DEFENDING MODEL EXTRACTION ATTACKS WITH PROBABILISTIC ISOLATION

By

Hunter Baxter

Dissertation

Submitted to the Faculty of the

Graduate School of Vanderbilt University

in partial fulfillment of the requirements

for the degree of

MASTER OF SCIENCE

in

Computer Science

December 16, 2023

Nashville, Tennessee

Approved:

Kevin Leach, Ph.D.

Yu Huang, Ph.D.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

Deep Neural Networks (DNNs) have seen a significant surge in adoption due to their ability to provide high value in domains such as text generation, generative graphics, image recognition, robotics, and speech recognition [12, 34]. An existing and flourishing business model is serving DNNs through a web client for various purposes in the framework as Machine Learning as a Service (MLaaS) [33]. In the MLaaS model, a customer sends some input to a service provider over a network, the input is processed, and results are sent back to the customer. The value of a DNN is broadly determined by the dataset and the architecture. Thus, a threat to the MLaaS business model is leaking information from the training set or the DNN architecture including operators, weights, size, and structure. The consequences of data leakage are concrete: privacy law violations, ethical violations regarding the usage of personal data, and a competitive data set edge lost. The consequences of model leakage, in theory, are quite severe as well. An actor with a leaked model no longer needs to pay the same R&D costs (labor, compute, data collection) to have an equivalent model, and it could have used that time elsewhere to secure an advantage. Since a potential actor had very little cost to obtain an equivalent performing model, they can sell services involving the model for less money. Finally, an unethical actor with a leaked model may have an easier time finding adversarial inputs to bypass or confuse an ML system, which can lead to brand damage and product failures - potentially lethal if in an autonomous vehicle [45].

Current defense mechanisms for Deep Neural Networks (DNNs) against model extraction attacks are all strong isolation techniques. The primary advantage of strong isolation lies in its robustness; it is impregnable to typical vulnerabilities, with the exception of certain side-channel attacks. However, this robustness does come with trade-offs. Strong isolation typically imposes significant overhead and limits the maximum size of the model, often necessitating that it fits within a trusted execution environment. This limitation can be a critical constraint, particularly in scenarios where larger, more complex models are required. In addition, the lack of defense against side-channel attacks is potentially problematic, given that an adversary targeting a datacenter would likely have considerable funds.

We introduce a novel defense method called *Jigsaw*. Jigsaw employs a moving target defense strategy that uses probabilistic pseudo-isolation to decrease the time value of information necessary to launch a model extraction attack. By trading the strong privacy guarantees of trusted execution environments or data-oblivious computation for probabilistic pseudo-isolation, we can secure larger models with fewer performance sacrifices. We show that with Jigsaw, there is an increase in the resources and time required to successfully complete an attack, which allows time to detect malicious behavior through well-studied anomaly detection systems. To bound the performance impact within the large random partitioning space of a DNN, we use a genetic algorithm to identify candidate partition plans that maintain reasonably strong performance while also providing a sufficient reduction in the attacker's window of opportunity. By balancing security with performance, Jigsaw aims to provide MLaaS providers with a practical security solution that scales to current large models and anticipates the needs of tomorrow's technologies.

CHAPTER 2

Background

2.1 Deep Neural Networks (DNNs)

Deep Neural Networks (DNNs) are a subset of statistical learning models that use a hierarchical organization of connected layers to approximate an unknown function $f(x) = y$ where x is the input, and y is the output. This organization can have arbitrary complexity, with popular forms including Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Long Short-Term Memory Networks (LSTMs), Autoencoders (AEs), and Deep Belief Networks (DBNs). The non-linearity and complexity of the network enable it to model complex functions that would otherwise be very difficult to model. The identity of a model can broadly be thought of as its architecture, hyper-parameters, and parameters. The architecture of a DNN can be described by the number of layers and the connection topology between each layer. A connection topology can be sequential or non-sequential, meaning that the output of layer i is the only input for layer $i+1$. Alternatively, more intricate topologies like skip connections, branching, and shared layers can be integrated [14, 12]. In the DNN architecture, each layer can be of different size and type, which include but aren't limited to: convolutional layers, recurrent layers, and fully connected layers [9]. The parameters of a DNN are configurable variables of the trained model whose values are set through training. Specifically, these are the weights and biases for every layer inside the DNN. The hyper-parameters of a DNN are values that control the training process but are not part of the data or architecture, like learning rate, regularization coefficients, batch size, etc.

A DNN broadly has two distinct phases: training and inference. Training a DNN is a compute-intensive process that takes a starter parameter model and a training dataset, and then uses back-propagation through the layers to set parameters that minimize a loss function. Typically, training is done offline with multiple GPUs and can often take hours or days. In inference, a model is deployed and used to make real-time predictions on novel inputs. Unlike training, this is often done on a CPU for smaller models to improve response time [11], but for large language models like GPT-3 [5], multiple GPUs may be needed [26].

2.2 Distributed Deep Learning

Several of the most recent advancements in the field of deep learning are direct consequences of increased model size or hardware advancements [42, 15]. The llama2 large language model has 70 billion parameters [49], and the GPT family of large language models by OpenAI requires over 325 GB of memory just to store the parameters of the layers [38, 5, 34]. The growth of these models seems only to continue [6, 7], so in the future, it should be expected that larger and larger infrastructure will be required. As a result of large models, and the belief that future models will be just as large, if not larger, significant engineering work has to be done from an infrastructure and algorithmic perspective to support the growth.

In current popular DNN frameworks [4, 1, 36], DNN computation is more often represented as a dataflow graph. The benefit of this approach is that it is a static scheduling problem, and all that is required is an execution engine that optimizes the schedule. When considering models that are too large to fit inside a conventional computer, like all recent large language models, typically a cluster of nodes with multiple hardware accelerators like Tensor Processing Units (TPU) or Graphics Processing Units (GPU) is utilized. In the case of the GPT-3 model [5], one needs at least five state-of-the-art 80 GB GPUs just to hold the weights and runtime information. In Megatron-LM, 512 GPUs in total were used to train large language models, which at the time, had state-of-the-art results [42]. In order to make use of these models, and efficiently train them, parallelization and/or distribution of work need to occur. The most common strategies are data parallelism and operator parallelism.

Data parallelism involves the partitioning of the training dataset among a set of worker nodes with identical models. Per iteration of training, each node computes a parameter update for its respective part of the dataset. After all nodes have computed an update, they synchronize their results before completing the update transaction. This allows all workers to have consistent model parameters throughout training. The downside of this approach is that although it allows training on larger datasets, alone, it does not enable using larger models.

Operator parallelism is the only strategy which allows for models that require more memory than what is feasible on a single device. Since a DNN is a dataflow graph, with each individual node being an operator, it can be partitioned across multiple devices. In some cases, it is done simplistically via splitting the layers [20], in others some manual effort is provided to the individual cluster [42], and in others the specific execution plan is algorithmically determined [56]. Operator parallelism, at a high level, can either be intra-operator or inter-operator. Intra-operator parallelism is when an operator has its respective tensor split across multiple nodes for computation as shown in figure 2.2. This is done by partitioning across arbitrary dimensions, assigning the results across different nodes, and computing the results on the different dimensions in parallel. Inter-operator parallelism is when an operator is computed in whole, but different operators in the computational graph are assigned to different nodes in the cluster. This is a more fine-grained approach than assigning layers to different nodes in the cluster and is shown in figure 2.1. Point-to-point communication between devices is used in inter-operator parallelism, which can be less than collective communication during intra-operator parallelism, but if used alone, it is possible that some nodes remain idle as there is a discrepancy in execution time between individual operators. So, one solution proposed by the Alpa project is to combine both inter-operator and intra-operator parallelism [56, 26]. Once a model is trained, with a large model, it is still necessary to employ some sort of model parallelism if the models are too large. If the model can fit into memory, a CPU is typically used as it can be faster [11], but most large language models cannot and need many GPUs even for inference [26].

2.3 Side Channel Attacks

A side channel attack can broadly be defined as using metadata from interactions with hidden data to infer the actual contents of that hidden data. In recent years, side channel attacks have emerged as a critical category of security vulnerabilities, due to their widespread application across almost

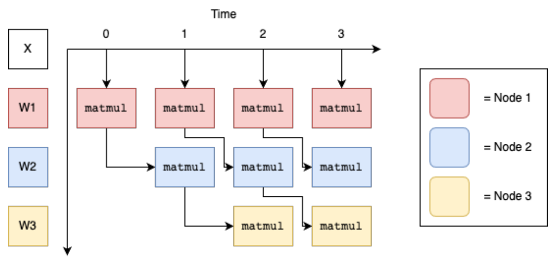

Figure 2.1: Pipeline Inter-Operator Parallelism

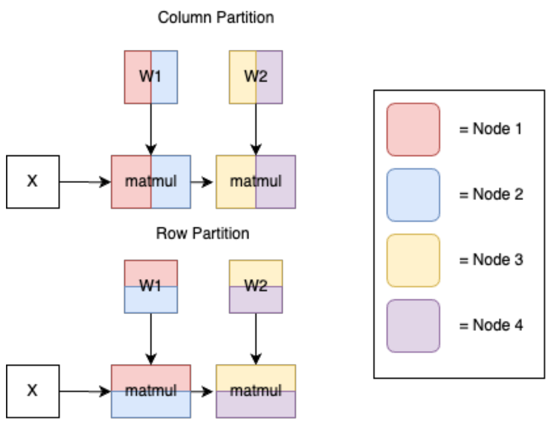

Figure 2.2: Intra-Operator Parallelism

all consumer and datacenter CPUs. These vulnerabilities are not easily addressed through traditional security measures and are often considered out of scope for many security solutions in the literature. The most common side channels rely on abusing timing behavior, exploiting transient execution, or inducing faults. While there is limited information regarding the sole usage of a side channel attack in a public attack, they have been used as building blocks for larger vulnerabilities in security research ([39, 47, 13]).

Microarchitectural Attack Type	Target/Form
Side-Channel	Cache (prime+probe [35], flush+reload [54], etc. [27, 10])
Transient Execution	Spectre [21], Meltdown [28], etc [40, 51].
Fault Attacks	DRAM Disturbance ([30]), DVFS ([48])

The most prevalent traditional timing attacks are prime & probe [35] and flush & reload [54]. Prime and probe attack typically runs in three phases: (1) filling the cache with an attackers own data so they can detect changes (2) victim running a process, which may change data the attacker set (3) the attacker probes the cache to see what data was replaced, using the time to access each piece of data as a way to infer which parts of the cache the victim's process used. Flush and reload operates in a similar manner. Instead of filling with own data, it flushes the cache, and then accesses the same data. If the data is now in the cache (due to the victim's access), it will load faster, indicating the victim's process accessed it.

2.4 DNN Attacks

In 2019, Gartner, a leading market research firm, published a report adising ”Application leaders must anticipate and prepare to mitigate potential risks of data corruption, model theft, and adversarial samples” [24]. The reason being that proprietary statistical learning models are becoming core to organizations value proposition, and there are considerable financial motivations for data corruption, model theft, and adversarial samples. Data poisoning can worsen a competitors product, model theft can save time and R&D costs, and adversarial samples could allow misuse of a product or damange the organizations reputation. The most common adversarial attacks are as follows:

- **Data poisoning:** [46] contaminating the training set for a model with the intent to increase errors or cause malicious behavior. One notable example of this attack is the Tay twitter bot, which users trained with hate speech, and as a result created a voice bot with undesirable behavior.
- **Model Extraction Attacks** [50]: an adversary has black-box access to a model f, and attempts to learn a model $\hat{f}$ which is a near equivalent for the desired effect.
- **Membership Inference** [43]: an adversary has black-box access to a model f. Determine if a record was a member of the training set of f.
- **Side Channel Data Identification** [17]: an adversary is able to identify the class of an input via performance metrics from GPU.
- **GPU Side Channel**: Using information leaked from the GPU during inference, an adversary is able to extract information about the input [17], or model parameters via runtime performance information [52, 31], PCIe [59, 47], Buses [13], or a physical side channel [2, 29].
- **CPU Side Channels**: Using traditional side channel attacks [35, 54, 21], model information about a DNN architecture is leaked [53].

2.5 Deep Neural Network Security Defenses

Defending Deep Neural Networks (DNNs) against side-channel and broad memory attacks generally involves two strategies: data oblivious techniques [17, 37] and robust memory isolation via trusted execution environments (TEEs) [57, 18, 19, 37, 17, 16, 58]. Implementing strong isolation typically requires hardware modifications to incorporate TEEs. This process is often expensive, especially in large-scale data centers already equipped with non-compatible hardware. Furthermore, TEEs usually impose limitations on memory capacity, leading to runtime inefficiencies when the required private memory exceeds the TEE's capacity. Additionally, most TEE-based methods do not adequately address side-channel vulnerabilities.

Three prominent DNN defense solutions are HIX [18], HETEE [58], and Chiron [16]. HIX focuses on secure isolation by altering the PCIe interface between a TEE-enabled CPU and a GPU. Despite offering GPU-level TEE features, HIX remains susceptible to conventional side-channel attacks and necessitates hardware modifications to the PCIe interface. Chiron aims to protect user data privacy in a Machine Learning as a Service (MLaaS) context by executing models in a sandboxed SGX enclave named Ryoan. This setup supports distributed training across multiple enclaves via a parameter server. However, Chiron's limitations include vulnerability to side-channel attacks, the need for hardware modification, and reliance on a trusted parameter server. HETEE employs a physically secured controller node to alter the PCIe fabric in a datacenter segment, isolating all computation accelerators without chip-level modifications. While this approach mitigates some hardware modification needs, it does not address side-channel threats and still incurs additional costs for the controller node.

Data-oblivious solutions, such as Telekine [17] and Visor [37], generally counteract side-channel attacks but often require hardware modifications and introduce significant runtime overhead, as all data must be treated uniformly. These solutions, designed mainly for image and video processing, are not universally applicable to all neural network models and face challenges with larger-scale implementations.

2.6 Moving Target Defense

Deep learning workloads exhibit a high level of predictability due to their bespoke structure, designed to achieve specific goals, with each layer entailing data dependencies with others. It's important to note that certain elements, such as learned parameters, are non-deterministic owing to random initialization and the use of stochastic algorithms.

The predictability of DNN workloads stems from their static nature, essentially being a composition of relatively straightforward processes like matrix multiplication and convolutions. Predictable workloads can be exploited by attacks that manipulate the system's state or extract private information [55]. An exploitation leveraging the timing of a workload is termed a schedule-based attack, which relies on a specific ordering in the execution of tasks by the victim and the attacker. This includes anterior, posterior, and pincer attacks. Anterior attacks precede the victim task, posterior attacks follow it, and pincer attacks sandwich the victim task, creating vulnerabilities at both ends [32].

One method to counter schedule-based attacks is schedule randomization. The aim here is to frequently permute the schedule, making it challenging for an attacker to infer a task's specific execution. Schedule randomization is a tactic of information concealment, opting for probabilistic pseudo-isolation over deterministic isolation [22]. Probabilistic pseudo-isolation is notably employed in address-space layout randomization (ASLR), safeguarding user and kernel space memory from memory-corruption attacks such as buffer overflows and code-reuse attacks [41, 3].

From an attacker's perspective, a static schedule can be breached given sufficient time or resources. However, with randomized scheduling, the temporal validity of any acquired state information is significantly reduced, posing a formidable challenge to such attacks.

2.7 Genetic Algorithm

Genetic algorithms are a subset of evolutionary algorithms inspired by the process of natural selection and can be used for optimization and search. They belong to a larger class of evolutionary algorithms, which includes differential evolution and evolutionary strategies. The basic principle involves starting with a randomly generated or guided population of potential solutions to the optimization problem. Solutions are then selected based on their fitness according to a specific fitness function, akin to how animals better suited for survival reproduce. After selecting the fittest solutions, pairs are crossed over to produce new solutions, mirroring biological reproduction. Along with this crossover, randomness is introduced to the population through mutation. This mutation maintains diversity and aims to prevent premature convergence to a solution that might be suboptimal or locally optimal. The process of pruning less fit solutions, followed by crossover and mutation, continues until a predetermined stopping point is reached, whether due to convergence or a time limit.

Genetic algorithms are particularly well-suited for combinatorial optimization problems, which involve finding the best solution from a finite or countably infinite set of possibilities. Many NP-Hard algorithmic problems, such as the traveling salesman or the knapsack problem, fall into this category.

The advantage of genetic algorithms lies in their global search approach, aiming not to settle for the first local optimum encountered. This isn't always guaranteed, but it's the primary objective. Another advantage is the potential to parallelize the evaluation of solution fitness within the population, a benefit given the expansive solution spaces of some problems. Furthermore, genetic algorithms can be applied to any problem where a fitness function is definable, without requiring prior knowledge. This contrasts with other methods, such as gradient descent in neural networks, which often need domain-specific knowledge [9].

CHAPTER 3

Approach

Jigsaw is a moving target defense library for sequential neural networks, offering probabilistic pseudo-isolation. In a given data center, we establish a set of available nodes as minion nodes, selecting one to act as the leader, termed the master node. Utilizing the specific cluster topology, composed of sets of these minion nodes, we apply Alpa [56] to devise an initial, approximately optimal partition of the layers. This initial partition is designed to minimize the end-to-end response time. Subsequently, we use a genetic algorithm scheme with PyGad [8] to slightly modify the solution, preserving the performance benefits while changing the memory locations of certain model parameters. This is achieved by penalizing schedules that place layers on the same node. Once a schedule is generated, each minion in the pool receives a message outlining their role in the computation, enabling them to load the required parameters into memory. When the master node receives an inference request, it forwards the tensor via zmq tcp sockets to the first minion for processing, which then continues to forward the tensors. Periodically, or upon request, the genetic algorithm scheme is revisited to generate a new solution, thus restarting the process. By placing each layer on a separate node in the cluster, we hypothesize that the weights become more challenging to steal. The rationale is that any effort to extract the weights for one model via an exploit must be replicated on every node containing a layer for full extraction.

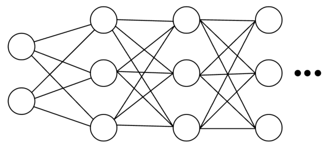
Figure 3.1: Sequential Network

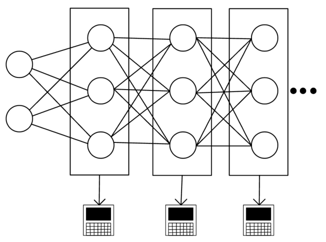
Figure 3.2: Partition Selection

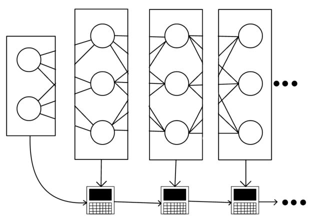
Figure 3.3: Tensor Forwarding

3.1 Threat Model

1. We assume the form of attack to obtain a neural network is a pincer attack via some form of side channel [21, 35, 54] that requires reading a cache recently [53], or a general anterior attack which relies on exploting some memory vulnerability which can read information directly from page tables.

2. The entire model can be dispersed to nodes without security issues via prior loading before inference, and it can be stored securely on disk. The only time model parameters can be stolen are when loaded into main memory.

3. We assume the following about the attacker: i) they cannot physically threaten the system via power analysis, fault injection, etc.; ii) they cannot manipulate the system's boot sequence or anticipate the output of the random number generator.

4. We suppose every single node is potentially untrusted in its entirety. An attacker, if compromised a node, would have full priviledges and hardware access. The implication being that an attacker will be able to inspect memory contents for a workload.

5. We assume that an attacker will have only compromised C nodes from a pool of nodes N.

3.2 Schedule Combinatorics

A deep learning model's inference process can be represented as a directed acyclic graph of computational units or tasks. When observed at a coarse level, this graph comprises sets of layers. We denote this ordered set of tasks as $\Gamma = \{\tau_1, \dots, \tau_k\}$.

In our cluster topology, we can schedule computations across a set of workers or nodes $N = \{n_1, \dots, n_m\}$. Each worker n_j is matched to an interval i_j in the set $I = \{i_1, \dots, i_m\}$. Each interval represents the tasks assigned to the corresponding node, effectively slicing the neural network for distributed computation.

We impose two conditions on I:

(i) $\bigcup_{i \in I} = \Gamma$ to ensure the completion of forward propagation, and

(ii) $\bigcap_{i \in I} = \emptyset$ to prevent duplication of tasks.

We want to find all possible scheduling configurations. To formalize the problem, we use the combinatorial stars-and-bars technique. If each task in Γ is represented by a star, then there are $k+1$ possible positions for bars or dividers. For a non-empty sets, we place $a-1$ separators, restricting us to k locations, to avoid creating an empty set. Thus, the number of combinations for a non-empty sets is $\binom{k}{a-1}$, and the number of permutations of these possibilities is $a!$.

Assuming a minimum of one computational node, we compute all possible configurations where computation is assigned to one or all m nodes. The total number of possible workloads is then $\sum_{a=1}^{m} \binom{m}{a} a! \binom{k-1}{a-1}$.

The Python function to generate a schedule looks like this:

```python
def create_schedule(
    minions: List[Any], m: int, k: int
) -> List[Tuple[Any, int, int]]:
    a = random.randint(1, m)
    chosen: List[Any] = random.sample(minions, a)
    bars: List[int] = sorted(random.sample(range(1, k), a - 1)) +
        [k]
    return [
        (chosen[i], bars[i - 1] if i > 0 else 0, bars[i] - 1) for
            i in range(a)
    ]
```

Given the algorithm, to compute the expected time it would take for an attacker to have all layers scheduled on a single infected node, we can do as follows:

(i) Given m minions, first amount of non-zero minions a is chosen from a uniform distribution

(ii) Then, we need to compute the likelihood a given node is selected inside $\binom{m}{a}$

(iii) Then, what amount of the permutations have a layer scheduled upon it.

To calculate the expected time required for a malicious node to schedule all layers, we:

(i) Draw a, the number of active nodes, from a uniform distribution,

(ii) Compute the likelihood of a given node being selected from $\binom{m}{a}$, and

(iii) Determine the fraction of permutations that have a layer assigned to it.

To solve (ii), we calculate the likelihood the malicious node is in the sample.

$$\frac{\binom{m-1}{a-1}}{\binom{m}{a}} = \frac{\frac{(m-1)!}{(a-1)!(m-1-a+1)!}}{\frac{m!}{a!(m-a)!}} = \frac{a!(m-a)!(m-1)!}{m!(a-1)!(m-a)!} = \frac{a}{m}$$

To solve (iii), we calculate the likelihood that minion is scheduled τ_i by fixing the set containing τ_i to the malicious minion, and only include the permutations where the condition is met:

$$\frac{(a-1)!}{a!} = \frac{1}{a}$$

So, for a given a, the likelihood a node has some layer τ_i scheduled upon it is $\frac{a}{m} \cdot \frac{1}{a} = \frac{1}{m}$

Given a uniform distribution, the odds of each a are equivalent. That is is $\sum_{a=1}^{m} \frac{1}{m} \cdot \frac{1}{m} = \frac{1}{m^2} \sum_{a=1}^{m} = \frac{1}{m}$ is the odds that a node is scheduled onto a layer for a given schedule.

So, the expected amount of schedules before a layer can be expected to occur on a node is m. Consider a naive attacker who waits until they are scheduled τ_i before checking if they are scheduled τ_{i+1}. This gives us km iterations, which sets an upper bound for the true expected value.

3.3 Generating New Solutions

Given a solution that is known to exhibit reasonable performance, as assessed by a tool such as Alpa [56], we periodically generate a new solution after a set interval of time. This new solution aims to minimize the time value of information, necessitating a significant difference in terms of parameter placement in memory. However, it is imperative to preserve reasonable performance; hence, simply assigning each layer to a distinct memory node may not suffice. Consequently, we frame this as a multi-objective optimization problem with two fitness functions: one targeting latency and the other focusing on the difference in the solution.

Consider a sequential neural network comprising L layers and N available nodes. To represent a solution, we construct an $N \times L$ binary matrix, denoted as S, which symbolizes our current strategy for partitioning DNN operators across the nodes. Our solution must adhere to the following constraints:

1. Operator Single Assignment

$$\sum_{i=1}^{N} S_{il} = 1 \text{ for all } l \in [1 \ldots L]$$

   The Operator Single Assignment Constraint ensures that a given operator is assigned to one and only one node within the cluster. This reduces the exposure of given operators weights by requiring that particular node to be compromised by an adversary.

2. Node Utilization

$$\sum_{i=1}^{N} f(S_i) = U$$

   The Node Utilization Constraint ensures that the network is partitioned to a particular amount of nodes. Here, U represents the amount of used nodes, and $f(S_i)$ is a boolean function which returns 1 if there is at least one layer scheduled for a node, and 0 otherwise.

3. Contiguous Layer Scheduling

```
    for row in potential_solution:
        zero_to_one = one_to_zero = 0
        for i in range(len(row) - 1):
            if row[i] == 0 and row[i + 1] == 1:
                zero_to_one += 1
            elif row[i] == 1 and row[i + 1] == 0:
                one_to_zero += 1

        if zero_to_one > 1 or one_to_zero > 1:
            return False
    return True
```

The Contiguous Layer Scheduling constraint mandates that the layers assigned to a node in a DNN must be scheduled contiguously. The provided Python code snippet is a validation check that ensures this requirement is met in a potential solution. It iterates through each row (representing a node's schedule) in a potential solution matrix. For each row, it counts transitions from 0 to 1 (indicating the start of a layer's schedule) and from 1 to 0 (indicating the end of a layer's schedule). If there are more than one such transitions in either direction for any row, the function returns False, signifying the solution violates the contiguous scheduling requirement. This constraint helps reduce the amount of tensor transfers over a network.

3.3.1 Latency Fitness

We have an $N \times N$ topology matrix L that describes the latency between nodes for a fixed size of information. This matrix approximates the latency of sending a given layer over the network by assuming that the speed of a fixed amount of megabytes scales linearly, an assumption that we beleive is sufficiently accurate. This matrix is computed in advance. Additionally, we possess an array T with count of the size ofmodel layers which contains the size of each respective layer output. We can use this array in conjunction with the latency matrix to approximate the transmission of the output tensor over the network.

We have an $m \times n$ matrix L of latency values, and we want to optimize an $m \times n$ matrix S consisting of ones or zeroes that represents our partitioning of the DNN layers across our nodes, where L_{ij} means the latency of node i with layer j. Our objective is to minimize the grand sum:

$$\text{Latency}(S) = \sum_{i=1}^{m} \sum_{j=1}^{n} (L_{ij} \cdot T_j) \odot S_{ij}$$

3.3.2 Difference Fitness

We have an $m \times n$ matrix C of one and zeroes representing our current partitioning of DNN layers across our nodes, and we want to optimize an $m \times n$ matrix N consisting of ones or zeroes that represents a potential next partition of the DNN layers across our nodes, where $N_{ij} = 1$ means that node i with compute layer j. Our objective is to minimize the grand sum of the hadamard product, where the operator is a binary and operation.

$$\text{Diff}(N) = \sum_{i=1}^{m} \sum_{j=1}^{n} C_{ij} \odot N_{ij}$$

3.3.3 Genetic Algorithm

Using the latency and difference fitness function, we set up a multi-objective optimization problem that we give a genetic algorithm engine such as PyGad to create a new solution. We set up our fitness function as:

$$\text{Fitness}(S) = \alpha \text{Latency}(S) - \beta \text{Diff}(S)$$

where α, β are parameters to tune the impact of latency and difference respectively.
We then use a genetic algorithm with the following parameters:

- 10,000 generations with 20 solutions per population, and 10 mating parents
- Parent Selection via Tournament Selection
- Two-Point Crossover Strategy

Generation and Population Parameters: The decision to set a cutoff at 10,000 generations with 20 solutions per population, and 10 mating parents, is a strategic balance between thoroughness and efficiency. In genetic algorithms, a larger number of generations allows for more extensive exploration of the solution space, increasing the likelihood of finding a globally optimal solution. However, this comes at the cost of increased computational time and resources. By limiting the process to 10,000 generations, the algorithm can still explore a significant portion of the solution space without becoming resource-intensive. The choice of 10 solutions per population and 5 mating parents is designed to maintain genetic diversity while avoiding the computational load of larger populations. We need this large amount of generations and quick iterations because we often have a population of invalid solutions, so we need to discard and reiterate many times.

Parent Selection via Tournament Algorithm: The use of a tournament algorithm for parent selection is aimed at consistently selecting high-quality solutions, which is crucial in environments with many invalid solutions. The tournament algorithm works by randomly selecting a subset of the population and then choosing the best solution from this subset to be a parent. Tournament selection is preferred over random selection as it tends to select individuals from the population who are valid, ensuring that offsprings are more likely to inherit desirable trains, which in our case are valid parts of a schedule. This also ensures that once we find partial valid partitions, the high-quality genes representing it will be present throughout the generations, which is helpful towards converging toward an optimal solution.

Two-Point Crossover Strategy: The choice of two-point crossover over single-point or more complex strategies balances the need for introducing genetic diversity with the risk of disrupting beneficial combinations of genes that lead to a valid solution. In single-point crossover, it is extremely unlikely that a crossover results in a valid solution from any non-identical pair of solutions. On the other hand, methods with multiple crossover points can excessively disrupt valid gene combinations, and led to even less valid solutions. Two-point crossover was the only strategy to occasionally allow for genetic mixing that led to valid solutions.

To ensure we are starting with a population that has the potential to reach a solution, a percentage of the initial population is duplicates of the minimal latency values, a percentage is random valid solutions, and the rest are random solutions. The reason for this approach is that if we start with a population of unviable solutions, it will take a long time to get to a viable solution, and experimentally, we have found that it greatly decreases convergence time to a new solution.

CHAPTER 4

Experiments

In this Thesis, we focus on answering the following research questions:

- How much isolation does Jigsaw provide?
- What value do new solutions generated by Jigsaw provide?
- What is the performance impact of partitioning a DNN?

For all plots, we use the parameters in table 4.1 unless directly comparing the effects of different minion pool sizes:

Parameter	**Value**
Nodes	5
Used Nodes	3
Generations	500
Number of Mating Parents	10
Solutions per Population	20
Crossover Type	Two Points
Parent Selection Type	Tournament
Mutation Type	Random
Proportion Current	0.2
Alpha	1
Beta	1

Table 4.1: Jigsaw Parameters Configuration

4.1 Research Question 1: Increased Isolation

In Section 3.2, our discussion centers on the theoretical implications of our partition scheme in the context of security. We propose that, according to our scheme, an attacker is likely to encounter a linear increase in the amount of work required for a successful attack. To empirically validate this theoretical proposition, we delve into an analysis of the unique memory pages occupied by the layers in the Jigsaw system. This approach allows us to directly assess the impact of our partition scheme on the security workload.

Our findings shown in figure 4.1 substantiate the theory: there is indeed a linear escalation in the number of page frame numbers, which which we argue causes an increased amount of resources by an attacker. This linear trend is a crucial indicator of the effectiveness of our partitioning strategy in enhancing the security of the system. Interestingly, we observe that the number of page frame numbers remains relatively constant over time. This consistency is due to the minion nodes reserving a set amount of space in memory, and then writing the scheduled layers into the region.

4.2 Research Question 2: Reshuffle Value

Jigsaw provides weak isolation for the parameters of a DNN. The isolation is achieved by rearranging the partition of the DNN after a configurable period, a technique that raises several questions about its effectiveness and implications. In Section 4.1, we demonstrated that if a solution adheres to our constraints and selection scheme, we can achieve enhanced security. However, this process is not without its challenges. A primary concern is the time required to find new solutions, which is a critical factor given that the network should be able to be repartitioned for a configurable period of time. Furthermore, it is essential for the new solutions to ensure isolation, maintaining the integrity and security of the DNN. Jigsaw's value lies in that it creates new solutions after a configurable period of time, which adds security benefits.

Our findings indicate that new solutions meeting our criteria can be identified in relatively few generations, as evidenced by the data shown in Figures 4.3, 4.4, and 4.5 for LeNet [25], AlexNet [23], and VGG [44] networks. This is encouraging as it suggests a level of efficiency in the reshuffling process. However, a notable observation is that with some networks, there is considerable overlap between generations of solutions. This can be seen in Table 4.3, where after the first repartition, all solutions are the same for AlexNet. This overlap is primarily due to our objective of minimizing the performance penalty, a crucial aspect for maintaining the practicality of Jigsaw over other defenses.

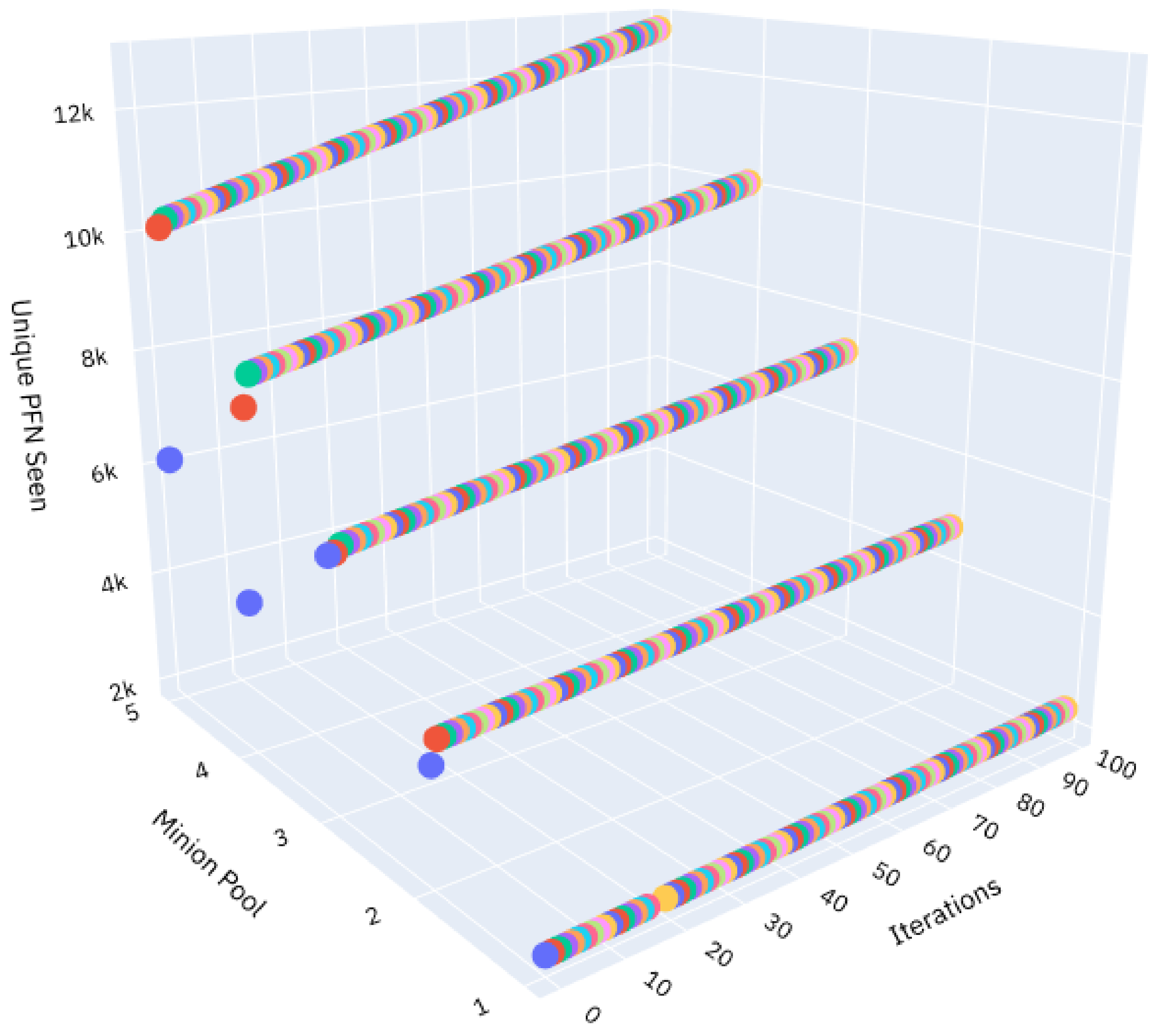

Figure 4.1: Unique Page Frame Numbers

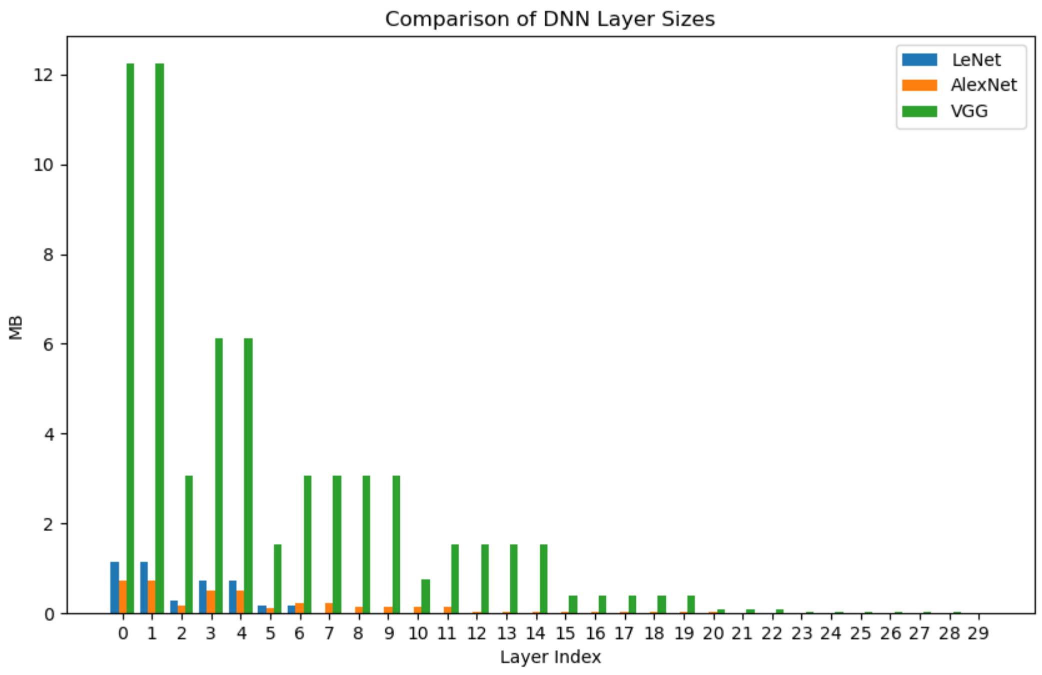

Figure 4.2: Size in MB of DNN Layers

To provide a more concrete understanding of these phenomena, we present sample generations for various DNN architectures, including VGG [44], AlexNet [23], and LeNet [25], as shown in Tables 4.2, 4.3, and 4.4. These examples offer insights into the reshuffling process's practical implications, demonstrating how the balance between security and performance is navigated in actual deployment scenarios.

4.3 Research Question 3: Performance Impact

The primary argument in favor of employing Jigsaw revolves around the comparative analysis of performance costs associated with different isolation strategies. Specifically, it is posited that the performance costs incurred due to weak isolation, as implemented in Jigsaw, are notably lower than those associated with strong isolation methods. In Figure 4.6, we present an analysis that shows the performance costs of Jigsaw are approximately a linear function in relation to the number of nodes utilized for computation. We argue a linear increase in computation time is acceptable, as

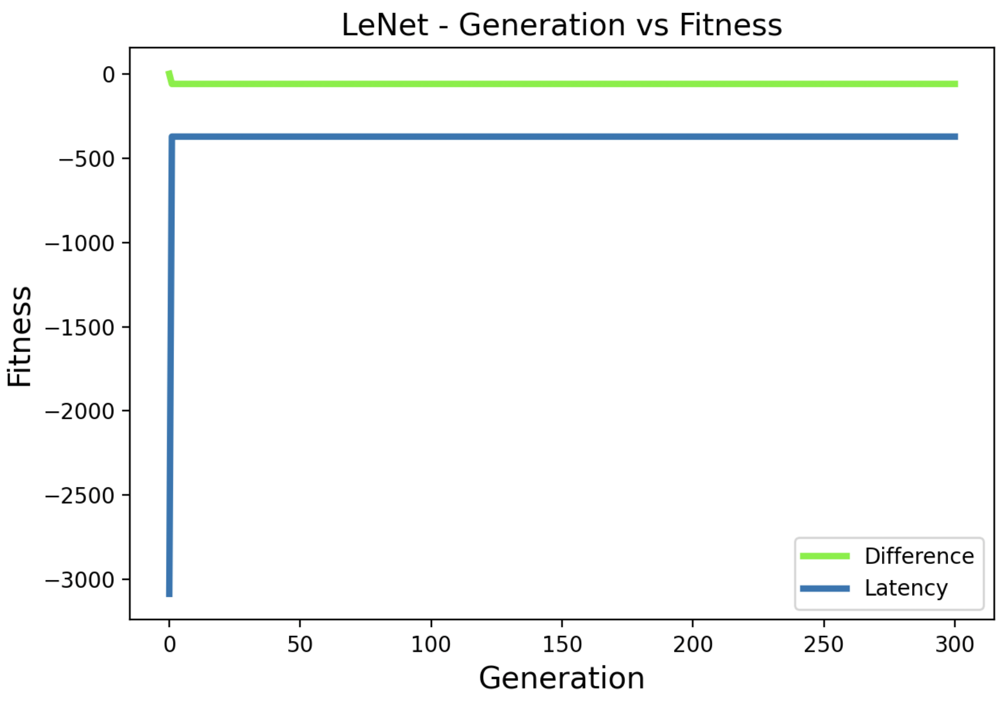

Figure 4.3: LeNet Generation vs Fitness

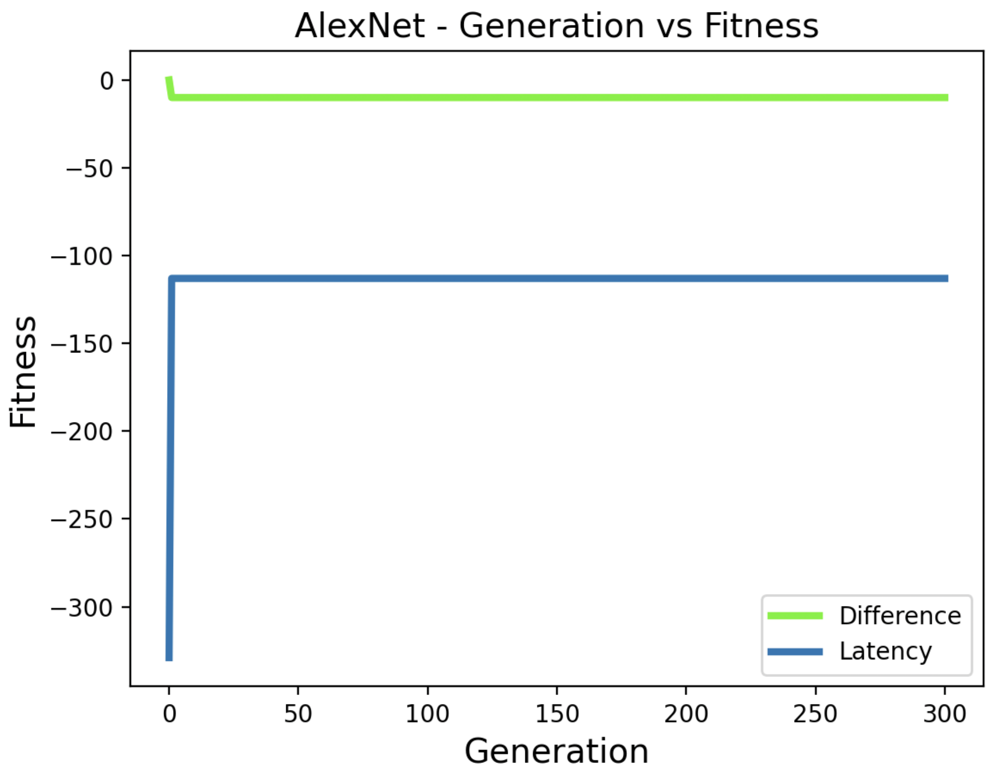

Figure 4.4: AlexNet Generation vs Fitness

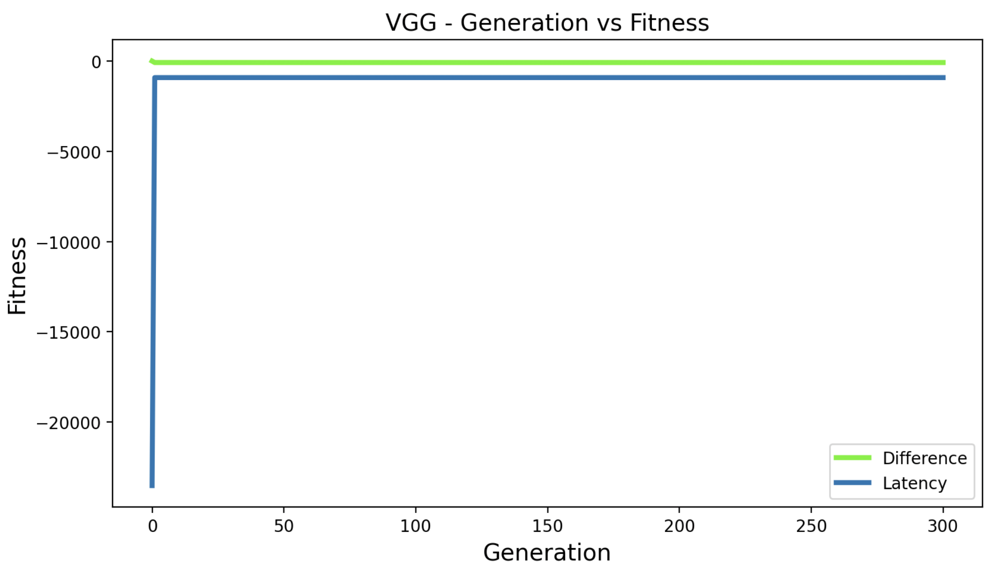

Figure 4.5: VGG Generation vs Fitness

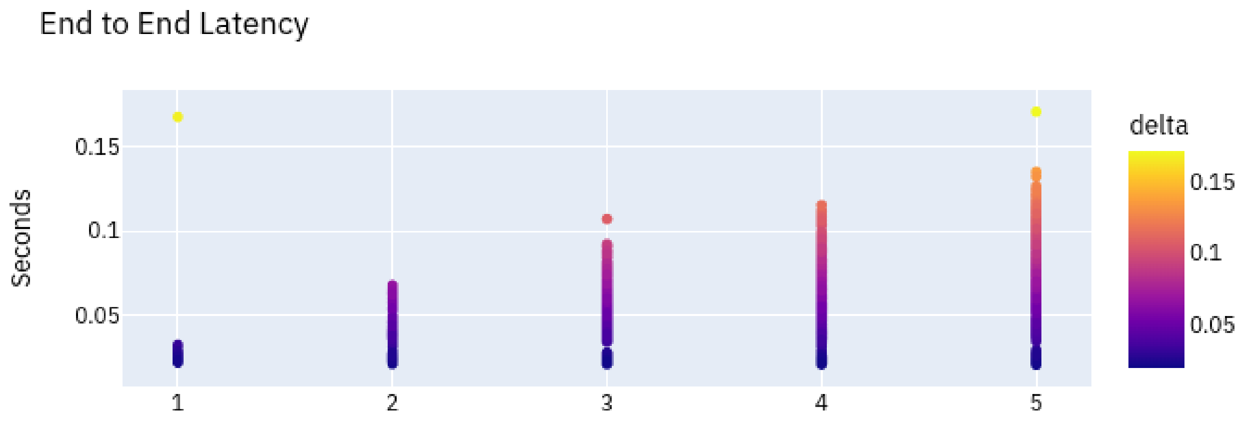

Figure 4.6: End to End Latency

with any form of inter operator parallelism, transfer of tensors over a network is inevitable.

Table 4.2: LeNet Schedule Change Example

Iteration and Previous Overlap	**Schedule**
LeNet(0)	1 1 1 1 0 0 0 0 0 0 0 0
	0 0 0 0 1 1 1 1 0 0 0 0
	0 0 0 0 0 0 0 0 1 1 1 1
	0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0
LeNet(1): 2 layer overlap	0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 1 1
	0 0 0 0 0 0 0 0 0 1 0 0
	1 1 1 1 1 1 1 1 1 0 0 0
LeNet(2): 0 layer overlap	1 1 1 1 1 1 1 1 1 0 0 0
	0 0 0 0 0 0 0 0 0 1 1 0
	0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 1
LeNet(3): 0 layer overlap	0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 1
	0 0 0 0 0 0 0 0 0 1 1 0
	1 1 1 1 1 1 1 1 1 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0
LeNet(4): 0 layer overlap	1 1 1 1 1 1 1 1 1 0 0 0
	0 0 0 0 0 0 0 0 0 1 1 0
	0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 1
LeNet(5): 0 layer overlap	0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 1
	0 0 0 0 0 0 0 0 0 1 1 0
	1 1 1 1 1 1 1 1 1 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0

Table 4.3: AlexNet Schedule Change Example

Iteration and Previous Overlap	**Schedule**

AlexNet(0)	1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

AlexNet(1): 5 layer overlap	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

AlexNet(2): 22 layer overlap	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

AlexNet(3): 22 layer overlap	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

AlexNet(4): 22 layer overlap	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

AlexNet(5): 22 layer overlap	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

Table 4.4: VGG Schedule Change Example

Iteration and Previous Overlap	**Schedule**

VGG(0)	1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

VGG(1): 0 layer overlap	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0 0 0

VGG(2): 2 layer overlap	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1

VGG(3): 0 layer overlap	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0

VGG(4): 0 layer overlap	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1

VGG(5): 0 layer overlap	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1 1 1
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0

CHAPTER 5

Conclusion

In this thesis, we introduce Jigsaw, a probabilistic isolation defense framework for Deep Neural Networks (DNNs). Jigsaw counters model extraction attacks by dynamically partitioning DNN computations across various nodes in a data center, providing a moving target defense against these attacks. We use the insight that partitioning DNN computation can improve latency, that modern large language models require operator parallelism to scale, and that side-channel attacks are already computationally demanding on a singular target, let alone multiple. Jigsaw fills a specific need that no other defense provides, shown in figure 5.1. Jigsaw is resistant to side channel attacks, does not require hardware modification, and is memory scalable. We provide some evidence that there is a linear increase in the amount of resources required by an attacker to steal a DNN model, and that we can generate new partitions which assign layers to different nodes in the data center, providing some isolation.

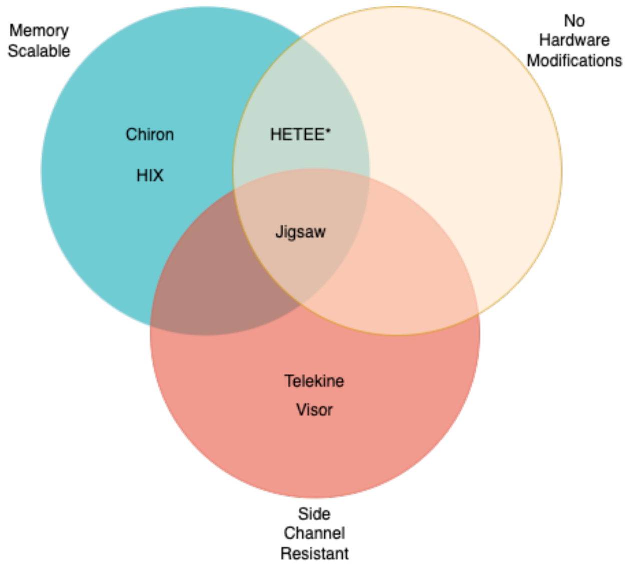

Figure 5.1: Characteristics of Current DNN Defenses

The Jigsaw framework has certain limitations that need to be acknowledged. Firstly, it can only handle sequential models, which does not capture the complexity of modern models. In addition, Jigsaw's effectiveness is partly contingent on tools like Alpa, which are used to generate strong initial partition solutions. In the event that a DNN is delicate in terms of latency, the solution space may be very narrow, which can lead to repetitive solutions, potentially undermining the desired isolation properties of Jigsaw. Also, given that genetic algorithms are heuristic in nature, there are no guarantees for solutions differing to a certain degree, nor with limited latency. Finally, we do not provide direct comparisons for performance versus strong isolation solutions.

Looking ahead, several areas for further development and research present themselves. Adapting Jigsaw to support inter-operator parallelism, not just per-layer parallelism, would enable it to handle modern DNN architectures. In addition, expanding the experimental analysis would provide evidence that Jigsaw is worthwhile compared to other strong isolation solutions. Finally, refining the genetic algorithm through tuned hyperparameters and incorporating parallel processing techniques could expedite the discovery of better solutions.

Bibliography

[1] Martin Abadi et al. "TensorFlow: a system for Large-Scale machine learning". In: *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 2016, pp. 265–283.

[2] Lejla Batina et al. "CSI NN: Reverse engineering of neural network architectures through electromagnetic side channel". In: *FIXME* (2019).

[3] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. "Address obfuscation: An efficient approach to combat a broad range of memory error exploits". In: *12th USENIX Security Symposium (USENIX Security 03)*. 2003.

[4] James Bradbury et al. "JAX: composable transformations of Python and NumPy programs". In: *FIXME* (2018).

[5] Tom Brown et al. "Language models are few-shot learners". In: *Advances in neural information processing systems* 33 (2020), pp. 1877–1901.

[6] Aakanksha Chowdhery et al. "Palm: Scaling language modeling with pathways". In: *arXiv preprint arXiv:2204.02311* (2022).

[7] William Fedus, Barret Zoph, and Noam Shazeer. "Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity". In: *The Journal of Machine Learning Research* 23.1 (2022), pp. 5232–5270.

[8] Ahmed Fawzy Gad. *PyGAD: An Intuitive Genetic Algorithm Python Library*. 2021. arXiv: 2106.06158 `[cs.NE]`.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. MIT press, 2016.

[10] Daniel Gruss et al. "Flush+Flush: a fast and stealthy cache attack". In: *Detection of Intrusions and Malware, and Vulnerability Assessment: 13th International Conference, DIMVA 2016, San Sebastián, Spain, July 7-8, 2016, Proceedings 13*. Springer. 2016, pp. 279–299.

[11] Kim Hazelwood et al. "Applied machine learning at facebook: A datacenter infrastructure perspective". In: *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2018, pp. 620–629.

[12] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[13] Xing Hu et al. "Deepsniffer: A dnn model extraction framework based on learning architectural hints". In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 385–399.

[14] Gao Huang et al. "Densely connected convolutional networks". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708.

[15] Yanping Huang et al. "Gpipe: Efficient training of giant neural networks using pipeline parallelism". In: *Advances in neural information processing systems* 32 (2019).

[16] Tyler Hunt et al. "Chiron: Privacy-preserving machine learning as a service". In: *arXiv preprint arXiv:1803.05961* (2018).

[17] Tyler Hunt et al. "Telekine: Secure Computing with Cloud GPUs." In: *NSDI*. 2020, pp. 817–833.

[18] Insu Jang et al. “Heterogeneous isolated execution for commodity gpus”. In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2019, pp. 455–468.

[19] Jianyu Jiang et al. “CRONUS: Fault-isolated, Secure and High-performance Heterogeneous Computing for Trusted Execution Environment”. In: *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2022, pp. 124–143.

[20] Yiping Kang et al. “Neurosurgeon: Collaborative intelligence between the cloud and mobile edge”. In: *ACM SIGARCH Computer Architecture News* 45.1 (2017), pp. 615–629.

[21] Paul Kocher et al. “Spectre Attacks: Exploiting Speculative Execution”. In: *Communications of the ACM* 63.7 (2020), pp. 93–101.

[22] Koen Koning et al. “No need to hide: Protecting safe regions on commodity hardware”. In: *Proceedings of the Twelfth European Conference on Computer Systems*. 2017, pp. 437–452.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012).

[24] Ram Shankar Siva Kumar et al. “Adversarial machine learning-industry perspectives”. In: *2020 IEEE Security and Privacy Workshops (SPW)*. IEEE. 2020, pp. 69–75.

[25] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *nature* 521.7553 (2015), pp. 436–444.

[26] Zhuohan Li et al. “{AlpaServe}: Statistical Multiplexing with Model Parallelism for Deep Learning Serving”. In: *17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23)*. 2023, pp. 663–679.

[27] Moritz Lipp et al. “{ARMageddon}: Cache attacks on mobile devices”. In: *25th USENIX Security Symposium (USENIX Security 16)*. 2016, pp. 549–564.

[28] Moritz Lipp et al. “Meltdown: Reading kernel memory from user space”. In: *Communications of the ACM* 63.6 (2020), pp. 46–56.

[29] Henrique Teles Maia et al. “Can one hear the shape of a neural network?: Snooping the GPU via Magnetic Side Channel”. In: *FIXME*. USENIX Security Symposium, 2022.

[30] Onur Mutlu and Jeremie S Kim. “Rowhammer: A retrospective”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.8 (2019), pp. 1555–1571.

[31] Hoda Naghibijouybari et al. “Rendered insecure: Gpu side channel attacks are practical”. In: *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*. 2018, pp. 2139–2153.

[32] Mitra Nasri et al. “On the pitfalls and vulnerabilities of schedule randomization against schedule-based attacks”. In: *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2019, pp. 103–116.

[33] OpenAI. *ChatGPT*. 2023. URL: https://chat.openai.com.

[34] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: 2303.08774 [cs.CL].

[35] Dag Arne Osvik, Adi Shamir, and Eran Tromer. “Cache attacks and countermeasures: the case of AES”. In: *Topics in Cryptology–CT-RSA 2006: The Cryptographers’ Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2005. Proceedings*. Springer. 2006, pp. 1–20.

[36] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *Advances in neural information processing systems* 32 (2019).

[37] Rishabh Poddar et al. “Visor: Privacy-preserving video analytics as a cloud service”. In: *Proceedings of the 29th USENIX Conference on Security Symposium*. 2020, pp. 1039–1056.

[38] Alec Radford et al. “Improving language understanding by generative pre-training”. In: *FIXME* (2018).

[39] Adnan Siraj Rakin et al. “Deepsteal: Advanced model extractions leveraging efficient weight stealing in memories”. In: *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2022, pp. 1157–1174.

[40] Michael Schwarz et al. “ZombieLoad: Cross-privilege-boundary data sampling”. In: *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 2019, pp. 753–768.

[41] Hovav Shacham et al. “On the effectiveness of address-space randomization”. In: *Proceedings of the 11th ACM conference on Computer and communications security*. 2004, pp. 298–307.

[42] Mohammad Shoeybi et al. “Megatron-lm: Training multi-billion parameter language models using model parallelism”. In: *arXiv preprint arXiv:1909.08053* (2019).

[43] Reza Shokri et al. “Membership inference attacks against machine learning models”. In: *2017 IEEE symposium on security and privacy (SP)*. IEEE. 2017, pp. 3–18.

[44] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[45] Zhichuang Sun et al. “Mind your weight (s): A large-scale study on insufficient machine learning model protection in mobile apps”. In: *30th USENIX Security Symposium (USENIX Security 21)*. 2021, pp. 1955–1972.

[46] Christian Szegedy et al. “Intriguing properties of neural networks”. In: *arXiv preprint arXiv:1312.6199* (2013).

[47] Mingtian Tan et al. “Invisible probe: Timing attacks with pcie congestion side-channel”. In: *2021 IEEE Symposium on Security and Privacy (SP)*. 2021, pp. 322–338.

[48] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. “{CLKSCREW}: Exposing the perils of {Security-Oblivious} energy management”. In: *26th USENIX Security Symposium (USENIX Security 17)*. 2017, pp. 1057–1074.

[49] Hugo Touvron et al. “Llama 2: Open Foundation and Fine-Tuned Chat Models”. In: *arXiv preprint arXiv:2307.09288* (2023).

[50] Florian Tramèr et al. “Stealing machine learning models via prediction {APIs}”. In: *25th USENIX security symposium (USENIX Security 16)*. 2016, pp. 601–618.

[51] Jo Van Bulck et al. “Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution”. In: *27th USENIX Security Symposium (USENIX Security 18)*. 2018, pp. 991–1008.

[52] Junyi Wei et al. “Leaky dnn: Stealing deep-learning model secret with gpu context-switching side-channel”. In: *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE. 2020, pp. 125–137.

[53] Mengjia Yan, Christopher Fletcher, and Josep Torrellas. “Cache telepathy: Leveraging shared resource attacks to learn DNN architectures”. In: *USENIX Security Symposium*. 2020.

[54] Yuval Yarom and Katrina Falkner. “FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack”. In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. 2014, pp. 719–732.

[55] Man-Ki Yoon et al. “Taskshuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems”. In: *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2016, pp. 1–12.

[56] Lianmin Zheng et al. “Alpa: Automating inter-and {Intra-Operator} parallelism for distributed deep learning”. In: *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 2022, pp. 559–578.

[57] Jianping Zhu, Rui Hou, and Dan Meng. “TACC: a secure accelerator enclave for AI workloads”. In: *Proceedings of the 15th ACM International Conference on Systems and Storage*. 2022, pp. 58–71.

[58] Jianping Zhu et al. “Enabling rack-scale confidential computing using heterogeneous trusted execution environment”. In: *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2020, pp. 1450–1465.

[59] Yuankun Zhu et al. “Hermes Attack: Steal DNN Models with Lossless Inference Accuracy.” In: *USENIX Security Symposium*. 2021, pp. 1973–1988.