and tuning, and they can be avoided in actual model training as explained below.

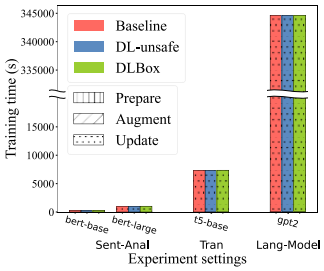
Overheads in Actual Training Process. Much of the time spent on training a model is dedicated to the actual training process, which involves computing gradients and updating the model while iterating over the samples. Thus, we trained the target models in §VIII-C on three different settings: i) `Baseline`, which conventionally runs a `Python` script using `PyTorch`, in a normal (unsafe) VM, ii) `DL-unsafe`, which uses DLBox to isolate the address space (by employing two different VMs) and taint trace the operations, but not using a confidential VM, and iii) DLBox, which employs a confidential VM (i.e., AMD SEV-SNP [21]) to protect the safe address space. We trained each model for 10 epochs, using `Adam` optimizer [73] with `CrossEntropyLoss` [74] for image classification and `BCELoss` [75] for image segmentation.

Learning times for each pair of dataset, model, and setting are shown in Figure 8. Overall, DLBox increases the learning time by 4% on average, showing maximum 38% overhead when training `SimpleFC` on `ChestXray`. The portion of overhead has decreased with larger models and datasets, as the training process is GPU-bound, where the GPU spends most of the time to compute gradients and update model weights. The processes involved in DLBox are carried out by the CPU, and their effect on the learning time is minimal.

Task	Model	Size (MB)	Dataset
S-A*	Bert-base	433	emotion [76]
	Bert-large	1334	
Tran†	T5-base	891	wmt16 [77]
L-M‡	Gpt2	497	wikipedia [78]

S-A*: sentiment-analysis, Tran†: translation
L-M‡: language-modeling

(a) Model specifications



(b) Learning time

Figure 9: Comparison of learning time for training language models on NLP tasks.

E. Performance Overhead on Language Models

In order to demonstrate the applicability of DLBox, we further evaluate its performance overhead on language models. To be specific, we used DLBox to train the models on following three NLP tasks: i) sentiment-analysis, which estimates the sentiment of a given sentence, ii) translation, which translates a sentence of source language to that of a target language, and iii) language modeling, which predicts the most appropriate word following a sentence. Depending on the tasks, we trained `Bert-base` [26], `Bert-large` [26], `T5-base` [27], and `Gpt2` [28] models, where the model sizes range from 400MB to 1.3GB as shown in Figure 9-(a). As a dataset for each task, we used **`emotion`** [76] (i.e., 3.4MB), **`wmt16`** [77] (i.e., 298MB), and **`wikipedia`** [78] (i.e., 30GB). We trained each model for one epoch due to the time limit, but we think the performance tendency would not deviate as the number of epochs increases [79].

We illustrate the performance overhead of DLBox on language models as shown in Figure 9-(b). DLBox increases the learning time by about 2% on average, showing almost 0% overhead when training `gpt2` model on `wikipedia` dataset. Similar to training image models on DLBox, the portion of performance overhead by DLBox decreases as the size of model and dataset increase, since the GPU spends most of the learning time computing gradients and updating models.

IX. Discussion

Generalizing `DGM-Rules` to Various Learning Techniques. While `DGM-Rules` are compatible with typical model training procedures as explained in §IV, it can be extended for other learning techniques. Nonetheless, the key observation of `DGM-Rules` still holds that the model training procedure is a statistical process of learning common patterns from a dataset. As an example, we discuss applying DLBox on following two learning techniques: i) resampling [80], ii) between-class learning [81], and

Resampling [80] oversamples or undersamples the samples of a specific class to balance the distribution of dataset. It does

not satisfy the model update rule $\mathbf{R_M}$ as some samples are used multiple times in a single epoch. However, we want to note that resampling also treats each sample equally in the same class, and $\mathbf{R_M}$ can be generalized to ensure the equal frequency of the samples used in the same class.

Between-class learning [81] mixes two samples in different classes into one to improve the performance of the model over noise. It does not satisfy the data augmentation rule $\mathbf{R_D}$ as the samples go through different augmentations depending on which samples they are mixed together. In order to cover between-class learning, $\mathbf{R_D}$ can be generalized to ensure equal distribution of samples used in the augmentation procedure.

Trade-offs between Security and Usability of DLBox. In order for the AI developers to train their models as usual, DLBox allows them to obtain a number of data flows (not satisfying `DGM-Rules`) configured by data owner. However, it also leads to data leaks as they may reconstruct the samples from the obtained data flows. DLBox needs to conservatively consider that the samples related to the obtained data flows are leaked. Then, the data owner could set the allowed number of samples that can be leaked.

However, we want to note that there is no need for the developers to manually observe all samples in the dataset (as it is the goal of deep learning—i.e., learn from the data on its own). Typical development procedure would include tens to hundreds trials of data visualization and metrics monitoring to debug and tune the models [82]. Thus, it would be enough to set the allowed number of samples to around tens to hundred. Furthermore, monitoring frameworks such as `TensorBoard` [72] could also be retrofitted to safely expose the computed metrics (e.g., accuracy) on the models.

Design Considerations for Implementing DLBox. DLBox can be implemented in various ways as long as it can safely enforce `DGM-Rules` while training a model. For instance, we can enforce `DGM-Rules` before training the model by statically analyzing the training code, or in runtime as we implemented in §VI (i.e., compartmentalization and RPC based approach).

We have chosen runtime based approach for i) usability, and ii) low false positives. First of all, our approach improves usability for the developers as they can interactively debug and tune the models as they did on `Python` REPL [7] or Jupyer Notebook [58]. Static approach needs to perform an analysis for every evaluation of the entire code given by developers [83]. Second, runtime based approach has lower false positives than the static approaches as it is previously studied [83]. Furthermore, it is more difficult for `Python`'s static analyzer to achieve low false positives due to the features of `Python` language (e.g., dynamic typing, metaprogramming [7], etc).

Potential Side-Channel Attacks on DLBox. Like other software frameworks that rely on confidential computing, DLBox can be vulnerable to side-channel attacks [47], [84], [85] as the computing resources are shared between different entities. In order to mitigate those attacks, DLBox can employ both micro-architectural [86], [87], and software-based [88], [89] approaches, which are widely studied to harden the computing infrastructure.

Effect of Auxiliary Dataset. While DLBox thwarts most of the attacks raised by untrusted developers, the adversaries with auxiliary dataset may be able to reconstruct the data partially through advanced model inversion attacks [19]. However, we want to note that such model inversion attacks cannot extract the entire distribution of the target dataset as the attack is also biased to the auxiliary dataset.

With prior knowledge on the dataset, attackers would be able to conduct unprecedented attacks to leak the data. However, we believe DLBox still provides the crucial protection bar for securing the training data, and it can be extended to employ an improved technique such as differential privacy [15] to protect against further attacks.

X. Related Work

In this section, we describe previous works that focus on preventing unauthorized data leaks in deep learning, and we discuss the differences between them and DLBox.

Federated Learning. Federated learning [13] was proposed to protect private data in data owner's end devices, while the gradients computed from the data are used to train an AI developer's model. In addition, recent works [14], [90] incorporated confidential computing to ensure the integrity of the developer's parameter server, thereby preventing gradient inversion attacks [20], [31], [32].

While federated learning has a similar goal with DLBox, it assumes much weaker adversary as he cannot control the original data but only receives the safely computed gradients. DLBox, on the other hand, assumes the malicious developers have full control over the data, and thus they can construct purely arbitrary data flows from the data to the model (e.g., encoding the data into the model's weights).

Furthermore, federated learning is limited in its scalability due to the challenges in synchronizing the model updates and large communication overhead [13], [91]. DLBox suffers fewer scalability issues as it can perform deep learning in the same machine, which accommodates both the training data and the model.

Differentially Private Machine Learning. The goal of differentially private machine learning [15], [92] is to ensure the statistical indistinguishability of the model with respect to an individual data. However, they also cannot be applied to DLBox as the malicious AI developers can perform arbitrary model training such as training the model with only a specific sample or maliciously computing the loss function [9]. Differential privacy needs further researches to be enforced on such general computations [93], [94], [95]. Additionally, differentially private machine learning suffers from an accuracy degradation [15], [92], while DLBox does not.

Machine Learning with Homomorphic Encryption. Homomorphic encryption [96], [97] does not work in DLBox's threat model as the model trained with data owner's data has to be eventually revealed to the AI developers. In other words, even if the model is trained with the homomorphically encrypted data

(by the data owner), the model should eventually be decrypted to be used by the developers.

Machine Learning with Confidential Computing. Researchers have used confidential computing for secure machine learning [16], [98], [99], but none of them have solved the problem of DLBox. Ohrimenko et al. [16] proposed oblivious multiparty machine learning, but they assume the model training code itself is benign. Chiron [98] works in MLaaS scenario, where the trained model belongs to the data owner. However, DLBox assumes an untrusted AI developers obtain the model, which is trained using an arbitrary model training code developed by themselves.

XI. Conclusion

In this paper, we present DLBox, which enables deep learning on shared training data while preventing data leakage through invalid paths. To this end, DLBox introduces `DGM-Rules` based on the key observation that a model training is a statistical process of learning common patterns from a dataset. Then, DLBox redesigns the model training framework on confidential computing to enforce `DGM-Rules`-based training. We implemented the prototype of DLBox on `PyTorch`, and the evaluation results clearly demonstrate that DLBox enhances the security with reasonable overheads.

Acknowledgment

We would like to thank anonymous reviewers for their insightful comments, which significantly improved the final version of this paper. This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. RS-2023-00209093). This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.RS-2024-00438729, Development of Full Lifecycle Privacy-Preserving Techniques using Anonymized Confidential Computing). This work was supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (RS-2023-00335515). The Institute of Engineering Research (IOER) and Automation and Systems Research Institute (ASRI) at Seoul National University provided research facilities for this work.

References

[1] Tim Hulsen. Sharing is caring—data sharing initiatives in healthcare. *International journal of environmental research and public health*, 17(9):3046, 2020.

[2] Shigao Huang, Jie Yang, Simon Fong, and Qi Zhao. Artificial intelligence in cancer diagnosis and prognosis: Opportunities and challenges. *Cancer letters*, 471:61–71, 2020.

[3] Laura C Bell and Efrat Shimron. Sharing data is essential for the future of ai in medical imaging. *Radiology: Artificial Intelligence*, 6(1):e230337, 2023.

[4] How ai helps to detect fraud in banking. https://www.gbgplc.com/en/blog/ai-a-key-player-in-financial-institutions-fight-against-fraud/.

[5] How ai will revolutionize financial fraud investigation. https://www.forbes.com/sites/forbestechcouncil/2023/07/03/the-digital-sentry-how-ai-will-revolutionize-financial-fraud-investigation/?sh=436f32f57e24.

[6] CNN. Meta agrees to pay $725 million to settle lawsuit over Cambridge Analytica data leak. https://edition.cnn.com/2022/12/23/tech/meta-cambridge-analytica-settlement/index.html, 2022.

[7] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[8] Air gap (networking). https://en.wikipedia.org/wiki/Air_gap_(networking).

[9] Congzheng Song, Thomas Ristenpart, and Vitaly Shmatikov. Machine learning models that remember too much. In *Proceedings of the 24th ACM Conference on Computer and Communications Security (CCS)*, Dallas, TX, October–November 2017.

[10] The bletchley park 1944 cryptographic dictionary formatted by tony sale (c) 2001. https://www.codesandciphers.org.uk/documents/cryptdict/cryptxtt.pdf.

[11] What is a data clean room and how does it work? https://clearcode.cc/blog/data-clean-room/.

[12] Share sensitive data with data clean rooms. https://cloud.google.com/bigquery/docs/data-clean-rooms?hl=en.

[13] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated learning: Challenges, methods, and future directions. *IEEE signal processing magazine*, 37(3):50–60, 2020.

[14] Fan Mo, Hamed Haddadi, Kleomenis Katevas, Eduard Marin, Diego Perino, and Nicolas Kourtellis. Ppfl: privacy-preserving federated learning with trusted execution environments. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 94–108, 2021.

[15] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. Deep learning with differential privacy. In *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, October 2016.

[16] Olga Ohrimenko, Felix Schuster, Cédric Fournet, Aastha Mehta, Sebastian Nowozin, Kapil Vaswani, and Manuel Costa. Oblivious {Multi-Party} machine learning on trusted processors. In *Proceedings of the 25th USENIX Security Symposium (Security)*, Austin, TX, August 2016.

[17] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

[18] Martín Abadi, Ashish Agarwal, Paul Barham, and et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[19] Yuheng Zhang, Ruoxi Jia, Hengzhi Pei, Wenxiao Wang, Bo Li, and Dawn Song. The secret revealer: Generative model-inversion attacks against deep neural networks. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 253–261, 2020.

[20] Jonas Geiping, Hartmut Bauermeister, Hannah Dröge, and Michael Moeller. Inverting gradients-how easy is it to break privacy in federated learning? *Advances in Neural Information Processing Systems*, 33:16937–16947, 2020.

[21] David Kaplan, Jeremy Powell, and Tom Woller. Amd memory encryption. *White paper*, 2016.

[22] Pytorch lightning. https://www.pytorchlightning.ai/.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[24] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[25] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3431–3440, 2015.

[26] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota, June 2019. Association for Computational Linguistics.

[27] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring

the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research*, 21(140):1–67, 2020.
[28] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. 2019.
[29] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.
[30] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *arXiv preprint arXiv:1712.04621*, 2017.
[31] Ligeng Zhu, Zhijian Liu, and Song Han. Deep leakage from gradients. *Advances in neural information processing systems*, 32, 2019.
[32] Hongxu Yin, Arun Mallya, Arash Vahdat, Jose M Alvarez, Jan Kautz, and Pavlo Molchanov. See through gradients: Image batch recovery via gradinversion. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 16337–16346, 2021.
[33] Matt Fredrikson, Somesh Jha, and Thomas Ristenpart. Model inversion attacks that exploit confidence information and basic countermeasures. In *Proceedings of the 22nd ACM SIGSAC conference on computer and communications security*, pages 1322–1333, 2015.
[34] Ziqi Yang, Jiyi Zhang, Ee-Chien Chang, and Zhenkai Liang. Neural network inversion in adversarial setting via background knowledge alignment. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 225–240, 2019.
[35] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. *Hasp@ isca*, 10(1), 2013.
[36] Intel trust domain extension (intel tdx). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html.
[37] Nvidia confidential computing. https://www.nvidia.com/en-us/data-center/solutions/confidential-computing/.
[38] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.
[39] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.
[40] Reddit's sale of user data for ai training draws ftc inquiry. https://www.wired.com/story/reddits-sale-user-data-ai-training-draws-ftc-investigation/.
[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
[42] Stable diffusion. https://github.com/CompVis/stable-diffusion.
[43] Mingxing Tan and Quoc Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning*, pages 6105–6114. PMLR, 2019.
[44] Cynthia Dwork, Aaron Roth, et al. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407, 2014.
[45] Katarzyna Janocha and Wojciech Marian Czarnecki. On loss functions for deep neural networks in classification. *arXiv preprint arXiv:1702.05659*, 2017.
[46] Suvrit Sra, Sebastian Nowozin, and Stephen J Wright. *Optimization for machine learning*. Mit Press, 2012.
[47] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *Proceedings of the 27th USENIX Security Symposium (Security)*, Baltimore, MD, August 2018.
[48] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *Proceedings of the 4th IEEE European Symposium on Security and Privacy (EuroS&P)*, Stockholm, Sweden, June 2019.
[49] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. 2018.
[50] Stephen Checkoway and Hovav Shacham. Iago attacks: Why the system call api is a bad untrusted rpc interface. *ACM SIGARCH Computer Architecture News*, 41(1):253–264, 2013.
[51] Li Deng. The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142, 2012.
[52] Pytorch dataset and dataloaders. https://pytorch.org/tutorials/beginner/basics/data_tutorial.html.
[53] Tensorflow dataset. https://www.tensorflow.org/datasets.
[54] Pytorch lightning module. https://lightning.ai/docs/pytorch/stable/common/lightning_module.html.
[55] Pytorch ignite engine. https://pytorch.org/ignite/engine.html.
[56] Tensorflow custom trainer. https://www.tensorflow.org/tutorials/customization/custom_training_walkthrough.
[57] Companies using pytorch lightning. https://theirstack.com/en/technology/pytorch-lightning.
[58] Thomas Kluyver, Benjamin Ragan-Kelley, Fernando Pérez, Brian Granger, Matthias Bussonnier, Jonathan Frederic, Kyle Kelley, Jessica Hamrick, Jason Grout, Sylvain Corlay, Paul Ivanov, Damián Avila, Safia Abdalla, and Carol Willing. Jupyter notebooks – a publishing format for reproducible computational workflows. In F. Loizides and B. Schmidt, editors, *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, pages 87 – 90. IOS Press, 2016.
[59] Python c extension. https://docs.python.org/3/extending/extending.html.
[60] grpc - an rpc library and framework. https://github.com/grpc/grpc.
[61] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. 2009.
[62] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987.
[63] Google remote procedure call. https://github.com/grpc/grpc.
[64] Pci passthrough via ovmf. https://wiki.archlinux.org/title/PCI_passthrough_via_OVMF.
[65] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on {GPUs}. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 681–696, 2018.
[66] Zhifei Zhang, Yang Song, and Hairong Qi. Age progression/regression by conditional adversarial autoencoder. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2017.
[67] Muhammad E. H. Chowdhury, Tawsifur Rahman, Amith Khandakar, Rashid Mazhar, Muhammad Abdul Kadir, Zaid Bin Mahbub, Khandakar Reajul Islam, Muhammad Salman Khan, Atif Iqbal, Nasser Al Emadi, Mamun Bin Ibne Reaz, and Mohammad Tariqul Islam. Can ai help in screening viral and covid-19 pneumonia? *IEEE Access*, 8:132665–132676, 2020.
[68] Tawsifur Rahman, Amith Khandakar, Yazan Qiblawey, Anas Tahir, Serkan Kiranyaz, Saad Bin Abul Kashem, Mohammad Tariqul Islam, Somaya Al Maadeed, Susu M. Zughaier, Muhammad Salman Khan, and Muhammad E.H. Chowdhury. Exploring the effect of image enhancement techniques on covid-19 detection using chest x-ray images. *Computers in Biology and Medicine*, 132:104319, 2021.
[69] Ferran Prados, John Ashburner, Claudia Blaiotta, Tom Brosch, Julio Carballido-Gamio, Manuel Jorge Cardoso, Benjamin N. Conrad, Esha Datta, Gergely Dávid, Benjamin De Leener, Sara M. Dupont, Patrick Freund, Claudia A.M. Gandini Wheeler-Kingshott, Francesco Grussu, Roland Henry, Bennett A. Landman, Emil Ljungberg, Bailey Lyttle, Sebastien Ourselin, Nico Papinutto, Salvatore Saporito, Regina Schlaeger, Seth A. Smith, Paul Summers, Roger Tam, Marios C. Yiannakas, Alyssa Zhu, and Julien Cohen-Adad. Spinal cord grey matter segmentation challenge. *NeuroImage*, 152:312–329, 2017.
[70] Yugeng Liu, Rui Wen, Xinlei He, Ahmed Salem, Zhikun Zhang, Michael Backes, Emiliano De Cristofaro, Mario Fritz, and Yang Zhang. {ML-Doctor}: Holistic risk assessment of inference attacks against machine learning models. In *Proceedings of the 31th USENIX Security Symposium (Security)*, Boston, MA, August 2022.
[71] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, pages 234–241. Springer, 2015.
[72] Tensorboard.
[73] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[74] Tilmann Gneiting and Adrian E Raftery. Strictly proper scoring rules, prediction, and estimation. *Journal of the American Statistical Association*, 102(477):359–378, 2007.

[75] Shruti Jadon. A survey of loss functions for semantic segmentation. In *2020 IEEE Conference on Computational Intelligence in Bioinformatics and Computational Biology (CIBCB)*, pages 1–7. IEEE, 2020.

[76] Elvis Saravia, Hsien-Chi Toby Liu, Yen-Hao Huang, Junlin Wu, and Yi-Shin Chen. CARER: Contextualized affect representations for emotion recognition. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pages 3687–3697, Brussels, Belgium, October-November 2018. Association for Computational Linguistics.

[77] Ond rej Bojar, Rajen Chatterjee, Christian Federmann, Yvette Graham, Barry Haddow, Matthias Huck, Antonio Jimeno Yepes, Philipp Koehn, Varvara Logacheva, Christof Monz, Matteo Negri, Aurelie Neveol, Mariana Neves, Martin Popel, Matt Post, Raphael Rubino, Carolina Scarton, Lucia Specia, Marco Turchi, Karin Verspoor, and Marcos Zampieri. Findings of the 2016 conference on machine translation. In *Proceedings of the First Conference on Machine Translation*, pages 131–198, Berlin, Germany, August 2016. Association for Computational Linguistics.

[78] Wikimedia Foundation. Wikimedia downloads.

[79] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Stephen McGough. Predicting the computational cost of deep learning models, 2018.

[80] Andrew Estabrooks, Taeho Jo, and Nathalie Japkowicz. A multiple resampling method for learning from imbalanced data sets. *Computational intelligence*, 20(1):18–36, 2004.

[81] Sangdoo Yun, Dongyoon Han, Seong Joon Oh, Sanghyuk Chun, Junsuk Choe, and Youngjoon Yoo. Cutmix: Regularization strategy to train strong classifiers with localizable features. In *Proceedings of the IEEE/CVF international conference on computer vision*, pages 6023–6032, 2019.

[82] scikit-optimize. https://scikit-optimize.github.io/stable/.

[83] Mark Dowd, John McDonald, and Justin Schuh. *The art of software security assessment: Identifying and preventing software vulnerabilities*. Pearson Education, 2006.

[84] Jo Van Bulck, Frank Piessens, and Raoul Strackx. Sgx-step: A practical attack framework for precise enclave execution control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, pages 1–6, 2017.

[85] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. {CIPHERLEAKS}: Breaking constant-time cryptography on {AMD}{SEV} via the ciphertext side channel. In *Proceedings of the 31th USENIX Security Symposium (Security)*, Virtual, August 2021.

[86] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *Proceedings of the 51th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Fukuoka, Japan, December 2018.

[87] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. {AEX-Notify}: Thwarting precise {Single-Stepping} attacks through interrupt awareness for intel {SGX} enclaves. In *Proceedings of the 32th USENIX Security Symposium (Security)*, Anaheim, CA, August 2023.

[88] Marco Guarnieri, Boris Köpf, Jan Reineke, and Pepe Vila. Hardware-software contracts for secure speculation. In *Proceedings of the 42st IEEE Symposium on Security and Privacy (Oakland)*, Virtual, May 2021.

[89] Adil Ahmad, Kyungtae Kim, Muhammad Ihsanulhaq Sarfaraz, and Byoungyoung Lee. Obliviate: A data oblivious filesystem for intel sgx. In *Proceedings of the 2018 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, February 2018.

[90] Chengliang Zhang, Junzhe Xia, Baichen Yang, Huancheng Puyang, Wei Wang, Ruichuan Chen, Istemi Ekin Akkus, Paarijaat Aditya, and Feng Yan. Citadel: Protecting data privacy and model confidentiality for collaborative learning. In *Proceedings of the 12th ACM Symposium on Cloud Computing (SoCC)*, Seattle, WA & Online, November 2021.

[91] Jakub Konečnỳ, H Brendan McMahan, Felix X Yu, Peter Richtárik, Ananda Theertha Suresh, and Dave Bacon. Federated learning: Strategies for improving communication efficiency. *arXiv preprint arXiv:1610.05492*, 2016.

[92] Ashkan Yousefpour, Igor Shilov, Alexandre Sablayrolles, Davide Testuggine, Karthik Prasad, Mani Malek, John Nguyen, Sayan Ghosh, Akash Bharadwaj, Jessica Zhao, et al. Opacus: User-friendly differential privacy library in pytorch. *arXiv preprint arXiv:2109.12298*, 2021.

[93] Phillip Nguyen, Alex Silence, David Darais, and Joseph P Near. Duetsgx: Differential privacy with secure hardware. *arXiv preprint arXiv:2010.10664*, 2020.

[94] Joseph P Near, David Darais, Chike Abuah, Tim Stevens, Pranav Gaddamadugu, Lun Wang, Neel Somani, Mu Zhang, Nikhil Sharma, Alex Shan, et al. Duet: an expressive higher-order language and linear type system for statically enforcing differential privacy. In *Proceedings of the 30th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, Athens, Greece, October 2019.

[95] Noah Johnson, Joseph P Near, and Dawn Song. Towards practical differential privacy for sql queries. In *Proceedings of the 44th International Conference on Very Large Data Bases (VLDB)*, Rio De Janeiro, Brazile, August 2018.

[96] Joon-Woo Lee, HyungChul Kang, Yongwoo Lee, Woosuk Choi, Jieun Eom, Maxim Deryabin, Eunsang Lee, Junghyun Lee, Donghoon Yoo, Young-Sik Kim, et al. Privacy-preserving machine learning with fully homomorphic encryption for deep neural network. *IEEE Access*, 10:30039–30054, 2022.

[97] Miran Kim, Yongsoo Song, Shuang Wang, Yuhou Xia, Xiaoqian Jiang, et al. Secure logistic regression based on homomorphic encryption: Design and evaluation. *JMIR medical informatics*, 6(2):e8805, 2018.

[98] Tyler Hunt, Congzheng Song, Reza Shokri, Vitaly Shmatikov, and Emmett Witchel. Chiron: Privacy-preserving machine learning as a service. *arXiv preprint arXiv:1803.05961*, 2018.

[99] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, Simon Peter, and Emmett Witchel. Ryoan: A distributed sandbox for untrusted computation on secret data. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, Savannah, GA, November 2016.