Decepticon: Attacking Secrets of Transformers

Mujahid Al Rafi[1] Yuan Feng[1] Fan Yao[2] Meng Tang[1] Hyeran Jeon[1]

[1]*University of California, Merced*
[2]*University of Central Florida*

Abstract

With the growing burden of training deep learning models with huge datasets, transfer learning has been widely adopted (e.g., Transformers like BERT, GPT). Transfer learning significantly reduces the time and effort of model training. However, the security impact of using shared pre-trained models has not been evaluated. In this paper, we provide in-depth characterizations of the fine-tuning process and reveal the security vulnerabilities of transfer-learned models. Then, we show a novel two-level model extraction attack; 1) identifying the pre-trained model of a victim transfer-learned model through model fingerprint collected from off-the-shelf GPUs and 2) extracting the entire weights of the victim black-box model based on the hints in the pre-trained model. The extracted model shows almost alike prediction accuracy with over 94% matching prediction outputs with the victim model. The two-level model extraction enables large model weight extraction that is considered as challenging if not impossible through significantly reduced extraction effort.

1. Introduction

Machine learning (ML) models are widely used for almost all computing solutions, including smart homes, autopilot programs of self-driving vehicles, and so on. In many applications that handle critical data for the users' safety or privacy, any perturbed predictions lead to tragic consequences. Such an ML prediction perturbation is one of the major targets of cyber attacks, which is enabled by stealing important parameters of the target ML model. Recently, a few studies demonstrated that it is feasible to steal ML model topology and hyperparameters through various side channels (e.g., performance counters, cache access timing, etc.) [23, 37, 51]. Most existing attacks targeted convolutional neural network (CNN) models. While the model topology and hyperparameters provide important hints about the victim model, the weight values shift the security surface to another level by enabling more advanced model extractions, including constructing local (clone) models with extremely high fidelity (i.e., compromising model privacy) and empowering adversarial inputs attacks (i.e., tampering model integrity).

Stealing model weights is extremely challenging for two main reasons. First, modern ML workloads are very different from traditional security-sensitive applications (e.g., crypto programs) in that there is no explicit secret dependent control/data flows, making it almost impractical to accurately reveal model weights using a conventional side channel (e.g., via caches). Second, even if stealing can be done in real system, state-of-the-art ML models are very large with tremendous amount of weights. Any such attack will require model stealing at an extremely large scale. Such challenge is further exacerbated in today's gigantic transformer models that come with billions/trillions of weights (e.g., BERT [21], GPT [19, 38], Llama 2 [44]). As a result, pure side channel-based model stealing in such use cases is unlikely to be practical. For instance, very recent work in [40] repurposed row-hammer as a side channel to exfiltrate bit-level information for model weights. However, to reveal part of the weight, thousands of rounds of row-hammer are needed. Such a method may not scale to today's large-scale models. Then, can we conclude that large-scale (billion- to trillion-parameter) models are secure enough?

In this paper, we present the first investigation of model extraction attacks on large-scale models. We first show an in-depth characterizations of large models (240 models downloaded from popular model zoos [3, 9, 12, 13, 15]) and propose a novel two-level model extraction attack, namely *Decepticon*. Unlike existing approaches that run rigorous memory probings directly on every single bit of individual weights, we show that an indirect (two-level) approach makes large-scale model extraction practical. The two-level approach especially leverages the unique characteristics of transfer learning that most large models use.

Transfer learning enables large-scale model development with significantly reduced training time. With transfer learning, individual developers and research teams can develop their own models by fine-tuning a pre-trained model with task-specific datasets. The pre-trained models are either publicly shared through popular model zoos [3, 9, 10] or privately shared within individual companies or institutions. In any scale, pre-trained models are purposely shared by multi to many ML developers. For example, some BERT-base and GPT-2 pre-trained models on Huggingface model repositories have been downloaded more than 10 million and 20 million times, respectively [4, 7]. Security and privacy-sensitive domains such as medical and defense applications also share their pre-trained models with limited accesses [2, 36, 41, 49]. We focus on this unique training process of large-scale models.

What if an adversary grabs an access to a pre-trained model? Can he/she derive the secrets of a black-box fine-tuned model with the pre-trained model? According to our experiments, pre-trained models

. This work was supported by NSF CCF-2114514 and NSF SaTC-2019536.

2835-2238/23/$31.00 ©2023 IEEE
DOI 10.1109/IISWC59245.2023.00028

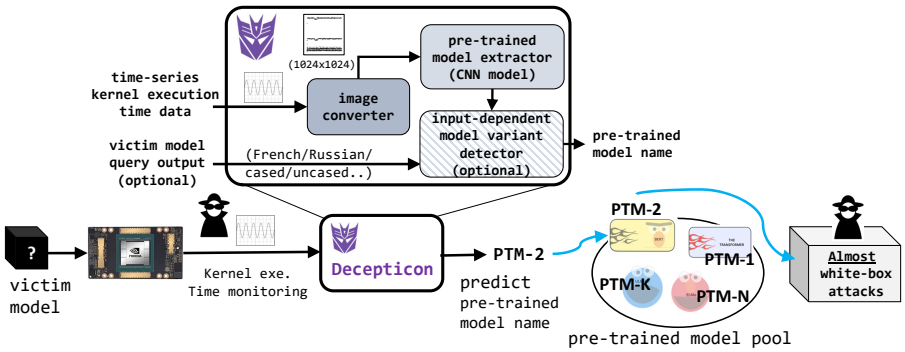


Figure 1: Decepticon Architecture

provide ample secrets of the fine-tuned model including very similar weight values. But, **can we locate the pre-trained model?** Our exhaustive investigation shows that individual ML models have unique execution fingerprints. Unlike existing studies assumed [23], the fingerprints are substantially different across models even when the models use the same architecture, dataset, and task. More importantly, the fingerprint is inherited from a pre-trained model to its fine-tuned models. Thus, the fingerprint can be leveraged to identify the pre-trained model.

With these observations, *Decepticon* first identifies the pre-trained model with the fingerprint of the blackbox model's execution on off-the-shelf GPUs. Then, it extracts the entire model-worth weights based on the pre-trained model's weight values. Decepticon uses two design methods to increase the attack success rate with minimal extraction effort; 1) selective weight extraction that exploits the diverse impact of individual weights and layers towards the final prediction output and 2) model fingerprint classification with CNN that is inherently noise tolerant. The clone model that is created by the extracted model secrets is used for an adversarial attack to perturb the victim model's prediction. We also show that Decepticon is applicable for any ML models that use transfer learning by using a CNN model example (Section 7.7). Figure 1 shows the end-to-end attack scenario of Decepticon.

Our contributions are like below:

1) To our best knowledge, this is the first study that demonstrates model extraction attack (including entire weight extraction) for large-scale models (e.g., Transformers). The proposed attack is applicable for any ML models that use transfer learning.
2) We provide in-depth characterization of transfer-learned models.
3) We apply a noise-tolerant image classification algorithm for model architecture extraction and introduce a selective weight extraction.
4) We demonstrate the impact of weight extraction through a clone model construction and an adversarial attack evaluation.

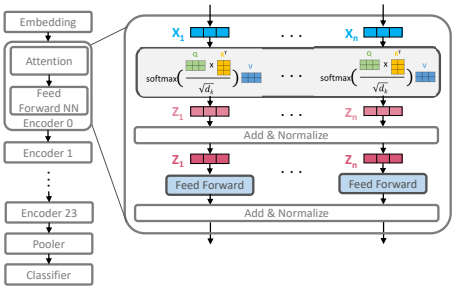


Figure 2: BERT-large Architecture

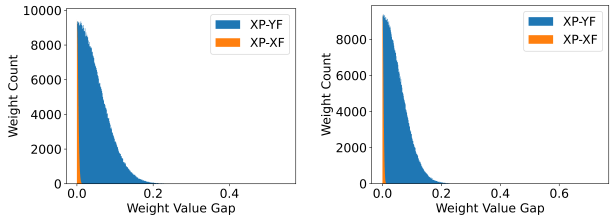
2. Background

2.1. Model Extraction Attacks

Model extraction attack is a security concern where an attacker aims to recover the secrets of ML models and reconstruct a copy of a victim model. The architecture and parameters of a victim model are the main targets of these attacks. The architecture of a model includes the number, type, and dimension of individual layers of the target ML model. The parameters include hyper-parameters, weights, and biases. Model extraction attack is a threat to the intellectual property of the ML solution providers. Moreover, the detailed knowledge about the victim model helps the attacker craft adversarial examples to fool the victim model or even extract sensitive training data. Several studies demonstrated various model extraction attacks [23, 33, 37, 40, 42, 45, 51, 52, 54, 56]. These studies leveraged various leakage vectors through EM side-channels, PCI-e bus snooping and message trapping, cache timings, and memory probing.

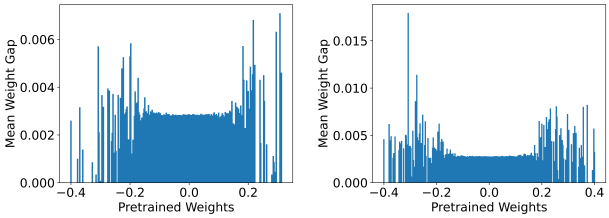
2.2. Transformer Models

Though any models that use transfer learning can be a target of Decepticon, we explain our attack scenario mainly with Transformer models. Transformer models use transfer learning as the default training method. Transformers have proven their effectiveness in various domains including natural language processing and computer vision [46]. Transformers are efficient to be adapted to several diverse tasks such as question answering, sentiment analysis, etc.



(a) Attention Self Key (b) Attention Self Query

Figure 3: Weight Value Gap Distribution



(a) Attention Self Key (b) Attention Self Query

Figure 4: Impact of Weight Value to Amount of Updates

with a task-specific last layer. A Transformer model runs multiple rounds of attention computations that check relations across all tokens of the given input in parallel. Transformer models consist of either *encoder* or *decoder* (or both). Each *encoder* is comprised of two layers: self-attention and feed-forward, as illustrated in Fig. 2. In a self-attention layer, input tokens are multiplied with three weight matrixes; Key, Query, and Value and generate K, Q, and V vectors. These vectors for each token are used for understanding the importance of the token in the input context. Decoders are similar to encoders, except the masked self-attention. Popular Transformer models are BERT [21], RoBERTa [31], ALBERT [28], and GPT-2 [38]. These reference models use fixed number of encoders/decoders such as 12 (BERT-base) and 24 (BERT-large).

3. Threat Model

The goal of our threat model is to steal the secrets of a black-box large model. The secrets are used for creating a clone model, which can be used for an adversarial attack. The victim model is assumed to be developed through transfer learning. The attacker is assumed to have no information about the victim model, while he/she can 1) collect architectural hints such as GPU kernel execution time and memory addresses and 2) query the victim model and check the prediction outputs. Note that various studies demonstrated that kernel execution time and memory addresses can be monitored through EM-side channels and bus probing on the interconnects between CPU and GPU [20, 23, 37, 51]. We leverage the existing side-channels but exploit the unique/novel characteristics of transfer-learned models for a novel model extraction attack. We assume that the attacker has a pool of candidate pre-trained models. The models can be either collected from public repositories or from internal spy routes if the victim model is designed with private pre-trained models. But, the attacker does not know which one was used for fine-tuning the victim model.

4. Transformer Model Vulnerabilities

We tested 70 pre-trained and 170 fine-tuned models downloaded from various model repositories on an NVIDIA GeForce RTX 3050 (Ampere) GPU with CUDA v11.8. We used Python v3.8, PyTorch v1.11.0, and TensorFlow v2.8.0 for evaluations. More details can be found from Section 7.

4.1. Pre-trained vs. Fine-tuned Models

Amount of weight updates during fine-tuning: To understand the relation between a pre-trained model and its fine-tuned model, we checked the significance of weight updates. We measured the absolute weight value gap for each pair of a pre-trained model and a fine-tuned model that use the same model architecture; weights on the same location of the two models were compared. Fig. 3 shows example results where the graphs with (XP-XF) are the comparisons between a pre-trained model and its fine-tuned model and those with (XP-YF) are between a pre-trained model and a fine-tuned model that used different pre-trained model. Note that there was not a pair of pre-trained and fine-tuned models designed for the same task.

In all cases, the distribution follows long tail. For (XP-XF) cases, the majority of weights encounter less than ± 0.01 value distance (thus looks like a sharp vertical line in the graph). Almost 50% weights show less than ± 0.002 distance. Any pair of pre-trained and its downstream models in our tested models showed similarly small weight gap. The weight value ranges of the tested models were at least 1.74 up to over 26.3.

On the other hand, (XP-YF) cases show at least $20\times$ higher gap. While most weights encounter less than ± 0.2 distance, the difference stretches up to over ± 0.6. Only less than 3% weights are within ± 0.002 distance.

Impact of weight values on updates: To understand the impact of weight values for the update amount, we measured the average value gap per pre-trained model weight value range. Fig. 4 shows example results of the models of (XP-XF) cases in Fig. 3. X-axis is the pre-trained weight value and Y-axis is the update amount during fine-tuning. Most of the layers show U-shape graph, which means that the weights further from zero encounter over $3\times$ higher weight changes than those closer to zero. Relating to the weight value gap result, the outliers such as outermost 10% weights (outside the ± 0.25 boundary in Fig. 4) are the sources of the long tail region in Fig. 3.

Impact of downstream tasks on updates: As a pre-trained model can be fine-tuned for different tasks, we also evaluated if we can still observe weight similarities if different task models share the same pre-trained model. We downloaded a BERT-base pre-trained model, fine-tuned it for nine tasks by using Huggingface GLUE benchmark [8],

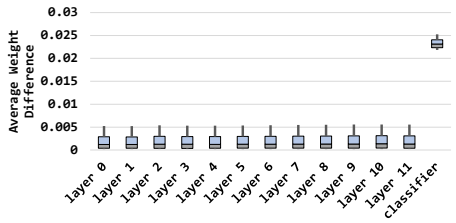


Figure 5: Avg. Weight Differences of 9 BERT-base Models Trained for Different Tasks

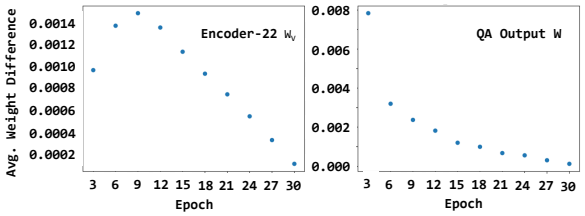


Figure 6: Avg. Weight Updates during Fine-tuning

and compared the weight values among the nine models. As shown in Fig. 5, all layers except for the task-dependent last layer show almost zero distances ($<$ 0.002 on average). This shows that the main updates are made in the last layer during the fine-tuning and the remaining layers might be very similar across fine-tuned models.

Impact of learning rates and epochs in fine-tuning: Fine-tuning typically uses small learning rates (and weight decay) and a limited number of epochs (up to three epochs typically), which may lead to little change in weight values during fine-tuning. To understand the impact of the short epochs, we tested the weight changes along longer fine-tuning epochs. Fig. 6 shows the average weight value changes of a layer (encoder 22) and the task-specific last layer of a BERT-large pre-trained model during 30 epochs of fine-tuning. Until epoch 9, the average weight value gap between epochs increases up to 0.0015 and then drops linearly to below 0.0002 at epoch 30. In the meantime, the output layer's weight value gap saturates exponentially as plotted in the second graph, which means that the fine-tuning converges well. This result shows that the weight gap may be increased with longer epochs but it is not significant (quickly saturated).

The theoretical reason of using small learning rates and epochs in fine-tuning is to prevent *catastrophic forgetting* [26, 35], which is a known phenomenon for neural networks that forget what it has previously learned given a new task or new data distribution. Catastrophic forgetting is one of the risks in the context of foundation models [18], particularly when the models are adapted to a new task with small data (e.g., few-shot learning). Therefore, the weight value similarity is not sourced from premature fine-tuning but is an unavoidable side-effect of transfer learning.

Contribution of layers towards model prediction: Inspired by the small weight value change during fine-tuning, we examined if pre-trained model's weights can be reused without hurting fine-tuning prediction output. For a BERT-base model [5] fine-tuned for question answering task, we tested downstream task accuracy by replacing the first a few layers with the pre-trained model's weights. Table 1 shows the results when replacing up to sixth layers. When freezing the first 2-3 layers, the fine-tuned model experiences only 1-3% accuracy drop.

TABLE 1: Downstream Task Accuracy with Frozen Layers

Number of Frozen Layers	Accuracy (%)
0	80.1325
1	80.2271
2	79.1485
3	76.5279
4	71.9111
5	68.543
6	64.579

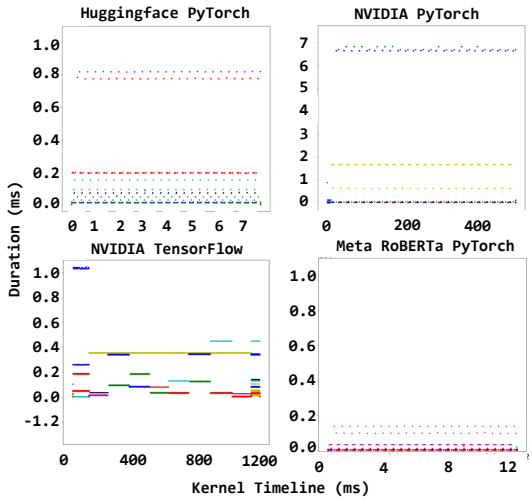


Figure 7: Diversity in Time-series Kernel Execution Times

Summary: Here are our observations.

Observation 1. Fine-tuned models have meaningfully close weight values to their baseline pre-trained models, which is at least 20× closer than with other pre-trained models.

Observation 2. Few outlier weights change by over 3× more and are the sources of the long tail weight gap distribution.

Observation 3. Fine-tuned models show very similar weights for all layers except for the last layer if they use the same pre-trained model even when fine-tuned for different tasks.

Observation 4. The impact of layers towards the downstream task's accuracy is different. First a few layers may reuse pre-trained model's weights without hurting accuracy.

Decepticon leverages these for extracting secrets of a blackbox victim model from its pre-trained model.

4.2. Model Execution Fingerprints

To extract the weights of a victim model by leveraging the weight value similarity, it is required to locate the pre-trained model from the victim black-box model. To identify the pre-trained model, there should be distinguishable model signatures. We tested if models have unique signatures.

Model signature in architecture hints: We compared kernel execution times of the models having the same

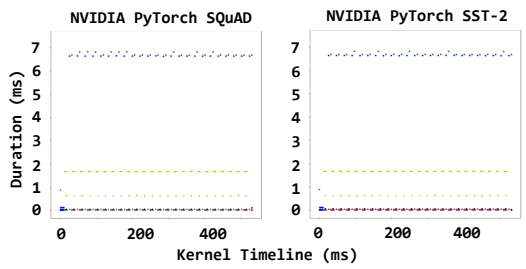


Figure 8: Consistency in Time-series Kernel Execution Times

Huggingface PyTorch Squad:
Total **687** executions of **11** kernels

```
splitKreduce_kernel
LayerNormForwardCUDAKernel
RowwiseMomentsCUDAKernel
indexSelectLargeIndex
vectorized_elementwise_kernel
volta_sgemm_128x128_tn
volta_sgemm_128x64_nn
volta_sgemm_32x128_tn
...
```

Meta (RoBERTa) PyTorch MNLI:
Total **697** executions of **17** kernels

```
splitKreduce_kernel
CatArrayBatchedCopy
volta_sgemm_128x32_sliced1x4_tn
DeviceScanKernel
dot_kernel
gemv2T_kernel_val
indexSelectLargeIndex
reduce_1Block_kernel
vectorized_layer_norm_kernel
...
```

Nvidia TensorFlow Squad:
Total **4754** executions of **403** kernels

```
splitKreduce_kernel
AddV2_GPU_DT_FLOAT_DT_FLOAT_k...
Mul_GPU_DT_FLOAT_DT_FLOAT_ker...
ampere_fp16_s16816gemm_fp16_1...
ampere_sgemm_128x128_nn
...
convert_411
convert_413
fusion
fusion_10
```

Nvidia PyTorch Squad:
Total **987** executions of **11** kernels

```
splitKreduce_kernel
cuApplyLayerNorm
elementwise_kernel_with_index
gemv2T_kernel_val
softmax_warp_forward
unrolled_elementwise_kernel
volta_fp16_s884gemm_fp16_64x1...
...
```

Figure 9: Kernels Executed by BERT-large Models

architecture. Fig. 7 shows the example results of different versions of BERT-large models measured on the same GPU with the same inputs. Each dot indicates the execution time of a kernel and the same-colored dots are the multiple invocations of the same kernel. Each BERT-large model is fine-tuned and released by different repository/developer as stated. Note that RoBERTa uses the same architecture with BERT. Interestingly, we couldn't find common patterns among them. On the other hand, when we tested the models released by the same repository/developer, the statistics showed high consistency even when they were fine-tuned for different tasks, as shown in Fig. 8.

Impact of algorithms and software interfaces: We observed that such model fingerprint is highly influenced by the algorithms and software interfaces (e.g., framework), which lead to different GPU kernel selections. Fig. 9 shows an example list of kernels executed by BERT-large models of different sources. Though the same model architecture is executed, only handful of kernels out of hundreds are commonly used across the models. We found that the framework is one of the main contributors that make the differences. TensorFlow models run up to 8× more kernel executions and use almost 40× more unique kernels than PyTorch models. Also, TensorFlow models tend to use their GPU backends, while PyTorch models use more GPU library functions. Developer-specific kernel preferences were also observed. Though different frameworks were used, NVIDIA models were commonly optimized to leverage their Tensor Core by running functions using half-precision data types. On the other hand, Meta models tend to run many short kernels such as reduction operations and hence the statistics had crowded kernel executions on the bottom of the graph as shown in Fig. 7. Similarly, we observed unique signatures from most of the 70 pre-trained models used in our evaluation (Section 7). Fine-tuned models showed very similar signatures with their pre-trained models.

TABLE 2: Impact of Model Fingerprint for CNN Layer Detection Accuracy

Models	Error Rate (LER)	Kernel Seq. Length	# of Unique Kernels
DeepSniffer Original Results [23]	0.091	222	16
DeepSniffer Pytorch Model [1]	0.567	256	16
Nvidia PyTorch Model [14]	2.628	1235	38
Google Tensorflow Model [16]	6.274	3399	50
Amazon Mxnet Model [11]	6.768	2652	59

Existing model extraction attacks do not work due to fingerprint: We also observed the model fingerprints from non-Transformer models but existing model extraction attack studies ignored the impact. When we fed a state-of-the-art CNN extraction framework [23] with statistics of CNN models that were developed by different sources, the layer prediction accuracy dropped significantly, as shown in Table 2. LER means how many layer sequences are incorrectly predicted per layer [23]. Thus, the prediction results with LER over 1 are not countable. We instead leverage the fingerprint for locating pre-trained model.

Model signature in query outputs: Though most of the models are recognizable with the unique model fingerprints, we found that there are cases that can't be distinguished through architectural hints. For example, in large language models (e.g., Transformers), 1) models that are trained with different languages (CamemBERT [34] and RuBERT [55] are French and Russian versions of BERT), 2) models that are trained with different training datasets (RoBERTa is trained with more datasets than BERT), and 3) models that are trained differently (e.g., cased vs. uncased) may not be distinguished if they are released by the same source and use the same architecture. For these cases, we found that query outputs can be used as a secondary fingerprint. For example, the dataset differences in RoBERTa and BERT can be checked with their vocabulary file (vocab.txt or vocab.json). When BERT was tested with queries including {`debugging, capitalize, cloves, indignation, hijab, selfies, misogynist, acupuncture`}, the predictions were incorrect while RoBERTa performed well.

Summary: Here are our observations.
Observation 1. Models have unique execution fingerprint even when using the same model architecture. The fingerprint is inherited from a pre-trained model to its fine-tuned models.
Observation 2. Query outputs can be used as a secondary model fingerprint if architecture hint is not distinguishable.
Decepticon leverages these for finding the pre-trained model of the victim model.

5. Decepticon Design

We propose *Decepticon*, a new model extraction attack that exploits weight similarity and model fingerprints.

5.1. Decepticon Architecture

Fig. 1 shows the architecture of Decepticon. Decepticon uses one *architectural hint* (time-series kernel execution time) and *model query outputs* as leakage vectors. The *pre-trained model extractor* receives the inputs in 2-dimensional image formats and finds the most matching pre-trained model out of the candidate models with an image recognition algorithm. For some models that show very similar model fingerprint, query outputs are used to improve the prediction accuracy in the *input-dependent model variant detector*. The output of Decepticon is the *pre-trained model name*. Once the pre-trained model is revealed, the attacker can try a variety of gray (or almost white) box attacks such as weight extraction and adversarial attacks.

5.2. Architectural Hints

Out of various side-channels (Section 2), we observed that time-series GPU kernel execution time reflect model fingerprints well. A model runs hundreds to thousands of kernels. Thus, individual kernel execution time does not reflect the model identity. Instead, we use a time-series kernel execution time, which shows the invocation timing and duration of all kernels during the model inference time. Thus, the attacker collects ($T_{invocation}$, $T_{termination}$) of all kernels, where T is timestamp.

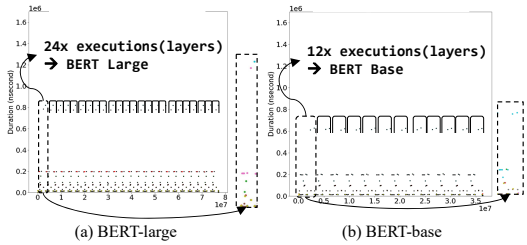
5.3. Query Outputs

As noted in Section 4.2, if there are multiple models having similar architecture hints, the attacker uses a special set of query inputs as the secondary fingerprint. By targeting large language models (e.g., BERT-uncased/cased, CamemBERT, RuBERT, RoBERTa, etc.), the set is compiled with several queries in different languages, special vocabularies that each candidate is uniquely trained with (by using the vocabulary file released with the pre-trained model or through exhaustive testing), and special words that have different meanings in upper/lower characters (e.g., Apple as a company name vs. apple as a fruit name). As attacker knows the differences of candidate pre-trained models, he/she can compile a query list with such knowledge. Note that the attacker is only aware of the pre-trained models in his/her model pool, not the downstream training datasets or architecture of the victim fine-tuned model.

5.4. Pre-trained Model Extractor

5.4.1. Model Fingerprint Recognition. The shapes of the model fingerprint are different for different types of ML models (e.g., CNN and Transformers use totally different architecture). Pre-trained model extractor uses a proper pattern recognition algorithm for the target model architecture. We describe Transformer model extraction.

Transformer models run identically-shaped encoders or decoders repeatedly (Section 2). Thus, architecture hints also show repetitive patterns. Therefore, a group of repeated measurements can be considered as one layer. However, the detection of the repeating groups is challenging because the number of layers, the volume of per-layer computations, and the intra-layer structure are pretty diverse in different Transformer models. Thus, the pre-trained model extractor for Transformer is designed to recognize diverse patterns of repetition from time-series kernel execution data. Fig. 10 shows example time-series kernel execution graphs where the group of kernels in the box is repeatedly executed. Though the shape and size of the group in the two models are very different, it is clearly seen that the repeating count matches the number of layers of each model (BERT-base has 12 group executions and BERT-large has 24). We observed similar patterns from most of the tested Transformers, while the shape of each group is pretty diverse due to model signatures.



(a) BERT-large (b) BERT-base

Figure 10: Layer Boundary Identification

Along with the number of layers, the size of layers is also an important parameter that determines different Transformer architectures. For example, DeBERTa-xsmall [22] uses 12 layers with 384 hidden states. GPT-2 [38] also uses 12 decoders but includes 768 hidden states. Due to the different number of hidden states, the layer size of the two models is notably different. The model extractor recognizes the layer size through the peak execution time in each kernel group. In Fig. 10, BERT-base's peak kernel execution time is around 0.6*ms* while BERT-large's is around 0.8*ms* because BERT-base uses 768 hidden states while BERT-large uses 1024 states.

5.4.2. Model Extraction through Image Recognition. As the shape and count of kernel groups vary from one model to another, it is undesirable to manually detect the model fingerprint. Therefore, the pre-trained model extractor internally runs an automated detection method.

To automate the attack process, there are two challenges: the automation needs to 1) detect kernel groups having different size and shape and 2) process 2-dimensional information of time-series kernel execution data (($T_{invocation}$, $T_{termination}$) as noted in Section 5.2). We found that this problem can be reduced to a pattern recognition problem for 2-dimensional inputs, which is similar to *image recognition*. The repetitive kernel groups are the target patterns to detect and the time-series kernel execution graph can be considered as an input image. We chose to use a CNN model that can detect patterns from images efficiently.

Architecture Hint Data Conversion: To apply CNN on the kernel execution time data, we convert the data to 2-dimensional images. We first plot the time-series kernel execution graphs with the same x- and y-scales (square shape) and convert them into images. Then, all other

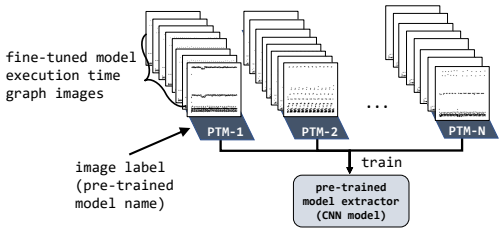


Figure 11: CNN Model Training

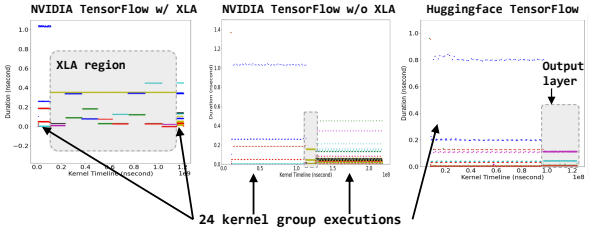


Figure 12: Irregular Execution Patterns

information (x- and y-axis, numbers, and texts) except for the execution patterns are removed. The images are then converted to gray-scale to remove any color bias. Finally, the images are re-shaped to 1024x1024 equal-sized images to be fed to the same CNN model. The example images are shown in Fig. 11. We collected 1787 images from 70 pre-trained and 170 fine-tuned models (Section 7). 80% of them were used for training and 20% for testing. As the goal of the detection is to recognize the pre-trained model, we labeled each graph image with each model's pre-trained model name. For example, a MobileBERT fine-tuned model's image was labeled as *google_mobilebert-uncased*.

CNN Model Training: The CNN model receives a kernel execution time image of a fine-tuned model and predicts the pre-trained model name. We explored various CNN architectures and finalized the CNN model with seven hidden layers: two convolution and pooling layers interleaving each other and followed by three fully connected layers - conv (in: 3, out: 6, kernel: 5×5), pool (kernel: 8×8, stride: 8), conv (in: 6, out: 16, kernel: 5×5), pool (kernel: 8×8, stride: 8), fc (in: 3600, out: 120), fc (in: 120, out: 84), fc (in: 84). We used PyTorch for training with learning rate and epochs as 0.001 and 10 respectively. The ground truth of each prediction is the pre-trained model name of the input image. If multiple pre-trained models are high-ranked in the CNN classification, we locate one through the query outputs (Section 5.3).

5.4.3. Handling Corner Cases. There are some models that have indistinguishable layer boundaries, mainly due to some optimizations (e.g., NVIDIA TensorFlow in Fig. 7 shows irregular execution pattern due to such optimizations (e.g., XLA [17])). Fig. 12 shows three example execution patterns of BERT-large models that a simple layer boundary detection can't be used. For these cases, layers can be detected in the regions excluding the gray-colored regions (marked as XLA region and output layer). XLA optimization runs compiler optimization operations in the middle of the inference and the executions for encoders are at the beginning and the end of the inference. In both encoder regions, we can find 24 repetitive kernel group executions. For these models, we do pre-processing to find the encoder regions and feed the CNN model with the image of the encoder regions.

6. Gray/White-box Attacks with Decepticon

Decepticon enables various gray-box attacks. We explore two attack scenarios 1) stealing intellectual property by creating a clone of the victim model and 2) perturbing victim model's prediction through adversarial attack.

6.1. Stealing Proprietary Fine-tuned Model

Once the pre-trained model is recovered, we can use the weights of the pre-trained model for extracting the victim model weights. The weights of the pre-trained model and the fine-tuned model are very similar but not identical. According to our experiments (Fig. 17), the pre-trained model itself cannot be used for downstream tasks, unless the attacker has the fine-tuning datasets. In our threat model, we do not assume to have fine-tuning datasets because fine-tuning is normally done with multiple in-house training datasets. Thus, the attacker needs to further reduce the weight value gap to clone the victim model.

We leverage an existing side-channel (row-hammer-based bit value checking [40]) to recover the actual weight value (weight address is collected (Section 3)). Unlike the existing study that had to check every weight value bits [40], we use the recovered pre-trained model as a baseline and check only the essential bits that are likely to be updated during fine-tuning. We design a *selective weight extraction* that systemically finds these essential bits to check.

6.1.1. Selective Weight Extraction. Selective weight extraction significantly reduces the scope of bit checking to those that cover the weight value gap between a pre-trained model and its fine-tuned model. We describe the process with float32 data while the algorithm can be applied to any data type. From our experiments, sign and exponent fields rarely change; an average of 99% weights keep their sign when fine-tuned. Only the fraction fields are updated mostly where only a handful of bits are corresponding to the small value gap (e.g., 0.002-0.01 in our earlier examples). Fig. 13 shows an example when weights use IEEE 754 format. Suppose that a weight in a pre-trained model is 0.018 and is fine-tuned to 0.01908. The fine-tuned weight value is black-box. Thus, the attacker should estimate the value, 0.01908, from the pre-trained model's weight value, 0.018. As illustrated, the sign, exponent, and first a few bits of the fraction field are identical. Only the two bits in blue color are responsible for the value gap, thus need checking. The remaining 18 bits in the fraction field do not need to be checked because those make very subtle differences (less than 0.001). From our experiments, the model prediction accuracy is almost the same (F1 score is dropped by less than 0.01) when discarding all weight values below 0.001.

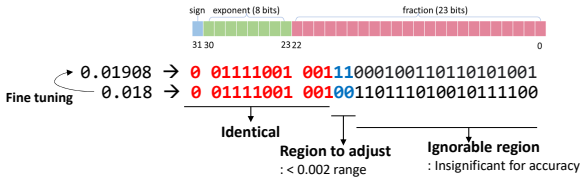


Figure 13: Selective Weight Extraction

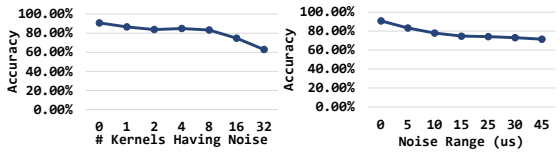


Figure 14: Extraction Accuracy: (left) impact of kernel count having noise (right) impact of noise lengths

The selective extraction identifies the bits of individual weights that *do not need to be checked* based on the pre-trained model weight values through two steps; 1) excluding the weights less than 0.001 and 2) for the remaining weights, excluding all the bits except for a small number of bits that correspond to the weight value gap. In the second step, the bit locations to check may vary depending on the exponent field value. For example, in Fig. 13, the blue bits are checked because they are corresponding to 0.00097 (=2^{-10}) and 0.00048 (=2^{-11}) respectively (together cover the estimated weight value gap (e.g., 0.002)) as the first bit value of fraction field is 2^{-7} (=$2^{121-127-1}$) according to IEEE 754. As weights are updated differently (U-shape distribution in Section 4.1), we check the essential bits for the estimated weight value gap based on the pre-trained weight value. We found that checking up to two bits per weight is sufficient for most cases. Algorithm 1 summarizes the process.

Algorithm 1: Selective Weight Extraction

```
target_weight ← W_i in fine-tuned model
base_weight ← W_i in pre-trained model
abs_base_weight ← abs(W_i) in pre-trained model
if abs_base_weight < 0.001 then
    clone model's W_i ← base_weight
end
else
    min ← base_weight − weight_dist
    max ← base_weight + weight_dist
    k ← most significant non-zero bit in fraction
    int_base ← 2^(exp−127) of abs_base_weight
    while up to 2 iterations do
        fr_base ← 2^(exp−127−k) of abs_base_weight
        if min ≤ (int_base + fr_base) ≤ max then
            check k_th bit of target_weight.
            clone model's k_th bit of W_i ← k_th bit of
              target_weight
        end
        k ← k + 1
    end
end
```

Leveraging the lower impact of first a few layers towards the downstream task accuracy (See Table 1), we begin the extraction from later layers and gradually recover earlier layers while checking the prediction accuracy of the clone model. The extraction stops when the clone model's accuracy becomes similar to the victim model.

For the task-dependent last layer, the pre-trained model does not have the baseline weight values because the layer is newly added while fine-tuning. Thus, we use row-hammer for all bit values. As the last layer has much fewer weights than the other layers, our selective extraction still reduces significant bit checking efforts. For all Transformer model sizes (from tiny to large), the last layer only contributes from 0.0005% to 0.009% of the total weight count (Section 7).

6.2. Adversarial Attack

The cloned model enables various white-box attacks, such as an adversarial attack. The adversarial attack finds adversarial inputs that perturb the prediction accuracy of the victim model. Suppose a victim model M has an output Y for a given input X (i.e., $Y = M(X)$). The attacker wants to find α that makes $Z = M(X+\alpha)$ where $Y \neq Z$. To demonstrate an adversarial attack, we generated adversarial inputs by using our clone model, tested the victim model with the adversarial inputs, and checked how many of the adversarial inputs led to incorrect predictions. We compared the effectiveness of our clone model with eight other substitute models. The substitute models (M') were fine-tuned from randomly selected pre-trained models with the victim model's (M) prediction records (i.e., $Y = M'(X)$ *where* $Y = M(X)$).

7. Evaluation

7.1. Methodology

We examined 70 pre-trained models and 170 fine-tuned models downloaded from various model repositories [5, 6, 8–10, 12, 13, 15]. The models use diverse Transformer architectures and sizes such as tiny, mini, distill, medium, base, large, xlarge, and xxlarge of BERT, GPT-2, RoBERTa, MobileBERT [43], CamemBERT [34], ALBERT [28], DeBERTa [22], XLNet [53], BART [29], T5 [39], and SpanBERT [24]. We selected these models based on their download counts (at least 100K downloads per month). The models are trained for various tasks but there is no single pair of pre-trained and fine-tuned models that are trained for the same task. A ResNet-18 model was also downloaded from PyTorch vision repository [15] and fine-tuned to examine the generalization of our approach to non-Transformer models. We tested these models on an NVIDIA GeForce RTX 3050 (Ampere) GPU with CUDA v11.8.

7.2. Model Extraction Accuracy

We tested the model extraction accuracy with 350 data (of 1787 total training data). We evaluated the prediction accuracy by adding two types of noise to the kernel execution time measurements: 1) varying the scope of kernels impacted by noise and 2) varying significance of noise for individual kernel. Based on the typical kernel duration,

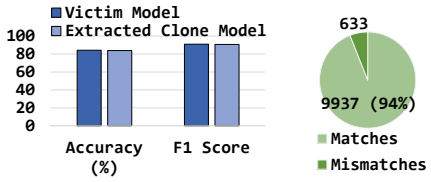


Figure 15: (left) Prediction Accuracy of Victim BERT model and Extracted model, (right) Fraction of Matched Predictions

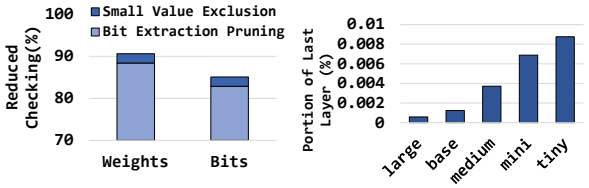


Figure 16: (left) Reduced Weight Value Checking w/o Errors, (right) Fraction of Last Layer in Total Model Weight Count

we set the noise in the first test as $20\mu s$. We randomly selected 1 to 64 kernels from each input image and adjusted their execution times to *original execution time* $\pm$ $20\mu s$. For the second test, we selected 16 kernels from each validation image and changed their execution times to *original execution time* $\pm K\mu s$, where K is varied from 5 to $45\mu s$. Then, the adjusted time-series kernel execution images were fed to the CNN model for prediction. Fig. 14 shows the averaged accuracy results. The accuracy is 90.78% without noise and dropped slowly for both types of noises thanks to inherently error-tolerant CNN [30].

7.3. Cloned Model Accuracy

We tested the cloning accuracy by comparing the inference outputs of the victim model and the cloned model. Fig. 15 shows the results on a BERT-large victim model and the extracted clone model on 10570 SQuAD inputs. The accuracy and F1 score of the clone model had only 0.2% difference from the victim model. 94% of the cloned model's predictions matched with victim model.

7.4. Weight Extraction Efficiency

To understand the efficiency of selective weight extraction, we examined the weight values of a randomly selected fine-tuned model and its pre-trained model and counted the total number of bits that needed to be checked. If the actual weight value gap was larger than expected amount or the sign bit was changed, we considered that the selective weight extraction led to incorrect extraction. Fig. 16 shows the breakdown of the reduced weight checking. The *Weights* bar shows the total number of weights correctly pruned. The *Bits* bar shows the total number of bits correctly excluded from checking. 90% weights and 85% bits of weights were correctly excluded from checking. The last layer hammering did not incur much overhead because the last layer contributes only up to 0.009% of the entire model weight count even in the smallest Transformer model, as plotted in Fig. 16.

Figure 17: Cloning Accuracy with Fine-tuning Data

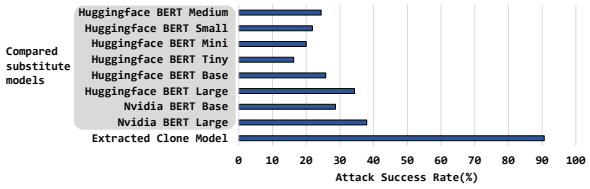


Figure 18: Effectiveness on Adversarial Attack

7.5. Weight Extraction Necessity

To check if weight extraction is necessary, we also measured cloning accuracy by assuming that an attacker has access to limited amount of fine-tuning dataset. Fig. 17 shows the results of fine-tuning a pre-trained BERT-base with different amounts of downstream data. The attacker needs at least 40% fine-tuning data to copy the model with less than 5% accuracy drop, which is unrealistic.

7.6. Adversarial Attack Effectiveness

To compare the effectiveness of adversarial attack, we developed eight other substitute models besides our extracted clone model. The eight models were developed by downloading random pre-trained models and fine-tuning them with the victim model's prediction records that were collected from 18K inferences. Fig. 18 shows the success rate of the nine models when 10K sample inputs were tested to locate adversarial inputs. The adversarial inputs identified by our clone model showed a superior attack success rate (90.62%) than the other models (up to 38% accuracy).

7.7. Attack Generalization

To check if weight value similarity is a universal symptom of transfer learning, we measured weight similarity of a CNN model, ResNet-18, between a fine-tuned model and its pre-trained model as plotted in Fig. 19. We fine-tuned a pre-trained model with Hymenoptera dataset. For a comparison, we trained another ResNet-18 model with the same dataset from scratch (without using transfer learning) and compared the weight values with the fine-tuned model. The darker blue bars are the weight difference between the fine-tuned model and its pre-trained model. The lighter bars are the difference from the model trained from scratch. The fine-tuned model had almost zero weight difference from its pre-trained model in almost all layers, while it showed at least 20$\times$ higher difference from the other model, even when they were trained with the same dataset, which is consistent with our observations from Transformer models.

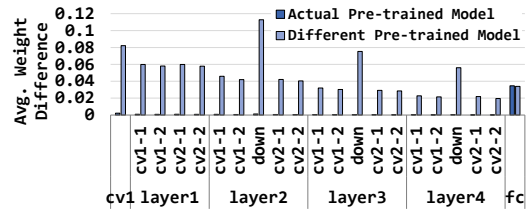
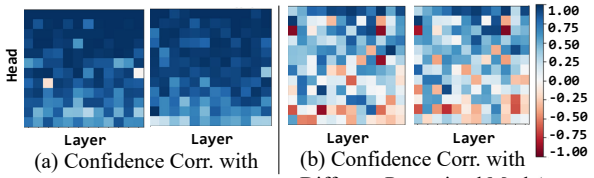


Figure 19: Weight Similarity in a CNN Model (ResNet-18)



(a) Confidence Corr. with Baseline Pre-trained Model

(b) Confidence Corr. with Different Pre-trained Model

Figure 20: Confidence Correlation used for Head Pruning

8. Discussions

Supporting Quantization and Pruning: We explained the weight extraction with float32 data. However, models can be quantized during/after fine-tuning. Our selective weight extraction is applicable for other data types. For example, compared to float32, float16 uses shorter (5-bit) exponent and (10-bit) fraction fields, while bfloat16 uses the same-length (8-bit) exponent with a shorter (7-bit) fraction field. If bfloat16 is used in the example of Fig. 13, the same bits (the blue bits) can be checked as bfloat16 uses the same length exponent with float32. Likewise, our selective weight extraction is applicable with slight bit adjustment.

Models can be optimized during fine-tuning. Head pruning is a popular optimization that removes insignificant heads from computation [47, 48]. To find the insignificant heads, *Confidence* is calculated by averaging the maximum attention weights. By exploiting the weight similarity, the attacker can calculate similar confidence values with the pre-trained model and locate the pruned heads. We evaluated the confidence value similarity between a pre-trained model and its two fine-tuned models that are trained for different tasks as shown in Fig. 20(a). Each cell is a Pearson correlation coefficient between the confidence of the heads on the same location in a pre-trained model and a fine-tuned model. Darker blue cells mean high correlation. In both fine-tuned models, confidence of all heads are highly correlated. When we compared the fine-tuned models with a different pre-trained model, the correlation dropped significantly as plotted in Fig. 20(b). The number of pruned heads also can be detected from kernel execution time. Fig. 21 shows time-series kernel execution time images where different numbers of heads are pruned (y-axis is kernel duration). Groups of short kernels (near the bottom) executed faster as more heads were pruned. By combining these two observations, the attacker can figure out exactly which heads are pruned. We also found that each head is placed in a specific location in the weight matrixes (i.e., the weights of head 0 come first and are followed by weights of head 1, and so on). By removing the pruned heads' weights from the weight file, the attacker can match the dimension of the weight matrix of the pruned fine-tuned and unpruned pre-trained model.

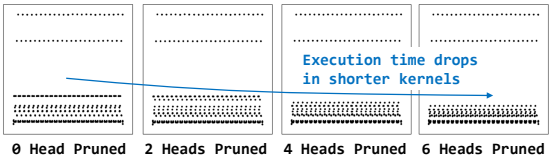


Figure 21: Impact of Head Pruning on Execution Time

Counter Measures: As we may not be able to perfectly prevent pre-trained model leakage due to its sharing nature, we can consider removing the fingerprint from model executions by randomizing the selections of GPU kernels/libraries and usages of various optimizations. Though there is only a handful of GPU deep learning libraries (e.g., cuDNN, cuBLAS), an algorithm can be executed with various combinations of library functions. If the combination selection is randomly determined at run time, it would be challenging to extract computation patterns.

9. Related Work

Several studies showed ML model extraction attacks. [23, 37] proposed a framework to extract a black-box CNN model architecture by using performance counter and bus/memory probing on GPUs. [56] proposed a technique to clone CNN model weights by monitoring unencrypted PCI-e bus traffic. [51] extended the cache-timing attack for model extraction on CNNs. [40] leveraged row-hammer attack [25] to recover weight values of victim models that use int8 weights. **All these studies targeted small CNN models, while we propose a new model extraction attack for billion-parameter models. Decepticon is the first approach using two-level attack model.**

Some studies [27, 32, 50] demonstrated security concerns in Transformer models by cloning a BERT model with prediction records. **These studies did not extract model architecture and Decepticon showed a superior cloning accuracy than these approaches as can be seen in the adversarial attack result (Fig. 18).**

10. Conclusion

This paper raises a new security concern for transfer-learned models. We show that model fingerprints can be leveraged to locate the pre-trained model. The identified pre-trained model becomes a good source of revealing weight values in the entire model level, which enables various gray/white-box attacks. This is the first study that extracts entire weights of large-scale models (e.g., Transformer) that use billions/trillions of weights.

Acknowledgments

We thank Murali Annavaram for his valuable feedback, and Xavier Ybarra and Aishwaria Rangasamy for their help in data collection.

References

[1] "DeepSniffer ResNet Models," https://github.com/xinghu7788/DeepSniffer/.

[2] Finding Startups with BERT. [Online]. Available: https://medium.com/axel-springer-tech/finding-startups-with-bert-4ae29f686a9e

[3] Google BERT TensorFlow. https://github.com/google-research/bert.

[4] Hugging Face BERT-base Pre-trained Model. https://huggingface.co/bert-base-uncased.

[5] Hugging Face BERT PyTorch. https://huggingface.co/docs/transformers/v4.17.0/en/model_doc/bert.

[6] Hugging Face BERT TensorFlow. https://huggingface.co/docs/transformers/model_doc/bert#transformers.TFBertForQuestionAnswering.

[7] Hugging Face Distilled GPT-2 Pre-trained Model. https://huggingface.co/distilgpt2.

[8] Hugging Face General Language Understanding Evaluation (GLUE) Benchmark. https://github.com/huggingface/transformers/tree/main/examples/pytorch/text-classification#glue-tasks.

[9] Hugging Face Models. https://huggingface.co/models.

[10] Meta RoBERTa PyTorch. https://github.com/pytorch/fairseq/tree/main/examples/roberta.

[11] MXNet ResNet Models. https://github.com/apache/incubator-mxnet/blob/master/python/mxnet/gluon/model_zoo/vision/resnet.py.

[12] NVIDIA BERT PyTorch. https://github.com/NVIDIA/DeepLearningExamples/tree/master/PyTorch/LanguageModeling/BERT.

[13] NVIDIA BERT TensorFlow. https://github.com/NVIDIA/DeepLearningExamples/tree/master/TensorFlow2/LanguageModeling/BERT.

[14] NVIDIA ResNet v1.5 for PyTorch. https://catalog.ngc.nvidia.com/orgs/nvidia/resources/resnet_50_v1_5_for_pytorch.

[15] Pytorch Hub: Repository of Pre-trained models. https://pytorch.org/docs/stable/hub.html.

[16] "TensorFlow ResNet Models," https://tfhub.dev/google/imagenet/resnet_v2_50/classification/5.

[17] XLA: Optimizing Compiler for Machine Learning. https://www.tensorflow.org/xla.

[18] R. Bommasani, D. A. Hudson, E. Adeli, R. Altman, S. Arora, S. von Arx, M. S. Bernstein, J. Bohg, A. Bosselut, E. Brunskill, E. Brynjolfsson, S. Buch, D. Card, R. Castellon, N. S. Chatterji, A. S. Chen, K. A. Creel, J. Davis, D. Demszky, C. Donahue, M. Doumbouya, E. Durmus, S. Ermon, J. Etchemendy, K. Ethayarajh, L. Fei-Fei, C. Finn, T. Gale, L. E. Gillespie, K. Goel, N. D. Goodman, S. Grossman, N. Guha, T. Hashimoto, P. Henderson, J. Hewitt, D. E. Ho, J. Hong, K. Hsu, J. Huang, T. F. Icard, S. Jain, D. Jurafsky, P. Kalluri, S. Karamcheti, G. Keeling, F. Khani, O. Khattab, P. W. Koh, M. S. Krass, R. Krishna, R. Kuditipudi, A. Kumar, F. Ladhak, M. Lee, T. Lee, J. Leskovec, I. Levent, X. L. Li, X. Li, T. Ma, A. Malik, C. D. Manning, S. P. Mirchandani, E. Mitchell, Z. Munyikwa, S. Nair, A. Narayan, D. Narayanan, B. Newman, A. Nie, J. C. Niebles, H. Nilforoshan, J. F. Nyarko, G. Ogut, L. Orr, I. Papadimitriou, J. S. Park, C. Piech, E. Portelance, C. Potts, A. Raghunathan, R. Reich, H. Ren, F. Rong, Y. H. Roohani, C. Ruiz, J. Ryan, C. R'e, D. Sadigh, S. Sagawa, K. Santhanam, A. Shih, K. P. Srinivasan, A. Tamkin, R. Taori, A. W. Thomas, F. Tramèr, R. E. Wang, W. Wang, B. Wu, J. Wu, Y. Wu, S. M. Xie, M. Yasunaga, J. You, M. A. Zaharia, M. Zhang, T. Zhang, X. Zhang, Y. Zhang, L. Zheng, K. Zhou, and P. Liang, "On the opportunities and risks of foundation models," in *arXiv:2108.07258*, 2021.

[19] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, 2020.

[20] R. Callan, A. Zajić, and M. Prvulovic, "A Practical Methodology for Measuring the Side-Channel Signal Available to the Attacker for Instruction-Level Events," in *IEEE/ACM International Symposium on Microarchitecture*, 2014.

[21] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv:1810.04805*, 2018.

[22] P. He, X. Liu, J. Gao, and W. Chen, "DeBERTa: Decoding-enhanced BERT with Disentangled Attention," *arXiv:2006.03654*, 2021.

[23] X. Hu, L. Liang, S. Li, L. Deng, P. Zuo, Y. Ji, X. Xie, Y. Ding, C. Liu, T. Sherwood, and Y. Xie, "DeepSniffer: A DNN Model Extraction Framework Based on Learning Architectural Hints," in *Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[24] M. Joshi, D. Chen, Y. Liu, D. S. Weld, L. Zettlemoyer, and O. Levy, "SpanBERT: Improving Pre-training by Representing and Predicting Spans," *arXiv:1907.10529*, 2020.

[25] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu, "Flipping Bits in Memory Without Accessing Them: An experimental study of DRAM disturbance errors," *ACM SIGARCH Computer Architecture News*, vol. 42, no. 3, pp. 361–372, 2014.

[26] J. Kirkpatrick, R. Pascanu, N. Rabinowitz, J. Veness, G. Desjardins, A. A. Rusu, K. Milan, J. Quan, T. Ramalho, A. Grabska-Barwinska, D. Hassabis, C. Clopath, D. Kumaran, and R. Hadsell, "Overcoming Catastrophic Forgetting in Neural Networks," in *Proceedings of the National Academy of Sciences*, 2017.

[27] K. Krishna, G. S. Tomar, A. P. Parikh, N. Papernot, and M. Iyyer, "Thieves on Sesame Street! Model Extraction of BERT-based APIs," in *International Conference on Learning Representations*, 2020.

[28] Z. Lan, M. Chen, S. Goodman, K. Gimpel, P. Sharma, and R. Soricut, "ALBERT: A Lite BERT for Self-supervised Learning of Language Representations," in *International Conference on Learning Representations*, 2020.

[29] M. Lewis, Y. Liu, N. Goyal, M. Ghazvininejad, A. Mohamed, O. Levy, V. Stoyanov, and L. Zettlemoyer, "BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension," *arXiv:1910.13461*, 2019.

[30] G. Li, S. K. S. Hari, M. Sullivan, T. Tsai, K. Pattabiraman, J. Emer, and S. W. Keckler, "Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017.

[31] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "RoBERTa: A Robustly Optimized BERT Pretraining Approach," *arXiv:1907.11692*, 2019.

[32] L. Lyu, X. He, F. Wu, and L. Sun, "Killing Two Birds with One Stone: Stealing Model and Inferring Attribute from BERT-based APIs," *abs/2105.10909*, 2021.

[33] H. T. Maia, C. Xiao, D. Li, E. Grinspun, and C. Zheng, "Can one hear the shape of a neural network?: Snooping the GPU via Magnetic Side Channel," in *31st USENIX Security Symposium*, 2022.

[34] L. Martin, B. Muller, P. J. Ortiz Suárez, Y. Dupont, L. Romary, É. de la Clergerie, D. Seddah, and B. Sagot, "CamemBERT: a tasty French language model," in *Proceedings of the Association for Computational Linguistics*, 2020.

[35] M. McCloskey and N. J. Cohen, "Catastrophic Interference in Connectionist Networks: The Sequential Learning Problem," in *Psychology of Learning and Motivation*, 1989.

[36] G. Moy, S. Shekh, M. Oxenham, S. E.-S. (Joint, O. A. D. D. Science, and T. Group), "Recent advances in artificial intelligence and their impact on defence," *DST-Group-TR-3716*, 2020.

[37] H. Naghibijouybari, A. Neupane, Z. Qian, and N. Abu-Ghazaleh, "Rendered Insecure: GPU Side Channel Attacks are Practical," in *ACM Conference on Computer and Communications Security*, 2018.

[38] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language Models are Unsupervised Multitask Learners," *OpenAI*, vol. 1, no. 8, p. 9, 2019.

[39] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the limits of transfer learning with a unified text-to-text transformer," *The Journal of Machine Learning Research*, 2020.

[40] A. S. Rakin, M. H. I. Chowdhuryy, F. Yao, and D. Fan, "DeepSteal: Advanced Model Extractions Leveraging Efficient Weight Stealing in Memories," *IEEE Symposium on Security and Privacy*, 2022.

[41] A. T. Ray, B. F. Cole, O. J. P. Fischer, R. T. White, and D. N. Mavris, "aeroBERT-Classifier: Classification of Aerospace Requirements Using BERT," in *Aerospace 2023, 10(3), 279*, 2023.

[42] N. Roberts, V. U. Prabhu, and M. McAteer, "Model Weight Theft With Just Noise Inputs: The Curious Case of the Petulant Attacker," *abs/1912.08987*, 2019.

[43] Z. Sun, H. Yu, X. Song, R. Liu, Y. Yang, and D. Zhou, "MobileBERT: a compact task-agnostic BERT for resource-limited devices," in *Proceedings of the Association for Computational Linguistics*, 2020.

[44] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, D. Bikel, L. Blecher, C. C. Ferrer, M. Chen, G. Cucurull, D. Esiobu, J. Fernandes, J. Fu, W. Fu, B. Fuller, C. Gao, V. Goswami, N. Goyal, A. Hartshorn, S. Hosseini, R. Hou, H. Inan, M. Kardas, V. Kerkez, M. Khabsa, I. Kloumann, A. Korenev, P. S. Koura, M.-A. Lachaux, T. Lavril, J. Lee, D. Liskovich, Y. Lu, Y. Mao, X. Martinet, T. Mihaylov, P. Mishra, I. Molybog, Y. Nie, A. Poulton, J. Reizenstein, R. Rungta, K. Saladi, A. Schelten, R. Silva, E. M. Smith, R. Subramanian, X. E. Tan, B. Tang, R. Taylor, A. Williams, J. X. Kuan, P. Xu, Z. Yan, I. Zarov, Y. Zhang, A. Fan, M. Kambadur, S. Narang, A. Rodriguez, R. Stojnic, S. Edunov, and T. Scialom, "Llama 2: Open Foundation and Fine-Tuned Chat Models," in *arXiv:2307.09288*, 2023.

[45] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing Machine Learning Models via Prediction APIs," in *USENIX Security Symposium*, 2016.

[46] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention Is All You Need," *arxiv:1706.03762*, 2017.

[47] E. Voita, R. Sennrich, and I. Titov, "Analyzing the Source and Target Contributions to Predictions in Neural Machine Translation," in *Annual Meeting of the Association for Computational Linguistics*, 2021.

[48] E. Voita, D. Talbot, F. Moiseev, R. Sennrich, and I. Titov, "Analyzing Multi-Head Self-Attention: Specialized Heads Do the Heavy Lifting, the Rest Can Be Pruned," in *Annual Meeting of the Association for Computational Linguistics*, 2019.

[49] T. Wachi, "Business application of bert, a general-purpose natural-language-processing model," *Feature Articles: ICT Solutions Offered by NTT Group Companies*, 2021.

[50] Q. Xu, X. He, L. Lyu, L. Qu, and G. Haffari, "Beyond Model Extraction: Imitation Attack for Black-Box NLP APIs," *abs/2108.13873*, 2021.

[51] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures," in *USENIX Security Symposium*, 2020.

[52] ——, "Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures," in *USENIX Security Symposium*, 2020.

[53] Z. Yang, Z. Dai, Y. Yang, J. Carbonell, R. Salakhutdinov, and Q. V. Le, "XLNet: Generalized Autoregressive Pretraining for Language Understanding," *arXiv:1906.08237*, 2020.

[54] Z. Yue, Z. He, H. Zeng, and J. McAuley, "Black-Box Attacks on Sequential Recommenders via Data-Free Model Extraction," in *ACM Conference on Recommender Systems*, 2021.

[55] M. A. Yuri Kuratov, "Adaptation of Deep Bidirectional Multilingual Transformers for Russian Language," *arXiv:1905.07213*, 2019.

[56] Y. Zhu, Y. Cheng, H. Zhou, and Y. Lu, "Hermes Attack: Steal DNN Models with Lossless Inference Accuracy," in *USENIX Security Symposium*, 2021.