Reverse Engineering Deep Neural Networks

Ding Dao Xian
School of Electrical and Electronic Engineering

Assoc Prof Chang Chip Hong
School of Electrical and Electronic Engineering

Abstract **–** With the recent advances in Deep Neural Networks (DNN), the applications of DNNs are becoming wide-spread with a variety of applications, but each model is specialized to their respective problems. Due to the heavy investment on tremendous computing power, human expertise and labelled training data, well-trained DNN models are valuable assets. However, recent reported intellectual properties (IP)-theft attacks can compromise the confidentiality of the DNN models by utilizing side-channel information leakage. In this paper, a technique of reverse engineering DNNs is explored in a cloud setting where the DNN is deployed on a CPU. First, the equation-solving attacks are used to infer an approximated model with weights similar to the original, thus allowing the adversary to reverse engineer the DNN. For those complex models on large scale problems, Cache Telepathy mechanism is investigated to reduce the architectural and hyperparameter search space of network structures. Once the architecture is known, similar equation-solving approach can be adopted to extract the DNNs. Our results show that the API-based extraction method can provide a substitutional model with functionality similar to the original. Reverse engineering results on different model architectures are also presented.

Keywords – Reverse Engineering, Deep Neural Networks, Side Channel Attacks.

1. INTRODUCTION

In recent years, the use of Deep Neural Networks (DNNs) have been employed in many diverse areas such as facial recognition, autonomous driving and medical practices, with recent research allowing DNNs to obtain high accuracy and flexibility in its applications, including making decision for security applications and malware detection as well as various military applications. [1]. However, owning to its flexibility, this results in DNN being heavily dependent on its architecture and hyper-parameters, as well as the training dataset used to generate the model. Significant time, and effort including collection of datasets and performance improvements have been invested into the model development by companies, which make well-trained DNNs valuable assets and intellectual properties (IP) for companies [3].

Unfortunately, due to their high commercial value, these models make them targets to attack for reverse engineering. Yet, due to the multitude of hyper-parameters and architectures, the search-space of DNNs are extremely large. Recent research has proposed using side-channel attacks where physical or remote access to the device running the DNN is assumed, allowing the adversary to infer the DNN attributes. The side-channel attack we have chosen: FLUSH+RELOAD exploits the shared last-level cache (L3 cache) between CPU cores. [5]

Additionally, the recent Machine Learning as a Service (MLaaS) business model [7] allows companies to commercialize their DNNs without revealing any further information on the DNN except the input and output. This mechanism can keep the DNN as a black box. In MLaaS, clients (trusted users) submit data to the MLaaS service providers to design, train and host the DNNs, thus allowing the remote untrusted users to adopt a pay-per-query business model. Through this framework, an additional source of exploit can be taken advantage of when reverse engineering the victim's model [2].

Yan et al proposed a method of reverse engineering DNN architectural hyperparameters by use of cache based side channels, and have proposed a possible methodology using this to reverse engineer a substitutional model [2]. Furthermore, Tramèr et al proposed the Equation Solving Attack, whereby the solutions to a Multilayer Perceptron can be reverse engineered through a set of solutions [4].

Our paper takes advantage of both these methods and thus propose a more "complete" methodology under a cloud setting where the DNN is deployed on CPUs. A 3-phase solution is used as the methodology whereby we take advantage of the Equation Solving Attack to solve for an initial substitutional model for basic problems, with more advance problems using the Cache Telepathy methodology alongside a similar equation-solving approach to extract the DNN weights.

We make the following contributions:

(i) Propose an attack to reverse engineer (i.e.: obtain a substitutional model) DNNs in a cloud setting.

(ii) Demonstrate the attack through several scenarios by implementing it on a state-of-the-art platform.

(iii) Compiled observations that would assist in speeding up the methodology, as well as scenarios where the methodology would fail.

2. BACKGROUND

2.1 EQUATION SOLVING ATTACK

The equation solving attack was proposed as a method of attacking black-box DNNs. For this report, 2 types of DNNs are explored: Multilayer Perceptron (MLP) and Convolutional Neural Networks (CNN).

2.1.1 Multilayer Perceptrons

Assuming the substitutional multilayer perceptron that first uses a kernel transforming the input space to the kernel space, then a SoftMax regression at the kernel space, the equation solving attack can be used to obtain an approximation to a given black-box model [4].

The concept of the equation solving attack is such that the adversary treats their model's ith layer (f_i) with the unknown parameters $\boldsymbol{W_i}$ and $\boldsymbol{\beta_i}$ as unknowns. Then by querying the victim's model, the adversary can obtain a set of solutions and thus "solve" for the unknown parameters.

It has been shown that this process can solve linear systems such as Binary Logistic Regression, to get an exact solution, thus an "Equation Solving Attack", but for models where nonlinear systems are solved such as Multiclass Logistic Regression (MLR) models, and MLPs, a loss function (usually strongly convex) is considered for minimization and thus, solves for a global minimum. As MLPs solve for non-linear systems, it is unknown the number of samples necessary to find a solution. Additionally, as the loss functions for MLPs during the equation solving is usually a categorical cross entropy loss function [8], or negative log likelihood [9] which are both not strong convex, the stochastic gradient descent algorithm is chosen when solving for the local minimum. Thus, the equation solving attack can be analogous to training an MLP with a dataset (albeit with "misclassified" data due to querying) and optimized by stochastic gradient descent.

Despite the deceptively simple concept of training a neural net by querying its data, Tramèr et al has shown its high effectiveness in obtaining a model with similar weights. However, although not written explicitly, the data obtained by Tramèr et al (i.e.: the similarity in weights between the adversarial and substitutional models) are made by assuming that the architectures of both models are the same (as shown by the "extract" function in the published code [9]). For our methodology, we assume that this is not necessarily the case.

2.1.2 Convolutional Neural Networks

Although it is possible to apply the equation solving attack on CNNs, the amount of computation and queries required to solve for the weight parameters induces overhead in terms of cost [8]. Thus, for our methodology, solving for CNN weights through this method is not considered or recommended.

2.2 FLUSH+RELOAD

Our methodology adopts the Flush+Reload technique applied on CPUs [5]. The attack works under the assumption that a "spy" process (by the adversary) can be run on the host machine, allowing the adversary to monitor shared data of the victim process. In the attack, the adversary performs a "cflush" operation to evict the L3 cache line containing the victim's data, thus pushing it out of the cache. Since the data is out, the victim will reload its content and the adversary can then measure the reload time (latency), with a quick reload time indicating a "hit" (i.e.: data/process was used) and a long reload time indicating a "miss".

2.3 CACHE TELEPATHY

Cache Telepathy [2] is an attack that infers DNN architectures and hyper-parameters from cache-based side-channel attacks by monitoring CPU optimized BLAS libraries. It takes advantage of the reliance of DNNs on Generalized Matrix Multiplication (GEMM). By monitoring specific functions in the BLAS libraries, the adversary can reduce the search space for the hyperparameters dramatically.

2.3.1 GEMM and BLAS libraries

For our experiment, we use the OpenBLAS library (which runs on Goto's algorithm [19]). During DNN inferences, GEMM: $C_{i+1} = \alpha A_i \cdot B_i + \beta C_i$ is computed where A_i is an $m\ x\ k$ matrix of the ith layer, B_i is the $k\ x\ n$ matrix of the ith layer, C_i is a $m\ x\ n$ matrix of the ith layer, with α and β being scalar quantities. Additionally, note that matrix multiplication runs on Goto's algorithm which consists of 3 nested loops: $Loop\ 1$ (outermost), $Loop\ 2$, $Loop\ 3$ (innermost). The GEMM algorithm in

OpenBLAS and a visual diagram of Goto's algorithm are shown in the Appendix (Fig. 16, 17).

By knowing: (i) Blocked matrix sizes implemented in Goto's algorithm (labelled P, Q, R in this paper). (ii) The number of times each loop of the algorithm structure is being iterated (labelled $iter_1$, $iter_2$, $iter_3$) with $iter_1$ being the outermost and $iter_3$ being the innermost by probing the addresses of the $itcopy$ and $oncopy$ functions (obtained via binary analysis). The relationship between the iterations and blocked matrix sizes is also given in the Appendix (Fig. 13). (iii) The measured times: t_{normal}, t_{small} which are the normal and last execution time of $Loop\ 3$ respectively, and t'_{normal}, t'_{small} which are the execution time of 2 iterations of $Loop\ 2$ (still within $Loop\ 1$) and the last iteration of $Loop\ 2$ respectively, we can use the formulas:

$$m = (iter_3 - 2)\,x\,P + 2\,x\,\frac{t_{small}}{t_{normal}}\,x\,P$$

$$n = (iter_1 - 2)\,x\,R + 2\,x\,iter_4\,x\,R$$

$$k = (iter_2 - 2)\,x\,Q + 2\,x\,\frac{t'_{small}}{t'_{normal}}\,x\,Q$$

to derive an approximation to the matrix sizes of each layer.

Note that: $iter_1$ can be counted by counting the number of "large repeated patterns of $itcopy$ and $oncopy$ functions", $iter_2$ can be counted by counting the number of "1 $itcopy$ followed by $oncopies$, then $itcopies$", $iter_3$ can be obtained by counting the number of "$itcopy$ numbers after the 1st $itcopy$ and the consecutive $oncopy$ calls in the Loop3", and lastly $iter_4$ can be obtained by counting the number of "$oncopy$ numbers after the 1st $itcopy$ call in the Loop3" (i.e: the number of $oncopy$ found in $iter_2$).

2.3.2 Matrix Hyper-parameters

For both fully-connected networks (FC network) and convolutional networks (Conv network), the hyper-parameters and the values needed to calculate it are summarized as follow:

Structure	Hyper-Parameter	Value
FC network	# of layers	# of matrix muls
FC layer$_i$	N_i: # of neurons	$n\_row(\theta_i)$
Conv network	# of Conv layers	# of matrix muls / B
Conv layer$_i$	D_{i+1}: # of filters	$n\_row(F'_i)$
	R_i: filter width and height[1]	$\sqrt{\frac{n\_row(in'_i)}{n\_row(out'_{i-1})}}$
	P_i: padding	difference between: $n\_col(out'_{i-1}), n\_col(in'_i)$
Pool$_i$ or Stride$_{i+1}$	pool or stride width and height	$\approx \sqrt{\frac{n\_col(out'_i)}{n\_col(in'_{i+1})}}$

Fig 1: Table mapping between Hyper-Parameters and the values to obtain. [2]

where: in_i is the input matrix of the ith layer, out_i is the output matrix of the ith layer, θ_i is the weight matrix of the ith layer, D_i and F_i are the depth (number of filters) and filter of the a convolutional layer respectively, $n\_row()$ and $n\_col()$ being functions that return the number of rows and columns of a matrix respectively, and $\boldsymbol{B}$ being the input batch size during the convolution process.

Note that:
(i) $\boldsymbol{B}$ is determined by counting the number of consecutive matrix multiplications with the same computational pattern
(ii) # of matrix_muls is determined by judgement of the "gap" between side-channel calls. A "large gap" signifies that a separate layer has been called.
(iii) A table of the relationship between the matrix dimensions and DNN layers is provided in the Appendix (Fig. 14, 15)
(iv) Filter width (W_i) and filter height (R_i) are assumed to be the same.

To obtain the parameters, the side channel information of the matrix dimensions mapped out in 2.3.1 can be used to determine the $n\_row()$ and $n\_col()$ of the matrixes, thus achieving the Hyper-Parameter values.

2.3.3 Layer Connections

Cache Telepathy maps out 2 types of inter-layer connections: Sequential and Non-Sequential, and their methods of identification. For a Sequential if:

(i) Since the filter widths and heights of layers i and $i+1$ are connected, and $R_i = k \in \mathbb{Z}$, following the formula from 2.3.2, we can get:

$$\exists k \in \mathbb{Z}\ s.t\ n\_row(in'_{i+1}) = k^2\,x\,n\_row(out'_i)$$

(ii) Similarly, since the pool size of layers i and $i+1$ are connected and $P_i = k \in \mathbb{Z}$, following the formula from 2.3.2, we can get:

$$\exists k \in \mathbb{Z}\ s.t\ n\_col(out'_i) = k^2\,x\,n\_col(in'_{i+1})$$

If any of these 2 conditions are not fulfilled, it is then taken that the layer is Non-Sequential.

Additionally, for Non-Sequential layers, we define the "shortcut" as the extra connection from layer j to layer $i+1$, "source" layer as layer j and "sink" layer as layer $i+1$. To identify the "shortcut", we need to find extra latency between the GEMM calls due to the merge operation, thus finding the "shortcut" in between layers i and layer j. Then from this, since there is a connection from the "source" to "sink" layer,

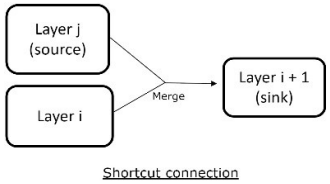


Fig 2: Illustration of the source and sink layers

and since a merge requires layer i and j to have the same output dimensions, we can thus identify the possible "sink" layer by identifying their respective dimensions.

2.3.4 Activation function

The post-processed functions are obtained by monitoring accesses to mathematical libraries, e.g.: $tanh$ and $sigmoid$ activations require access to mathematical libraries, whereas $relu$ activations do not.

2.4 THREAT MODEL

For the attack used in this paper, a few standard assumptions are required.

Black-Box Access: Following [4], a black box threat model assumes the DNN being accessible only via an official query interface with no prior knowledge to the architecture or hyper-parameters of the DNN.

Hardware and Setting: The methodology is an inference attack with the target model assumed to be deployed on CPUs (as most Cloud service providers use CPUs for inference [20] with GPUs being used for training). Additionally, we assume the adversary can launch a co-located process on the same host machine as the victim's DNN process, allowing a shared cache for the side-channel attack. We also assume the victim utilizes BLAS libraries (such as OpenBLAS and Intel MKL) for optimization during inference.

Evaluation: To measure the effectiveness of the functionality, 2 different error measures are being used:

(i) $Test\ Error: R_{test}(f, \hat{f}) = \sum_{(\vec{x},y) \in D} \frac{d(f(\vec{x}),\ \hat{f}(\vec{x}))}{|D|}$

(ii) $Uniform\ Error: R_{unif}(f, \hat{f}) = \sum_{\vec{x} \in U} \frac{d(f(\vec{x}),\ \hat{f}(\vec{x}))}{|U|}$

where: D is the test set used to evaluate the models, U is the set of input vectors uniformly taken from an input space and d is defined as the total variational distance: $d(\boldsymbol{y}, \hat{\boldsymbol{y}}) = \frac{1}{2} \sum |\boldsymbol{y}[i] - \hat{\boldsymbol{y}}[i]|$.

To reduce confusion, we denote the definition Model Test Error when comparing the models, and Test Error when denoting the error when compared against the test dataset when training the model. This is likewise true for Uniform Error.

Note that the model test error measures the similarity for an input distribution similar to the training data and the model uniform error measures the similarity for an input distribution that is uniform across the possible input space. The model test and uniform accuracy are then defined as $1 - R_{test}$ and $1 - R_{unif}$ respectively, with a high accuracy implying that the adversarial model $\hat{f}$ matches the victim's model f closely.

3. ATTACK METHODOLOGY

The complete methodology can be broadly categorized into 3 phases:

(i) Equation Solving Phase
(ii) Extraction Phase
(iii) Evaluation Phase

where the Equation Solving Phase is uses the equation solving attack to approximate a substitutional MLP, the Extraction Phase uses the Cache Telepathy mechanism to obtain a better architecture and train it through the use of queries, with the evaluation phase dictating the adequacy of the model, as shown in Fig. 3.

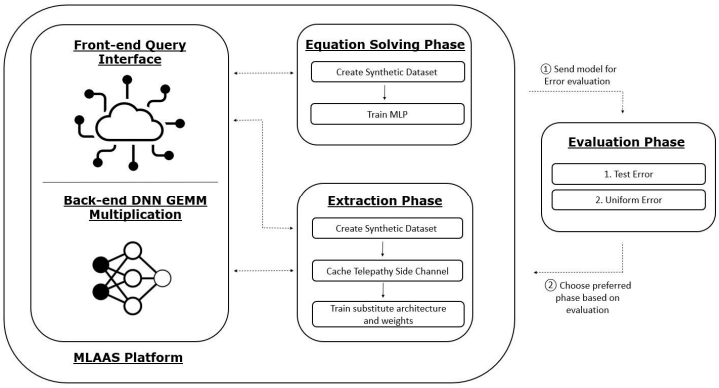


Fig. 3: Overview of the Attack Methodology and its place in an MLAAS setting

3.1 EQUATION SOLVING PHASE

At the very beginning of the model extraction, the Equation Solving Phase is assumed to be operated. This is to allow a "quick overview" of the extraction process and to hasten the model extraction whilst reducing the extraction cost if the model used by the victim is "simple". The exception to this would be if

the adversary knows prior that the problem statement is complex enough that MLPs would not be sufficient, such as facial recognition or object detection.

Presumptions: Initially, the adversary starts with the assumption that an MLP model with 1 hidden layer and a given number of hidden nodes (usually 20) with epochs of training is sufficient to provide an approximation to the adversarial model. By doing so, the adversary assumes that the equation solving attack provides a decent approximation to the victim's model.

Synthetic Dataset: Following the Equation Solving Attack, the adversary would first generate a synthetic dataset for solving. At the initial stage, although sophisticated methods of generating the dataset (such as using model inversion attacks) could in theory generate a dataset that corresponds well to the underlying data distribution, since the purpose of the Equation Solving Phase is for simplicity, using such methods could incur additional overhead cost and is deemed unsuitable for this stage. Hence, the proposed method for this phase is to randomly generate vectors $\vec{x}$ and query them, obtaining their "supposed class labels" y, and thus, obtaining a synthetic dataset used for solving [4]. A similar method can be done to generate the training, testing and evaluation when evaluating the model for errors. Additionally, the dataset needs to be sufficiently large as the loss function for MLPs are not strongly convex, thus a local minimum may not be found, incurring inaccuracy [4]. We note that there are exceptions to this, for example, if the problem is sufficiently complex, then complex methods of synthetic dataset generation should be used (explored further in 3.2).

Equation Solving Attack: After a sufficiently large synthetic dataset is generated, the equation solving attack can be used to generate a substitutional model. In the equation solving attack, it has been shown that with just 1 hidden layer with 20 hidden nodes, the accuracy of the MLP trained on the "Adult" Dataset can achieve an accuracy of up to 99% with just over 2000 queries [4]. Additionally, with more hidden layers, the number of queries grows exponentially, thus a possible recommendation for the initial MLP model (for relatively simple problems) is 1 hidden layer and 20 hidden nodes with 1000 epochs of training.

3.2 EXTRACTION PHASE

If the initial model is determined to be inadequate as a substitutional model at the Evaluation phase, the Extraction Phase is then done. At the Extraction Phase, the Cache Telepathy mechanism takes place where the adversary attempts to generate a more accurate architecture.

Search Space: Since this phase operates under the cache telepathy mechanism, cache access to the host machine is assumed. Thus, when the victim infers the DNN model, the adversary can quickly obtain traces of the function calls and thus generate a possible search space for the DNN architecture. Once a search space is generated, the adversary can choose an architecture to be used for training.

Synthetic Dataset: At the Extraction Phase, it has been determined that an MLP approximation is not sufficient, so a more complex model is needed, thus more data is required. Additionally, although it is possible, random vector generation will likely be inadequate and possibly detrimental when training the substitutional model if the synthetic dataset distribution is highly statistically different from the trained dataset, thus the substitutional model may be high in Model Test Accuracy, but low in Model Uniform Accuracy.

Possible improvements include knowing some general statistics or obtaining background about the dataset such as the marginal distribution of the feature values. If background knowledge of the dataset is known, assumptions could be made on the underlying data distribution, thus allowing multiple shadow models to be generated to determine if a given query $\vec{x}$ belongs to the victim's training dataset [10]

If such information is unknown, it would be beneficial to only collect data that have high confidence values through a "hill-climbing" algorithm, thus generating a dataset that is likely statistically similar to the victim's training dataset [11]. The complete algorithm (Synthesize) is provided in the Appendix (Fig. 19).

3.3 EVALUATION PHASE

At the Evaluation Phase, the synthetic test dataset, alongside the trained substitutional model generated from the respective phases, are then used to query and calculate the Model Test Accuracy and Model Uniform Accuracy. Once the accuracies are calculated, the adversary can decide whether to continually improve the model by repeating the respective phases and gaining more data, or to move to a different phase. If the model uniform accuracy is much larger than the model test accuracy, it is then suggested that a "hill climbing" method be used, whereas when the network's Model Test accuracy is suitably high, the substitutional model can be considered a success.

3.4 INTUITION

The entire methodology follows the principle of "training another network through querying for a dataset", with the Equation Solving Phase being an approximation to MLPs and the Extraction Phase being a generalized method of reversing any neural networks to a relatively high degree of accuracy. Despite the flexibility of the Extraction Phase, we still decide that although not crucial, the assumption that an MLP as a sufficient model is extremely helpful in speeding up the efficiency when reverse engineering, whilst also lowering costs by means of reduce querying and reduce energy spent on the side-channel co-location process. To further justify this, we illustrate the importance of the Equation Solving phase in 5.1 by testing it on different problems and showing the versatility of the Equation Solving Phase.

4. EXPERIMENTAL SETUP

Attack Platform: Our attacks are mounted on an ASUS VivoBook S410UN with an Intel x86_64. 4-core, i7-8550U Processor @ 1.8GHz, with 8GB RAM. The cache information summarized are such that in the L1 cache, both the data (L1d) and instruction (L1i) cache are 128KB each, and both the L2 and L3 cache have 8MB each. The OS used is Ubuntu 20.04.2 LTS. Note that during all inferences and training, no GPU acceleration is used, only CPU.

Extraction Phase Configuration: For the Extraction Phase, we configured the probing of $sgemm\_itcopy$ and $sgemm\_oncopy$ every 2000 cycles with an access cycle of less than 70 cycles for a hit, with most calls ending up around 30-45 cycles. The chosen OpenBLAS block sizes are P = 320, Q = 512, R = 4096, 3UNROLL = 3*4 = 12

5. RESULTS AND EVALUATION

To test the effectiveness of the methodology proposed, we considered the type of problem statement the victim's model would be suited to, and drafted a black-box using "appropriate" models. Then we subject the adversary to such situations and noted down some notable observations as well as some problems the adversary could face, and some possible improvements to the initial parameters to hopefully optimize the methodology. Note that it is impossible to consider all possible problem statements and scenarios.

5.1 UNREASONABLE EFFECTIVENESS OF THE EQUATION SOLVING PHASE

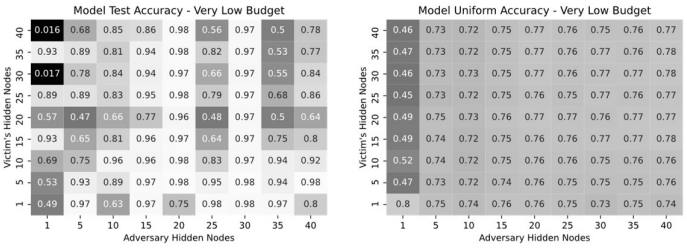
When exploring the effectiveness of the Equation Solving Phase in the methodology, we consider the "difficulty" of the problem the victim's DNN attempts to solve.

5.1.1 Basic Problem Statements

To test the effectiveness of the Equation Solving Phase for solving basic problems, following [4], we used the "Adult" dataset with the target "Race" trained under 1000 epochs. To simulate the experiment, we trained several MLP models composed of 1 hidden layer with 1 to 40 hidden nodes, because as shown by Tramèr et al, this model is more than sufficient to create a "black box model" with high accuracy (>99%). Additionally, through this experiment, we can observe the relationship between number of hidden nodes chosen by both the victim and the adversary, and thus find an optimal number of hidden nodes to start with, if the victim were to use an MLP.

For our test, we trained MLPs with hidden nodes in multiples of 5 to act both as the victim's model and the substitutional model. We then subject each of the substitutional model to the equation solving attack when solving for the model parameters. We have also separated the experiment into 4 cases depending on the query_budget: Very Low: 100 queries; Low: 1000 queries; Medium: 1000 queries; High: 10000 queries and trained each model with 1000 epochs.

Very Low query_budget:



(a) (b)

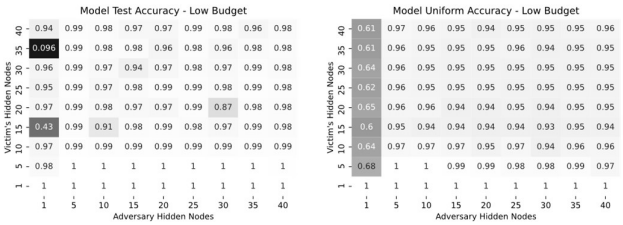
Fig. 4 (a): Heatmap of the Model Test Accuracy for Very Low Budget. (b): Heatmap of the Model Uniform Accuracy for Very Low Budget.

At first glance, an almost random-like structure is seen for the test accuracy with some "trends" seen when the adversary has a significantly higher number of hidden nodes (e.g.: adversary: 40, victim: 10 gives 92%), thus allowing a higher Model Test Accuracy.

However, in contrast, the Model Uniform Accuracy is extremely consistent (except for the first column), and this is to be expected because as stated in 3.1, we use a uniform querying system when obtaining the equations to solve. Hence, when we compare

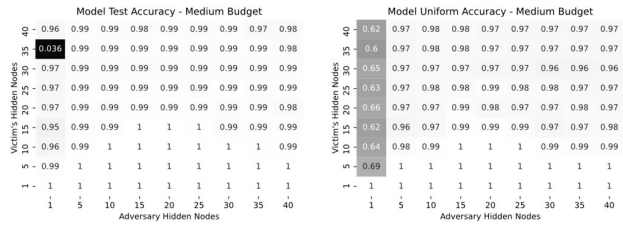
the model accuracies, if the dataset for comparison is uniform, we can also expect the accuracy to be "relatively high" despite the low querying budget, i.e.: both models act the same if given 'the same type of dataset". But when a dataset with an unknown distribution is used, the model acts almost randomly at a low querying budget. This can be solved using additional training as shown later. Thus, highlighting the need for the "hill climbing" method as stated in 3.2, and showing that the method of distribution of data extraction plays a crucial role.

Low and Medium query_budget:



(a) (b)

Fig. 5 (a), (b): Heatmap of the Model Test and Uniform Accuracy for "Low"

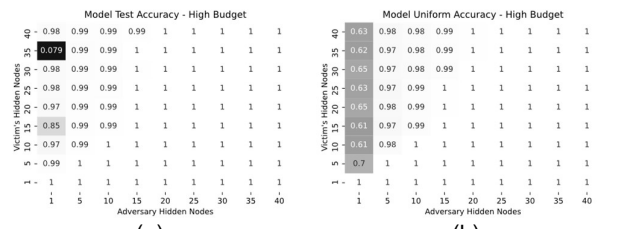


(a) (b)

Fig. 6 (a), (b): Heatmap of the Model Test and Uniform Accuracy for "Medium"

When a more appropriate query budget is given, the model displays some expected trends such as having a much higher accuracy when a "suitable" number of hidden nodes is chosen. Similarly, when the adversary's hidden node choice is much higher than the victim's the Model Test and Uniform Accuracies are much higher.

High query_budget:



(a) (b)

Fig. 7 (a), (b): Heatmap of the Model Test and Uniform Accuracy for "High

As expected, with the exception of the case with 1 hidden node, the adversary has a high model test and uniform accuracy even when the initial node numbers are significantly smaller than the victims'. Thus, implying that provided the adversary has a high enough budget, a successful extraction could be made even without guessing accurately the initial starting node (as long as it isn't 1 hidden node).

5.1.2 More Advance Problems

For more complex problems, sometimes, MLPs with more layers are needed. MLPs are also able to provide a relatively high accuracy. As the number of layers of an MLP increases, the representational capability for complex functions of the MLP increases. From the adversary's perspective, the number of layers the victim uses is unknown, hence the adversary will need to modify either their number of layers or number of hidden nodes to achieve better results.

The "Universal Approximation Theorem" states that there exists a neural network whereby a single hidden layer with a sufficiently large number of hidden nodes and the proper use of activation functions can approximate most continuous functions [12,13, 17, 18]. Thus, for the Equation Solving Phase, we propose that instead of guessing the number of layers needed, since the purpose of the Equation Solving Phase is to quickly obtain an approximation, the adversary should maintain using a single hidden layer but with increasing number of hidden nodes, and possibly increased budget. This is because modifying the hidden nodes would allow solving than guessing the "golden formula" for hidden layers and nodes. Plus, if more layers are needed, it will be shown quickly that no matter how high the number of hidden nodes become, the Equation Solving Phase will be insufficient due to the MLP approximation. Furthermore, layer tuning is more effectively done through the Extraction Phase than guessing at the Equation Solving Phase.

To illustrate an example, we tested the Equation Solving Phase on an MLP modelled for more number identification using the standard 28 x 28 MNIST dataset. The victim's model is a 4 layered (2 hidden layers) MLP with an input size of 784, hidden nodes of 120 and 80 in the first and second hidden layers respectively, and an output vector of 10. The relu activation function is chosen for activation after the input and first hidden layer. Whereas for the adversarial model, we assume the adversary would choose a decently sized model considering the 784-input size. For this (adversary), we consider 3 cases: 500, 1000, 5000 hidden nodes with a much higher budgeting query of 50,000 due to the complexity of the problem. Additionally, as the problem is more complex, we also considered using

the hill climbing method instead of just uniform querying. The results are summarised:

MLP Model	Hidden Nodes	Test Accuracy	1 - Rtest (Model)	1 - Runif (Model)	Runtime (seconds)
Victim's Model	2 layers (120, 80)	98.42%	-	-	271 (30 epochs)
Adversary (Unif Query)	1 layer (500)	20.76%	20.78%	94.14%	91 (30 epochs)
Adversary (Unif Query)	1 layer (1000)	20.97%	20.94%	94.04%	96 (30 epochs)
Adversary (Unif Query)	1 layer (5000)	21.55%	19.55%	92.1%	121 (10 epochs)
Adversary (Hill Climbing)	1 layer (500)	92.33%	92.47%	7.55%	93 (30 epochs)
Adversary (Hill Climbing)	1 layer (1000)	91.99%	92.44%	2.4%	93 (30 epochs)
Adversary (Hill Climbing)	1 layer (5000)	91.35%	91.69%	2.04%	145 (10 epochs)

Fig 8: Summary of the models trained and extracted

Notice that even at relatively low number of hidden nodes, as long as the proper synthetic dataset generation method is chosen, the respective accuracies will be high (e.g.: at 500 nodes, uniform querying, the model test accuracy = 20.78% but model uniform accuracy = 94.14%). From this, it can be noted that "perfectly" replicating an advance model through this method is difficult as when the number of hidden layers and nodes increases, so does the number of possible minima that the model could settle in. Hence, given the time, node choices and budget constraints, for these large models, it is simply unlikely to be able to reach a stage with both high model test and uniform accuracy.

However, in terms of functionality, often, the model test accuracy is considered as a reflection of realistic scenarios. Hence, it has been shown that with the hill climbing method, even a small number of nodes is more than enough to create a model with high model test accuracy.

5.1.3 Point of Failure

It is difficult to pinpoint a point of failure where the Equation Solving Phase fails as it is highly dependent on the problem/task the victim's DNN is trained to solve on as well as the judgement of the adversary. However, when it is shown (either through low accuracy scores or by complex problem statements such as facial recognition) that the Equation Solving Phase will be insufficient, the adversary can then move on to the Extraction Phase.

5.2 EXTRACTION PHASE

5.2.1 Example of the use of the Extraction Phase

To demonstrate the use of the Cache Telepathy, several examples are provided for the demonstration for identification of matrix parameters.

Number of Layers: The number of layers can be obtained by counting the number of "large" traces. An example of MLP with multiple layers is shown below where the orange arrows denote the gap between the layers (i.e.: 4 layers are identified below):

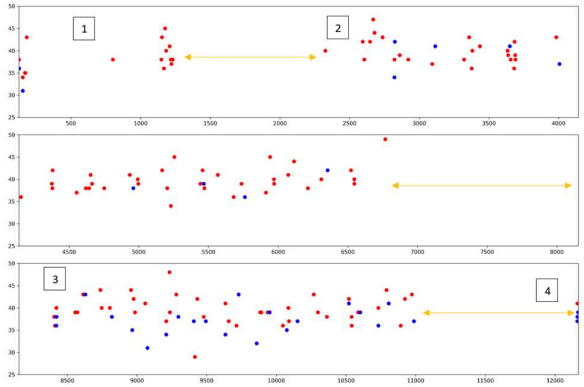


Fig 9: Image of an MLP with 4 layers identified

Iterations: Within each layer, the sequence of $itcopy$ ("blue") and $oncopy$ ("red") are found. Hence within each "layer", the matrix sizes can be found. Below shows a sub section of layer as an example:

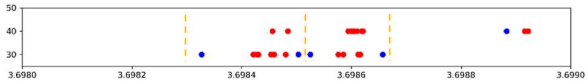


Fig 10: Image of an MLP with 4 layers identified

The orange lines separate the 3rd Loop in the OpenBLAS GEMM code. Within each "section", there's 1 $itcopy$ being called followed by 3 $oncopy$ and 1 $itcopy$ after. Following the GEMM code [2] provided in the appendix (Fig. 17), the first $itcopy$ in each section is likely the $itcopy$ in the first iteration of "Loop 3", then the $oncopy$ within the "1st iteration" implies $iter_4 = 3$, and the $itcopy$ at the end implies $iter_3 = 1$, with the "overall repetition" of the section implying $iter_2 = 2$. Note that $iter_1$ can sometimes be considered only runned once as the block size R can sometimes be up to size 104512.

Additional parameters: Additional methods of identification for some of the parameters have been shown in the original paper by Yan et al [2] and by Chabanne et al [14] specifically on identifying the Pooling layers and the convolutional layers respectively.

5.2.2 Effectiveness of the Extraction Phase

In terms of results, Yan et al [2] has shown that the trace manages to reduce the upper-bound of the search space using the formula below:

$$Search\ Space\ = \sum_{i=1}^{C} \prod_{j=1}^{L} x_j$$

where C: total number of connection configurations, L: total number of layers, x_j: total number of possible combinations of hyper-parameters for layer j. Hence using this formula, the search spaces of VGG-16 and Resnet-50 can be reduced from over 10^{46} to just 16 and 512 respectively [2], which is significant.

5.2.3 Issues with the Extraction Phase

Noise and effort: An obvious issue is that the Flush+Reload trace has sparse and random noise, which can be filtered out. Hence for larger sized models such as Resnets and VGG, much human manual effort is needed to reconstruct the model, but we do note that this is much lesser than the cost of obtaining a hyperparameter through handcraft.

Side Channel Tuning: Additionally, there is sometimes inconsistencies with the function calls due to possible Flush+Reload cache misses when the victim's call fully or partially overlaps with the reload phase, thus missing the call. Hence, some fine-tuning of the side channel is necessary for the attack.

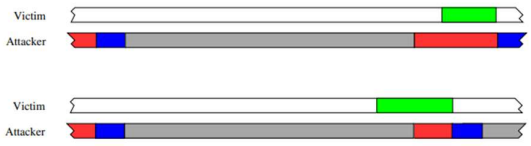


Fig 11: Illustration of the reload (red) and victim call (green) fully and partially overlapping [5]

Block Size: Another crucial issue is that cache telepathy fails when the matrix dimension m is smaller than 2*P where P is the block size. In this scenario, the matrix size m cannot be detected and this is important because the smallest matrix size for P in OpenBLAS is 128 ranging up to 1280. Thus, depending on what model and block size the victim decides to use, the model obtained may be to a certain extent misleading. An example of a layer with a low m size (m = 100) is shown below:

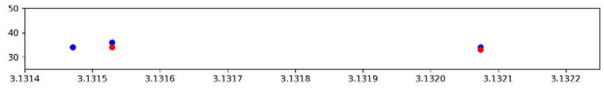


Fig 12: Image of a layer with m < 2P

Search Space: For some models, the search space is reduced to a narrow size for quick vetting (e.g.: VGG-16 being reduced to just 16 possible architectures), whereas for other models, although the possible architecture is significantly reduced, it is still large (e.g.: Resnet50 being reduced to 512 possible architectures). A possible solution to this is to use Reinforcement Learning similar to what Duddu et al [1] did where they trained a model to choose the best architecture from the Timing reduced space.

6. FURTHER DISCUSSIONS

The attack methodology can be considered naive and simple as there are multiple countermeasures to this methodology.

Hardware Countermeasures: There are a few crucial assumptions that this attack requires that might not be feasible. One of which is the requirement of a side-channel access. The "Flush+Reload" exploits the L3 cache and thus page-deduplication, but this is often not the case in cloud settings [15]. This can be solved by using the "Prime+Probe" side channel mechanism [6] but, existing cache side-channel defences such as Cache partitioning [21], and other cache security mechanisms such as SHARP [22] and CEASER [23] prevent such exploits.

Specialized Neural Network Countermeasures: There has been research on methods to reduce the effectiveness of the Equation Solving Attack. Guiga et al [8] proposed adding noise into the input, resulting in "unclear" confidence values. Furthermore, as shown earlier, the methodology fails when matrix size m < 2*P and this can easily be done if the victim were to use Integer Quantization. Additionally, Chabanne et al [14] has proposed adding randomness to the matrix block size and computing the neurons depth-wise, thus deterring a general formula from being formed.

Related works: There have been other related works that provide methods for fully reverse-engineering DNNs. One such is Duddu et al, who provided a comprehensive method of using timing-based side-channel attack on CPUs during inference to reduce the search space, and then using reinforcement learning, they can train a model to identify which model layer architecture is most likely the victim's from the timing obtained [1]. Additionally, Foo et al introduced NASPY [24], whereby a specialized use of Cache Telepathy is used to automate using NAS models for the extraction of NAS models.

7. CONCLUSION

In this paper, we propose a "complete" methodology of reverse engineering a DNN in a cloud setting under CPU inference. For simple problems, we can approximate the victim's model

as a DNN, and for more difficult problems, we can use the Cache Telepathy mechanism to obtain a possible architecture with hyperparameters to train the substitutional model. This methodology however, requires a certain amount of intuition and manual labour and have certain scenarios whereby it may fail. Hence the adversary may need to exercise judgement and experience in both the analysis and side channel attack to be able to use the methodology effectively.

ACKNOWLEDGEMENT

I would like to express my appreciation to my supervisor, Assoc Prof Chang Chip Hong, for giving me the opportunity to work on this project and explore this area of research. His continual guidance, vision, and support has provided me with a valuable opportunity to explore the research methodology. Additionally, I would like to express my gratitude and warmest thanks to my mentor, Dr Liu Wenye from the School of Electrical and Electronics Engineering at NTU for his constant guidance throughout this project. His advice and mentorship have not only helped to bring this project into fruition, but also ensure an enjoyable research experience.

I would also like to acknowledge [4, 5, 12, 13, 25] for providing their source codes as a reference. Special thank is given to Dr Xiaoxuan Lou from the School of Computer Science and Engineering at NTU for his advice, help, and source code in regard to the Cache Telepathy implementation.

I would like to acknowledge the funding support from Nanyang Technological University – URECA Undergraduate Research Programme for this research project.

APPENDIX

$$iter_3 = \lceil m/P \rceil$$
$$iter_2 = \lceil k/Q \rceil$$
$$iter_1 = \lceil n/R \rceil$$
$$iter_4 = \lceil R/3UNROLL \rceil$$
$$or \quad iter_4 = \lceil (n \bmod R)/3UNROLL \rceil$$

Fig 13: Relationship between matrix dimensions and OpenBLAS iteration counts [2]

Matrix	n_row	n_col
Input: In_i	B	N_i
Weight: θ_i	N_i	N_{i+1}
Output: O_i	B	N_{i+1}

Fig 14: Relationship between matrix dimensions and DNN layers for FC network [2]

Matrix	n_row	n_col
in_i'	$D_i \times R_i^2$	$(W_i - R_i + P_i)(H_i - R_i + P_i)$
F_i'	D_{i+1}	$D_i \times R_i^2$
out_i'	D_{i+1}	$(W_i - R_i + P_i)(H_i - R_i + P_i) = W_{i+1} \times H_{i+1}$

Fig 15: Relationship between matrix dimensions and DNN layers for conv network [2]

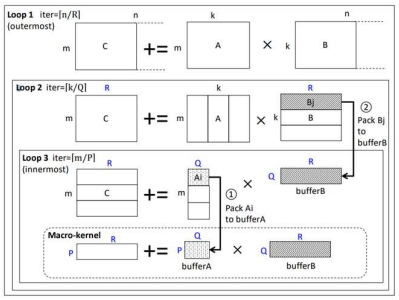


Fig 16: Goto's algorithm for cache-based matrix multiplication [2]

Algorithm 1: gemm_nn in OpenBLAS.

Input : Matrix A, B, C; Scalar α, β; Block size P, Q, R; *UNROLL*
Output : $C := \alpha A \cdot B + \beta C$

```
1  for j = 0, n, R do // Loop 1
2      for l = 0, k, Q do // Loop 2
           // Loop 3, 1st iteration
3          itcopy(A[0,l], buf_A, P, Q)
4          for jj = j, j+R, 3UNROLL do // Loop 4
5              oncopy(B[l,jj], buf_B + (jj-j) × Q, Q, 3UNROLL)
6              kernel(buf_A, buf_B + (jj-j) × Q, C[l,j], P, Q, 3UNROLL
7          end
           // Loop 3, rest iterations
8          for i = P, m, P do
9              itcopy(A[i,l], buf_A, P, Q)
10             kernel(buf_A, buf_B, C[l,j], P, Q, R)
11         end
12     end
13 end
```

Fig 17: GEMM multiplication in OpenBLAS [2]

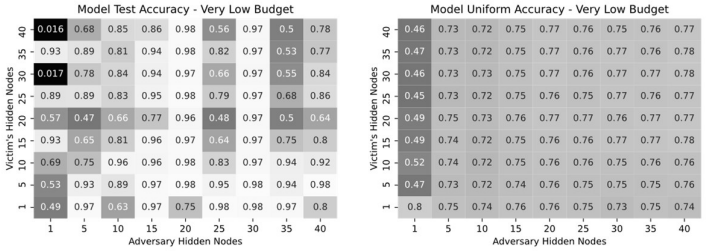


Fig 18: Repeat of the Very Low Query Budget.

Algorithm 1 Data synthesis using the target model

procedure SYNTHESIZE(class : c)
  $\mathbf{x} \leftarrow$ RANDRECORD( ) ▷ *initialize a record randomly*
  $y_c^* \leftarrow 0$
  $j \leftarrow 0$
  $k \leftarrow k_{max}$
  for $iteration = 1 \cdots iter_{max}$ **do**
    $\mathbf{y} \leftarrow f_{\text{target}}(\mathbf{x})$ ▷ *query the target model*
    if $y_c \geq y_c^*$ **then** ▷ *accept the record*
      if $y_c > \text{conf}_{min}$ and $c = \arg\max(\mathbf{y})$ **then**
        if $\text{rand}() < y_c$ **then** ▷ *sample*
          return $\mathbf{x}$ ▷ *synthetic data*
        end if
      end if
      $\mathbf{x}^* \leftarrow \mathbf{x}$
      $y_c^* \leftarrow y_c$
      $j \leftarrow 0$
    else
      $j \leftarrow j + 1$
      if $j > rej_{max}$ **then** ▷ *many consecutive rejects*
        $k \leftarrow \max(k_{min}, \lceil k/2 \rceil)$
        $j \leftarrow 0$
      end if
    end if
    $\mathbf{x} \leftarrow$ RANDRECORD($\mathbf{x}^*$, k) ▷ *randomize k features*
  end for
  return $\perp$ ▷ *failed to synthesize*
end procedure

Fig 19: "Hill Climbing" algorithm known as SYNTHESIZE [11]

REFERENCES

[1] V. Duddu, D. Samanta, D. V. Rao, and V. E. Balas, "Stealing Neural Networks via timing side channels," *arXiv [cs.CR]*, 2018.

[2] M. Yan, C. W. Fletcher, and J. Torrellas, "Cache telepathy: Leveraging shared resource attacks to learn {DNN} architectures," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 2003–2020.

[3] J. Wei, Y. Zhang, Z. Zhou, Z. Li, and M. A. Al Faruque, "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)*, 2020, pp. 125–137.

[4] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction APIs," *arXiv [cs.CR]*, 2016.

[5] Y. Yarom and K. 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.

[6] F. Liu, Y. Yarom, Q. Ge, G. Heiser, and R. B. Lee, "Last-Level Cache Side-Channel Attacks are Practical," in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 605–622.

[7] M. Ribeiro, K. Grolinger, and M. A. M. Capretz, "MLaaS: Machine Learning as a Service," in *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*, 2015, pp. 896–902.

[8] L. Guiga and A. Roscoe, "Neural network security: Hiding CNN parameters with guided grad-CAM," in *Proceedings of the 6th International Conference on Information Systems Security and Privacy*, 2020.

[9] F. Tramer, "Steal-ML." [Online]. Available: https://github.com/ftramer/Steal-ML.

[10] A. Salem, Y. Zhang, M. Humbert, P. Berrang, M. Fritz, and M. Backes, "ML-leaks: Model and data independent membership inference attacks and defenses on machine learning models," in *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[11] R. Shokri, M. Stronati, C. Song, and V. Shmatikov, "Membership inference attacks against machine learning models," *arXiv [cs.CR]*, 2016.

[12] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[13] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. London, England: MIT Press, 2016.

[14] H. Chabanne, J.-L. Danger, L. Guiga, and U. Kühne, "Telepathic headache: Mitigating cache side-channel attacks on convolutional neural networks," in *Applied Cryptography and Network Security*, Cham: Springer International Publishing, 2021, pp. 363–392.

[15] "VMware knowledge base," *Vmware.com*. [Online]. Available: https://kb.vmware.com/s/article/2080735.

[16] S. Hong *et al.*, "Security analysis of deep neural networks operating in the presence of cache side-channel attacks," *arXiv [cs.CR]*, 2018.

[17] T. Lim, "Understand Universal approximation theorem with code," *Towards Data Science*, 12-Sep-2020. [Online]. Available:

https://towardsdatascience.com/understand-universal-approximation-theorem-with-code-774dcef55731.

[18] K. Upadhyay, "Beginner's guide to universal Approximation Theorem," *Analytics Vidhya*, 29-Jun-2021. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/06/beginners-guide-to-universal-approximation-theorem/.

[19] K. Goto and R. A. van de Geijn, "Anatomy of high-performance matrix multiplication," *ACM Trans. Math. Softw.*, vol. 34, no. 3, pp. 1–25, 2008.

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

[21] W. Paper, "Improving real-time performance by utilizing cache allocation technology enhancing performance via allocation of the processor's cache," *Intel.com*, 2015. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf.

[22] M. Yan, B. Gopireddy, T. Shull, and J. Torrellas, "Secure hierarchy-aware cache replacement policy (SHARP): Defending against cache-based side channel atacks," in *Proceedings of the 44th Annual International Symposium on Computer Architecture*, 2017

[23] M. K. Qureshi, "CEASER: Mitigating conflict-based cache attacks via encrypted-address and remapping," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[24] X. Lou, S. Guo, J. Li, Y. Wu, and T. Zhang, "NASPY: Automated extraction of automated machine learning models," 2022.

[25] "DeepLearning.AI," *Coursera*. [Online]. Available: https://www.coursera.org/deeplearning-ai.

[26] H. Wang, S. M. Hafiz, K. Patwari, C.-N. Chuah, Z. Shafiq, and H. Homayoun, "Stealthy inference attack on DNN via cache-based side-channel attacks," in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2022.