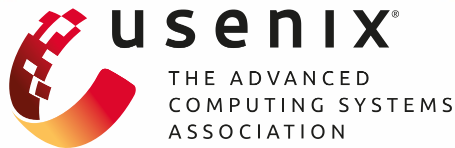

Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures

Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas,
University of Illinois at Urbana-Champaign

https://www.usenix.org/conference/usenixsecurity20/presentation/yan

This paper is included in the Proceedings of the
29th USENIX Security Symposium.

August 12–14, 2020

978-1-939133-17-5

Open access to the Proceedings of the
29th USENIX Security Symposium
is sponsored by USENIX.

Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures

Mengjia Yan
myan8@illinois.edu
University of Illinois at Urbana-Champaign

Christopher W. Fletcher
cwfletch@illinois.edu
University of Illinois at Urbana-Champaign

Josep Torrellas
torrella@illinois.edu
University of Illinois at Urbana-Champaign

Abstract

Deep Neural Networks (DNNs) are fast becoming ubiquitous for their ability to attain good accuracy in various machine learning tasks. A DNN's architecture (i.e., its hyper-parameters) broadly determines the DNN's accuracy and performance, and is often confidential. Attacking a DNN in the cloud to obtain its architecture can potentially provide major commercial value. Further, attaining a DNN's architecture facilitates other existing DNN attacks.

This paper presents *Cache Telepathy*: an efficient mechanism to help obtain a DNN's architecture using the cache side channel. The attack is based on the insight that DNN inference relies heavily on tiled GEMM (Generalized Matrix Multiply), and that DNN architecture parameters determine the number of GEMM calls and the dimensions of the matrices used in the GEMM functions. Such information can be leaked through the cache side channel.

This paper uses Prime+Probe and Flush+Reload to attack the VGG and ResNet DNNs running OpenBLAS and Intel MKL libraries. Our attack is effective in helping obtain the DNN architectures by very substantially reducing the search space of target DNN architectures. For example, when attacking the OpenBLAS library, for the different layers in VGG-16, it reduces the search space from more than 5.4×10^{12} architectures to just 16; for the different modules in ResNet-50, it reduces the search space from more than 6×10^{46} architectures to only 512.

1 Introduction

For the past several years, Deep Neural Networks (DNNs) have increased in popularity thanks to their ability to attain high accuracy and performance in a multitude of machine learning tasks — e.g., image and speech recognition [26, 63], scene generation [45], and game playing [51]. An emerging framework that provides end-to-end infrastructure for using DNNs is Machine Learning as a Service (MLaaS) [2, 19]. In MLaaS, trusted clients submit DNNs or training data to MLaaS service providers (e.g., an Amazon or Google datacenter). Service providers host the DNNs, and allow remote *untrusted* users to submit queries to the DNNs for a fee.

Despite its promise, MLaaS provides new ways to undermine the privacy of the hosted DNNs. An adversary may be able to learn details of the hosted DNNs beyond the official query APIs. For example, an adversary may try to learn the DNN's architecture (i.e., its *hyper-parameters*). These are the parameters that give the network its shape, such as the number and types of layers, the number of neurons per layer, and the connections between layers.

The architecture of a DNN broadly determines the DNN's accuracy and performance. For this reason, obtaining it often has high commercial value. Furthermore, once a DNN's architecture is known, other attacks are possible, such as the model extraction attack [55] (which obtains the weights of the DNN's edges), and the membership inference attack [39, 49] (which determines whether an input was used to train the DNN).

Yet, stealing a DNN's architecture is challenging. DNNs have a multitude of hyper-parameters, which makes brute-force guesswork unfeasible. Moreover, the DNN design space has been growing with time, which is further aggravating the adversary's task.

This paper demonstrates that despite the large search space, attackers can quickly reduce the search space of DNN architectures in the MLaaS setting using the cache side channel. Our insight is that DNN inference relies heavily on tiled GEMM (Generalized Matrix Multiply), and that DNN architecture parameters determine the number of GEMM calls and the dimensions of the matrices used in the GEMM functions. Such information can be leaked through the cache side channel.

We present an attack that we call *Cache Telepathy*. It is the first cache side channel attack targeting modern DNNs on general-purpose processors (CPUs). The reason for targeting CPUs is that CPUs are widely used for DNN inference in existing MLaaS platforms, such as Facebook's [25] and Amazon's [4].

We demonstrate our attack by implementing it on a state-of-the-art platform. We use Prime+Probe and Flush+Reload to attack the VGG and ResNet DNNs running OpenBLAS and Intel MKL libraries. Our attack is effective at helping obtain the architectures by very substantially reducing the search space of target DNN architectures. For example, when attacking the OpenBLAS library, for the different layers in VGG-16, it reduces the search space from more than 5.4×10^{12} architectures to just 16; for the different modules in ResNet-50, it reduces the search space from more than 6×10^{46} architectures to only 512.

This paper makes the following contributions:

1. It provides a detailed analysis of the mapping of DNN hyper-parameters to the number of GEMM calls and their arguments.
2. It implements the first cache-based side channel attack to extract DNN architectures on general purpose processors.
3. It evaluates the attack on VGG and ResNet DNNs running OpenBLAS and Intel MKL libraries.

2 Background

2.1 Deep Neural Networks

Deep Neural Networks (DNNs) are a class of Machine Learning (ML) algorithms that use a cascade of multiple layers of nonlinear processing units for feature extraction and transformation [35]. There are several major types of DNNs in use today, two popular types being fully-connected neural networks (or multi-layer perceptrons) and Convolutional Neural Networks (CNNs).

DNN Architecture The *architecture* of a DNN, also called the *hyper-parameters*, gives the network its shape. DNN hyper-parameters considered in this paper are:

a) Total number of layers.
b) Layer types, such as fully-connected, convolutional, or pooling layer.
c) Connections between layers, including sequential and non-sequential connections such as shortcuts. Non-sequential connections exist in recent DNNs, such as ResNet [26]. For example, instead of directly using the output from a prior layer as the input to a later layer, a shortcut involves summing up the outputs of two prior layers and using the result as the input for a later layer.
d) Hyper-parameters for each layer. For a fully-connected layer, this is the number of neurons in that layer. For a convolutional layer, this is the number of filters, the filter size, and the stride size.
e) The activation function in each layer, e.g., `relu` and `sigmoid`.

DNN Weights The computation in each DNN layer involves many multiply-accumulate operations (MACCs) on input neurons. The DNN *weights*, also called *parameters*, specify operands to these multiply-accumulate operations. In a fully-connected layer, each edge out of a neuron is a MACC with a weight; in a convolutional layer, each filter is a multi-dimensional array of weights, which is used as a sliding window that computes dot products over input neurons.

DNN Usage DNNs usage has two distinct phases: training and inference. In training, the DNN designer starts with a network architecture and a training set of labeled inputs, and tries to find the DNN weights to minimize mis-prediction error. Training is generally performed offline on GPUs and takes a relatively long time to finish, typically hours or days [12, 25]. In inference, the trained model is deployed and used to make real-time predictions on new inputs. For good responsiveness, inference is generally performed on CPUs [4, 25].

2.2 Prior Privacy Attacks Need the DNN Architecture

To gain insight into the importance of DNN architectures, we discuss prior DNN privacy attacks [39, 49, 55, 59]. There are three types of such attacks, each with a different goal. All of them require knowing the victim's DNN architecture. In the following, we refer to the victim's network as the oracle network, its architecture as the oracle DNN architecture, and its training data set as the oracle training data set.

In the *model extraction attack* [55], the attacker tries to obtain a network that is close enough to the oracle network. It assumes that the attacker knows the oracle DNN architecture at the start, and tries to estimate the weights of the oracle network. The attacker creates a synthetic data set, requests the classification results from the oracle network, and uses such results to train a network that uses the oracle architecture.

The *membership inference attack* [39, 49] aims to infer the composition of the oracle training data set, which is expressed as the probability of whether a data sample exists in the training set or not. This attack also requires knowledge of the oracle DNN architecture. Attackers create multiple synthetic data sets and train multiple networks that use the oracle architecture. Then, they run the inference algorithm on these networks with some inputs in their training sets and some not in their training sets. They then compare the results to find the patterns in the output of the data in the training sets. The pattern information is used to infer the composition of the oracle training set. Specifically, given a data sample, they run the inference algorithm of the oracle network, obtain the output and check whether the output matches the pattern obtained before. The more the output matches the pattern, the more likely the data sample exists in the oracle training set.

The *hyper-parameter stealing attack* [59] steals the loss function and regularization term used in ML algorithms, in-

cluding DNN training and inference. This attack also relies on knowing the oracle DNN architecture. During the attack, attackers leverage the model extraction attack to learn the DNN's weights. They then find the loss function that minimizes the training misprediction error.

2.3 Cache-based Side Channel Attacks

In a cache-based side channel attack, the attacker infers a secret from the victim by observing the side effects of the victim's cache behavior. Recently, multiple variations of cache-based side channel attacks have been proposed. Flush+Reload [69] and Prime+Probe [38, 43] are two powerful ones. Flush+Reload requires that the attacker share security-sensitive code or data with the victim. This sharing can be achieved by leveraging the page de-duplication technique. In an attack, the attacker first performs a `clflush` operation to the shared cache line, to push it out of the cache. It then waits to allow the victim to execute. Finally, it re-accesses the same cache line and measures the access latency. Depending on the latency, it learns whether the victim has accessed the shared line.

Prime+Probe does not require page sharing. It is more practical than Flush+Reload as most cloud providers disable page de-duplication for security purposes [58]. The attacker constructs a collection of addresses, called conflict addresses, which map to the same cache set as the victim's line. In an attack, the attacker first accesses the conflict addresses to cause cache conflicts with the victim's line, and evict it from the cache. After waiting for an interval, it re-accesses the conflict addresses and measures the access latency. The latency is used to infer if the victim has accessed the line.

2.4 Threat Model

This paper develops a cache-timing attack that quickly reduces the search space of DNN architectures. The attack relies on the following standard assumptions.

Black-box Access We follow a black-box threat model in an MLaaS setting similar to [55]. In a black-box attack, the DNN model is only accessible to attackers via an official query interface. Attackers do not have prior knowledge about the target DNN, including its hyper-parameters, weights and training data.

Co-location We assume that the attacker process can use techniques from prior work [7, 8, 14, 46, 57, 66, 73] to co-locate onto the same processor chip as the victim process running DNN inference. This is feasible, as current MLaaS jobs are deployed on shared clouds. Note that recent MLaaS, such as Amazon SageMaker [3] and Google ML Engine [18] allow users to upload their own code for training and inference, instead of using pre-defined APIs. In this case, attackers can disguise themselves as an MLaaS process and the cloud scheduler will have difficulty in separating attacker processes from victim processes.

Code Analysis We also assume that the attacker can analyze the ML framework code and linear algebra libraries used by the victim. These are realistic assumptions. First, open-source ML frameworks are widely used for efficient development of ML applications. The frameworks supported by Google, Amazon and other companies, including Tensorflow [1], Caffe [32], and MXNet [6] are all public. Our analysis is applicable to almost all of these frameworks. Second, the frameworks' backends are all supported by high-performance and popular linear algebra libraries, such as OpenBLAS [64], Eigen [23] and MKL [60]. OpenBLAS and Eigen are open sourced, and MKL can be reverse engineered, as we show in Section 6.

3 Attack Overview

The goal of Cache Telepathy is to substantially reduce the search space of target DNN architectures. In this section, we first discuss how our attack can assist other DNN privacy attacks, and then give an overview of the Cache Telepathy attack procedure.

Cache Telepathy's Role in Existing DNN Attacks In settings where DNN architectures are not known, our attack can serve as an essential initial step for many existing DNN privacy attacks, including model extraction attacks [55] and membership inference attacks [49].

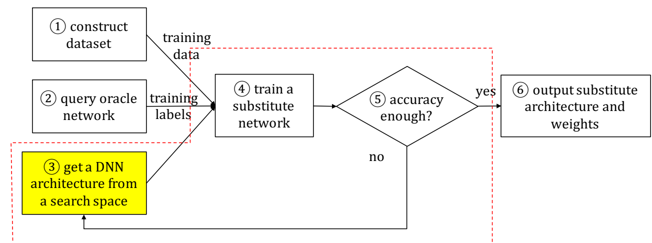

Figure 1: Cache Telepathy assists model extraction attacks.

Figure 1 demonstrates how Cache Telepathy makes the model extraction attack feasible. The final goal of the model extraction attack is to obtain a network that is close enough to the oracle network (Section 2.2). The attack uses the following steps. First, the attacker generates a synthetic training data set (①). This step can be achieved using a random feature vector method [55] or more sophisticated techniques, such as hill-climbing [49]. Next, the attacker queries the oracle network via inference APIs provided by MLaaS providers to get labels or confidence values (②). The synthetic data set and corresponding query results will be used as training data

and labels later. In the case that the oracle architecture is not known, the attacker needs to choose a DNN architecture from a search space (③) and then train a network with the chosen architecture (④). Steps ③-④ repeat until a network is found with sufficient prediction accuracy (⑤).

This attack process is extremely compute intensive, since it involves many iterations of step ④. Considering the depth and complexity of state-of-the-art DNNs, training and validating each network can take hours to days. Moreover, without any information about the architecture, the search space of possible architectures is often intractable, and thus, the model extraction attack is infeasible. However, Cache Telepathy can reduce the architecture search space (③) to a tractable size and make the attack feasible in settings where DNN architectures are unknown.

Membership inference attacks suffer from a more serious problem if the DNN architecture is not known. Recall that the attack aims to figure out the composition of the oracle training data set (Section 2.2). If there are many different candidate architectures, the attacker needs to consider the results generated by all the candidate architectures and statistically summarize inconsistent results from those architectures. A large search space of candidate architectures, not only significantly increases the computation requirements, but also potentially hurts attack accuracy. Consider a candidate architecture which is very different from the oracle architecture. It is likely to contribute incorrect results, and in turn, decrease the attack accuracy. However, Cache Telepathy can reduce the search space to a reasonable size. Moreover, the candidate architectures in the reduced search space have the same or very similar hyper-parameters as the oracle network. Therefore, they perform very similarly to the oracle network on various data sets. Hence, our attack also plays an important role in membership inference attacks.

Overall Cache Telepathy Attack Procedure Our attack is based on two observations. First, DNN inference relies heavily on GEMM (Generalized Matrix Multiply). We conduct a detailed analysis of how GEMM is used in ML frameworks, and figure out the mapping between DNN hyper-parameters and matrix parameters (Section 4). Second, high-performance GEMM algorithms are vulnerable to cache-based side channel attacks, as they are all tuned for the cache hierarchy through matrix blocking (i.e., tiling). When the block size is public (or can be easily deduced), the attacker can use the cache side channel to count blocks and learn the matrix sizes.

The Cache Telepathy attack procedure includes a cache attack and post processing steps. First, it uses a cache attack to monitor matrix multiplications and obtain matrix parameters (Sections 5 and 6). Then, the DNN architecture is reverse-engineered based on the mapping between DNN hyper-parameters and matrix parameters (Section 4). Finally, Cache Telepathy prunes the possible values of the remaining undiscovered hyper-parameters and generates a pruned search space for the target DNN architecture (Section 8.3). We consider the attack to be successful if we can generate a reasonable number of candidate architectures whose hyper-parameters are the same or very similar to the oracle network.

4 Mapping DNNs to Matrix Parameters

DNN hyper-parameters, listed in Section 2.1, can be mapped to GEMM execution. We first discuss how the layer type and configurations within each layer map to matrix parameters, assuming that all layers are sequentially connected (Section 4.1 and 4.2). We then generalize the mapping by showing how the connections between layers map to GEMM execution (Section 4.3). Finally, we discuss what information is required to extract the activation functions of Section 2.1 (Section 4.4).

4.1 Analysis of DNN Layers

There are two types of neural network layers whose computation can be mapped to matrix multiplications, namely fully-connected and convolutional layers.

4.1.1 Fully-connected Layer

In a fully-connected layer, each neuron computes a weighted sum of values from all the neurons in the previous layer, followed by a non-linear transformation. The ith layer computes $out_i = f_i(in_i \otimes \theta_i)$ where in_i is the input vector, θ_i is the weight matrix, $\otimes$ denotes a matrix-vector operation, f is an element-wise non-linear function such as tanh or sigmoid, and out_i is the resulting output vector.

The feed-forward computation of a fully-connected DNN can be performed over a batch of a few inputs at a time (B). These multiple input vectors are stacked into an input matrix In_i. A matrix multiplication between the input matrix and the weight matrix (θ_i) produces an output matrix, which is a stack of output vectors. We represent the computation as $O_i = f_i(In_i \cdot \theta_i)$ where In_i is a matrix with as many rows as B and as many columns as N_i (the number of neurons in the layer i); O_i is a matrix with as many rows as B and as many columns as N_{i+1} (the number of neurons in the layer $i+1$); and θ_i is a matrix with N_i rows and N_{i+1} columns. Table 1 shows the number of rows and columns of all the matrices.

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}

Table 1: Matrix sizes in a fully-connected layer.

4.1.2 Convolutional Layer

In a convolutional layer, a neuron is connected to only a spatial region of neurons in the previous layer. Consider the upper row of Figure 2, which shows the computation in the ith layer. The layer generates an output out_i (right part of the upper row) by performing convolution operations on an input in_i (center of the upper row) with multiple filters (left part of the upper row). The input volume in_i is of size $W_i \times H_i \times D_i$, where the depth ($D_i$) also refers to the number of channels of the input. Each filter is of size $R_i \times R_i \times D_i$.

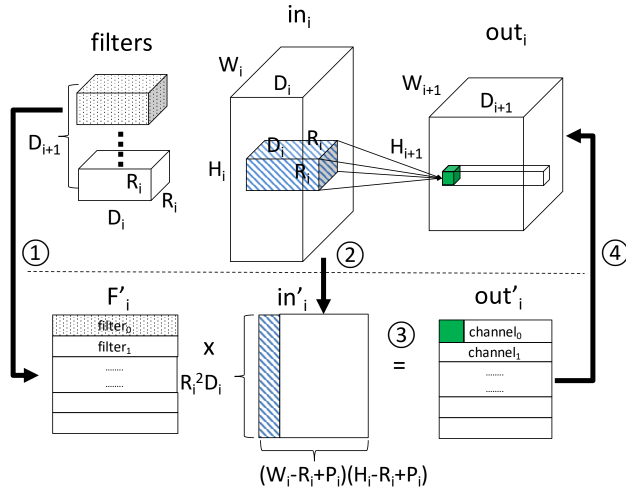

Figure 2: Mapping a convolutional layer (upper part of the figure) to a matrix multiplication (lower part).

To see how a convolution operation is performed, the figure highlights the process of generating one output neuron in out_i. The neuron is a result of a convolution operation – an elementwise dot product of the filter shaded in dots and the subvolume in in_i shaded in dashes. Both the subvolume and the filter have dimensions $R_i \times R_i \times D_i$. Applying one filter on the entire input volume (in_i) generates one channel of the output (out_i). Thus, the number of filters in layer i (D_{i+1}) is the number of channels (depth) in the output volume.

The lower row of Figure 2 shows a common implementation that transforms the multiple convolution operations in a layer into a single matrix multiply. First, as shown in arrow ①, each filter is stretched out into a row to form a matrix F'_i. The number of rows in F'_i is the number of filters in the layer.

Second, as shown in arrow ②, each subvolume in the input volume is stretched out into a column. The number of elements in the column is $D_i \times R_i^2$. For an input volume with dimensions $W_i \times H_i \times D_i$, there are $(W_i - R_i + P_i)(H_i - R_i + P_i)$ such columns in total, where P_i is the amount of zero padding. We call this transformed input matrix in'_i. Then, the convolution becomes a matrix multiply: $out'_i = F'_i \cdot in'_i$ (③).

Finally, the out'_i matrix is reshaped back to its proper dimensions of the out_i volume (arrow ④). Each row of the resulting out'_i matrix corresponds to one channel in the out_i volume. The number of columns of the out'_i matrix is $(W_i - R_i + P_i)(H_i - R_i + P_i)$, which is the size of one output channel, namely, $W_{i+1} \times H_{i+1}$. Table 2 shows the number of rows and columns of the matrices involved.

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}	$(W_i - R_i + P_i)(H_i - R_i + P_i) = W_{i+1} \times H_{i+1}$

Table 2: Matrix sizes in a convolutional layer.

The matrix multiplication described above processes a single input. As with fully-connected DNNs, CNN inference can consume a batch of B inputs in a single forward pass. In this case, a convolutional layer performs B matrix multiplications per pass. This is different from fully-connected layers, where the entire batch is computed using only one matrix multiplication.

4.2 Resolving DNN Hyper-parameters

Based on the previous analysis, we can now map DNN hyper-parameters to matrix operation parameters assuming all layers are sequentially connected.

4.2.1 Fully-connected Networks

Consider a fully-connected network. Its hyper-parameters are the number of layers, the number of neurons in each layer (N_i) and the activation function per layer. As discussed in Section 4.1, the feed-forward computation performs one matrix multiplication per layer. Hence, we extract the number of layers by counting the number of matrix multiplications performed. Moreover, according to Table 1, the number of neurons in layer i (N_i) is the number of rows of the layer's weight matrix (θ_i). The first two rows of Table 3 summarize this information.

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	$\approx \sqrt{\frac{n_col(out'_i)}{n_col(in'_{i+1})}}$

Table 3: Mapping between DNN hyper-parameters and matrix parameters. FC stands for fully connected.

[1]Specifically, we learn the filter spatial dimensions. If the filter is not square, the search space grows depending on factor combinations (e.g., 2 by

4.2.2 Convolutional Networks

A convolutional network generally consists of four types of layers: convolutional, Relu, pooling, and fully connected. Recall that each convolutional layer involves a batch B of matrix multiplications. Moreover, the B matrix multiplications that correspond to the same layer, always have the same dimension sizes and are executed consecutively. Therefore, we can count the number of consecutive matrix multiplications which have the same computation pattern to determine B.

In a convolutional layer i, the hyper-parameters include the number of filters (D_{i+1}), the filter width and height (R_i), and the padding (P_i). We assume that the filter width and height are the same, which is the common case. Note that for layer i, we consider that the depth of the input volume (D_i) is known, as it can be obtained from the previous layer.

We now show how these parameters for a convolutional layer can be reverse engineered. From Table 2, we see that the number of filters (D_{i+1}) is the number of rows of the filter matrix F_i'. To attain the filter width (R_i), we note that the number of rows of the in_i' matrix is $D_i \times R_i^2$, where D_i is the number of output channels in the previous layer and is equal to the number of rows of the out_{i-1}' matrix. Therefore, as summarized in Table 3, the filter width is attained by dividing the number of rows of in_i' by the number of rows of out_{i-1}' and performing the square root. In the case that layer i is the first one, directly connected to the input, the denominator (out_0') of this fraction is the number of channels of the input of the network, which is public information.

Padding results in a larger input matrix (in_i'). After resolving the filter width (R_i), the value of padding can be deduced by determining the difference between the number of columns of the output matrix of layer $i-1$ (out_{i-1}'), which is $W_i \times H_i$, and the number of columns of the in_i' matrix, which is $(W_i - R_i + P)(H_i - R_i + P)$.

A pooling layer can be located in-between two convolutional layers. It down-samples every channel of the input along width and height, resulting in a small channel size. The hyper-parameter in this layer is the pool width and height (assumed to be the same value), which can be inferred as follows. Consider the channel size of the output of layer i (number of columns in out_i') and the channel size of the input volume in layer $i+1$ (approximately equals to the number of columns in in_{i+1}'). If the two are the same, there is no pooling layer; otherwise, we expect to see the channel size reduced by the square of the pool width. In the latter case, the exact pool dimension can be found using a similar procedure used to determine R_i. Note that a non-unit stride operation results in the same dimension reduction as a pooling layer. Thus, we cannot distinguish between non-unit striding and pooling. Table 3 summarizes the mappings.

4 looks the same as 1 by 8). We note that filters in modern DNNs are nearly always square.

4.3 Connections Between Layers

We now examine how to map inter-layer connections to GEMM execution. We consider two types of inter-layer connections, i.e., sequential connections and non-sequential connections.

4.3.1 Mapping Sequential Connections

A sequential connection is one that connects two consecutive layers, e.g., layer i and layer $i+1$. The output of layer i is used as the input of its next layer $i+1$. According to the mapping relationships in Table 3, a DNN places several constraints on GEMM parameters for sequentially-connected convolutional layers.

First, since the filter width and height must be integer values, there is a constraint on the number of rows of the input and output matrices in consecutive layers. Considering the formula used to derive the filter width and height in Table 3, if layer i and layer $i+1$ are connected, the number of rows in the input matrix of layer $i+1$ ($n_row(in_{i+1}')$) must be the product of the number of rows in the output matrix of layer i ($n_row(out_i')$) and the square of an integer number.

Second, since the pool size and stride size are integer values, there is another constraint on the number of columns of the input and output matrix sizes between consecutive layers. According to the formula used to derive pool and stride size, if layer i and layer $i+1$ are connected, the number of columns in the output matrix of layer i ($n_col(out_i')$) must be very close to the product of the number of columns in the input matrix of layer $i+1$ ($n_col(in_{i+1}')$) and the square of an integer number.

The two constraints above help us to distinguish non-sequential connections from sequential ones. Specifically, if one of these constraints is not satisfied, we are sure that the two layers are not sequentially connected.

4.3.2 Mapping Non-sequential Connections

In this paper, we consider that a non-sequential connection is one where, given two consecutive layers i and $i+1$, there is a third layer j, whose output is merged with the output of layer i and the merged result is used as the input to layer $i+1$. We call the extra connection from layer j to layer $i+1$ a shortcut, where layer j is the source layer and layer $i+1$ is the sink layer. Shortcut connections can be mapped to GEMM execution.

First, there exists a certain latency between consecutive GEMMs, which we call inter-GEMM latency. The inter-GEMM latency before the sink layer in a non-sequential connection is longer than the latency in a sequential connection. To see why, consider the operations that are performed between two consecutive GEMMs: post-processing of the prior GEMM's output (e.g., batch normalization) and pre-processing of the next GEMM's input (e.g., padding and striding). When there is no shortcut, the inter-GEMM latency

is linearly related to the sum of the prior layer's output size and the next layer's input size. However, a shortcut requires an extra merge operation that incurs extra latency between GEMM calls.

Second, the source layer of a shortcut connection must have the same output dimensions as the other source layer of the non-sequential connection. For example, when a shortcut connects layer j and layer $i+1$, the output matrices of layer j and layer i must have the same number of rows and columns. This is because one can only merge two outputs whose dimension sizes match.

These two characteristics help us identify the existence of a shortcut, its source layer, and its sink layer.

4.4 Activation Functions

So far, this section discussed how DNN parameters map to GEMM calls. Convolutional and fully-connected layers are post-processed by elementwise non-linear functions, such as `relu`, `sigmoid` and `tanh`, which do not appear in GEMM parameters. We can distinguish `relu` activations from `sigmoid` and `tanh` by monitoring whether the non-linear functions access the standard mathematical library `libm`. `relu` is a simple activation which does not need support from libm, while the other functions are computationally intensive and generally leverage libm to achieve high performance. We remark that nearly all convolutional layers use `relu` or a close variant [26, 33, 52, 53, 65].

5 Attacking Matrix Multiplication

We now design a side channel attack to learn matrix multiplication parameters. Given the mapping from the previous section, this attack will allow us to reconstruct the DNN architecture.

We analyze state-of-the-art BLAS libraries, which have extensively optimized blocked matrix multiply. Examples of such libraries are OpenBLAS [64], BLIS [56], Intel MKL [60] and AMD ACML [5]. We show in detail how to extract the desired information from the GEMM implementation in OpenBLAS. In Section 6, we generalize our attack to other BLAS libraries, using Intel MKL as an example.

5.1 Analyzing GEMM from OpenBLAS

Function `gemm_nn` from the OpenBLAS library performs blocked matrix-matrix multiplication. It computes $C = \alpha A \cdot B + \beta C$ where α and β are scalars, A is an $m \times k$ matrix, B is a $k \times n$ matrix, and C is an $m \times n$ matrix. Our goal is to extract m, n and k.

Like most modern BLAS libraries, OpenBLAS implements Goto's algorithm [20]. The algorithm has been optimized for modern multi-level cache hierarchies. Figure 3 depicts the way Goto's algorithm structures blocked matrix multiplication for a three-level cache. The *macro-kernel* at the bottom performs the basic operation, multiplying a $P \times Q$ block from matrix A with a $Q \times R$ block from matrix B. This kernel is generally written in assembly code, and manually optimized by taking the CPU pipeline structure and register availability into consideration. The block sizes are picked so that the $P \times Q$ block of A fits in the L2 cache, and the $Q \times R$ block of B fits in the L3 cache.

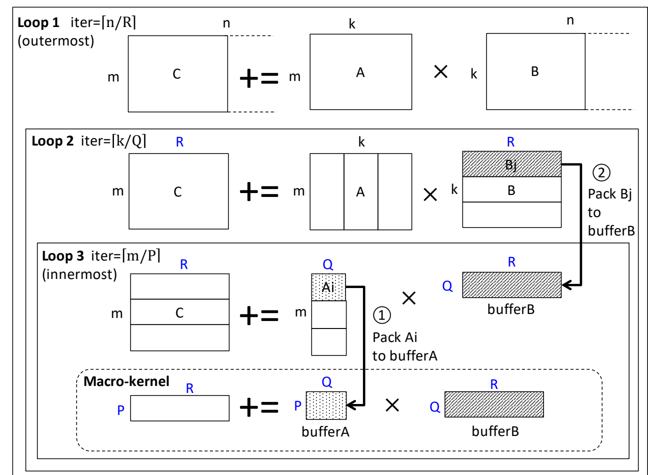

Figure 3: Blocked GEMM with matrices in column major.

As shown in Figure 3, there is a three-level loop nest around the macro-kernel. The innermost one is Loop 3, the intermediate one is Loop 2, and the outermost one is Loop 1. We call the iteration counts in these loops $iter_3$, $iter_2$, and $iter_1$, respectively, and are given by:

$$\begin{aligned} iter_3 &= \lceil m/P \rceil \\ iter_2 &= \lceil k/Q \rceil \\ iter_1 &= \lceil n/R \rceil \end{aligned} \quad (1)$$

Algorithm 1 shows the corresponding pseudo-code with the three nested loops. Note that Loop 3 is further split into two parts, to obtain better cache locality. The first part performs only the first iteration, and the second part performs the rest.

The first iteration of Loop 3 (Lines 3-7) performs three steps as follows. First, the data in the $P \times Q$ block from matrix A is packed into a buffer (bufferA) using function `itcopy`. This is shown in Figure 3 as arrow ① and corresponds to line 3 in Algorithm 1. Second, the data in the $Q \times R$ block from matrix B is also packed into a buffer (bufferB) using function `oncopy`. This is shown in Figure 3 as arrow ② and corresponds to line 5 in Algorithm 1. The $Q \times R$ block from matrix B is copied in units of $Q \times 3UNROLL$ sub-blocks. This breaks down the first iteration of Loop 3 into a loop, which is labeled as Loop 4. The iteration count in Loop 4, $iter_4$, is

Algorithm 1: `gemm_nn` in OpenBLAS.

```
   Input  : Matrix A, B, C; Scalar α, β; Block size P,Q,R; UNROLL
   Output : C := αA·B + βC
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
```

given by:

$$\begin{aligned} & iter_4 = \lceil R/3UNROLL \rceil \\ or \quad & iter_4 = \lceil (n \bmod R)/3UNROLL \rceil \end{aligned} \tag{2}$$

where the second expression corresponds to the last iteration of Loop 1. Note that bufferB, which is filled by the first iteration of Loop 3, is also shared by the rest of iterations. Third, the macro-kernel (function `kernel`) is executed on the two buffers. This corresponds to line 6 in Algorithm 1.

The rest iterations (line 8-11) skip the second step above. These iterations only pack a block from matrix A to fill bufferA and execute the macro-kernel.

The BLAS libraries use different P, Q, and R for different cache sizes to achieve best performance. For example, when compiling OpenBLAS on our experimental machine (Section 7), the GEMM function for double data type uses $P = 512$; $Q = 256$, $R = 16384$, and $3UNROLL = 24$.

5.2 Locating Probing Addresses

Our goal is to find the size of the matrices of Figure 3, namely, m, k, and n. To do so, we need to first obtain the number of iterations of the 4 loops in Algorithm 1, and then use Formulas 1 and 2. Note that we know the values of the block sizes P, Q, and R (as well as $3UNROLL$) — these are constants available in the open-source code of OpenBLAS.

In this paper, we propose to use, as probing addresses, addresses in the `itcopy`, `oncopy` and `kernel` functions of Algorithm 1. To understand why, consider the dynamic invocations to these functions. Figure 4 shows the Dynamic Call Graph (DCG) of `gemm_nn` in Algorithm 1.

Each iteration of Loop 2 contains one invocation of function `itcopy`, followed by $iter_4$ invocations of the pair `oncopy` and `kernel`, and then $(iter_3 - 1)$ invocations of the pair `itcopy` and `kernel`. The whole sequence in Figure 4 is executed $iter_1 \times iter_2$ times in one invocation of `gemm_nn`.

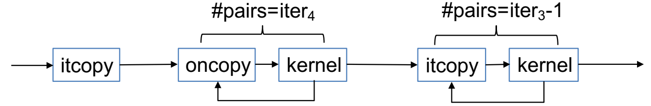

Figure 4: DCG of `gemm_nn`, with the number of invocations per iteration of Loop 2.

We will see in Section 5.3 that these invocation counts are enough to allow us to find the size of the matrices.

We now discuss how to select probing addresses inside the three functions—`itcopy`, `oncopy` and `kernel`—to improve attack accuracy. The main bodies of the three functions are loops. To distinguish these loops from the GEMM loops, we refer to them in this paper as *in-function* loops. We select addresses that are located inside the in-function loops as probing addresses. This strategy helps improve attack accuracy, because such addresses are accessed multiple times per function invocation and their access patterns can be easily distinguished from noise (Section 8.1).

5.3 Procedure to Extract Matrix Dimensions

To understand the procedure we use to extract matrix dimensions, we show an example in Figure 5(a), which visualizes the execution time of a `gemm_nn` where Loop 1, Loop 2 and Loop 3 have 5 iterations each. The figure also shows the size of the block that each iteration operates on. Note that the OpenBLAS library handles the last two iterations of each loop in a special manner. When the last iteration does not have a full block to compute, rather than assigning a small block to the last iteration, it assigns two equal-sized small blocks to the last two iterations. In Figure 5(a), in Loop 1, the first three iterations use R-sized blocks, and each of the last two use a block of size $(R+n \bmod R)/2$. In Loop 2, the corresponding block sizes are Q and $(Q+k \bmod Q)/2$. In Loop 3, they are P and $(P+m \bmod P)/2$.

Figure 5(b) shows additional information for each of the first iterations of Loop 3. Recall that the first iteration of Loop 3 is special, as it involves an extra packing operation that is performed by Loop 4 in Algorithm 1. Figure 5(b) shows the number of iterations of Loop 4 in each invocation ($iter_4$). During the execution of the first three iterations of Loop 1, $iter_4$ is $\lceil R/3UNROLL \rceil$. In the last two iterations of Loop 1, $iter_4$ is $\lceil ((R+n \bmod R)/2)/3UNROLL \rceil$, as can be deduced from Equation 2 after applying OpenBLAS' special handling of the last two iterations.

Based on these insights, our procedure to extract m, k, and n has four steps.

Step 1: Identify the DCG of a Loop 2 iteration and extract $iter_1 \times iter_2$**.** By probing one instruction in each of `itcopy`, `oncopy`, and `kernel`, we repeatedly obtain the DCG pattern of a Loop 2 iteration (Figure 4). By counting the number of such patterns, we obtain $iter_1 \times iter_2$.

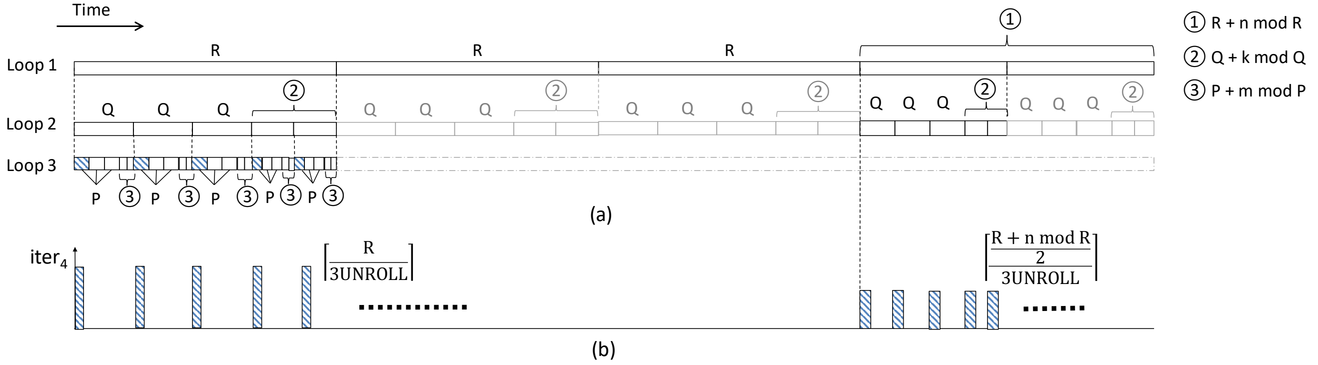

Figure 5: Visualization of execution time of a gemm_nn where Loop 1, Loop 2, and Loop3 have 5 iterations each (a), and value of $iter_4$ for each first iteration of Loop 3 (b).

Step 2: Extract $iter_3$ and determine the value of m. In the DCG pattern of a Loop 2 iteration, we count the number of invocations of the itcopy-kernel pair (Figure 4). This count plus 1 gives $iter_3$. Of all of these $iter_3$ iterations, all but the last two execute a block of size P; the last two execute a block of size $(P+m \bmod P)/2$ each (Figure 5(a)). To estimate the size of this smaller block, we assume that the execution time of an iteration is proportional to the block size it processes — except for the first iteration which, as we indicated, is different. Hence, we time the execution of a "normal" iteration of Loop 3 and the execution of the last iteration of Loop 3. Let's call the times t_{normal} and t_{small}. The value of m is computed by adding P for each of the ($iter_3$ - 2) iterations and adding the estimated number for each of the last two iterations:

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

Step 3: Extract $iter_4$ and $iter_2$, and determine the value of k. In the DCG pattern of a Loop 2 iteration (Figure 4), we count the number of oncopy-kernel pairs, and obtain $iter_4$. As shown in Figure 5(b), the value of $iter_4$ is $\lceil R/3UNROLL \rceil$ in all iterations of Loop 2 except those that are part of the last two iterations of Loop 1. For the latter, $iter_4$ is $\lceil ((R+n \bmod R)/2)/3UNROLL \rceil$, which is a lower value. Consequently, by counting the number of DCG patterns that have a low value of $iter_4$, and dividing it by 2, we attain $iter_2$. We then follow the procedure of Step 2 to calculate k. Specifically, all Loop 2 iterations but the last two execute a block of size Q; the last two execute a block of size $(Q+k \bmod Q)/2$ each (Figure 5(a)). Hence, we time the execution of two iterations of Loop 2 in the first Loop 1 iteration: a "normal" one (t'_{normal}) and the last one (t'_{small}). We then compute k like in Step 2:

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

Step 4: Extract $iter_1$ and determine the value of n. If we take the total number of DCG patterns in the execution from Step 1 and divide that by $iter_2$, we obtain $iter_1$. We know that all Loop 1 iterations but the last two execute a block of size R; the last two execute a block of size $(R+n \bmod R)/2$ each. To compute the size of the latter block, we note that, in the last two iterations of Loop 1, $iter_4$ is $\lceil ((R+n \bmod R)/2)/3UNROLL \rceil$. Since both $iter_4$ and $3UNROLL$ are known, we can estimate $(R+n \bmod R)/2$. We neglect the effect of the ceiling operator because $3UNROLL$ is a very small number. Hence, we compute n as:

$$n = (iter_1 - 2) \times R + 2 \times iter_4 \times 3UNROLL$$

Our attack cannot handle the cases when m or k are less than or equal to twice their corresponding block sizes. For example, when m is less than or equal to $2 \times P$, there is no iteration of Loop 3 that operates on a smaller block size. Our procedure cannot compute the exact value of m, and can only say that $m \leq 2P$.

6 Generalization of the Attack on GEMM

Our attack can be generalized to other BLAS libraries, since all of them use blocked matrix-multiplication, and most of them implement Goto's algorithm [20]. We show that our attack is still effective, using the Intel MKL library as an example. MKL is a widely used library but is closed source. We reverse engineer the scheduling of the three-level nested loop in MKL and its block sizes. The information is enough for us to apply the same attack procedure in Section 5.3 to obtain matrix dimensions.

Constructing the DCG We apply binary analysis [41, 73] techniques to construct the DCG of the GEMM function in MKL, shown in Figure 6. The pattern is the same as the DCG of OpenBLAS in Figure 4. Thus, the attack strategy in Section 5 also works towards MKL.

Extracting Block Sizes Similar to OpenBLAS, in MKL, there exists a linear relationship between matrix dimensions

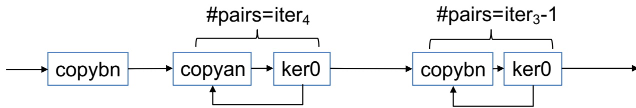

Figure 6: DCG of blocked GEMM in Intel MKL, with the number of invocations per iteration of Loop 2.

and iteration count for each of the loops, as shown in Formulas 1 and 2. When the matrix size increases by a block size, the corresponding iteration count increases by 1. We leverage this relationship to reverse engineer the block sizes for MKL. Specifically, we gradually increase the input dimension size until the number of iterations increments. The stride on the input dimension that triggers the change of iteration count is the block size.

Special Cases According to our analysis, MKL follows a different DCG when dealing with small matrices. First, instead of executing three-level nested loops, it uses a single-level loop, tiling on the dimension that has the largest value among m, n, k. Second, the kernel computation is performed directly on the input matrices, without packing and buffering operations.

For these special cases, we slightly adjust the attack strategy in Figure 5. We use side channels to monitor the number of iterations on that single-level loop and the time spent for each iteration. We then use the number of iterations to deduce the size of the largest dimension. Finally, we use the timing information for each iteration to deduce the product of the other two dimensions.

7 Experimental Setup

Attack Platform We evaluate our attacks on a Dell workstation Precision T1700, which has a 4-core Intel Xeon E3 processor and an 8GB DDR3-1600 memory. The processor has two levels of private caches and a shared last level cache. The first level caches are a 32KB instruction cache and a 32KB data cache. The second level cache is 256KB. The shared last level cache is 8MB. We test our attacks on a same-OS scenario using Ubuntu 4.2.0-27, where the attacker and the victim are different processes within the same bare-metal server. Our attacks should be applicable to other platforms, as the effectiveness of Flush+Reload and Prime+Probe has been proved in multiple hardware platforms [38, 69].

Victim DNNs We use a VGG [52] instance and a ResNet [26] instance as victim DNNs. VGG is representative of early DNNs (e.g., AlexNet [33] and LeNet [34]). ResNet is representative of state-of-the-art DNNs. Both are standard and widely-used CNNs with a large number of layers and hyper-parameters. ResNet additionally features shortcut connections.

There are several versions of VGG, with 11 to 19 layers. All VGGs have 5 types of layers, which are replicated a different number of times. We show our results on VGG-16.

There are several versions of ResNet, with 18 to 152 layers. All of them consist of the same 4 types of modules, which are replicated a different number of times. Each module contains 3 or 4 layers, which are all different. We show our results on ResNet-50.

The victim programs are implemented using the Keras [10] framework, with Theano [54] as the backend. We execute each DNN instance with a single thread.

Attack Implementation We use Flush+Reload and Prime+Probe attacks. In both attacks, the attacker and the victim are different processes and are pinned to different cores, only sharing the last level cache.

In Flush+Reload, the attacker and the victim share the BLAS library via page de-duplication. The attacker probes one address in `itcopy` and one in `oncopy` every 2,000 cycles. There is no need to probe any address in `kernel`, as the access pattern is clear enough. Our Prime+Probe attack targets the last level cache. We construct two sets of conflict addresses for the two probing addresses using the algorithm proposed by Liu et al. [38]. The Prime+Probe uses the same monitoring interval length of 2,000 cycles.

8 Evaluation

We first evaluate our attacks on the GEMM function. We then show the effectiveness of our attack on neural network inference, followed by an analysis of the search space of DNN architectures.

8.1 Attacking GEMM

8.1.1 Attack Examples

Figure 7 shows raw traces generated by Flush+Reload and Prime+Probe when monitoring the execution of the GEMM function in OpenBLAS. Due to space limitations, we only show the traces for one iteration of Loop 2 (Algorithm 1).

Figure 7(a) is generated under Flush+Reload. It shows the latency of the attacker's reload accesses to the probing addresses in the `itcopy` and `oncopy` functions for each monitoring interval. In the figure, we only show the instances where the access took less than 75 cycles. These instances correspond to cache hits and, therefore, cases when the victim executed the corresponding function. Figure 7(b) is generated under Prime+Probe. It shows the latency of the attacker's probe accesses to the conflict addresses. We only show the instances where the accesses took more than 500 cycles. These instances correspond to cache misses of at least one conflict address. They are the cases when the victim executed the corresponding function.

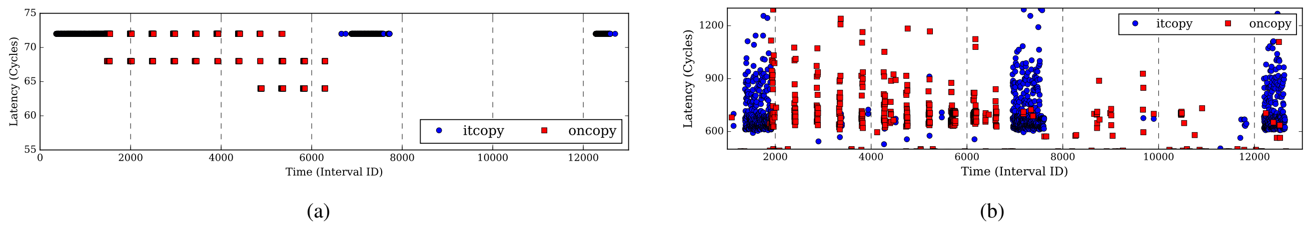

Figure 7: Flush+Reload (a) and Prime+Probe (b) traces of the GEMM execution. The monitoring interval is 2,000 cycles.

Since we select the probing addresses to be within in-function loops (Section 5.2), a cluster of hits in the Flush+Reload trace (or misses in the Prime+Probe trace) indicates the time period when the victim is executing the probed function.

In both traces, the victim calls `itcopy` before interval 2,000, then calls `oncopy` 11 times between intervals 2,000 and 7,000. It then calls `itcopy` another two times in intervals 7,000 and 13,000. The trace matches the DCG shown in Figure 4. We can easily derive that $iter_4 = 11$ and $iter_3 = 3$.

8.1.2 Handling Noise

Comparing the two traces in Figure 7, we can observe that Prime+Probe suffers much more noise than Flush+Reload. The noise in Flush+Reload is generally sparsely and randomly distributed, and thus can be easily filtered out. However, Prime+Probe has noise in consecutive monitoring intervals, as shown in Figure 7(b). It happens mainly due to the non-determinism of the cache replacement policy [13]. When one of the cache ways is used by the victim's line, it takes multiple "prime" operations to guarantee that the victim's line is selected to be evicted. It is more difficult to distinguish the victim's accesses from such noise.

We leverage our knowledge of the execution patterns in GEMM to handle the noise in Prime+Probe. First, recall that we pick the probing addresses within tight loops inside each of the probing functions (Section 5.2). Therefore, for each invocation of the functions, the corresponding probing address is accessed multiple times, which is observed as a cluster of cache misses in Prime+Probe. We count the number of consecutive cache misses in each cluster to obtain its size. The size of a cluster of cache misses that are due to noise is smaller than size of a cluster of misses that are caused by the victim's accesses. Thus, we discard the clusters with small sizes. Second, due to the three-level loop structure, each probing function, such as `oncopy`, is called repetitively with consistent interval lengths between each invocation (Figure 4). Thus, we compute the distances between neighboring clusters and discard the clusters with abnormal distances to their neighbors.

These two steps are effective enough to handle the noise in Prime+Probe. However, when tracing MKL's special cases that use a single-level loop (Section 6), we find that using Prime+Probe is ineffective to obtain useful information. Such environment affects the accuracy of the Cache Telepathy attack, as we will see in Section 8.3.

8.2 Extracting Hyper-parameters of DNNs

We show the effectiveness of our attack by extracting the hyper-parameters of VGG-16 [52] and ResNet-50 [26]. Figures 8(a), 8(b), and 8(c) show the extracted values of the *n*, *k*, and *m* matrix parameters, respectively, using Flush+Reload. In each figure, we show the values for each of the layers (*L1*, *L2*, *L3*, and *L4*) in the 4 distinct modules in ResNet-50 (*M1*, *M2*, *M3*, and *M4*), and for the 5 distinct layers in VGG-16 (*B1*, *B2*, *B3*, *B4*, and *B5*). We do not show the other layers because they are duplicates of the layers shown.

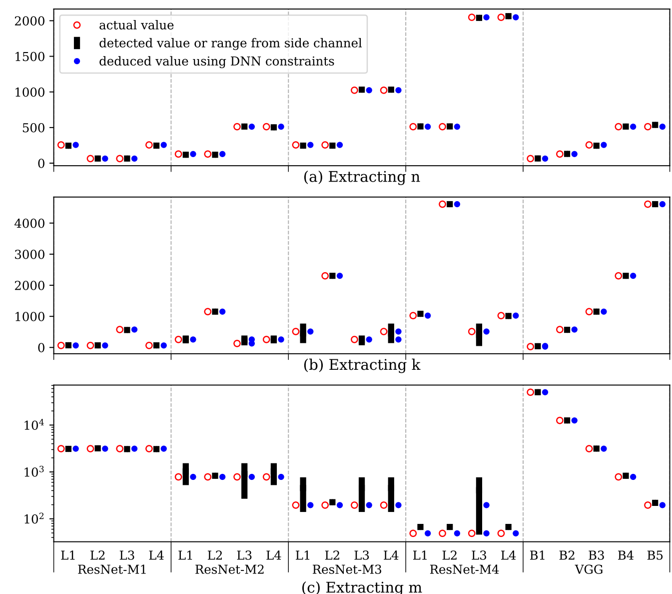

Figure 8: Extracted values of the *n*, *k*, and *m* matrix parameters for VGG-16 and ResNet-50 using Flush+Reload on OpenBLAS.

The figures show three data points for each parameter (e.g., *m*) and each layer (e.g., L1 in ResNet-M2): a hollowed circle, a solid square or rectangle, and a solid circle. The hollowed circle indicates the *actual* value of the parameter. The solid square or rectangle indicates the value of the parameter *de-*

tected with our side channel attack. When the side channel attack can only narrow down the possible value to a range, the figure shows a rectangle. Finally, the solid circle indicates the value of the parameter that we *deduce*, based on the detected value and some DNN constraints. For example, for parameter m in layer L1 of ResNet-M2, the actual value is 784, the detected value range is $[524, 1536]$, and the deduced value is 784.

We will discuss how we obtain the solid circles later. Here, we compare the actual and the detected values (hollowed circles and solid squares/rectangles). Figure 8(a) shows that our attack is always able to determine the n value with negligible error. The reason is that, to compute n, we need to estimate $iter_1$ and $iter_4$ (Section 5.3), and it can be shown that most of the noise comes from estimating $iter_4$. However, since $iter_4$ is multiplied by the small *3UNROLL* parameter in the equation for n, the impact of such noise is small.

Figures 8(b) and (c) show that the attack is able to accurately determine the m and k values for all the layers in ResNet-M1 and VGG, and for most layers in ResNet-M4. However, it can only derive ranges of values for most of the ResNet-M2 and ResNet-M3 layers. This is because the m and k values in these layers are often smaller than twice of the corresponding block sizes (Section 5.3).

In Figure 9, we show the same set of results by analyzing the traces generated using Prime+Probe. Compared to the results from Flush+Reload, there are some differences of detected values or ranges, especially in ResNet-M3 and ResNet-M4.

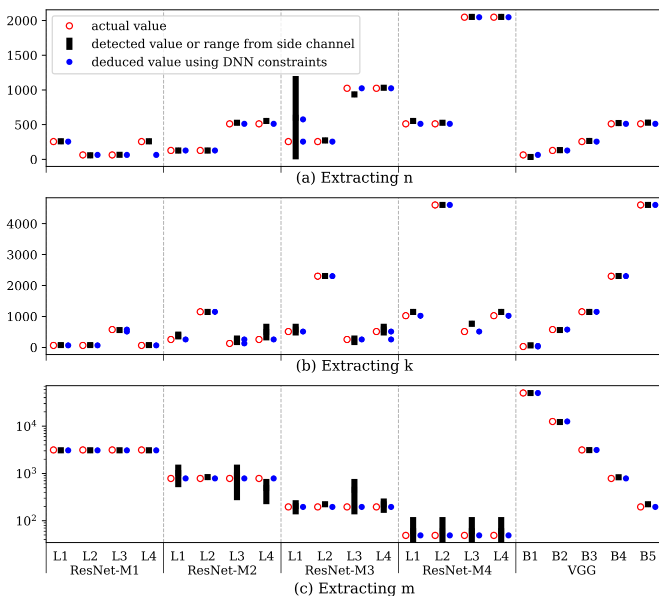

Figure 9: Extracted values of the n, k, and m matrix parameters for VGG-16 and ResNet-50 using Prime+Probe on OpenBLAS.

In summary, our side channel attacks, using Flush+Reload or Prime+Probe, can either detect the matrix parameters with negligible error, or can provide a range where the actual value falls in. We will next show that, in many cases, the imprecision from the negligible error and the ranges can be eliminated after applying DNN constraints (Section 8.3.2).

8.3 Size of Architecture Search Space

In this section, we compare the number of architectures in the search space without Cache Telepathy (which we call *original* space), and with Cache Telepathy (which we call *reduced* space). In both cases, we only consider reasonable hyper-parameters for the layers as follows. For fully-connected layers, the number of neurons can be 2^i, where $8 \leq i \leq 13$. For convolutional layers, the number of filters can be a multiple of 64 ($64 \times i$, where $1 \leq i \leq 32$), and the filter size can be an integer value between 1 and 11.

8.3.1 Size of the Original Search Space

To be conservative, when computing the size of the original search space, we assume that the attacker knows the number of layers and type of each layer in the oracle DNN. There exist 352 different configurations for each convolutional layer without considering pooling or striding, and 6 configurations for each fully-connected layer. Moreover, considering the existence of non-sequential connections, given L layers, there are $L \times 2^{L-1}$ possible ways to connect them.

A network like VGG-16 has five *different* layers (*B1*, *B2*, *B3*, *B4*, and *B5*), and no shortcuts. If we consider only these five different layers, the size of the search space is about 5.4×10^{12} candidate architectures. A network like ResNet-50 has 4 *different* modules (*M1*, *M2*, *M3*, and *M4*) and some shortcuts inside these modules. If we consider only these four different modules, the size of the search space is about 6×10^{46} candidate architectures. Overall, the original search space is intractable.

8.3.2 Determining the Reduced Search Space

Using the detected values of the matrix parameters in Section 8.2, we first determine the possible connections between layers by locating shortcuts. Next, for each possible connection configuration, we calculate the possible hyper-parameters for each layer. The final search space is computed as

$$search\ space = \sum_{i=1}^{C} \left(\prod_{j=1}^{L} x_j \right) \tag{3}$$

where C is the total number of possible connection configurations, L is the total number of layers, and x_j is the number of possible combinations of hyper-parameters for layer j.

Determining Connections Between Layers We show how to reverse engineer the connections between layers using ResNet-M1 as an example.

First, we leverage inter-GEMM latency to determine the existence of shortcuts and their sinks using the method discussed in Section 4.3. Figure 10 shows the extracted matrix dimensions and the inter-GEMM latency for the 4 layers in ResNet-M1. The inter-GEMM latency after M1-L4 is significantly longer than expected, given its output matrix size and the input matrix size of the next layer. Thus, the layer after M1-L4 is a sink layer.

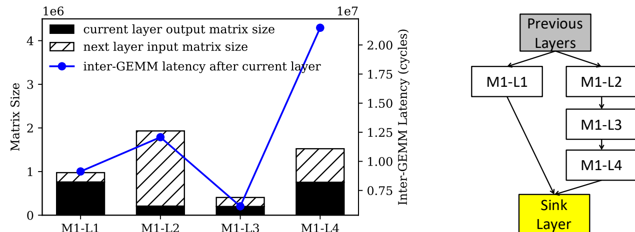

Figure 10: Extracting connections in ResNet-M1.

Next, we check the output matrix dimensions of previous layers to locate the source of the shortcut. Note that a shortcut only connects layers with the same output matrix dimensions. Based on the extracted dimension information (values of n and m) in Figure 8, we determine that M1-L1 is the source. In addition, we know that M1-L1 and M1-L2 are not sequentially connected by comparing the output matrix of M1-L1 and the input matrix of M1-L2 (Section 4.3).

Figure 10 summarizes the reverse engineered connections among the 4 layers, which match the actual connections in ResNet-M1. We can use the same method to derive the possible connection configurations for the other modules. Note that this approach does not work for ResNet-M3 and ResNet-M4. In these layers, the input and output matrices are small and operations between consecutive layers take a short time. As a result, the inter-GEMM latency is not effective in identifying shortcuts.

Determining Hyper-parameters for Each Layer We plug the detected matrix parameters into the formulas in Table 3 to deduce the hyper-parameters for each layer. For the matrix dimensions that cannot be extracted precisely, we leverage DNN constraints to prune the search space.

As an example, consider reverse engineering the hyper-parameters for Layer 3 in ResNet-M2. First, we extract the number of filters. We round the extracted $n_{M2\text{-}L3}$ from Figure 8(a) (the number of rows in F') to the nearest multiple of 64. This is because, as discussed at the beginning of Section 8.3, we assume that the number of filters is a multiple of 64. We get that the number of filters is 512. Second, we use the formula in Table 3 to determine the filter width and height. We consider the case where L2 is sequentially connected to L3. The extracted range of $k_{M2\text{-}L3}$ from Figure 8(b) (the number of rows in in' of current layer) is $[68, 384]$, and the value for $n_{M2\text{-}L2}$ from Figure 8(a) (the number of rows in out' of the previous layer) is 118. We need to make sure that the square root of $k_{M2\text{-}L3}/n_{M2\text{-}L2}$ is an integer, which leads to the conclusion that the only possible value for $k_{M2\text{-}L3}$ is 118 (one of the solid circles for $k_{M2\text{-}L3}$), and the filter width and height is 1. The same value is deduced if we consider, instead, that L1 is connected to L3. The other solid circle for $k_{M2\text{-}L3}$ is derived similarly if we consider that the last layer in M1 is connected to layer 3 in M2.

We apply the same methodology for the other layers. With this method, we obtain the solid circles in Figures 8 and 9.

Determining Pooling and Striding We use the difference in the m dimension (i.e., the channel size of the output) between consecutive layers to determine the pool or stride size. For example, in Figure 8(c) and 9(c), the m dimensions of the last layer in ResNet-M1 and the first layer in ResNet-M2 are different. This difference indicates the existence of a pool layer or a stride operation. In Figure 8(c), the extracted value of $m_{M1\text{-}L4}$ (the number of columns in out' for the current layer) is 3072, and the extracted range of $m_{M2\text{-}L1}$ (the number of columns in in' for the next layer) is $[524, 1536]$. We use the formula in Table 3 to determine the pool or stride width and height. To make the square root of $m_{M1\text{-}L4}/m_{M2\text{-}L1}$ an integer, $m_{M2\text{-}L1}$ has to be 768, and the pool or stride width and height have to be 2.

8.3.3 Size of the Reduced Search Space

Using Equation 3, we compute the number of architectures in the search space without Cache Telepathy and with Cache Telepathy. Table 4 shows the resulting values. Note that we only consider the possible configurations of the *different* layers in VGG-16 (*B1*, *B2*, *B3*, *B4*, and *B5*) and of the *different* modules in ResNet-50 (*M1*, *M2*, *M3*, and *M4*).

DNN		ResNet-50	VGG-16
Original: No Cache Telepathy		$> 6 \times 10^{46}$	$> 5.4 \times 10^{12}$
Flush+Reload	OpenBLAS	512	16
	MKL	6144	64
Prime+Probe	OpenBLAS	512	16
	MKL	5.7×10^{15}	1936

Table 4: Comparing the original search space (without Cache Telepathy) and the reduced search space (with Cache Telepathy).

Using Cache Telepathy to attack OpenBLAS, we are able to significantly reduce the search space from an intractable size to a reasonable size. Both Flush+Reload and Prime+Probe obtain a very small search space. Specifically, for VGG-16, Cache Telepathy reduces the search space from more than 5.4×10^{12} architectures to just 16; for ResNet-50, Cache Telepathy reduces the search space from more than 6×10^{46} to 512.

Cache Telepathy is less effective on MKL. For VGG-16, Cache Telepathy reduces the search space from more

than 5.4×10^{12} to 64 (with Flush+Reload) or 1936 (with Prime+Probe). For ResNet-50, Cache Telepathy reduces the search space from more than 6×10^{46} to 6144 (with Flush+Reload) or 5.7×10^{15} (with Prime+Probe). The last number is large because the matrix dimensions in Module M1 and Module 4 of ResNet-50 are small, and MKL handles these matrices with the special method described in Section 6. Such method is not easily attackable by Prime+Probe. However, if we only count the number of possible configurations in Modules M1, M2, and M3, the search space is 41472.

Implications of Large Search Spaces A large search space means that the attacker needs to train many networks. Training DNNs is easy to parallelize, and attackers can request many GPUs to train in parallel. However, it comes with a high cost. For example, assume that training one network takes 2 GPU days. On Amazon EC2, the current price for a single-node GPU instance is ∼\$3/hour. Without Cache Telepathy, since the search space is so huge, the cost is unbearable. Using Cache Telepathy with Flush+Reload, the reduced search space for the different layers in VGG-16 and for the different modules in ResNet-50 running OpenBLAS means that the training takes 32 and 1024 GPU days, respectively. The resulting cost is only ∼\$2K and ∼\$74K. When attacking ResNet-50 running MKL, the attacker needs to train 6144 architectures, requiring over \$884K.

9 Countermeasures

We overview possible countermeasures against our attack, and discuss their effectiveness and performance implications.

We first investigate whether it is possible to stop the attack by modifying the BLAS libraries. All BLAS libraries use extensively optimized blocked matrix multiplication for performance. One approach is to disable the optimization or use less aggressive optimization. However, it is unreasonable to disable blocked matrix multiplication, as the result would be very poor cache performance. Using a less aggressive blocking strategy, such as removing the optimization for the first iteration of Loop 3 (lines 4-7 in Algorithm 1), only slightly increases the difficulty for attackers to recover some matrix dimensions. It cannot effectively eliminate the vulnerability.

Another approach is to reduce the dimensions of the matrices. Recall that in both OpenBLAS and MKL, we are unable to precisely deduce the matrix dimensions if they are smaller than or equal to the block size. Existing techniques, such as quantization, can help reduce the matrix size to some degree. This mitigation is typically effective for the last few layers in a convolutional network, which generally use small filter sizes. However, it cannot protect layers with large matrices, such as those using a large number of filters and input activations.

Alternatively, one can use existing cache-based side channel defense solutions. One approach is to use cache partitioning, such as Intel CAT (Cache Allocation Technology) [30]. CAT assigns different ways of the last level cache to different applications, which blocks cache interference between attackers and victims [37]. Further, there are proposals for security-oriented cache mechanisms such as PLCache [62], SHARP [67] and CEASER [44]. If these mechanisms are adopted in production hardware, they can mitigate our attack with moderate performance degradation.

10 Related Work

Recent research has called attention to the confidentiality of neural network hyper-parameters. Hua et al. [29] designed the first attack to steal CNN architectures running on a hardware accelerator. Their attack is based on a different threat model, which requires the attacker to be able to monitor all of the memory addresses accessed by the victim. Our attack does not require such elevated privilege. Hong et al. [27] proposed to use cache-based side channel attacks to reverse engineer coarse-grained information of DNN architectures. Their attack is less powerful than Cache Telepathy. They can only obtain the number and types of layers, but are unable to obtain more detailed hyper-parameters, such as the number of neurons in fully-connected layers and filter size in convolutional layers. Batina et al. [9] proposed to use electromagnetic side channel attacks to reverse engineer DNNs in embedded systems.

Cache-based side channel attacks have been used to trace program execution to steal sensitive information. A lot of attacks target cryptography algorithms [15–17, 24, 31, 38, 40, 47, 68–70, 72], such as AES, RSA and ECDSA. Recent works also target application fingerprinting [28, 42, 48, 50, 73] to steal web content or server data, monitor user behavior [22, 36, 46, 71], and break system protection mechanisms such as SGX and KASLR [11, 21, 61].

11 Conclusion

In this paper, we proposed Cache Telepathy, an efficient mechanism to help obtain a DNN's architecture using the cache side channel. We identified that DNN inference relies heavily on blocked GEMM, and provided a detailed security analysis of this operation. We then designed an attack to extract the matrix parameters of GEMM calls, and scaled this attack to complete DNNs. We used Prime+Probe and Flush+Reload to attack VGG and ResNet DNNs running OpenBLAS and Intel MKL libraries. Our attack is effective at helping obtain the architectures by very substantially reducing the search space of target DNN architectures. For example, when attacking the OpenBLAS library, for the different layers in VGG-16, it reduces the search space from more than 5.4×10^{12} architectures to just 16; for the different modules in ResNet-50, it reduces the search space from more than 6×10^{46} architectures to only 512.

Acknowledgments

This work was funded in part by NSF under grant CCF-1725734.

References

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: A System for Large-scale Machine Learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation*, 2016.

[2] Amazon. Amazon Machine Learning. https://aws.amazon.com/machine-learning/, 2018.

[3] Amazon. Amazon SageMaker. https://aws.amazon.com/sagemaker/, 2018.

[4] Amazon. Amazon SageMaker ML Instance Types. https://aws.amazon.com/sagemaker/pricing/instance-types/, 2018.

[5] AMD. Core Math Library (ACML). https://developer.amd.com/amd-aocl/amd-math-library-libm/, 2012.

[6] Apache. Apache MXNet. https://mxnet.apache.org/, 2018.

[7] Ahmed Osama Fathy Atya, Zhiyun Qian, Srikanth V Krishnamurthy, Thomas La Porta, Patrick McDaniel, and Lisa Marvel. Malicious Co-Residency on the Cloud: Attacks and Defense. In *IEEE Conference on Computer Communications*. IEEE, 2017.

[8] Ahmed Osama Fathy Atya, Zhiyun Qian, Srikanth V Krishnamurthy, Thomas La Porta, Patrick McDaniel, and Lisa M Marvel. Catch Me if You Can: A Closer Look at Malicious Co-Residency on the Cloud. *IEEE/ACM Transactions on Networking*, 2019.

[9] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. CSI NN: Reverse Engineering of Neural Network Architectures Through Electromagnetic Side Channel. In *28th USENIX Security Symposium*, 2019.

[10] François Chollet. Keras. https://github.com/fchollet/keras, 2015.

[11] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. Cachequote: Efficiently Recovering Long-Term Secrets of SGX EPID via Cache Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018.

[12] Christopher De Sa, Matthew Feldman, Christopher Ré, and Kunle Olukotun. Understanding and optimizing asynchronous low-precision stochastic gradient descent. In *ACM SIGARCH Computer Architecture News*, 2017.

[13] Craig Disselkoen, David Kohlbrenner, Leo Porter, and Dean Tullsen. Prime+Abort: A Timer-Free High-Precision L3 Cache Attack Using Intel TSX. In *26th USENIX Security Symposium*, 2017.

[14] Paul Devadoss Ezhilchelvan and Isi Mitrani. Evaluating the Probability of Malicious Co-Residency in Public Clouds. *IEEE Transactions on Cloud Computing*, 2017.

[15] Cesar Pereida García and Billy Bob Brumley. Constant-Time Callees with Variable-Time Callers. In *26th USENIX Security Symposium*, 2017.

[16] Daniel Genkin, Lev Pachmanov, Eran Tromer, and Yuval Yarom. Drive-by Key-extraction Cache Attacks from Portable Code. In *International Conference on Applied Cryptography and Network Security*, 2018.

[17] Daniel Genkin, Luke Valenta, and Yuval Yarom. May the Fourth be With You: A Microarchitectural Side Channel Attack on Several Real-world Applications of Curve25519. In *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017.

[18] Google. Cloud ML Engine Overview. https://cloud.google.com/ml-engine/docs/technical-overview, 2018.

[19] Google. Google Machine Learning. https://cloud.google.com/products/machine-learning/, 2019.

[20] Kazushige Goto and Robert A. van de Geijn. Anatomy of High-performance Matrix Multiplication. *ACM Trans. Math. Softw.*, 2008.

[21] Daniel Gruss, Clémentine Maurice, Anders Fogh, Moritz Lipp, and Stefan Mangard. Prefetch Side-Channel Attacks: Bypassing SMAP and Kernel ASLR. In *ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016.

[22] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-level Caches. In *Proceedings of the 24th USENIX Conference on Security Symposium*, 2015.

[23] Gaël Guennebaud, Benoît Jacob, et al. Eigen v3. http://eigen.tuxfamily.org, 2010.

[24] David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games–Bringing Access-based Cache Attacks on AES to Practice. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2011.

[25] Kim Hazelwood, Sarah Bird, David Brooks, Soumith Chintala, Utku Diril, Dmytro Dzhulgakov, Mohamed Fawzy, Bill Jia, Yangqing Jia, Aditya Kalro, James Law, Kevin Lee, Jason Lu, Pieter Noordhuis, Misha Smelyanskiy, Liang Xiong, and Xiaodong Wang. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *International Symposium on High-Performance Computer Architecture*, 2018.

[26] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep Residual Learning for Image Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016.

[27] Sanghyun Hong, Michael Davinroy, Yiğitcan Kaya, Stuart Nevans Locke, Ian Rackow, Kevin Kulda, Dana Dachman-Soled, and Tudor Dumitraş. Security Analysis of Deep Neural Networks Operating in the Presence of Cache Side-Channel Attacks. *arXiv preprint arXiv:1810.03487*, 2018.

[28] Taylor Hornby. Side-Channel Attacks on Everyday Applications: Distinguishing Inputs with FLUSH+RELOAD. BlackHat, 2016.

[29] W. Hua, Z. Zhang, and G. E. Suh. Reverse engineering convolutional neural networks through side-channel information leaks. In *Design Automation Conference (DAC)*. ACM, 2018.

[30] Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf, 2015.

[31] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack that Works Across Cores and Defies VM Sandboxing – and Its Application to AES. In *IEEE Symposium on Security and Privacy (SP)*, 2015.

[32] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*, 2014.

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2012.

[34] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 1998.

[35] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep Learning. *Nature*, 2015.

[36] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. ARMageddon: Last-level Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium*, 2015.

[37] Fangfei Liu, Qian Ge, Yuval Yarom, Frank McKeen, Carlos Rozas, Gernot Heiser, and Ruby B. Lee. CATalyst: Defeating Last-level Cache Side Channel Attacks in Cloud Computing. In *IEEE International Symposium on High Performance Computer Architecture*, 2016.

[38] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-level Cache Side-Channel Attacks are Practical. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2015.

[39] Yunhui Long, Vincent Bindschaedler, Lei Wang, Diyue Bu, Xiaofeng Wang, Haixu Tang, Carl A. Gunter, and Kai Chen. Understanding Membership Inferences on Well-Generalized Learning Models. *CoRR*, abs/1802.04889, 2018.

[40] Michael Neve and Jean P. Seifert. Advances on Access-Driven Cache Attacks on AES. In *Proceedings of the 13th International Conference on Selected Areas in Cryptography*. Springer-Verlag, 2007.

[41] Ryan O'Neill. *Learning Linux Binary Analysis*. Packt Publishing Ltd., 2016.

[42] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015.

[43] DagArne Osvik, Adi Shamir, and Eran Tromer. Cache Attacks and Countermeasures: The Case of AES. In *Topics in Cryptology*, Lecture Notes in Computer Science. Springer, 2006.

[44] M. K. Qureshi. CEASER: Mitigating Conflict-Based Cache Attacks via Encrypted-Address and Remapping. In *51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018.

[45] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks. *CoRR*, abs/1511.06434, 2015.

[46] Thomas Ristenpart, Eran Tromer, Hovav Shacham, and Stefan Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-party Compute

Clouds. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*. ACM, 2009.

[47] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 Lives of Bleichenbacher's CAT: New Cache Attacks on TLS Implementations. In *IEEE Symposium on Security and Privacy*, 2019.

[48] Michael Schwarz, Florian Lackner, and Daniel Gruss. JavaScript Template Attacks: Automatically Inferring Host Information for Targeted Exploits. In *Network and Distributed System Security Symposium (NDSS)*, 2019.

[49] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. Membership Inference Attacks against Machine Learning Models. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2017.

[50] Anatoly Shusterman, Lachlan Kang, Yarden Haskal, Yosef Meltser, Prateek Mittal, Yossi Oren, and Yuval Yarom. Robust Website Fingerprinting through the Cache Occupancy Channel. In *28th USENIX Security Symposium*, 2019.

[51] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the Game of Go with Deep Neural Networks and Tree Search. *Nature*, 2016.

[52] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-scale Image Recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[53] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going Deeper with Convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

[54] Theano Development Team. Theano: A Python Framework for Fast Computation of Mathematical Expressions. *arXiv e-prints*, abs/1605.02688, 2016.

[55] Florian Tramèr, Fan Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Stealing Machine Learning Models via Prediction APIs. In *USENIX Security*, 2016.

[56] Field G Van Zee and Robert A Van De Geijn. BLIS: A Framework for Rapidly Instantiating BLAS Functionality. *ACM Transactions on Mathematical Software (TOMS)*, 2015.

[57] Venkatanathan Varadarajan, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. A Placement Vulnerability Study in Multi-tenant Public Clouds. In *24th USENIX Security Symposium*, 2015.

[58] VMWare. Security Considerations and Disallowing Inter-Virtual Machine Transparent Page Sharing, 2018.

[59] Binghui Wang and Neil Zhenqiang Gong. Stealing Hyperparameters in Machine Learning. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018.

[60] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel Math Kernel Library. In *High-Performance Computing on the Intel® Xeon Phi*. Springer, 2014.

[61] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Hazards in SGX. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2017.

[62] Zhenghong Wang and Ruby B. Lee. New Cache Designs for Thwarting Software Cache-based Side Channel Attacks. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. ACM, 2007.

[63] Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V. Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, Jeff Klingner, Apurva Shah, Melvin Johnson, Xiaobing Liu, Lukasz Kaiser, Stephan Gouws, Yoshikiyo Kato, Taku Kudo, Hideto Kazawa, Keith Stevens, George Kurian, Nishant Patil, Wei Wang, Cliff Young, Jason Smith, Jason Riesa, Alex Rudnick, Oriol Vinyals, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. *CoRR*, abs/1609.08144, 2016.

[64] Zhang Xianyi, Wang Qian, and Zaheer Chothia. OpenBLAS. `http://www.openblas.net/`, 2019.

[65] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical Evaluation of Rectified Activations in Convolutional Network. *CoRR*, abs/1505.00853, 2015.

[66] Zhang Xu, Haining Wang, and Zhenyu Wu. A Measurement Study on Co-Residence Threat Inside the Cloud. In *24th USENIX Security Symposium*, 2015.

[67] Mengjia Yan, Bhargava Gopireddy, Thomas Shull, and Josep Torrellas. Secure Hierarchy-Aware Cache Replacement Policy (SHARP): Defending against Cache-based Side Channel Attacks. In *ACM/IEEE 44th An-*

nual International Symposium on Computer Architecture (ISCA). IEEE, 2017.

[68] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. Attack Directories, Not Caches: Side Channel Attacks in a Non-inclusive World. In *IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019.

[69] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium*. USENIX Association, 2014.

[70] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A Timing Attack on OpenSSL Constant-time RSA. *Journal of Cryptographic Engineering*, 2017.

[71] Xiaokuan Zhang, Yuan Xiao, and Yinqian Zhang. Return-Oriented Flush-Reload Side Channels on ARM and Their Implications for Android Devices. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2016.

[72] Yinqian Zhang, Ari Juels, Michael K Reiter, and Thomas Ristenpart. Cross-VM Side Channels and Their Use to Extract Private Keys. In *ACM conference on Computer and Communications Security*, 2012.

[73] Yinqian Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. Cross-tenant Side-Channel Attacks in PaaS Clouds. In *SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, 2014.