JAXED: Reverse Engineering DNN Architectures Leveraging JIT GEMM Libraries

Malith Jayaweera*, Kaustubh Shivdikar*, Yanzhi Wang* and David Kaeli†
Northeastern University, USA
*{malithjayaweera.d, shivdikar.k, yanz.wang}@northeastern.edu, †kaeli@ece.neu.edu

***Abstract*—General matrix multiplication (GEMM) libraries on x86 architectures have recently adopted Just-in-time (JIT) based optimizations to dramatically reduce the execution time of small and medium-sized matrix multiplication. The exploitation of the latest CPU architectural extensions, such as the AVX2 and AVX-512 extensions, are the target for these optimizations. Although JIT compilers can provide impressive speedups to GEMM libraries, they expose a new attack surface through the built-in JIT code caches. These software-based caches allow an adversary to extract sensitive information through carefully designed timing attacks. The attack surface of such libraries has become more prominent due to their widespread integration into popular Machine Learning (ML) frameworks such as PyTorch and Tensorflow.**

In our paper, we present a novel attack strategy for JIT-compiled GEMM libraries called JAXED. We demonstrate how an adversary can exploit the GEMM library's vulnerable state management to extract confidential CNN model hyperparameters. We show that using JAXED, one can successfully extract the hyperparameters of models with fully-connected layers with an average accuracy of 92%. Further, we demonstrate our attack against the final fully connected layer of 10 popular DNN models. Finally, we perform an end-to-end attack on MobileNetV2, on both the convolution and FC layers, successfully extracting model hyperparameters.

***Index Terms*—JIT, Compilers, GEMM library, Software Cache, Security, Machine Learning, Timing Attack, JAXED**

I. Introduction

With recent advances in deep learning, many businesses have adopted or integrated deep learning into their core technology stack [1]. Although the performance of Deep Neural Networks (DNNs) has created new business opportunities, the cost of designing and training a high-performance DNN model remains an expensive process. The main reason is due to the highly specialized knowledge required on DNN architectures, and also the challenges faced when building large data sets that are representative of real-world scenarios. Increasingly, companies treat their deep learning model hyperparameters as prized possessions, given that minor improvements provide a competitive edge over their competition. Thus, it has become increasingly important to protect this valuable intellectual property.

DNN architectures are powered by convolution operations, which are performed using highly-optimized General Matrix Multiply (GEMM) libraries. For many DNN architectures, it has been reported that traditional BLAS libraries provide limited benefits when performing small and medium-sized matrix multiplications, as BLAS is optimized to utilized to leverage the memory hierarchy effectively, rather than maximize compute resource usage [2], [3]. Recently, there has been a growing trend towards optimizing GEMM performance through run-time optimizations [4]. Traditionally, JIT-based compilation has been used for interpreted languages such as Python and Java, though similar optimizations are being incorporated into libraries written in compile-time languages such as C/C++. Run-time optimizations can be more aggressive than compile-time optimizations due to the availability of problem dimensions at run-time. Hence, a JIT compiler can improve the performance of small and medium-sized matrix multiplication kernels by orders of magnitude [2], [4]. The latest generation of GEMM libraries with runtime code generation build an internal software code cache to allow faster execution of GEMM kernels [2]–[5]. The runtime library manages these caches, relieving the user from explicitly managing them, making them transparent to the user. This code cache is a vital aspect of JIT optimizations, as they allow the dynamic optimization system to amortize the cost of run-time code generation over many iterations. For machine learning (ML) frameworks, where hyperparameters remain constant for a period (until they are eventually updated), a significant speedup can be obtained by amortizing the code generation cost [3], [4].

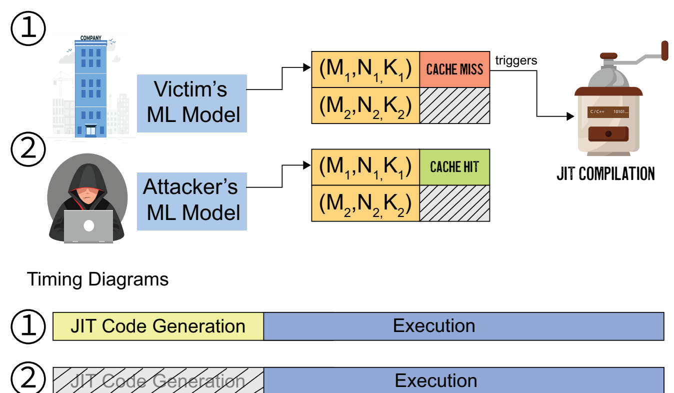


Fig. 1: The victim's DNN execution calls the GEMM library, passing model hyperparameters, which invokes the JIT process (step 1). Thus, subsequent calls by the attacker use the cached instructions, resulting in faster execution (step 2).

However, internal code caches introduce a new attack surface that has never before been exploited, mainly because

JIT-based optimization of DNN libraries is a fairly recent development. Due to the JIT code generation process, the execution of a benign user application can leave behind a trace of instructions that can be exploited later by an attacker to extract problem dimensions, as shown in Fig 1.

Our attack, JAXED (*Just-In-Time Axed*), is the first end-to-end demonstration of a JIT attack, demonstrating a complete hyperparameter extraction in the context of DNNs. One of the main goals of our work is to raise awareness of the potential vulnerabilities due to just-in-time (JIT)-optimized libraries, attack surfaces that may go unnoticed by developers. This class of attack has real-world implications, especially given the widespread use of open-source machine learning frameworks and JIT-optimized GEMM libraries.

Consider a scenario where a software vendor provides a face detection API service as a machine learning service to its users. Assume a user (victim) is performing speech inference using a DNN model, and another user (adversary) performs object detection through a separate DNN model.

The DNN models are not visible to each other; However, with such a configuration, when running DNN inference, the victim's model will make a GEMM call through the underlying ML framework (e.g., PyTorch or Tensorflow). This call would trigger JIT code generation for the specified model hyperparameters. In the context of GEMM, these model hyperparameters will map to dimensions M, N and K, which represent the number of rows of matrix $\mathbf{A}$, the number of columns of matrix $\mathbf{B}$ and the number of columns of matrix $\mathbf{A}$, respectively, in standard GEMM notation. The resulting instruction sequence will then be cached to provide faster servicing during future GEMM calls, using the same values for the (M, N, K) parameters. Thus, an adversary's model using the same (M, N, K) parameters would also observe faster execution, as shown in Fig 1 (step 2).

By combining the knowledge of ML model characteristics and timing measurements, an adversary can now reduce the search space for discovering the model hyperparameters. Thus, adversaries are able to steal valuable intellectual property associated with the ML model without any detectable privilege escalation nor tampering.

In our work, we consider scenarios where a legitimate user can act as a malicious adversary, performing a timing side-channel attack on the victim's model. We demonstrate an end-to-end strategy, showing how an attack is performed, and provide an example attack using PyTorch [6] and Intel's extension for PyTorch [7]. Our goal is to increase awareness among both library developers and DNN users about this new class of attack surface. We also consider potential mitigation strategies.

Our contributions in this paper include:

- We demonstrate a JAXED (*Just-In-Time Axed*) attack strategy by providing an example attack against fully connected layers and convolution layers using PyTorch and Intel's extension for PyTorch.
- We show that our attack works against the fully connected layers of 10 widely adopted DNN architectures used in production environments. We also show a complete attack against MobileNetV2.
- We discuss mitigation strategies from both the library developer's perspective as well as the DNN user's perspective.

II. Attack Scenarios

A. An Example Attack Scenario on ML Inference

In today's markets, we see companies specializing in providing machine learning as a service (MLaaS), delivered either through a platform or as a software package. Such services require off-the-shelf ML frameworks as the development effort required to build a customized ML framework is a significant undertaking. Therefore, most vendors rely on open-source ML frameworks and build their own frameworks on top of existing solutions. Our focus here assumes an ML vendor that provides an API for DNN inference. For the sake of discussion, let us assume that the underlying ML model is a DNN.

"Best DNN" is a private cloud ML vendor that provides an ML inference API service for users. "SNAPAPP" is a mobile application company that leverages the face detection API service to generate decorative artwork (e.g., image filters) on end-user mobile images. "TikTikAPP", a competitor of SNAPAPP, is a mobile application company, where users access this ML inference service to render colorful filters.

In this case, SNAPAPP (the adversary) does not have access to the underlying DNN model of TikTikAPP (the victim). However, since the underlying framework is shared, SNAPAPP can expose a timing side channel through JIT-optimized GEMM libraries. The information leakage is due to the internal software cache of JIT-optimized GEMM libraries, as we will see in the next sections.

The above approach can be deployed via two possible privilege scenarios: i) API access (read privilege) or ii) private cloud access (read + write privilege):

API access: SNAPAPP only has access to the API endpoint and waits until the victim sends an API request to the corresponding endpoint. In the case where DNN primitives (operations) are shared between the two models, SNAPAPP would observe a significant reduction in execution times. Thus, using our JAXED attack strategy, the adversary can exploit the existence of a side channel vulnerability and narrow down the search space of shared primitives.

Private cloud access: Although previous work [8]–[10] has demonstrated that co-location vulnerabilities can be exploited in the public cloud, modern infrastructure has been made more secure and co-location on a public cloud environment has become almost impossible. However, due to the fast pace in the ML domain, there is a higher probability of such co-location vulnerabilities being present in a private cloud environment.

In such a private cloud environment, assume a server such as Torchserve [11] or Multi Model Server [12] is hosting SNAPAPP and TikTikAPP's models. SNAPAPP will not have direct visibility into TikTikAPP's model execution due to restricted permissions. However, since the ML framework is shared within the same process, the underlying library code cache

timing side channel can be exploitable by SNAPAPP. In this scenario, the adversary is able to extract the hyperparameters through DNN design space exploration by modifying her own model and monitoring the execution time.

Compared to prior works on JIT-based attacks [13], our attack is extremely practical since we do not rely on fine-grained control. Nor do we assume observability over the victim's model execution, but rather rely on observing the adversary's own execution. Having introduced the attack scenarios, next, we formally present our threat model.

B. Threat Model

We develop a timing attack that employs JIT-induced cache timing side-channels to reduce the search space and narrow down candidate guesses for DNN hyperparameters.

Black-box Access - We employ a black-box threat model [14], [15]. The victim's DNN model hyperparameters and weights are not visible to the adversary and cannot be extracted due to in-built security mechanisms. The adversary also does not have the luxury of any observability over the victim's execution and can only measure her own actions.

Shared Platform - We assume that the underlying ML framework (and accompanying GEMM library) is shared between the victim's DNN model and the adversary's DNN model within the same process space. In the context of JIT security attacks against JVMs, Brennan et al. [13] demonstrated that this is feasible, as shared services are provided through the cloud.

Timing Information - The attacker should be able to gain fine-grained timing details for her own DNN model. The layer-wise information will enable the attacker to map candidate guesses to the victim's execution easily. Our attack does not assume any observability over the timing of the victim.

III. JIT-OPTIMIZED GEMM

This section will look at the fundamentals behind JIT-optimized GEMM libraries and how they potentially expose a new attack surface.

A. Software Architecture

For small and medium-sized matrix multiplications, traditional libraries do not deliver the best possible performance. The reason is that well-known and established BLAS libraries, such as Intel MKL [16] and OpenBLAS [17], are tuned for large problem sizes. Traditional GEMM libraries tune memory accesses to best leverage the available memory hierarchy.

LIBXSMM [2] is a GEMM library designed to address this problem, providing runtime code generation. The availability of new CPU instruction set extensions, such as AVX-2 and AVX-512, have paved the way towards more aggressive optimizations. However, at compile-time, the runtime problem dimensions are typically unavailable since the GEMM library is often dynamically linked during program execution. When using run-time code generation, aggressive optimizations that depend on problem dimensions can be performed. We will analyze the architecture of this library, as it is open-source, and will help us understand the internals of JIT-optimized GEMM libraries. A simplified view of the software architecture of LIBXSMM is shown in Fig. 2.

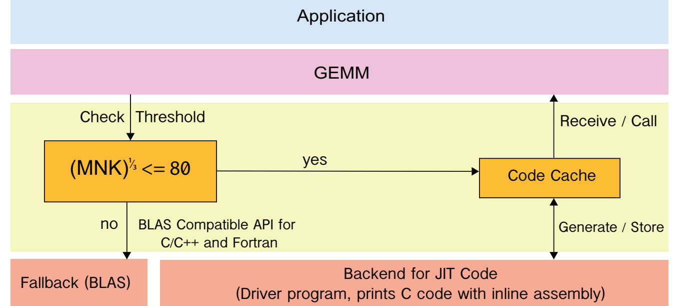


Fig. 2: Software architecture of the LIBXSMM library.

The application invokes the LIBXSMM library through the GEMM API. For example, if the application works with single-precision data stored in matrices floating-point numbers, the call would be SGEMM (compatible with the standard BLAS API). The library would first check M, N, and K values and evaluate whether $M \times N \times K \leq 80$. According to the developers of LIBXSMM [2], this threshold has been chosen based on the L2-cache size, ensuring that the matrices will fit nicely in the memory hierarchy. The threshold plays an important role, as it determines which GEMM library is used for the computation.

- If the threshold is met, LIBXSMM checks for the availability of optimized code in the code cache. If the optimized code is not available in the code cache, it will trigger the backend to run code generation. LIBXMM maintains a code registry and a small code cache to hold the most recently used GEMM kernels.
- If the threshold is not met, the library invokes a traditional BLAS library such as MKL BLAS or OpenBLAS because the library determines that BLAS would perform faster than LIBXSMM.

B. Comparison Between Existing JIT GEMM Libraries

In terms of the scale of this attack surface, is this present in all ML model execution? The Intel MKL library provides a JIT-optimized GEMM method for small and medium-sized matrix multiplications [4]. Intel's OneDNN [5] also provides a JIT-optimized GEMM method for ML applications. Both of these libraries provide two methods for integration: 1) using a conventional BLAS API and 2) using a customized JIT API. When using a conventional API, code cache management is handled by the library, whereas with the customized JIT API, the user is responsible for code cache creation and destruction. Similarly, LIBXSMM provides a conventional BLAS API, which allows the user to fully utilize the library without requiring extensive modifications to the original C/C++ application. The built-in software code cache removes the burden

of explicit memory management in the application from the user. Therefore, all GEMM calls to the extension are directed through a shared code cache (in LIBXSMM) or a shared JIT engine (in oneDNN).

C. JIT Code Cache Timing Behavior

For an attack to be successful, there should be a noticeable difference in the execution time between different input parameter settings. We perform an investigation, establishing the viability of our attack in the context of LIBXSMM, and compare against a traditional BLAS library [17]. Fig. 3 shows the execution time when only the adversary is executed (hereafter referred to as an "isolated run"), and when the victim would have executed before the adversary's execution with the same parameters as the victim (referred to as a "shared run [similar params]." We will use the terminology "shared run" when the adversary executes after the victim execution (irrespective of parameters). In a shared environment, the JIT GEMM library will result in a faster execution time when run with similar parameters. However, in the absence of JIT optimizations, we would not observe this phenomenon.

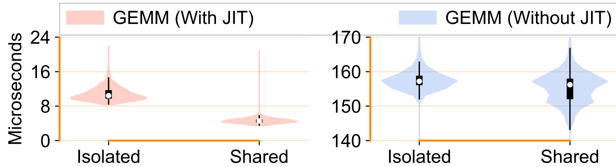


Fig. 3: LIBXSMM (with JIT) vs. OpenBLAS (without JIT). In a shared environment (with similar parameters), JIT optimizations will be performed, resulting in faster execution due to JIT-optimized GEMM.

Next, to evaluate the viability of our attack, we run an experiment with the LIBXSMM library, where the attacker attempts to execute a matrix multiplications of different dimensions in a shared environment. Our goal is to distinguish the difference in execution time when the adversary's parameters are the same as the victim's (code cache hit timing), versus when the parameters are different from the victim (code cache miss timing). The difference between the two is the time required by the JIT library to compile code and store it in cache (code cache miss penalty).

In Fig. 4, we observe the aforementioned difference (from the attacker's perspective) between the execution times of the different sized matrix multiplications, where $M = N = K$ in standard GEMM notation. We test each matrix configuration 1000 times and present the average execution times. The attacker experiences a significant difference in execution time in a shared environment where the parameters (i.e., when M, N, K) are the same as the victim's parameters. Fig. 4 shows the difference in execution time of the attacker when parameters are the same as the victim (a code cache hit) and when parameters are different (a code cache miss).

In the next sections, we will describe how the behavior of JIT-code generation, combined with the internal software code cache, can be exploited to reveal sensitive model hyperparameters.

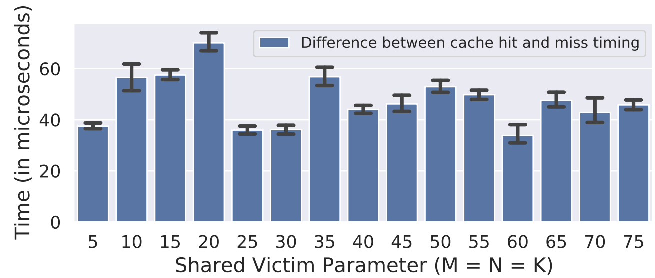


Fig. 4: Code cache timing attack: Bar plot showing the average difference in execution time between a code cache miss and a code cache hit.

IV. JAXED Attack Strategy

We will first explain how an adversary would perform a timing side-channel attack with the knowledge of the ML framework and the underlying JIT-optimized GEMM library to extract model hyperparameters.

A. Attack Strategy

We provide a general description of the attack to help conceptualize the attacker's strategy. The adversary first needs to understand the targeted system's behavior in the absence of JIT-optimized kernels. Thus, she first selects a set of educated guesses. For example, the attacker can select a candidate set containing powers-of-two, as these are frequent choices in popular DNN designs [18]–[26]. For each candidate guess, the attacker profiles the execution time on her system (referred to as an isolated run). Next, the adversary waits for the execution of the victim's ML model so that the corresponding JIT code cache will be filled in the library. Finally, the attacker runs her model with the predicted parameters and observes the execution runtime (referred to as an observed run). Having obtained the isolated vs. observed runtimes over many iterations, the attacker computes a score. The score is computed by determining how much faster each observed runtime is compared to the isolated runtime, as shown in Eq. 1. The candidate hyperparameters corresponding to the highest score (i.e., largest difference) should identify the victim model's hyperparameters. In a JAXED (*Just-In-Time Axed*) attack, the adversary relies solely on observations of her own behavior and not the victim's behavior, which may be impossible in some circumstances.

$$\text{score} = \frac{\text{difference in execution time}}{\text{isolated runtime}} \% \tag{1}$$

B. A Successful JAXED Attack

Having presented an overview of the attack steps, we will discuss why our attack strategy works. First, the adversary waits for the victim's model execution. We will see the execution steps involved and the artifacts left behind as a result of the execution.

As seen in Fig. 5, when executing the vendor's ML inference library, in step 1, the binary is loaded. The weights of the DNN are subsequently loaded in step 2. As the execution proceeds, when the ML model computes a convolution layer or a fully connected layer, the underlying GEMM driver will be invoked in step 3. The functionality of the GEMM API is encapsulated within the PyTorch framework.

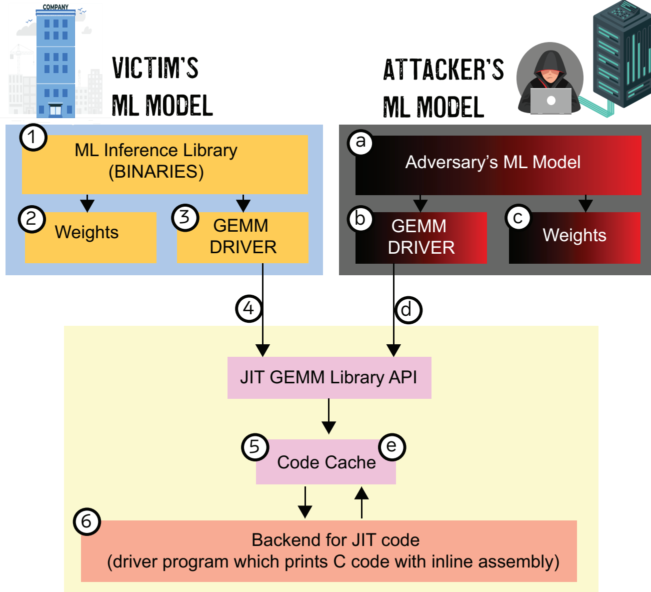


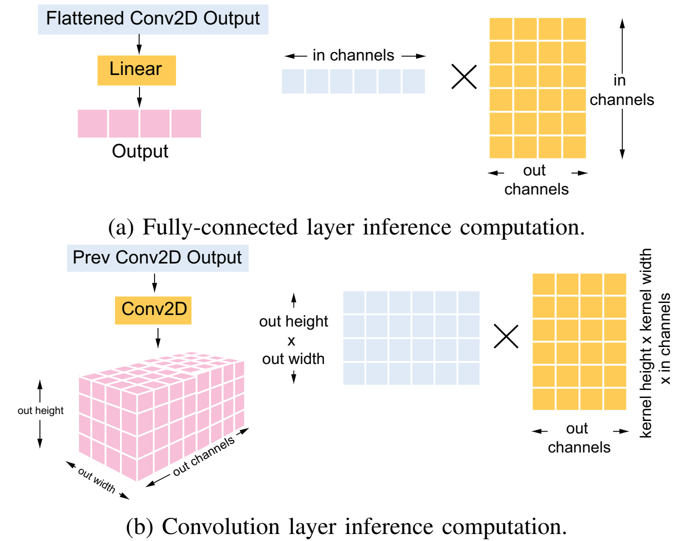
Fig. 5: Attack Surface: After the victim's execution, the victim leaves behind information about its model hyperparameters in the JIT code cache. The attacker probes this JIT code cache through the attacker's ML model and observes timing information to determine the victim's model hyperparameters.

In this case, we will assume that the PyTorch framework has been configured to use the LIBXSMM library. The GEMM API would be called in step 4, resulting in a code cache lookup in step 5. If the code cache does not contain the GEMM kernel (i.e., a cache miss), the back-end is invoked, and the code generation is performed in step 6. The victim's model execution will leave behind an execution trace through the JIT-optimized code stored in the cache. The attacker can now investigate the timing behavior to identify the secret model hyperparameters.

In step a (shown in the attacker's ML model in Fig. 5), the attacker builds a hypothesized model. For example, if the attacker is trying to guess the parameters of a fully connected layer, the attacker will develop a similar model with a configurable layer size. The attacker would then use random weights in step c and rely on the GEMM driver in step b to perform inference. The GEMM call will be directed to the JIT library. In this scenario, if the attacker guesses the same model hyperparameters as the victim, the call will be serviced through the code cache in step e (i.e., a code cache hit). Therefore, the attacker will observe a decrease in the expected execution time for the correct parameter guess, as the back-end for JIT code generation will not be triggered. By measuring the difference in execution time, the attacker will be able to verify her hypothesis.

V. Experimental Methodology

Next, we discuss the relationship between the GEMM calls and the model hyper-parameters. Based on the specific layer type, we found that using different guessing mechanisms was warranted. We will first discuss the parameters of each layer and how an educated guess could help reduce the candidate set space.



(a) Fully-connected layer inference computation.

(b) Convolution layer inference computation.

Fig. 6: Fully-connected layer and convolution layer computations. Each computation can be formulated as a matrix multiplication, performed using a GEMM library.

A. Parameters for the Fully Connected Layers

FC layers have two parameters: 1) the input channels and 2) the number of neurons (also referred to as output channels). The activation input to an FC layer is flattened into a one-dimensional structure. Thus, the size of the input would be equal to $1 \times$ input channels. As shown in Fig. 6a, this would result in a multiplication between the flattened input and the FC layer weights, which can be thought of as a matrix with the dimensions input channels $\times$ output channels.

B. Parameters for the Convolution Layers

A convolution layer has eight parameters: 1) the input height, 2) the input width, 3) the input channels, 4) the number of filters (also referred to as output channels), 5) the kernel height, 6) the kernel width, 7) the padding and 8) the stride. A convolution layer first transforms the activation input into a column format using a technique widely known as *im2col*. This is done in order to use optimized BLAS methods provided by widely available GEMM libraries. In a 2D convolution, the output size of the 2D input can be computed as follows.

$$\text{out dim} = \frac{(\text{input dim} - \text{kernel dim} + 2 \times \text{padding})}{\text{stride}} \quad (2)$$

Having computed the output size, the *im2col* is performed. A single channel is transformed, such that the multiplication of the activation matrix (input) with the filter weights matrix (layer) results in the convolution output. If there are multiple input channels, the same operation is performed for each activation.

C. Attacking FC Layers

Fig. 7: Deep Neural Network diagram. Final layer is often a Linear (FC) layer in order to map to a probability vector.

Let us assume the attacker is attempting to detect the parameters for the FC layers in the DNN model shown in Fig. 7. Most DNN's (in the context of image classification) employ an FC layer at the very end of the model, mapping the DNN result to a probability vector, selecting the most probable object class and object boundaries. Thus, the easiest way to begin an attack would be to target the final FC layer. Since the output vector specifications are provided to the user ("SNAPAPP") by the ML vendor ("Best DNN"), one of the hyperparameters (i.e., the number of neurons) for the FC layer is already known. Therefore, the adversary only needs to perform a linear sweep to identify the other hyperparameter (number of input channels). We detail the FC layer attack algorithm in 1.

Algorithm 1 FC Layer Attack Algorithm.

Input: c - out dimension
Output: input channels (i)
1: I = {input channel candidate set}
2: **for** i $\in$ I **do**
3:     record the run-time for (c, i) ▷ Profiling step
4: Run the victim model ▷ Inference Step
5: **for** i $\in$ I **do**
6:     record the run-time for (c, i) ▷ Attack step
7: compute the time difference $\forall$ (i) $\in$ I
8: sort and find the maximum time difference
9: return corresponding (i)

D. Attacking Convolution Layers

For the convolution layers, the guess involves simultaneous parameter guesses due to the complexity of this layer. Assume that the attacker has successfully identified the number of input channels for the FC layer, as shown in Fig. 7. This means that the flattened output of the previous convolution layer is known by the attacker (because the output of the convolution layer becomes the input to the FC layer). The adversary first decides the range of kernel sizes, which will be between 1×1, 3×3, 5×5. Almost all modern DNNs limit themselves to these kernel sizes, given the high computational overhead with large kernel sizes in convolution layers [19]. Next, the adversary guesses the input stride and the padding size. The stride value is limited to either 1 or 2. Because as the stride size increases, most of the layer input is discarded. The padding value is dependent upon the kernel size and the decision of whether to use padding. Most DNN's use padding to avoid the destruction of boundary pixel information in images.

Having established guesses for these parameters, the adversary builds a candidate guess set, consisting of the number of input channels and the number of filter combinations. The attacker then iterates through the combinations to identify the correct combination. We have observed that when using an incorrect kernel size, incorrect stride value, and incorrect padding assumptions, the success rate of the attack drops below 10%. This provides a good indication that the adversary needs to revise the initial assumptions and re-attempt the attack. In the general case (if initial assumptions on kernel sizes and layers are observed to be contradictory), the adversary must increase the search space to include more unconventional DNN designs.

Since the attacker already knows the flattened output shape, the attacker can devise an algorithm to map hyper-parameters. Usually, images used for inference are square in shape, So the out height$\times$out width can be considered as the out dim^2. By figuring out the perfect squares (i.e., n $\in \mathbb{N}$, such that $\exists$ i $\in \mathbb{N}$, $n = i^2$) that are less than the output shape size, the adversary can build a set of all possible candidate guesses for out dim^2. But the attacker is more interested in the number of filters in the layer. Thus, the attacker converts the set of guesses built for the output dimension into guesses to find out the number of filters. This can be done by dividing the output shape by the guesses for the output dimension and choosing the result only if the remainder is zero. We summarize our algorithm for a successful convolution attack in Algorithm 2.

Algorithm 2 Convolution Layer Attack Algorithm.

Input: m - FC input dimension (or flattened input dimension of the subsequent layer)
Output: input channels (i), number of filters (c)
1: I = {input channel candidate set}
2: S = {s | perfect squares $\leq$ m, s $\in \mathbb{N}$}
3: C = { c = $\frac{m}{s}$ | m is divisible by s $\in$ S} ▷ Compute candidate set for number of filters
4: **for** c $\in$ C **do**
5:     **for** i $\in$ I **do**
6:         record the run-time for (c, i) ▷ Profiling step
7: Run the victim model ▷ Inference Step
8: **for** c $\in$ C **do**
9:     **for** i $\in$ I **do**
10:         record the run-time for (c, i) ▷ Attack step
11: compute the percentage time difference $\forall$ (c, i) $\in$ C $\times$ I
12: sort and find the maximum time difference
13: return corresponding (c, i)

E. Discovering DNN Network Architecture

The attacker will begin the attack from the last layer and will proceed to detect subsequent layers. For each layer-wise attack, the adversary will assume a layer type (e.g., FC layer or convolution layer) and will verify the correctness of her hypothesis through experimentation. We will show how this technique can be used to recover the complete architecture of a DNN.

VI. Results

To demonstrate our attack, we use a system with an Intel(R) Xeon(R) W-2295 CPU (hyper-threading enabled) with AVX-512 extensions. The system is running Red Hat Enterprise Linux release 8.4. We use the latest PyTorch version 1.7.0, supported by Intel extensions for PyTorch [7]. To set up PyTorch, we follow the help guide and make no modifications to the PyTorch framework, nor to the Intel extensions library, other than those specified in the help guide. We demonstrate our examples using Python, and the same principles can be extended to compile-time languages such as C++.

A. Qualitative Comparison Against Existing Attacks

We compare JAXED with a state-of-the-art JIT attack (JIT Leaks [13]) and state-of-the-art DNN parameter extraction attack in the context of GEMM (Cache Telepathy [14]). TABLE I shows that JAXED is the only attack which is able to perform a successful JIT-based DNN attack in the context of JIT optimized GEMM libraries.

B. Attacking Fully Connected Layers

We will first look at attacks on fully connected layers as they are easier to attack than the convolution layers. Consider a toy victim model which has a final output vector of size 1×200. Assume that the size of the victim's input channels to the last FC layer is 25, which is the secret hyperparameter being targeted by the adversary. The adversary would iterate through the candidate set and record the execution time difference, as shown in Fig. 8. In order to identify the secret parameter, the adversary would compute the score by using Equation 3. The value corresponding to the largest time difference percentage-wise would reveal the secret hyperparameter. Fig. 8 shows the time possessing the largest difference for an input dimension of 25. Thus, the adversary can figure out the secret hyperparameter quite easily.

Next, we will explore the prediction accuracy over a larger search space. We select two boundaries for the output shape size, 10 (representing CIFAR-10 [27]) and 1000 (representing ImageNet [28] datasets). We select increments of 100 and choose the secret parameter (number of input channels of the FC layer) to be within $[2, 2^{10}]$, since most CNNs opt to have parameters as powers-of-two. Due to the variability in execution times, we perform the experiment 100 times and compute the success rate. Fig. 9 shows the mean success rate over 100 iterations. Our analysis has revealed that the mean accuracy rate is 92.3%, and the mean lies between [92.1%, 92.4%], with a confidence level of 95%.

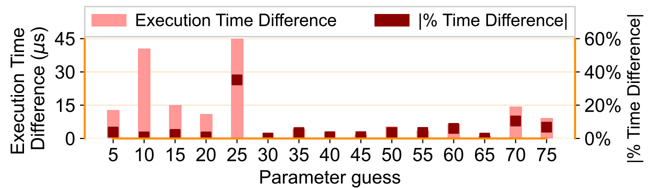


Fig. 8: Difference in execution times when the attacker executes their own model in isolation and when their model is executed after the victim (a higher value is better). When the attacker observes the largest difference, this indicates that the code cache contains the JIT generated code.

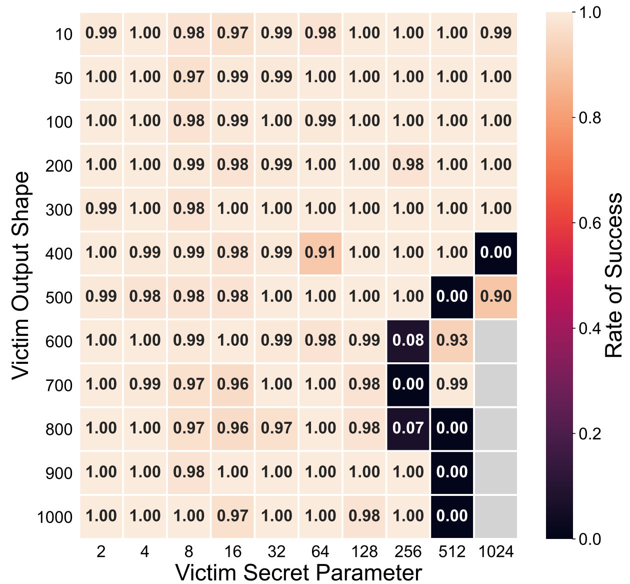


Fig. 9: Mean success rate of different output shape and secret parameter combinations. A rate of 1.00 indicates that our algorithm was able to successfully determine the victim's secret parameter for all 100 experiments.

As seen in Fig. 9, our attack strategy seems to produce inconsistent results when the victim's secret parameter is ≥ 256 and the output shape is ≥ 400. We analyzed the relationship between the success rate and the mean JIT code generation time, computed as a fraction of the total execution time. In Fig. 10, we can observe a clear relationship between the proportion of time spent on code generation time and the rate of success. The median $\frac{\text{mean JIT code generation time}}{\text{mean execution time}}$ of 0.22 represents the point were predictability diverges. For computations below this value, the success rate is low (the group marked with blue x's in Fig. 10). However, for computations above 0.22, the success rate is very high (the group marked with red o's). When the JIT code generation time is only a small portion of the total execution time, the timing characteristics are no longer dominated by the code cache available within the library and instead dominated by other factors, including

TABLE I: Qualitative comparison between JAXED and existing attack strategies.

Criteria	JIT Leaks [13]	Cache Telepathy [14]	JAXED
Targeted software	JVM / Virtual Machine	Intel MKL / OpenBLAS	oneDNN
Context	Runtime JIT optimizations performed by JVM and virtual machines on hot paths.	GEMM libraries (without JIT) utilized by DNN frameworks.	GEMM libraries (with JIT) utilized by DNN frameworks.
Side channel	Timing side channel.	Hardware cache side channel.	Timing side channel + software cache side channel.
Vulnerability	Timing side channel attack on optimized hot path execution.	Cache side channel attack on GEMM instructions.	Timing and Software cache side channel attack on runtime optimized GEMM calls stored in the code cache.
Attack DNN frameworks?	✗	✓	✓
Dynamic instruction sequences?	✓	✗	✓
Open source?	✓	✗	✓

the cache hierarchy, operating system behavior, and PyTorch framework characteristics. Therefore, the rate of success drops when a lower percentage of the total time is spent in code generation.

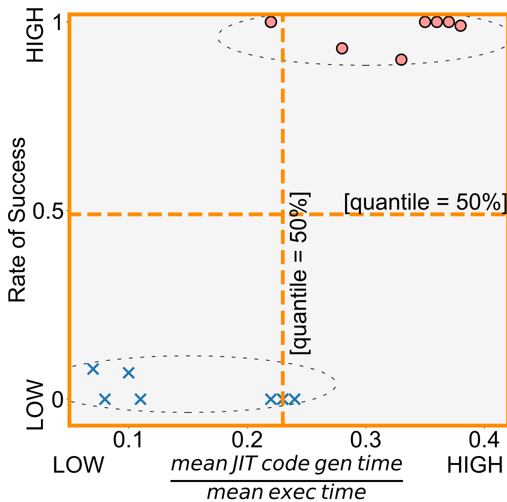


Fig. 10: Outliers in Fig. 9; The points correspond to output shape $\geq$ 400 and victim parameter $\geq$ 256.

C. Attacking Convolution Layers

As we did in the case with the FC layers, let us first consider a toy example. Consider a victim model that has a convolution layer followed by an FC layer. Assume that the attacker has already discovered the secret parameter of the FC layer by following the attack steps described earlier. Assume that the convolution layer has a kernel size of 3×3, a stride equal to 1, and uses padding, such that the activation height and width would remain constant.

With reference to our detailed convolution attacks in Section V, the input dimension of the FC layer is already known (which is the subsequent layer). Therefore, we can compute the FC's input dimension as follows:

$$\text{FC input dimension} = \text{out dim}^2 \times \text{number of filters} \tag{3}$$

We can use this information and Algorithm 2 to deduce the number of input channels and the number of filters. Fig. 11 shows a set of different input channels and number of filters (output channel) combinations. We identify the largest difference in execution times to correctly identify the convolution layer secret hyperparameter pair: the number of input channels and the number of filters.

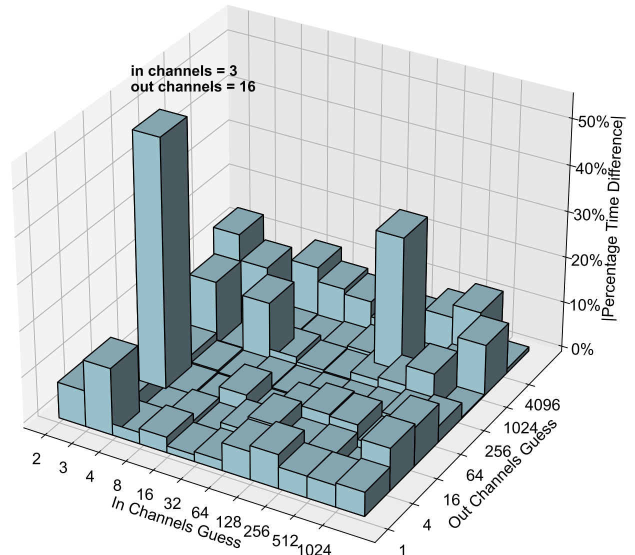


Fig. 11: Convolution layer attacks require two simultaneous guesses: the number of input channels and the number of filters in a layer. The highest bar shows the correct prediction of the victim's number of input channels and number of filters.

D. DNN Evaluation

We first select 10 popular DNNs, all trained with the CIFAR-10 dataset [27]. The reason for the dataset selection is to ensure that the JIT GEMM library operates within its specified range. For larger datasets and network architectures, the JIT GEMM library backend would switch to MKL BLAS, rendering the attack ineffective.

Table II shows the success rate of our attack against the last layer (fully connected) of ResNet-18, ResNet-34, ResNet-50 [20], VGG16, VGG19 [19], MobileNetV2 [26], DenseNet121, DenseNet169 [22], Inception v3 [23] and GoogLeNet [24], all trained on CIFAR-10 [27]. To ensure that our results are reproducible, we have used models available in the open-source domain. As we can see, independent of the model selected, our attack strategy achieves a success rate of 99%-100% across a range of popular DNNs.

Next, we attempt an attack on MobileNetV2 to discover the complete network architecture. First, we review the architecture of MobileNetV2, provided in Table III. MobileNetV2 introduces the concept of a bottleneck, also referred to as an inverse residual block [26]. A bottleneck layer consists of a 1×1 convolution (a point-wise convolution), followed by a 3×3 depth-wise convolution, and finally followed by another 1×1 point-wise convolution depth-wise convolutions are detected by the PyTorch framework when the number of input channels is equal to the number of groups specified by the user [29]. PyTorch will take a different execution path, one that does not involve GEMM. However, for point-wise convolutions, the execution path is directed through GEMM, as it results in faster convolutions. Due to the presence of point-wise convolution operators in all bottleneck structures, we are able to successfully extract hyperparameters end-to-end.

Figure 12 (left) shows the success rate for each convolution and fully connected layer when considering the top-1 candidate. This means that the attacker computes the score and selects the candidate to guess which ranked most probable. The success of the attack is measured by the percentage of selected guesses that matched the secret hyperparameters of the layer being considered. In Figure 12 (left), we observe that, for a subset of convolution layers, the attacker observes a very low success rate. This is due to the fact that when the attacker is iterating through all possible candidate guesses, multiple code cache entries are being activated (since multiple convolution layers have cached GEMM parameters in the library), as shown in Table IV.

Therefore, the attacker can modify the scoring mechanism to select the top-3 candidate choices for each layer instead of selecting the top-1 choice, as shown in Fig. 12 (right). The success rate of a top-3 choice can be verified by summing up the horizontal lines in Table IV. For example, consider conv2d $[\mathbf{4^2}, \mathbf{384}, \mathbf{96}]$. For the row that corresponds to 384 input channels and 96 filters shows that the combination was ranked as the second choice in 14% of the experiments and ranked third in 63% of the experiments. Therefore, [384, 96] achieves a 77% [=63% + 14%] success rate when considering the top-3 choices. This shows that by selecting the top-3 choices, the adversary can successfully narrow down the search space from over the 1300 possible combinations for each layer, reducing the number to just 3.

TABLE II: Last Fully Connected Layer Attack Results.

DNN	Input Channels	Output Channels	Success rate
ResNet-18	512	10	99%
ResNet-34	512	10	99%
ResNet-50	2048	10	99%
VGG16	4096	10	99%
VGG19	4096	10	100%
MobileNetV2	1280	10	99%
DenseNet121	1024	10	100%
DenseNet169	1664	10	100%
Inception v3	2048	10	100%
GoogLeNet	1024	10	99%

TABLE III: MobileNetV2 Architecture. Each line describes a set of 1 or more identical layers, repeated n times. All layers in the same set have the same number of output channels - c. The first layer of each set has a stride s and all others use stride 1. Expansion factor t is used for spatial convolution.

Input	Operator	t	c	n	s
$32^2 \times 3$	conv2d	-	32	1	2
$16^2 \times 32$	bottleneck	1	16	1	1
$16^2 \times 16$	bottleneck	6	24	2	1
$16^2 \times 24$	bottleneck	6	32	3	2
$8^2 \times 32$	bottleneck	6	64	4	2
$4^2 \times 64$	bottleneck	6	96	3	1
$4^2 \times 96$	bottleneck	6	160	3	2
$2^2 \times 160$	bottleneck	6	320	1	1
$2^2 \times 320$	conv2d 1×1	-	1280	1	1
-	dropout 0.2	-	-	-	-
$1 \times 1 \times 1280$	linear	-	10	-	-

VII. Mitigation Strategies

In this section we will discuss mitigation strategies that can be followed to avoid such attacks on valuable intellectual property. We will discuss them in the order of simplicity in implementation.

A. *User Awareness*

Currently, JIT based optimizations in the context of GEMM libraries remain at a very early stage and the exposure of new threat surfaces is not widely known. More user awareness provided by library developers through the documentation will be beneficial in identifying the exposure to timing side channel attacks. Reporting vulnerabilities, such as JAXED, helps, will alert library users of this class of potential exploits.

B. *Explicit Code Cache Flushing*

Almost all ML frameworks provide a device level abstraction to run the same DNN on a CPU, GPU or a custom accelerator. The framework level can be modified to include device specific initializer and a destroyer so that internal library caches can be explicitly flushed by the framework to avoid a timing leakage. For example, if PyTorch detects the availability of the x86_64 architecture with AVX-512 and the presence of a JIT GEMM library, device specific actions can be activated.

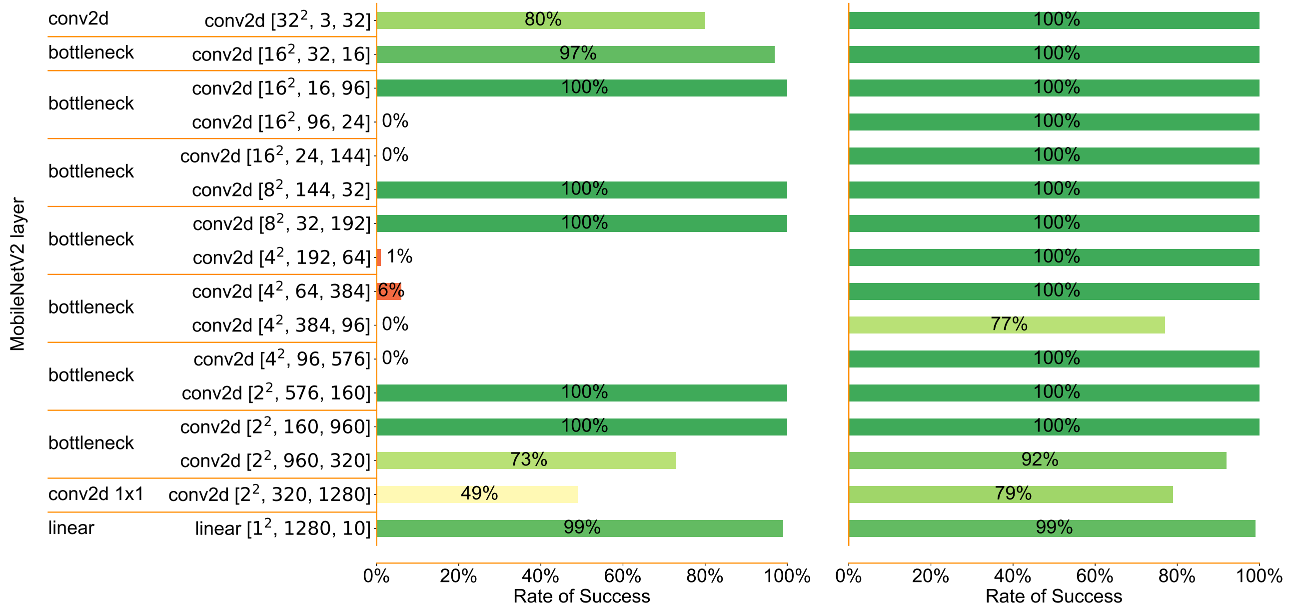


Fig. 12: Attack to discover model parameters of MobileNetV2. Our end-to-end attack attempts to recover model dimensions of each layer. The Top-1 success rate (left) and the Top-3 success rate (right).

C. *Modifying Library Design*

Although an internal code cache improves the user friendliness, had the cache management responsibility been transferred to the user, the transparency of JIT GEMM libraries would be improved. Therefore, in terms of security, the best practice would be to provide a pointer to the generated code and the user will have to build her own software caches. Thus, the user would be responsible for creating any threat surfaces and not the library.

VIII. Related Work

In the context of recent work on security attacks, we focus on the domain of JIT compilation and interpreted languages where optimization is performed. We begin by reviewing related work on cache timing side-channel attacks and GEMM-based attacks on ML models.

A. *Prior work on attacking JIT behavior*

JIT compilation techniques have been used in the domain of interpreted languages in order to execute programs using an interpreter (e.g., Python) or to use a virtual machine (e.g., Java). Modern JIT compiler implementations involve techniques to dynamically adjust the optimization level. Page [30] first reported that JIT compilation optimizations could lead to timing channel vulnerabilities. In this work, Page presents a case study on a Java implementation of a double-and-add-based multiplication program, highlighting the timing difference in the compilation. Based on this observation, Brennan et al. [13] introduced how non-uniform input distributions could induce a side channel in a Java virtual machine / Javascript engine. This information can be used by an adversary to infer sensitive information and actively exploit the JIT's focus on optimization. Based on Brennan et al.'s classification, our attack closely resembles a natural-priming model, a model where the attacker times their own probing call. However, it differs from this definition given that we do not explore the majority observation (which leads to an optimized execution path) but rather explore the availability of an instruction sequence in the cache managed by the JIT compilation. Our focus also differs in that our attack targets JIT-optimized GEMM libraries and DNN applications. In that sense, we introduce a novel side-channel attack surface, where JIT compilation employs an in-built cache for optimization, in contrast to a "hot" path execution optimization targeted by the related work.

Interestingly, countermeasures for JIT run-time attacks have also been proposed. Cleemput et al. [31] reported that statically compiled programs could still have side channels present at runtime because side channels present that are related to the processor pipeline behavior or device availability (online/offline) cannot be addressed unless a dynamic strategy is adopted. In this work, they demonstrate the use of JIT compilation to mitigate timing side channels by proposing to collect application profiles across a range of inputs in a predetermined training set. Each profile collects information about the program call graph. Then the methods most vulnerable to timing attacks are selected. Following that, the control flow transformations (e.g., if-conversion) and the data flow transformations are applied to protect critical functions.

There is prior work on transformation approaches that try to defend against timing side channels on modern x86 pro-

TABLE IV: The success rate of the top-3 candidate choices for the secret parameters of each convolution layer. C and f denotes the number of input channels and the number of filters, respectively. R0, R1 and R2 denote that the choice was ranked as first, second or third.

Layer	c	f	R0	R1	R2
conv2d [$\mathbf{32^2}$, **3**, **32**]	**3**	**32**	80%	20%	0%
	320	512	20%	72%	5%
	576	128	0%	6%	60%
conv2d [$\mathbf{16^2}$, **96**, **24**]	16	96	100%	0%	0%
	144	24	0%	68%	32%
	96	**24**	0%	32%	68%
conv2d [$\mathbf{16^2}$, **24**, **144**]	32	16	100%	0%	0%
	24	**144**	0%	100%	0%
	512	576	0%	0%	59%
conv2d [$\mathbf{4^2}$, **192**, **64**]	32	16	99%	0%	0%
	192	**64**	1%	99%	0%
	576	1,024	0%	1%	97%
conv2d [$\mathbf{4^2}$, **64**, **384**]	16	96	94%	6%	0%
	64	**384**	6%	94%	0%
	144	24	0%	0%	92%
conv2d [$\mathbf{4^2}$, **384**, **96**]	16	96	100%	0%	0%
	96	24	0%	86%	14%
	384	**96**	0%	14%	63%
conv2d [$\mathbf{4^2}$, **96**, **576**]	32	16	100%	0%	0%
	96	**576**	0%	87%	13%
	24	144	0%	13%	87%
conv2d [$\mathbf{2^2}$, **960**, **320**]	**960**	**320**	73%	13%	6%
	24	1,280	3%	13%	11%
	32	1,280	0%	1%	1%
conv2d [$\mathbf{2^2}$, **320**, **1280**]	**320**	**1280**	49%	21%	9%
	4	1,280	7%	3%	4%
	3	1,280	1%	0%	4%

cessors [32], [33]. Cleemput et al. [31] combine JIT profiling with these mitigation approaches to prevent adversarial timing attacks. Randomization of the control flow, which involves generating unique execution paths for each program, is an effective approach proposed by Crane et al. [34] to avoid timing-based side-channel attacks through JIT compilation.

Frassetto et al. [35] show that by using Intel's Software Guard Extensions (SGX), the JIT code compiler can be isolated, eliminating the possibility of code injection, code-reuse, or data-only attacks [36], guarding against multiple threats to software security [37]. However, their approach does not consider side-channel attacks that take a passive (i.e., observatory) approach rather than a tampering approach.

B. CPU induced side channels during runtime

Cache side channels rely on program-dependent behavior and the use of the CPU hardware caches to perform side-channel attacks. Cache-based side-channel attacks have been shown to have a widespread impact due to the growing dependence on cloud resources [38]–[42]. Acıiçmez et al. [43], [44] first demonstrated that the CPU's branch predictor could be used to perform timing attacks [43], [44]. Many classes of attack have been demonstrated that employ CPU-induced side channels [45]–[49].

Building on prior work on cache side channels, Yan et al. show that the same principles can be applied in other contexts, such as GEMM (General Matrix Multiplication) execution [14]. By carefully targeting specific instructions in the GEMM library, Yan et al. show that matrix dimensions can be extracted [14]. Cache side-channel attacks can be deployed against DNN models that utilize GEMM to perform convolutions. If DNNs are deployed in a shared cloud environment, this exposes an attack surface where an adversary can extract features and model dimensions that may be a valuable intellectual property of the organization. Yan et al. employ a hardware cache side-channel attack where the attack strategy is based on flush+reload [42] or prime and probe. In addition to model hyperparameter extraction, there has also been an interest in the extraction of model parameters (weights) from DNN models [50]. In the context of ML models (not limited to DNNs), much research has been done on model extraction [15], [51], [52]. Compared to existing hyperparameter estimation techniques [51], approaches that require knowledge of the training dataset, JAXED exposes a new attack surface which does not need knowledge of the details of the training dataset.

Recently, due to the diminishing returns on optimization on CPUs, a trend has emerged where matrix multiplications for small and medium matrices are optimized through JIT-based code generators. These optimizations can employ new architectural advances, such as Intel's AVX2 and AVX-512 extensions [2]–[4]. These libraries have already been integrated into existing machine learning frameworks, such as TensorFlow [53] and PyTorch [6].

In our JAXED attack, the first successful attack of its kind, we show that JIT optimizers that leverage code caches can expose behavior in a JIT environment which can be easily exploited by an adversary. We demonstrate a real attack using the PyTorch framework and successfully extract the model hyperparameters of a complete DNN. We also show that many existing DNNs are vulnerable to information leakage with the introduction of JIT-optimized GEMM libraries.

IX. Conclusion

In this paper, we described and demonstrated a novel timing attack on JIT-optimized GEMM libraries, successfully extracting model hyperparameters. Although previous research has reported the possibility of extraction of sensitive information in JIT environments, we are the first to demonstrate an end-to-end hyperparameter extraction in the context of DNNs using this new side-channel attack. We believe that our work will educate both library developers and model users of the existence of such threat surfaces and motivate new security research in JIT-optimized GEMM libraries.

Acknowledgment

We would like to thank our shepherd Tianwei Zhang, and the anonymous reviewers, who provided thoughtful feedback to improve our paper.

REFERENCES

[1] "From the acr team: Super resolution." [Online]. Available: https://blog.adobe.com/en/publish/2021/03/10/from-the-acr-team-super-resolution.html

[2] A. Heinecke, G. Henry, M. Hutchinson, and H. Pabst, "Libxsmm: accelerating small matrix multiplications by runtime code generation," in *SC'16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2016, pp. 981–991.

[3] D. Khudia, J. Huang, P. Basu, S. Deng, H. Liu, J. Park, and M. Smelyanskiy, "Fbgemm: Enabling high-performance low-precision deep learning inference," *arXiv preprint arXiv:2101.05615*, 2021.

[4] I. MKL, "Intel® math kernel library improved small matrix performance using just-in-time (jit) code generation for matrix multiplication (gemm)," July 2020. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/intel-math-kernel-library-improved-small-matrix-performance-using-just-in-time-jit-code.html

[5] oneDNN, "oneapi deep neural network library (onednn)," July 2020. [Online]. Available: https://github.com/oneapi-src/oneDNN

[6] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[7] Intel, "intel/intel-extension-for-pytorch." [Online]. Available: https://github.com/intel/intel-extension-for-pytorch

[8] A. O. F. Atya, Z. Qian, S. V. Krishnamurthy, T. La Porta, P. McDaniel, and L. Marvel, "Malicious co-residency on the cloud: Attacks and defense," in *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 2017, pp. 1–9.

[9] A. O. F. Atya, Z. Qian, S. V. Krishnamurthy, T. La Porta, P. McDaniel, and L. M. Marvel, "Catch me if you can: A closer look at malicious co-residency on the cloud," *IEEE/ACM Transactions on Networking*, vol. 27, no. 2, pp. 560–576, 2019.

[10] P. D. Ezhilchelvan and I. Mitrani, "Evaluating the probability of malicious co-residency in public clouds," *IEEE Transactions on Cloud Computing*, vol. 5, no. 3, pp. 420–427, 2015.

[11] Pytorch, "Pytorch/serve: Model serving on pytorch." [Online]. Available: https://github.com/pytorch/serve

[12] Awslabs, "Awslabs/multi-model-server: Multi model server is a tool for serving neural net models for inference." [Online]. Available: https://github.com/awslabs/multi-model-server

[13] T. Brennan, N. Rosner, and T. Bultan, "Jit leaks: inducing timing side channels through just-in-time compilation," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1207–1222.

[14] 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.

[15] F. Tramèr, F. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Stealing machine learning models via prediction apis," in *25th {USENIX} Security Symposium ({USENIX} Security 16)*, 2016, pp. 601–618.

[16] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang, "Intel math kernel library," in *High-Performance Computing on the Intel® Xeon Phi™*. USA: Springer, 2014, pp. 167–188.

[17] [Online]. Available: http://www.openblas.net/

[18] A. Krizhevsky, "One weird trick for parallelizing convolutional neural networks," *arXiv preprint arXiv:1404.5997*, 2014.

[19] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[20] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[21] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <0.5mb model size," *arXiv:1602.07360*, 2016.

[22] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708.

[23] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[24] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, pp. 1–9.

[25] N. Ma, X. Zhang, H.-T. Zheng, and J. Sun, "Shufflenet v2: Practical guidelines for efficient cnn architecture design," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 116–131.

[26] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "Mobilenetv2: Inverted residuals and linear bottlenecks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 4510–4520.

[27] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)." [Online]. Available: http://www.cs.toronto.edu/ kriz/cifar.html

[28] J. Deng, K. Li, M. Do, H. Su, and L. Fei-Fei, "Construction and Analysis of a Large Scale Image Ontology." Vision Sciences Society, 2009.

[29] "Conv2d." [Online]. Available: https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

[30] D. Page, "A note on side-channels resulting from dynamic compilation." *IACR Cryptol. ePrint Arch.*, vol. 2006, p. 349, 2006.

[31] J. Van Cleemput, B. De Sutter, and K. De Bosschere, "Adaptive compiler strategies for mitigating timing side channel attacks," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 1, pp. 35–49, 2020.

[32] B. Coppens, I. Verbauwhede, K. De Bosschere, and B. De Sutter, "Practical mitigations for timing-based side-channel attacks on modern x86 processors," in *2009 30th IEEE Symposium on Security and Privacy*. IEEE, 2009, pp. 45–60.

[33] J. V. Cleemput, B. Coppens, and B. De Sutter, "Compiler mitigations for time attacks on modern x86 processors," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 8, no. 4, pp. 1–20, 2012.

[34] S. Crane, A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "Thwarting cache side-channel attacks through dynamic software diversity." in *NDSS*, 2015, pp. 8–11.

[35] T. Frassetto, D. Gens, C. Liebchen, and A.-R. Sadeghi, "Jitguard: hardening just-in-time compilers with sgx," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2405–2419.

[36] S. Chen, J. Xu, and E. C. Sezer, "Non-control-data attacks are realistic threats," in *14th USENIX Security Symposium (USENIX Security 05)*. Baltimore, MD: USENIX Association, Jul. 2005. [Online]. Available: https://www.usenix.org/conference/14th-usenix-security-symposium/non-control-data-attacks-are-realistic-threats

[37] H. Hu, S. Shinde, S. Adrian, Z. L. Chua, P. Saxena, and Z. Liang, "Data-oriented programming: On the expressiveness of non-control data attacks," in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 969–986.

[38] J. Kelsey, B. Schneier, D. Wagner, and C. Hall, "Side channel cryptanalysis of product ciphers," in *European Symposium on Research in Computer Security*. Springer, 1998, pp. 97–110.

[39] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel." *IACR Cryptol. ePrint Arch.*, vol. 2002, no. 169, pp. 1–23, 2002.

[40] B. B. Brumley and R. M. Hakala, "Cache-timing template attacks," in *International Conference on the Theory and Application of Cryptology and Information Security*. Springer, 2009, pp. 667–684.

[41] C. Percival, "Cache missing for fun and profit," 2005.

[42] 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.

[43] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, "On the power of simple branch prediction analysis," in *Proceedings of the 2nd ACM symposium on Information, computer and communications security*, 2007, pp. 312–320.

[44] O. Acıiçmez, Ç. K. Koç, and J.-P. Seifert, "Predicting secret keys via branch prediction," in *Cryptographers' Track at the RSA Conference*. Springer, 2007, pp. 225–242.
[45] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher *et al.*, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 1–19.
[46] D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh, "Jump over aslr: Attacking branch predictors to bypass aslr," in *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 2016, pp. 1–13.
[47] D. Evtyushkin, R. Riley, N. C. Abu-Ghazaleh, ECE, and D. Ponomarev, "Branchscope: A new side-channel attack on directional branch predictor," *ACM SIGPLAN Notices*, vol. 53, no. 2, pp. 693–707, 2018.
[48] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado, "Inferring fine-grained control flow inside {SGX} enclaves with branch shadowing," in *26th {USENIX} security symposium ({USENIX} security 17)*, 2017, pp. 557–574.
[49] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin *et al.*, "Meltdown: Reading kernel memory from user space," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 973–990.
[50] M. Jagielski, N. Carlini, D. Berthelot, A. Kurakin, and N. Papernot, "High accuracy and high fidelity extraction of neural networks," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1345–1362.
[51] B. Wang and N. Z. Gong, "Stealing hyperparameters in machine learning," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 36–52.
[52] V. Chandrasekaran, K. Chaudhuri, I. Giacomelli, S. Jha, and S. Yan, "Exploring connections between active learning and model extraction," in *29th {USENIX} Security Symposium ({USENIX} Security 20)*, 2020, pp. 1309–1326.
[53] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

Appendix

A. Noise Tolerance of the Attack Strategy

Fig. 9 shows the average success rate of varying output shape and secret parameter configurations. These experiments were performed in an environment where only the victim and adversary were present. Thus, a low level of noise was observed. In addition, we introduce an external compute intensive ML inference workload to observe the mean success rate in a noisy environment. Fig. 13 shows the distribution of success rates when only the attacker and victim are present, and when an external workload is also present (noisy environment). Even with noise, 80% of the cases show a average success rate above 0.5.

B. Attacker Model Design Details

In JAXED, the attacker designs her own model and times her model execution, attempting to match the parameters of the targeted FC / convolution layer in the victim.

C. Attacker Timing Methodology

Attacker can obtain detailed timing information for her own model, since she has full observability only over her model execution. For example, in PyTorch 1.9 this can be easily achieved using the built-in profiler, as shown in Listing 1. The profiler would provide detailed timing for the layer of interest.

When the attacker guesses different hyperparameters, the underlying attacker model would be modified and the execution times will be profiled. By observing the time differences, after having executed the victim, the attacker should be able to reveal whether the targeted layer matches that of the victim.

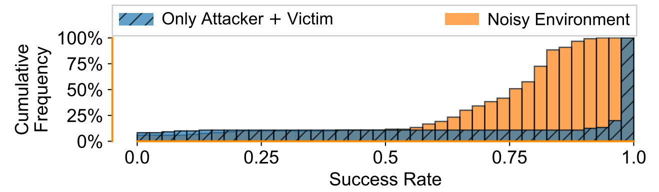


Fig. 13: Cumulative frequency of average success rates when only the attacker and victim are present and when an external workload is also present (noisy environment).

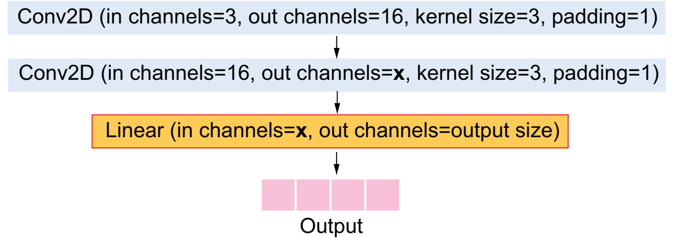


Fig. 14: Attacker model design to extract hyperparameters of the victim model's FC layer. Since the output size is known, by guessing the number of input channels, the linear layer (FC) parameter can be detected. Since the attacker is only interested in timing, the input image can be chosen to be 1×1 for fast execution.

Listing 1: Attacker timing method.

```
import torch
import models
from torch.profiler import profile,
    ↪ record_function, ProfilerActivity

model = models.attacker_model()
inputs = torch.randn(1, 3, 1, 1)

with profile(activities=[ProfilerActivity.
    ↪ CPU], record_shapes=True) as prof:
    with record_function("model_inference")
        ↪ :
        attacker_model(inputs)
```

D. Complete Attack

The attack would involve 3 steps: i) a profiling step where the attacker gathers her own system behavior and expected runtimes. ii) next, the attacker would run the victim model so that the optimized code would be stored in the JIT code

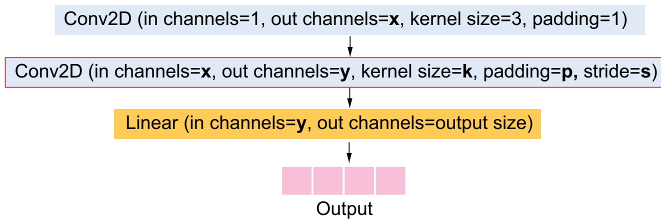


Fig. 15: Attacker model design to extract hyperparameters of the victim model's convolution layer. The parameters of the second convolution layer are guessed so that the execution would match a convolution layer of the victim.

cache, and finally, when the attacker runs her model, if layer hyperparameters are shared, a time difference will be observed. The attack is similar to Listing 2.

Listing 2: Complete attack code.

```
import intel_pytorch_extension as ipex

device = ipex.DEVICE
if args.option == 'profile':
      profile_sys(iterations, device,
          ↪ out_shape, candidate_guesses)
      write_profiling_data()
      return
elif args.option == 'attack':
      try:
          df = read_profiling_data()
      except:
          raise IOError('profiling␣data␣not
              ↪ ␣available.␣are␣you␣sure␣
              ↪ you␣did␣the␣profiling␣step?
              ↪ ')
      run_victim_model()
      perform_attack(df, iterations,
          ↪ device, out_shape,
          ↪ candidate_guesses)
      return
```