PROTEUS: PRESERVING MODEL CONFIDENTIALITY DURING GRAPH OPTIMIZATIONS

Yubo Gao [1 2 3] **Maryam Haghifam** [1] **Christina Giannoula** [1 3] **Renbo Tu** [1] **Gennady Pekhimenko** [1 2 3]
Nandita Vijaykumar [1 2]

ABSTRACT

Deep learning (DL) models have revolutionized numerous domains, yet optimizing them for computational efficiency remains a challenging endeavor. Development of new DL models typically involves two parties: the model developers and performance optimizers. The collaboration between the parties often necessitates the model developers exposing the model architecture and computational graph to the optimizers. However, this exposure is undesirable since the model architecture is an important intellectual property, and its innovations require significant investments and expertise. During the exchange, the model is also vulnerable to adversarial attacks via model stealing.

This paper presents PROTEUS, a novel mechanism that enables model optimization by an independent party while preserving the confidentiality of the model architecture. PROTEUS obfuscates the protected model by partitioning its computational graph into subgraphs and concealing each subgraph within a large pool of generated realistic subgraphs that cannot be easily distinguished from the original. We evaluate PROTEUS on a range of DNNs, demonstrating its efficacy in preserving confidentiality without compromising performance optimization opportunities. PROTEUS effectively hides the model as one alternative among up to 10^{32} possible model architectures, and is resilient against attacks with a learning-based adversary. We also demonstrate that heuristic based and manual approaches are ineffective in identifying the protected model. To our knowledge, PROTEUS is the first work that tackles the challenge of model confidentiality during performance optimization. PROTEUS will be open-sourced for direct use and experimentation, with easy integration with compilers such as ONNXRuntime.

1 INTRODUCTION

Deep learning (DL) has emerged as a highly effective approach with a wide range of use cases. The remarkable performance achieved by DL models in domains like computer vision, natural language processing, and recommendation systems has immensely fueled their popularity. Models for ChatGPT, stable diffusion (Rombach et al., 2021), and vision transformers (Dosovitskiy et al., 2021), all demonstrate the potential of DL models in solving complex tasks. This has led to widespread interest in the generation of new models and DL innovations for novel and more powerful capabilities in both academia and industry.

A major challenge with DL models is the significant computational overhead for training and inference. DL models may have millions to hundreds of billions of parameters (Labs, 2023), requiring significant memory resources and compute. Thus, training DL models and deploying trained models for inference can be extremely expensive and time-consuming. This issue is expected to be exacerbated in the future, as model sizes continue to grow. For example, OpenAI reports a daily cost of $700K to run ChatGPT (Insider, 2023).

Performance optimizations using ML compilers have thus become crucial to efficient training and inference to reduce latency, computational expenses, and energy consumption. Recently developed optimizing compilers/tools include TVM (Chen et al., 2018), TASO (Jia et al., 2019), ONNXRuntime (developers, 2021), and Hidet (Ding et al., 2023) and is an active area of research and development. Existing tools are already proven to be highly effective in generating significant speedups and are thus widely used. For example, TVM can provide up to $3.8\times$ speedup on model inference (Chen et al., 2018).

[1]Department of Computer Science, University of Toronto [2]Vector Institute [3]CentML Inc. Correspondence to: Yubo Gao <ybgao@cs.toronto.edu>.

Proceedings of the 5^{th} MLSys Conference, Santa Clara, CA, USA, 2024.

Model optimization and model creation/development are typically not done at the same time by *the same party*. Model development and optimization each requires different expertise and domain knowledge. For example, while model developers are good at designing neural network architectures, they often do not possess the domain skills necessary for performance optimization. This has led to the emergence of companies offering model optimization as a service such as OctoML (oct, a) and MosaicML (mos) to fill in the gap.

Several existing compilers (Chen et al., 2018; developers, 2021; Ding et al., 2023; NVIDIA Corporation, 2022) can be used to *automatically* optimize the model, potentially enabling model developers to directly produce performant implementations without a second party for optimization. However, *solely* relying on automatic compilers has limitations and does not eliminate the need for additional optimization expertise. First, effectively optimizing tensor computations is still challenging even when using an automatic compiler and often requires significant domain expertise and intervention. For example, correctly configuring the tensor compiler (e.g., selecting search space, using the correct floating point precision), adding previously-unsupported operators (TVM, 2023), or implementing scheduling templates for novel operators (Ding et al., 2023) requires systems expertise. Second, they are less effective at optimizing for proprietary hardware or at leveraging hardware features that are not fully exposed by hardware vendors (Zheng et al., 2020), and thus may require specialized expertise from hardware vendors about their hardware. For example, the optimizations/libraries specific to Google's TPUs in XLA and NVIDIA GPUs are closed source and require support from the hardware vendors when the provided tools are not effective (tfx). Third, the developers of novel optimizing tools may not provide open-source implementations or the entire toolset for automatic use due to proprietary optimizations or the need for manual intervention. For example, OctoML (oct, a) applies proprietary optimizations manually for customers(oct, b), a process which requires manual insight.

The necessity of two parties to effectively develop and optimize new DL models leads to an important novel challenge: ensuring confidentiality of the DL architecture. Highly effective performance optimizations include graph-level transformations which involve optimizing the computational graph (i.e. the graph of operators). Possible transformations include techniques such as operator fusion, constant folding, and functional approximations (ort). Graph level performance optimizations typically require providing the optimizing party direct access to the entire computational graph of the model. However, the model architecture itself is expensive intellectual property to the model developers, as innovating novel DL architectures requires domain experts and extensive resources for neural architecture search and training. For example, NASNet (Zoph et al., 2018) is discovered through thousands of GPU days spent on neural architecture search, and a single training of GPT-3 costs $4.6M(Labs, 2023). Additionally, exposing the model architecture exacerbates the threat of adversarial attacks (Goodfellow et al., 2014) by model stealing approaches that can then be used to perform gradient-based adversarial attacks (Goodfellow et al., 2015).

In this work, we present PROTEUS, an obfuscation mechanism that aims to preserve the confidentiality of the protected model during graph optimizations. PROTEUS effectively enables an independent party with a proprietary optimization tool such as a machine learning compiler to optimize a novel model architecture with no direct knowledge of the original model architecture. PROTEUS is largely agnostic to the optimizations themselves and can be generally used by any optimization tool.

The key idea behind PROTEUS is twofold. First, we propose to generate *sentinel* graphs, which are artificially generated graphs that resemble real world DL computational graphs. These sentinel graphs are provided alongside the original graph to the optimizing party such that the optimizing party cannot distinguish which graph is the *protected* graph. This approach alone, however, still involves providing the optimizing party with the protected graph in its entirety.

To address this challenge, our second idea leverages *graph partitioning*. Graph partitioning first partitions the protected graph into smaller subgraphs. We then generate sentinel subgraphs for each protected subgraph. The optimizing party is now provided with a bucket of sentinel subgraphs and protected subgraphs for optimization that are indistinguishable from each other. This approach requires that the adversary would have to correctly identify *every* protected subgraph to recover the protected model. At the same time, the optimizing party can optimize each of the subgraphs flexibly. The model owners can then trivially reconstruct the original model from the optimized subgraphs.

We demonstrate that with sentinel generation and graph partitioning, it would be infeasibly expensive to correctly identify the original protected graph. We illustrate the major steps of PROTEUS in Figure 1. At a high level, PROTEUS accepts a model to be optimized by the optimizer party. The obfuscation mechanism converts the graph that needs to be optimized (hereafter referred to as the "protected" graph) into a set of subgraphs that contain both parts of the protected graph (the "protected subgraphs") as well as artificially generated subgraphs (the "sentinel subgraphs"). The optimizer-party then performs optimizations on the collection of obfuscated subgraphs (indistinguishably includes both the original and artificial subgraphs). The optimized subgraphs are returned to the model owner who can trivially assemble the optimized protected graph.

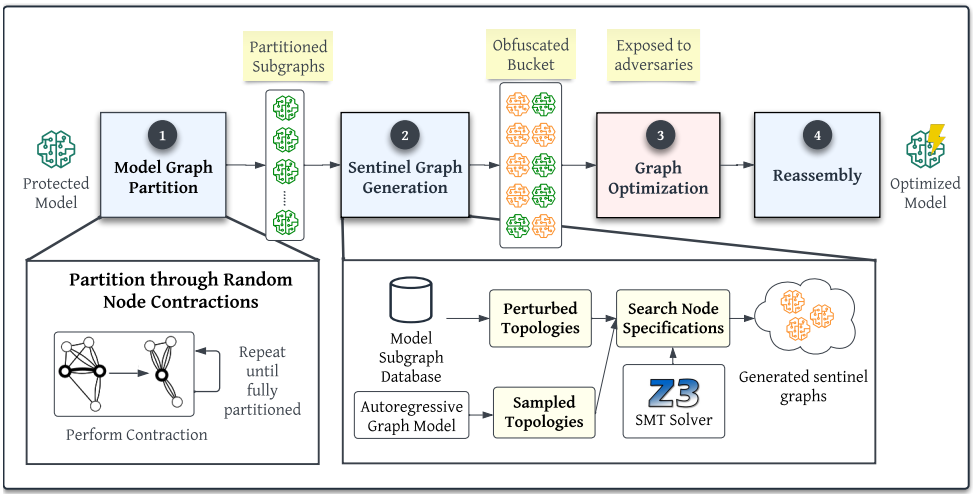


Figure 1. System Overview of PROTEUS

There are several major criteria that need to be met in designing PROTEUS. First, it is crucial that the generated sentinel subgraphs are difficult to differentiate from the protected subgraphs, while still ensuring that no general information regarding the protected graph can be inferred. Second, it is important to perform graph partitioning such that optimization opportunities are not lost, while generating enough subgraphs to sufficiently obfuscate the protected graph. We discuss how PROTEUS addresses these challenges in Section 4 and develop a mechanism that allows trading off obfuscation quality for less optimization overhead.

We evaluate PROTEUS using a range of common image and language models to evaluate the effectiveness of PROTEUS's obfuscation mechanism, we devise an adversarial attack using a learned classifier model to distinguish between real subgraphs and the generated sentinels. We demonstrate that such an attack is unable to distinguish the real protected model from a pool of 10^7 to 10^{32} potential model architectures, making recovery computationally infeasible using this method. Across all the evaluated models, PROTEUS retains the ability of the optimizer to provide significant speedups via graph-level optimization, with an average speedup within 10% of the maximum attainable by the optimizer. We demonstrate PROTEUS's overall use and effectiveness using two case studies., which show that PROTEUS remains within 1% and 10% of the speedup of the optimizer. In both cases, our method requires a learning-based adversary to evaluate more than 10^{23} models in which the original model is hidden. We will provide an open-source implementation of the tool that can be directly used with modern optimizing compilers and can serve as basis for future research on this topic.

To summarize, this work makes the following contributions:

(a) We motivate the need for a mechanism that effectively decouples model innovation and model optimization by preserving the confidentiality of the model architecture. Such a mechanism would flexibly enable model development and performance optimization to be performed by independent parties, without the optimization party having full knowledge of the confidential model architecture.

(b) We propose PROTEUS, the first mechanism to tackle this challenge of preserving confidentiality of any arbitrary DL model during performance optimization. PROTEUS partitions the protected DL model into subgraphs, and hides them within sentinel graphs. PROTEUS also effectively preserves the efficacy of various graph-level optimizations performed by optimizers.

(c) We propose a novel subgraph generation tool that is able to produce realistic artificial subgraphs to obfuscate the original subgraph. To demonstrate the robustness of our approach, we devise a learning-based attack attempting to identify the sentinels and demonstrate that it is ultimately ineffective in recovering the protected DL model. We also demonstrate that heuristic based and manual approaches are ineffective in identifying the protected model.

2 BACKGROUND AND RELATED WORK

2.1 Graph-Level Optimizations for DL Models

Deep learning (DL) compilers (Chen et al., 2018; Jia et al., 2019; Ding et al., 2023; Sabne, 2020; Rotem et al., 2018; Cyphers et al., 2018; Ye et al., 2023) provide graph-level and operator-level optimizations for DL models to accelerate their deployment and system performance. To apply these optimizations, the DL compiler operates on the graph representation of the model, i.e., the architectural structure of the model that determines how the layers and connections are organized to process input data and generate the desired outputs. A DL model is typically expressed as a

directed acyclic graph (DAG), hereafter named as *computational graph*, in which the nodes represent the DL operators (e.g., convolution, pooling, activation) and the edges represent the dependencies between the nodes, i.e., the tensors given as inputs/outputs to the operators (Bengio et al., 2009). DL compilers first apply graph-level optimizations on the computational graphs and then perform operator-level optimizations. The graph-level optimizations are rule-based transformations: they simplify executed computations, and they are either manually designed or automatically generated using heuristics algorithms. For instance, TVM (Chen et al., 2018), TensorRT (NVIDIA Corporation, 2022), and ONNXRuntime (developers, 2021) integrate generic rule-based transformations that assist in finding and applying optimizations. ONNXRuntime supports specific graph-level optimizations, such as Identity Elimination and Reshape Fusion. TASO (Jia et al., 2019), on the other hand, *automatically* generates rule transformations for a given set of operators and verifies their correctness. Recently, companies (such as OctoML and MosaicML (oct, a; mos)) have emerged that provide such optimizations as a service, i.e., developing DL compilers, tools, and services that accelerate the performance of emerging DL models.

2.2 Data Privacy Solutions

Differential Privacy. Differential Privacy (DP) (Dwork, 2006; Dwork et al., 2014) provides privacy-preserving data analysis and learning, i.e., extracting useful statistics and information from a dataset, while protecting the identity and privacy of individuals in the dataset. Typical DP methods, e.g., Laplace mechanism (Dwork et al., 2006), inject controlled noise or randomness to the data or statistical analysis process to protect the privacy of individuals, while preserving the overall utility of the data. DP approaches (Song et al., 2013) have been proposed to protect *user data* that is used to train models. However, it is unclear how these approaches can protect confidentiality of DL model architecture as adding noise to it damages functional correctness of the model.

Homomorphic Encryption. Homomorphic Encryption (HE) (Gentry, 2009) allows computations to be performed on encrypted data without the need for decryption. Homomorphism-based transformations are structure-preserving transformations, i.e., HE-based schemes preserve the additive and multiplicative structures of the data. Therefore, even though HE-based methods encrypt the parameters (Gilad-Bachrach et al., 2016; Hesamifard et al., 2017) of DL models (e.g., weights of matrices, tensor values), they do not encrypt operators and topology of the model. During performance optimization, HE-based methods can ensure the privacy of model parameters, but they cannot protect the model's structure.

2.3 Model Stealing Attacks

Model stealing consists of creating a functionally-equivalent model and carrying out gradient-based adversarial attacks on the model. Prior works (Oh et al., 2019; Papernot et al., 2017; Tramèr et al., 2016) design algorithmic-level analysis to create functionally-equivalent models. The initial phase of important adversarial attacks involves the extraction of the network architecture (topology) of the DL model : given the topology (network architecture) of a DL model is known, stealing attacks can infer the values of model parameters, hyper-parameters, and even training data (Tramèr et al., 2016; Wang & Gong, 2018). There are also several model topology extraction attacks, including DeepSniffer (Hu et al., 2020) and ReverseCNN (Hua et al., 2018), which extract the model architecture by leveraging architectural hints.

3 OUR PROPOSAL: PROTEUS

3.1 Threat Model

PROTEUS aims to protect the model architecture from exposure to third parties, where the risk of model exposure is incurred when the model leaves the model owner and is:

1. intercepted by a third party in transit from the model owner to the optimizer through conventional wiretapping techniques, or
2. leaked by the optimization party to a malicious third-party. This includes the possibility where the optimizer party is also the party performing the attacks.

At the same time, implementations of the optimizing compiler remain as crucial intellectual properties of the optimization service. Release of the optimizing compiler to the model owner incurs the risk of software piracy.

3.2 Goals

Our goal in this work is to effectively decouple model innovation/development and performance optimization by enabling performance optimization *while protecting the confidentiality of the model architecture.* Ensuring privacy of the model architecture enables an independent party/service to optimize DL models. Specifically, we aim to achieve the following goals with our proposed mechanism:

1. **Model Confidentiality.** Given a mechanism that *obfuscates* the graph to prevent the retrieval of the original architecture: an adversary with access to both the obfuscating algorithm used and the obfuscated graphs produced by the confidentiality mechanism should not be feasibly able to retrieve the initially protected graph.
2. **Agnosticity and Independence of Performance Optimizations.** An effective confidentiality mechanism should not constrain the optimizations that can be performed on the protected graph. This additionally enables

the potential for preserving the confidentiality of the model optimizations and the compilers themselves.
3. **High Performance Efficiency.** Since the key goal of the optimizer is to improve the runtime performance of DL models, the confidentiality mechanism needs to preserve the performance benefits and ensure similar speedups as that achieved without the obfuscation mechanism.
4. **Low Compilation Overhead.** The confidentiality mechanism should not cause a significant compilation overhead to the optimizer party or make the optimization process more challenging. In this work, we focus our efforts on making the overhead of confidential optimization by machine learning compiler feasible.

3.3 Overview

In this work, we propose PROTEUS, the first obfuscation mechanism for DNN computational graphs for performance optimization. PROTEUS involves optimization in three independent steps. First, the *obfuscation* step where the original computation graph is "obfuscated" such that an adversary cannot feasibly identify the original model, thus providing confidentiality. Second, the *optimization* step is carried out flexibly and independently by the optimizer party on the obfuscated computational graph, providing performance speedups. Finally, the *de-obfuscation* step where the original model is retrieved by the model owner in its optimized form.

3.4 Obfuscation: Key Ideas

The key ideas behind PROTEUS are twofold: *sentinel generation* and *graph partitioning*. We detail each below.

3.4.1 Sentinel Generation.

We propose to obfuscate the original graph by hiding the protected DL graph among a set of *sentinel* graphs, i.e., artificially generated realistic graphs. The idea behind this approach is that an adversary will be unable to distinguish the real graph from the sentinel models. As the set of sentinel graphs grows larger, it is more challenging to identify the protected graph. This approach enables the optimizing party to optimize the obfuscated graph – by essentially optimizing all the sentinel graphs.

However, directly applying this approach has important limitations. First, the original protected graph is still directly exposed in its entirety. Second, this approach adds significant overhead to the optimizing party as all the sentinel graphs need to be optimized. Generating k sentinel graphs requires the both optimizer and the adversary to carry out $\mathcal{O}(k)$ work, either to perform optimizations or to attempt recovery of the protected model. To address these limitations, we first perform *graph partitioning* as described below.

3.4.2 Graph Partitioning.

We observe that most graph-level transformations performed by tensor compilers are local: graph-level substitutions performed by compilers operate on an operator and its neighboring nodes. We leverage this observation to partition the computational graph into smaller subgraphs. These subgraphs are independently optimized and then reassembled to generate the entire optimized graph. We evaluate the implications on performance speedup in Section 5.2 and demonstrate that this only incurs small losses in performance speedups from optimizations. With graph partitioning, the sentinel graphs are now generated for subgraphs rather than the entire graph. Thus, our solution can be broken down into two steps: (i) we partition the protected model into smaller subgraphs and (ii) hide the protected subgraphs within a set of k sentinel subgraphs.

The use of sentinel subgraphs makes the recovery of the original model significantly more challenging for the adversary because every subgraph has to be correctly classified and identified to reconstruct the original protected graph. Thus, we can use fewer sentinel graphs while still making it infeasible to recover the original model (we quantitatively demonstrate this in Section 4). At the same time, the model architecture in its entirety is never exposed to the optimizer.

3.4.3 Design Challenges

There are two major challenges in effectively obfuscating the protected graph as described above:

(i) Effective partitioning strategy: The number of subgraphs in the graph plays a key role in our ability to retain the optimization performance benefits. If the subgraphs are too small or if the partitioning eliminates optimization opportunities, the optimization schemes may be rendered less effective. At the same time, increasing the number of subgraphs obfuscates the graph more effectively and would thus require fewer sentinel graphs.

(ii) Generating sentinel graphs: It is crucial to generate sentinel subgraphs that are difficult to identify as *artificial*. Thus, they must be *realistic*, syntactically accurate, and resemble real world subgraphs in terms of operations, topology, etc. In other words, the sentinels cannot be arbitrarily generated. We next describe PROTEUS's detailed design where we aim to address the above challenges.

4 PROTEUS: DETAILED DESIGN

We describe the three key steps of PROTEUS in more detail. Section 4.1 describes the obfuscation mechanism of PROTEUS. Section 4.2 describes the optimization step needed to be performed by the optimizer party on the obfuscated subgraphs produced by PROTEUS. Section 4.3 describes

the de-obfuscation step performed by PROTEUS to retrieve the original model in its optimized form, returning it to the model owner.

4.1 Obfuscation

PROTEUS is an effective obfuscation mechanism that preserves confidentiality in DL models consisting of two major steps: (i) *graph partitioning* splits the protected model into smaller subgraphs, and (ii) *sentinel graph generation* hides the protected subgraphs within a set of k sentinel graphs. Specifically, given an arbitrary DL model, PROTEUS generates n smaller subgraphs and hides each subgraph within a set of k sentinel (artificially generated) subgraphs. This way PROTEUS hides the given protected DL model within a set of $\mathcal{O}((k+1)^n)$ possible computational graphs.

4.1.1 Graph Partitioning

PROTEUS splits the protected model into n subgraphs of similar sizes. Our key goal is to generate many subgraphs, such that the adversary cannot feasibly identify the original model, while at the same time not affecting the graph-level optimizations performed by the optimizer party. Note that with PROTEUS the optimizer applies graph-level transformations at each subgraph individually, and cannot perform optimizations that span across multiple subgraphs. However, given that we do not have any information in advance on which graph-level transformations will be performed by the optimizer, we employ a randomized graph partitioning algorithm to split the computational graph to n subgraphs.

We develop a graph partitioning algorithmic scheme inspired by the Karger-Stein (K-S) algorithm (Karger, 1993). K-S is a randomized algorithm that solves the minimum-cut problem on a graph. At each step, it selects a random edge from the graph and merges the two nodes connected by the selected edge into a single node. This step is called "edge contraction", and it is iteratively repeated until n nodes remain in the graph. When the algorithm terminates, each of the n remaining nodes represents a subgraph of the initial graph.

However, since K-S algorithm is randomized, the resulting n subgraphs may significantly vary in size. Creating subgraphs with high disparity in their sizes brings two key issues. First, very large subgraphs may cause confidentiality issues, since they can potentially reveal many useful information to the adversary related to the initial protected graph. Second, small subgraphs might cause performance issues, since optimizers cannot perform very efficient graph-level transformations on small graphs. Therefore, we enhance the K-S algorithm to create n subgraphs of almost equal sizes. Specifically, we perform multiple iterations of the K-S algorithm and at each iteration we evaluate the standard deviation of the sizes of the subgraphs created. Then, we keep the graph partitioning scheme that minimizes the disparity in the sizes of the subgraphs, to provide a more balanced and less informative graph partitioning.

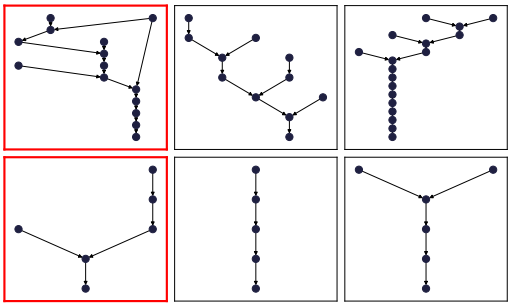


Figure 2. Examples of topologies sampled by PROTEUS (red: the original topologies)

4.1.2 Sentinel Graph Generation

In the sentinel graph generation step, PROTEUS hides the protected subgraphs within a set of k sentinel subgraphs. The generated sentinels must be syntactically correct to avoid immediate detection. Moreover, the sentinel graphs should resemble real world ones, so that the adversary cannot differentiate between real subgraphs (extracted from the initial protected model) and the sentinel subgraphs (artificially generated using PROTEUS).

In this step, we generate k sentinel graphs for each of the n subgraphs. We denote the n subgraphs extracted from the original protected model with $G_1, \ldots, G_n$. For each subgraph G_i, we generate k sentinel graphs denoted with $G_i^{(1)}, \ldots, G_i^{(k)}$. In other words, PROTEUS creates a bucket of $k+1$ subgraphs, one of them is the real subgraph G_i, and the remaining k subgraphs are sentinel (artificially generated) subgraphs. Therefore, the total search space for subgraphs has a size of approximately $\mathcal{O}((k+1)^n)$.

The k generated sentinel subgraphs $G_i^{(1)}, \ldots, G_i^{(k)}$ should resemble the original real subgraph G_i, such that the adversary should not be able to differentiate among them. We can categorize two types of metrics, which an adversary could use to identify the real subgraph G_i:

(a) **Topological Information.** A subgraph can be identified by the topological connections of its nodes. Specifically, computational DL graphs are acyclic graphs that describe the dataflow of DL operators, and their nodes typically have a small number of incoming edges.

(b) **Operator Information.** A subgraph can be identified by the patterns of computations performed at its nodes. In a computation DL graph, nodes represent the operation applied to a tensor. Therefore, real subgraphs follow similar computational patterns: e.g., a convolution operator is commonly followed by an activation.

Based on the two types of metrics listed above, PROTEUS generates sentinel graphs in two stages. First, in the *topology selection* stage, we generate the connections between the nodes in the computation graph. Second, we fill in operators into the graph in the *operator population* stage.

Topology Selection. We first generate the topologies of the sentinel graphs. The topology refers to the "shape" of the computation graph, i.e., how the nodes are connected. The topology selection process for sentinel subgraphs has two major steps: graph generation and sampling.

First, during the *graph generation step*, we generate a pool of realistic graph topologies with GraphRNN (You et al., 2018), an autoregressive graph generation model. However, a key limitation of this GraphRNN-based approach is that it will generate *undirected* graphs, while DL graphs are directed. To resolve this limitation, we transform the undirected graphs to *directed* graphs using the algorithm shown in Algorithm 3 in appendix section A.1, which traverses the undirected graph and assigns a direction to each edge, resulting in a DAG.

Next, we *sample* sentinel topologies from the previous step that are similar to the provided real subgraph. Specifically, we approximate the similarity measure by comparing various graph- and node-level statistics of the sentinel subgraphs with that observed of real-world subgraphs. The evaluated similarity metrics are the following: average degree, clustering coefficient, diameter, and graph size.

In algorithm 1, the SAMPLETOPOLOGIES function takes as input a protected subgraph G_i and generates a set of similar graph topologies that are statistically indistinguishable from G_i. This is achieved by ensuring that the graph statistics form a uniform distribution around the protected subgraph G_i, effectively adding random noise resulting in uncertainty. In other words, by observing the distribution of these subgraph statistics, each subgraph would have an equal chance of being the protected subgraph. Here, we extract a set of GraphRNN-generated graph topologies, denoted as $\mathcal{D}$, and control the range of the uniform distribution with β.

In algorithm 1, we establish bounds for uniform distribution in lines $2-8$, sample from this uniform distribution in lines $15-17$. Notably, if we sample graphs from $\mathcal{D}$ uniformly at random, the resulting distribution would follow that of $\mathcal{D}$ rather than being uniform. To tackle this, we employ *importance sampling* which applies a weight of $1/p$ to each sample where p describes the density of the topology under $\mathcal{D}$. This "corrects" the uneven densities under $\mathcal{D}$ and makes the resulting samples uniform.

Operator Population. After generating graph topologies, PROTEUS assigns a DL operator at each node in the generated graph. A DL operator describes the computations that will be applied to tensor given as input (e.g., matrix multiplication, convolution, activation). The DL operators assigned to nodes need to be (i) syntactically correct, i.e., they need to have correct configurations (e.g., the number of input and output arguments, the tensor dimensions) that are consistent with specifications of DL operators, and (ii) semantically consistent, i.e., the sequence (order) of operators within the generated graph needs to resemble realistic DL operator sequences.

Algorithm 1 Sampling similar topologies

1: **function** SAMPLETOPOLOGIES($G, \mathcal{D}, \beta$)
2: *Estimate the density p from the GraphRNN graphs*
3: $p(\mathbf{x}) \leftarrow$ ESTIMATEDENSITY($\mathcal{D}$)
4: *Sample the random position of G*
5: Sample $\alpha \sim \mathrm{Unif}([0, \beta]^n)$
6: *Compute the range of the uniform distribution*
7: $\ell \leftarrow p(\mathbf{x}) - \alpha$
8: $r \leftarrow \ell + \beta$
9: *Initialize the set of similar topologies*
10: $\mathrm{T} \leftarrow \emptyset$
11: **for** each graph $G \in \mathcal{D}$ **do**
12: *Transform G into a directed graph G′*
13: $G' \leftarrow$ INDUCEORIENTATION(G)
14: *Apply importance sampling*
15: $\mathbf{x} \leftarrow$ COMPUTEFEATURES(G')
16: $p' \leftarrow p(\mathbf{x})$
17: $\mathrm{T} \leftarrow \mathrm{T} \cup \begin{cases} \{G'\} & \text{w/ prob } \mathbb{1}(\mathbf{x} \in [\ell, r])/p' \\ \emptyset & \text{otherwise} \end{cases}$
18: **return** T

To ensure *syntactic correctness*, we can convert this problem into one of *constraint satisfaction*, i.e., given a set of constraints we need to find a solution that satisfies them. In our context, given a set of syntactic constraints for DL operators, we need to find an assignment of DL operators to the nodes of the graph, which satisfies the given syntactic constraints. We use Z3 (De Moura & Bjørner, 2008), an SMT solver to produce the assignment of the operators to the sentinel nodes.

Z3 takes as inputs (i) the graph topology, (ii) the list of operators, and (iii) their syntactic constraints, searches the solution space and returns a syntactically correct assignment of the operators to the nodes of the graph. For convolution and pooling, along with the operator we also need to specify the operator's kernel shape and number of input/output channels (for 2D convolutions).

To ensure *semantic consistency*, we need to quantify the *likelihood* of an operator assignment. If the likelihood is high, the operator assignment is more likely to be semantically similar to real-world DL operator assignments. To compute the likelihood, we calculate probabilities of operator

sequences generated by traversing the graph.

Algorithm 2 Generating opcode specifications

```
1:  function AssignOperators(G, pct, max_solns)
2:      Rules ← GenerateRuleSet(G)
3:      Solver ← Z3Solver
4:      S ← ∅
5:      Loop until no new solutions or maximum reached
6:      while satisfiable(S) and |Solns| ≤ max_solns do
7:          S ← GetSolution(Solver, Rules)
8:          Find logprob for solution S
9:          p ← logprob(S)
10:         Solns ← Solns ∪ {(S, p)}
11:         Prevent S from being returned again
12:         Rules ← Rules ∧ (¬S)
13:     return TopPercentile(Solns, pct)
```

Algorithm 2 describes the operator assignment process. The AssignOperators procedure takes as input the graph topology and returns a set of operator assignments. Specifically, we repeatedly query the solver to find syntactically valid operator assignments. For each solution, we compute its likelihood, and record it (line 8). We also exclude the solution from being returned in a subsequent iteration. We repeat this procedure until either the solver claims that there are no other solutions (i.e. unsatisfiable) or when the number of solutions exceeds a predefined limit (line 5). We return the operator assignments that are both *syntactically valid* and *semantically likely*.

Minor Modifications over Popular Models. To handle the scenarios where the original protected model is structurally very similar to commonly-used popular DL models, e.g., the protected model is a ResNet-like model, PROTEUS also generates graph topologies by modifying the topologies of popular DL models. Specifically, PROTEUS generates new DNN-like graph topologies by adding and/or removing nodes in the existing graph topologies of popular DL models. Then, PROTEUS fills DL operators to the newly added nodes using the process described above. In these cases, the opcodes of unperturbed nodes, except for the ones that are immediately adjacent to the perturbed nodes, are preserved.

4.2 Optimization

After obfuscation, the set of $n(k+1)$ obfuscated subgraphs (including the original and generated sentinels) are given to the optimizer party. The optimizer applies graph transformations to each of the provided subgraphs to minimize their runtime behavior and provide performance speedups.

The optimization step is carried *independently* by the optimizer party on the obfuscated subgraphs. Note that PROTEUS is largely *agnostic* to the optimizer, since it does not make *any* assumptions about the the optimizer's implementation other than that it preserves functional correctness. The optimizer will then return an optimized version of each obfuscated subgraph.

4.3 De-obfuscation

Upon receiving the optimized subgraphs from the optimizer, PROTEUS *reconstructs* the original model in its optimized form. It does so by extracting and concatenating the optimized "real" subgraphs.

Assuming that the optimization procedure performed by the optimizer is functionally correct, the optimized subgraphs are functionally equivalent to the original subgraphs (up to numerical differences). Thus, when we reassemble the model using the optimized subgraphs, we obtain a computation that is defined by the composition of these subgraphs. If the subgraphs are functionally correct, their composition would also be functionally correct. In our implementation, the obfuscation step generates the optimized graph graph by connecting the input and output edges of each adjacent subgraph. This can be done using information about subgraph connections tracked when the graph was partitioned. Finally, the de-obfuscated graph is returned to the model owner as the optimized version of his original model.

4.4 Parameterization

PROTEUS provides a number of tunable parameters to the model owner outlined in figure 8. These parameters allow for tradeoffs between (a) the complexity of recovery by an adversary, (b) the computational overhead for the optimizer, and (c) the quality of model optimizations (in particular, the slowdown compared to optimizing without partitioning).

Name	Description
n	Number of *graph partitions* generated from the protected graph
k	Number of *sentinel subgraphs* generated per protected subgraph

Figure 3. List of tunable parameters provided by PROTEUS

We tabulated the precise tradeoffs as a result of these parameters in Figure 9. These tunable parameters allow the model owner to tradeoff some potential speedups for additional and stronger obfuscation.

Manual Optimization. We note that while a $\mathcal{O}(k)$-fold increase is acceptable for an automatic optimizer or tensor compiler. However, if each subgraph requires manual engineering efforts to optimize, this overhead would be prohibitive, and PROTEUS would be ineffective for these cases. However, we observe that most manual efforts are spent on development and tuning of the machine learning compiler instead of manually applying optimizations on each subgraph.

5 EVALUATION

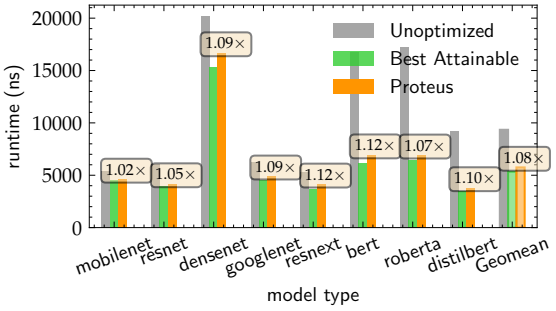
5.1 Methodology

Runtime Environment. PROTEUS uses ONNX (ONNX Contributors, 2023) for intermediate model representation, i.e., the initial DL model, its intermediate computational graph representation, and the optimized version of the given DL model are represented using the ONNX format.

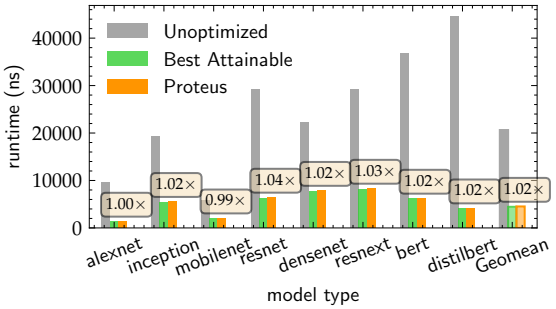
To demonstrate optimizer agnosticism, we use ONNXRuntime and Hidet for model optimizations and inference. ONNXRuntime is a performant optimizer and inference engine and Hidet (Ding et al., 2023) is a state of the art machine learning compiler.

We conduct our experiments on a `a2-highgpu-1g` instance on Google Cloud with 85GiB of RAM and an NVIDIA A100 GPU.

Models. We evaluate PROTEUS using representative widely-used convolutional neural networks (CNNs) that perform image classification as well as BERT-like language models (listed in figure 6). We obtained their implementations through the `torchvision` package (PyTorch, 2017) and HuggingFace Model Hub (Huggingface, 2023).



(a) Evaluation with ONNXRuntime



(b) Evaluation with Hidet

Figure 4. Execution time of DL models achieved by all evaluated schemes. The slowdown of Proteus over Best Attainable is labelled above each model.

5.2 Performance Efficiency

We first explore the ability of PROTEUS to retain the effectiveness of optimizations that generate performance speedups. ONNXRuntime performs a series of graph-level optimizations, from basic techniques such as constant folding to more complex operator fusion. Figure 4 depicts the resulting runtime for different DNNs, measured using the ONNXRuntime profiling tool across 500 iterations and computing the geometric mean of a single iteration. We evaluate three different mechanisms: (i) *Unoptimized*: without enabling *any* graph-level optimizations in the baseline graph, (ii) *Best Attainable*: enabling the best-performing graph-level optimizations available for the initial graph, and (iii) PROTEUS: using PROTEUS to protect confidentiality of the model by partitioning into subgraphs, and then enabling the best-performing graph-level optimizations available for each subgraph.

We observe that on average PROTEUS enables performance speedups close to the speedup of the optimizer without the confidentiality protection (within 8% of the maximum speedup on average and at most 12%). The small loss in performance speedups is due to the partitioning approach which reduces the effectiveness of some optimization techniques. For example, if a `conv` operator is following by an `add` operator in the original graph but the two are partitioned into different subgraphs, then fusion cannot be done between them. Performance loss due to graph partitioning is also dependent on the choice of the optimizer, i.e., the graph-level substitutions enabled by each particular optimizer. However, we argue that since various tensor compilers typically perform local graph transformations, we see similar behaviours with Hidet (Ding et al., 2023) and expect similar performance trends for other optimizers.

Loss of performance optimization opportunities is correlated with the average size of the subgraphs and this is further investigated in appendix A.3. We provide subgraph size as a tunable parameter as larger subgraph sizes would require more sentinels and thus, higher overheads for optimization. A subgraph size of $8 - 16$ offers a sweet spot where performance loss is less than 10% on average and incurs only small optimization overheads and is what we use in the remaining evaluations for PROTEUS.

The optimization overhead is also correlated with the number of sentinels generated per real partition. We include specifics of the tradeoff in appendix A.2.

5.3 Protection of Confidentiality

We demonstrate that PROTEUS generates sentinel subgraphs that are difficult to differentiate from real subgraphs by (i) evaluating graph statistics of sentinel subgraphs (Section 5.3.1), (ii) devising a learning-based adversarial attack on sentinel subgraphs (Section 5.3.2), and (iii) evaluating the feasibility of manual and expert intervention (Section 5.3.3).

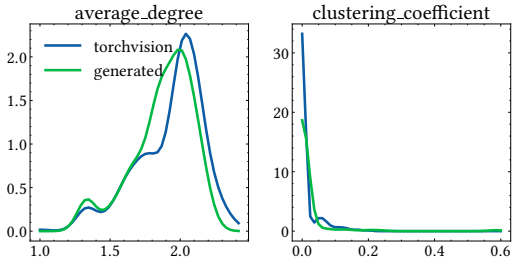


Figure 5. Comparing distributions of graph statistics between real and PROTEUS-generated subgraphs

5.3.1 *Statistical Quality of Sentinel Subgraphs*

We evaluate the quality of generated sentinel graphs with PROTEUS by comparing their distributions on various graph statistics with that of real graphs. Figure 5 compares the distributions between real and PROTEUS-generated graphs of their (a) average degree and (b) clustering coefficient. Appendix A.4 evaluates two additional metrics. We observe that sentinel subgraphs of PROTEUS have very similar distributions to that of real graphs in all evaluated metrics and would not distinguish sentinels from real graphs. We conclude that PROTEUS is robust against mechanisms that may leverage graph heuristics or employ statistical distributions of various metrics to identify the protected graph.

We note that even when using fewer metrics in algorithm 1, we still observe distributional similarities in other graph metrics. This is due to GraphRNN's ability to learn complex edge dependencies by observing real topologies, then imitating it to generate realistic sentinel topologies and leads to robustness across metrics.

5.3.2 *Learning-Based Adversarial Attack*

Given the mechanism proposed in this work, adversaries with the objective to recover the original protected graph fundamentally need to decide if a particular subgraph in the bucket is a PROTEUS-generated sentinel graph or part of the original protected graph.

In this section, we put ourselves into the position of one such adversary who attempts this task with a learning-based approach. Particularly, we evaluate the effectiveness of using a graph neural network (GNN) to perform such differentiation and investigate if the classifier helps to reduce the search space to one that compromises the model architecture.

Classifier Architecture. The classifier network accepts a graph as an input and outputs its confidence that the given graph is a sentinel. We depict the architecture of the classifier in Figure 7, and we elaborate further in Appendix A.5.

Datasets. During the training and evaluation of this learning-based adversary, we task the GNN-based adversary to differentiate real model subgraphs with the following:

1. *Random opcodes on* PROTEUS*-generated topologies.* We use the GraphRNN topologies with random operators.
2. *Sentinel graphs from* PROTEUS. We run the entire pipeline, using both GraphRNN and Z3.

In our experiments, we task PROTEUS with protecting one model at a time. To do so, we test the adversary on the protected model after training the classifier model on the remaining models.

Attack Mechanism. The GNN classifier outputs a confidence $y \in [0, 1]$ of the graph being sentinel. The adversary would fix a decision boundary γ such that a graph is eliminated as fake when $y \geq \gamma$. We make a pessimistic assumption that the adversary obtains γ. We note that the adversary must not erroneously eliminate any real subgraphs.

Metrics. We can measure the *sensitivity* (how many real subgraphs are correctly classified) and its *specificity* (how many fake subgraphs are correctly classified) of the adversary. Specifically, for each of the protected models we test and our choice of n, k and γ, we can measure the classifier's sensitivity (denoted α) and specificity (denoted β).

As established earlier, we must have $\alpha = 1$ such that all *real* subgraphs are correctly classified. In this case, for each of the n real subgraphs, approximately $(1 - \beta)k$ of its sentinel graphs are misidentified, resulting in a total number of $1 + (1 - \beta)k$ candidates for this particular subgraph. This results in a search space with size:

$$[1 + (1 - \beta)k]^n$$

within which the protected graph is hidden.

Search Space Reduction. In Figure 6, we compared the sizes of reduced search spaces with the learning-based approach. For each model type, we plotted the specificity (β) as well as the minimum decision threshold (γ) such that no real subgraphs are incorrectly identified. Using the cost computed above, we also tabulated the number of candidates in the reduced search space. The above is done for both the baseline (random opcode population) and PROTEUS.

We find that the resulting search space for differentiating PROTEUS-generated graphs from real graphs is orders-of-magnitude larger than that of the baseline (random opcodes). In many cases, a single candidate remains for the baseline, therefore making recovery trivial. Thus, such attacks are effective when the sentinels are not appropriately generated, as addressed by PROTEUS.

5.3.3 *User Survey on Sentinel Graphs*

To evaluate the realism of the graphs and the possibility of manually intervention by experts to identify sentinels, we conducted a survey amongst researchers in ML (details of

Protected Model	n	k	Random Opcodes Specificity	Random Opcodes $\min(\gamma)$	Random Opcodes Candidates	PROTEUS (Ours) Specificity	PROTEUS (Ours) $\min(\gamma)$	PROTEUS (Ours) Candidates
densenet	19	20	0.000	1.000	1.32×10^{25}	0.338	0.757	8.33×10^{21}
googlenet	11	20	0.990	0.356	7.00	0.346	0.899	4.30×10^{12}
inception	19	20	0.970	0.784	7.69×10^{3}	0.229	0.910	1.23×10^{23}
mnasnet	11	20	1.000	0.019	1.00	0.117	0.944	9.59×10^{13}
resnet	10	20	1.000	0.100	1.00	0.451	0.908	6.12×10^{10}
mobilenet	11	20	0.607	1.000	2.66×10^{10}	0.135	0.977	7.72×10^{13}
bert	16	20	0.996	0.474	3.00	0.910	0.653	1.37×10^{7}
roberta	16	20	0.990	0.634	2.00×10^{1}	0.862	0.799	1.54×10^{9}
xlm	25	20	1.000	0.300	1.00	0.906	0.816	2.99×10^{11}

Figure 6. Search space reduction for learning-based adversary

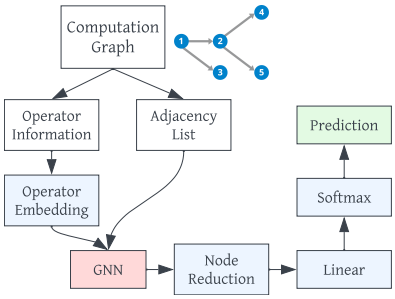


Figure 7. Architecture of GNN classifier

the methodology are in Appendix A.8). The survey contains 20 computational graphs and asks the participants to classify them as either real (i.e., taken from a real popular ML model) or fake (i.e., generated by PROTEUS). Out of 13 participants, the average accuracy is 52%, similar to that of random guesses. The survey can be accessed here and contains randomly chosen sample sentinel graphs that demonstrate the infeasibility of distinguishing sentinels simply by visual inspection from experts.

6 CASE STUDIES

We present two archetypal scenarios of graph-level model optimization: in the first case, the proposed model is unique and dissimilar to existing ones; and in the second, the proposed model largely resembles one that is widely used. In appendix A.7, we also evaluate the potential of GNN adversary presented in Section 5.3.2 in these case studies and present some visual examples of misclassified graphs.

6.1 Optimizing NAS Model

In the first scenario, the user optimizes a more "exotic" model that is very dissimilar to existing ones. For this purpose, we sample a model from NATSBench's (Dong et al., 2021) search space. Optimizing this model with ONNXRuntime results in a *slowdown* of 2.15×. This is potentially due to some optimizations that are typically beneficial but turned out harmful for this particular exotic model. When this model is optimized with PROTEUS, a similar outcome (slowdown of 2.164×) can be observed. Furthermore, the search space size is 1.18×10^{21} for the GNN adversary.

6.2 Optimizing a ResNet-like Model

In the second case study, the user attempts to optimize a model that resembles ResNet, SEResNet (Hu et al., 2019). The main difference lies in the addition of squeeze-excitation blocks. In the ideal case of optimizing the model directly, the maximum attainable speedup is 1.663×. With PROTEUS, we obtain a speedup of 1.494×, representing a $\approx$ 10% penalty. Furthermore, the search space size is 1.22×10^{87} for the GNN adversary.

7 CONCLUSION

In this paper, we introduce and motivate a novel and unexplored research question of confidential compiler graph-level optimization for deep learning models. Our proposed solution, PROTEUS, obfuscates the original model within realistic artificially generated computational graphs. PROTEUS is largely agnostic to the optimization approach and is thus generally applicable. The incurred computational overhead is trivial when using modern graph-level compilers. We demonstrate PROTEUS's robustness against learning-based, heuristic-based, and manual approaches, that are unable to distinguish the sentinel and protected subgraphs. A limitation of PROTEUS is that the additional sentinels make manual intervention more expensive (but still not infeasible when $k = 20$). The graph partitioning used in PROTEUS could also undermine optimization opportunities. Reducing the number of sentinels and partitioning more effectively are areas for future exploration. We hope that our work provides a first step for future work on confidential compiler optimization in deep learning.

8 ARTIFACT INSTRUCTIONS

Our artifact provides the code to reproduce two of our main results - those in figure 4 demonstrating the partition-optimize-reassemble routine preserves the optimization speedups and figure 6 showing the difficulty of graph recovery by a learning based adversary.

The PROTEUS source code and scripts are available at `github.com/proteus-mlsys24/mlsys24-artifact`.

For the full artifact appendix, please refer to appendix B.

REFERENCES

Mosaicml — brave new cloud. https://www.mosaicml.com/. Accessed: 2022-12-01.

Model optimization and automated deployment by the octoml platform. https://octoml.ai/, a. Accessed: 2022-12-01.

How 4x speedup on generative video model (film) created huge cost savings for wombo. OctoML, b. URL https://octoml.ai/blog/how-4x-speedup-on-generative-video-model-\film-created-huge-cost-savings-for-wombo/. Accessed on May 21, 2023.

Graph optimizations. ONNX Runtime website. URL https://onnxruntime.ai/docs/performance/model-optimizations/graph-optimizations.html. Accessed on May 21, 2023.

Issues · tensorflow/tensorflow · github. GitHub Issues. URL https://github.com/tensorflow/tensorflow/issues?q=is%3Aissue+is%3Aopen+XLA. Accessed on May 21, 2023.

Bengio, Y. et al. Learning deep architectures for ai. *Foundations and trends® in Machine Learning*, 2(1):1–127, 2009.

Chen, T., Moreau, T., Jiang, Z., Zheng, L., Yan, E., Cowan, M., Shen, H., Wang, L., Hu, Y., Ceze, L., Guestrin, C., and Krishnamurthy, A. Tvm: An automated end-to-end optimizing compiler for deep learning, 2018. URL https://arxiv.org/abs/1802.04799.

Cyphers, S., Bansal, A. K., Bhiwandiwalla, A., Bobba, J., Brookhart, M., Chakraborty, A., Constable, W., Convey, C., Cook, L., Kanawi, O., et al. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058*, 2018.

De Moura, L. and Bjørner, N. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pp. 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3540787992.

developers, O. R. Onnx runtime. https://onnxruntime.ai/, 2021. Version: x.y.z.

Ding, Y., Yu, C. H., Zheng, B., Liu, Y., Wang, Y., and Pekhimenko, G. Hidet: Task-mapping programming paradigm for deep learning tensor programs. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pp. 370–384, 2023.

Dong, X., Liu, L., Musial, K., and Gabrys, B. NATS-bench: Benchmarking NAS algorithms for architecture topology and size. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2021. doi: 10.1109/tpami.2021.3054824. URL https://doi.org/10.1109%2Ftpami.2021.3054824.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., and Houlsby, N. An image is worth 16x16 words: Transformers for image recognition at scale, 2021.

Dwork, C. Differential privacy. In *Automata, Languages and Programming: 33rd International Colloquium, ICALP 2006, Venice, Italy, July 10-14, 2006, Proceedings, Part II 33*, pp. 1–12. Springer, 2006.

Dwork, C., McSherry, F., Nissim, K., and Smith, A. Calibrating noise to sensitivity in private data analysis. In Halevi, S. and Rabin, T. (eds.), *Theory of Cryptography*, pp. 265–284, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg. ISBN 978-3-540-32732-5.

Dwork, C., Roth, A., et al. The algorithmic foundations of differential privacy. *Foundations and Trends® in Theoretical Computer Science*, 9(3–4):211–407, 2014.

Gentry, C. Fully homomorphic encryption using ideal lattices. In *Proceedings of the forty-first annual ACM symposium on Theory of computing*, pp. 169–178, 2009.

Gilad-Bachrach, R., Dowlin, N., Laine, K., Lauter, K., Naehrig, M., and Wernsing, J. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*, pp. 201–210. PMLR, 2016.

Goodfellow, I. J., Shlens, J., and Szegedy, C. Explaining and harnessing adversarial examples, 2014. URL https://arxiv.org/abs/1412.6572.

Goodfellow, I. J., Shlens, J., and Szegedy, C. Explaining and harnessing adversarial examples, 2015.

Hamilton, W. L., Ying, R., and Leskovec, J. Inductive representation learning on large graphs, 2018.

Hesamifard, E., Takabi, H., and Ghasemi, M. Cryptodl: Deep neural networks over encrypted data. *arXiv preprint arXiv:1711.05189*, 2017.

Hu, J., Shen, L., Albanie, S., Sun, G., and Wu, E. Squeeze-and-excitation networks, 2019.

Hu, X., Liang, L., Li, S., Deng, L., Zuo, P., Ji, Y., Xie, X., Ding, Y., Liu, C., Sherwood, T., et al. Deepsniffer: A dnn model extraction framework based on learning architec-

tural hints. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 385–399, 2020.

Hua, W., Zhang, Z., and Suh, G. E. Reverse engineering convolutional neural networks through side-channel information leaks. In *Proceedings of the 55th Annual Design Automation Conference*, pp. 1–6, 2018.

Huggingface. Hugging face – the ai community building the future. https://huggingface.co, 2023.

Insider, B. Chatgpt could cost over $700,000 per day to operate. microsoft is reportedly trying to make it cheaper. https://www.businessinsider.com/how-much-chatgpt-costs-openai-to-run-\estimate-report-2023-4, 2023.

Jia, Z., Padon, O., Thomas, J., Warszawski, T., Zaharia, M., and Aiken, A. Taso: Optimizing deep learning computation with automatic generation of graph substitutions. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, pp. 47–62, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450368735. doi: 10.1145/3341301.3359630. URL https://doi.org/10.1145/3341301.3359630.

Karger, D. R. Global min-cuts in rnc, and other ramifications of a simple min-out algorithm. In *ACM-SIAM Symposium on Discrete Algorithms*, 1993.

Labs, L. Demystifying gpt-3. https://lambdalabs.com/blog/demystifying-gpt-3, 2023. Blog post.

NVIDIA Corporation. TensorRT: Programmable Inference Accelerator, 2022. https://developer.nvidia.com/tensorrt.

Oh, S. J., Schiele, B., and Fritz, M. Towards reverse-engineering black-box neural networks. *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, pp. 121–144, 2019.

ONNX Contributors. ONNX: Open Neural Network Exchange. https://onnx.ai/, 2023. Accessed: May 21, 2023.

Papernot, N., McDaniel, P., Goodfellow, I., Jha, S., Celik, Z. B., and Swami, A. Practical black-box attacks against machine learning. In *Proceedings of the 2017 ACM on Asia conference on computer and communications security*, pp. 506–519, 2017.

PyTorch. torchvision: datasets, transforms and models specific to computer vision. https://pytorch.org/vision/, 2017.

Rombach, R., Blattmann, A., Lorenz, D., Esser, P., and Ommer, B. High-resolution image synthesis with latent diffusion models, 2021.

Rotem, N., Fix, J., Abdulrasool, S., Catron, G., Deng, S., Dzhabarov, R., Gibson, N., Hegeman, J., Lele, M., Levenstein, R., et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

Sabne, A. Xla: Compiling machine learning for peak performance. 2020.

Song, S., Chaudhuri, K., and Sarwate, A. D. Stochastic gradient descent with differentially private updates. In *2013 IEEE Global Conference on Signal and Information Processing*, pp. 245–248, 2013. doi: 10.1109/GlobalSIP.2013.6736861.

Tramèr, F., Zhang, F., Juels, A., Reiter, M. K., and Ristenpart, T. Stealing machine learning models via prediction apis. In *USENIX security symposium*, volume 16, pp. 601–618, 2016.

TVM. Adding an operator to relay — tvm 0.13.dev0 documentation. https://tvm.apache.org/docs/dev/how_to/relay_add_op.html, 2023.

Wang, B. and Gong, N. Z. Stealing hyperparameters in machine learning. In *2018 IEEE symposium on security and privacy (SP)*, pp. 36–52. IEEE, 2018.

Ye, Z., Lai, R., Shao, J., Chen, T., and Ceze, L. Sparsetir: Composable abstractions for sparse compilation in deep learning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pp. 660–678, 2023.

You, J., Ying, R., Ren, X., Hamilton, W. L., and Leskovec, J. Graphrnn: Generating realistic graphs with deep auto-regressive models, 2018.

Zheng, L., Jia, C., Sun, M., Wu, Z., Yu, C. H., Haj-Ali, A., Wang, Y., Yang, J., Zhuo, D., Sen, K., et al. Ansor: Generating high-performance tensor programs for deep learning. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*, pp. 863–879, 2020.

Zoph, B., Vasudevan, V., Shlens, J., and Le, Q. V. Learning transferable architectures for scalable image recognition, 2018.