New Passive and Active Attacks on Deep Neural Networks in Medical Applications

Invited Talk

Cheng Gongye, Hongjia Li, Xiang Zhang, Majid Sabbagh, Geng Yuan, Xue Lin, Thomas Wahl, and Yunsi Fei
Northeastern University, Boston, USA
{gongye.c,li.hongjia,zhang.xiang1,sabbagh.m,yuan.geng,xue.lin,t.wahl,y.fei}@northeastern.edu

ABSTRACT

Security of deep neural network (DNN) inference engines, i.e., trained DNN models on various platforms, has become one of the biggest challenges in deploying artificial intelligence in domains where privacy, safety, and reliability are of paramount importance, such as in medical applications. In addition to classic software attacks such as model inversion and evasion attacks, recently a new attack surface—implementation attacks which include both passive side-channel attacks and active fault injection and adversarial attacks—is arising, targeting implementation peculiarities of DNN to breach their confidentiality and integrity. This paper presents several novel passive and active attacks on DNN we have developed and tested over medical datasets. Our new attacks reveal a largely under-explored attack surface of DNN inference engines. Insights gained during attack exploration will provide valuable guidance for effectively protecting DNN execution against reverse-engineering and integrity violations.

CCS CONCEPTS

• **Security and privacy → Hardware attacks and countermeasures.**

KEYWORDS

deep neural networks, side-channel attacks, fault injection attacks

ACM Reference Format:
Cheng Gongye, Hongjia Li, Xiang Zhang, Majid Sabbagh, Geng Yuan, Xue Lin, Thomas Wahl, and Yunsi Fei. 2020. New Passive and Active Attacks on Deep Neural Networks in Medical Applications: Invited Talk. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD '20), November 2–5, 2020, Virtual Event, USA.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3400302.3418782

1 INTRODUCTION

Deep learning (DL) has become a foundational means for solving grand societal challenges, disrupting many application domains with superior performance. Trained Deep Neural Network (DNN) models have proven to be effective in solving various medical imaging problems. Commercial AI-assisted computer-aided diagnosis equipment has been approved and deployed [19, 21]. Despite the promising outcomes of DNN, protecting the security of trained models has become a challenge, in part because DNN inference engines create heretofore unknown attack surfaces.

Trained models constitute valuable intellectual property, for the following reasons. Customizing DNN models for applications requires access to high-quality, often proprietary, datasets and also demands a considerable amount of computational resources. Typically it also requires machine learning experts and domain experts to work together towards selecting network structures suitable for the task, pre-processing the dataset, and fine-tuning the model structure and hyperparameters. Given the commercial value of today's DNN models, an adversary has a strong incentive to reverse-engineer a trained DNN model to obtain a near-identical one. If the model is known to the adversary, active attacks that disrupt or sabotage the DNN inference engines can be enabled or strengthened. For example, attacks based on adversarial examples, which appear authentic to human eyes but contain deliberately added noise to yield a wrong output, can become more effective. Knowing the details of the model also facilitates fault injection attacks, which maliciously modify the model parameters to disrupt the deep learning applications in execution.

In this work, we present two passive attacks that steal the intellectual property of DNN models, and two active attacks that compromise DNN execution. We discover a new attack surface with a number of threats directed at DNN inference engines. Countermeasures are proposed to protect the confidentiality and integrity of DNN model execution against these new attacks.

The rest of this paper is organized as follows. Section 1.1 gives an overview of the attacks. Section 1.2 provides the necessary background for attacks on DNN. Section 2 and 3 describe the passive attacks and the active attacks in detail, respectively, where countermeasures are also outlined for each attack. Conclusions and future work are presented in Section 4.

1.1 Overview

To protect the confidentiality of trained DNN models, trusted execution environments like Intel SGX and ARM TrustZone can be adopted. However, valuable information can still be extracted from various side channels. We develop two passive side-channel attacks to steal the intellectual property of DNN models. The first one is a novel persistent cache monitoring attack, which relies on a newly developed utility to monitor the state of shared caches continuously and covertly. By monitoring carefully selected function call instructions, we are able to reverse-engineer the hyperparameters of DNN, as shown in Section 2.1. Knowing the hyperparameters, i.e., the structure of the DNN, enables another finer-grained side-channel attack, which exploits the floating-point timing side-channel to reverse-engineer all parameters of DNN models accurately, as detailed in Section 2.2. This is the first reverse-engineering work targeting weights and biases of DNN software implementations, published in DAC 2020 [9].

We also investigate the feasibility of active attacks, i.e., how to effectively and efficiently disrupt the execution of DNN inference engines. A ResNet-18 network that detects COVID-19 disease from chest X-ray images is trained to evaluate these attacks. In Section 3.1, we launch an adversarial attack against this model and demonstrate that adding human-imperceptible noise to the input images can effectively mislead the DNN inference. In Section 3.2 we perform a realistic fault injection attack on GPU kernels for such ResNet-18 model inference. We leverage the GPU overdrive attack [28], which was published in DAC 2020. This attack maliciously perturbs the operating voltage and frequency of the target GPU, inducing silent data corruption during model execution, which leads to a significant decrease of the DNN classification accuracy.

1.2 Background

1.2.1 Deep neural networks. Over the past decade, DNN have experienced rapid and tremendous progress thanks to the new era of big data. Especially for computer vision problems, DNN and large-scale annotated imaging datasets drastically improve the performance of classification, object localization, detection, and segmentation. Chest X-ray images can be quickly obtained from patients with inexpensive equipment. Various DNN models have been developed and trained on large datasets, which can rapidly extract and learn the complex features embedded in images. Applying the trained DNN models for inference can significantly aid diagnosing, disease detection and localization. One prior work [30] introduces a recurrent neural cascade model for disease detection. Another work [26] adopts convolutional layers to construct CheXNet, exceeding the average performance on detecting pneumonia by radiologists. Another model, Text-Image Embedding network (TieNet), is based on an end-to-end trainable CNN-RNN architecture and can be transferred to a chest X-ray reporting system [34].

1.2.2 Side-channel analysis. Side-channel analysis (SCA) targets the information leakage of a system due to peculiarities of its physical implementation, on various platforms including CPU, GPU, MCU, and FPGA. These leakages come in the form of physical signatures that include, among others, power consumption [6], execution time [9], electromagnetic radiation [18], and sound emission [8]. Various methods have been developed to extract secret information, e.g., key of cryptographic algorithms, from these physical signatures. SCA attacks are relatively cheap to perform, and hard and expensive to protect against. Recently SCA attacks have been applied to steal the IP of DNN models [4, 9, 14].

1.2.3 Fault injection attacks. Fault injection attacks actively modify intermediate states of a program to bypass verification [17], facilitate differential fault analysis for secret key retrieval [3], or simply disrupt or shut down the operation [16]. Physical fault injection methods include laser beaming [33], electromagnetic radiation [25], and voltage glitching [32], requiring physical access to the victim device. Fault injections can also be performed by software, including RowHammer [20] and DVFS attacks [31], possibly controlled remotely. Vulnerabilities of DNN models to fault injections attacks are evaluated by different algorithms [23, 39]. Our prior work [29] also considers the effect of model compression in fault resilience. A recent work implements practical fault attacks using laser beaming on a simple MLP inference engine running on a microcontroller [5].

2 STEALING MODEL IP VIA PASSIVE ATTACKS

This section presents two IP-stealing passive attacks: one cache side-channel attack for hyperparameters retrieval and the other floating-point timing attack to reverse-engineer all model parameters.

2.1 Persistent Cache Monitoring Attack

Cache Telepathy [37] is a recent work that leverages shared resource, cache, to learn the architecture of DNN. We devise a novel cache monitor that can run much faster and can retrieve the model architecture in real-time without instrumenting the victim code.

2.1.1 Attack Model. In our attack model, the victim is a trained DNN model running on an x86 processor. The adversary (spy) runs on the same processor and shares common software libraries with the victim, e.g., OpenSSL for network security and OpenBLAS [35] for deep learning applications. There is no synchronization between the victim and the spy processes, and they are executing concurrently, either on different cores, or on the same core with hyper-threading on. The spy and the victim only interact with each other through the shared resource - cache, and the contention on cache leaks victim information.

2.1.2 Attack Details. We propose a novel Flush+Flush based persistent cache state monitor, and apply this monitor to a DNN victim.

a. Spy - Flush+Flush Cache Monitor: Various cache-based side channels and covert channels have been presented, including Flush+Reload [38], Prime+Probe [22], and Flush+Flush [12]. They differ in the granularity of side-channel. Prime+Probe attacks deal with cache sets and are more general as the adversary is completely independent of the victim. The other two rely on a special x86 instruction - CLFLUSH to deal with individual cache lines, but require shared libraries between the spy and the victim.

The prior work [37] uses a Flush+Reload monitor, where the spy keeps running flushing and reloading one address from the shared library and times the reloading. The effect of `CLFLUSH, maddr` is, the cache lines corresponding to the memory address `maddr` are flushed from the entire cache hierarchy - L1, L2, and Last-level Cache (LLC). If the victim has accessed this memory address between the spy's flushing and reloading, the reload takes a shorter time (cache hit) because the cache line has been brought back to the processor by the victim. Otherwise, the reload experiences a cache miss due to the prior flushing event. The difference in a cache hit and a cache miss (can be 100+ cycles for last-level cache) forms a strong timing side-channel to indicate whether the victim has accessed a certain address or not.

However, Flush+Reload side channels experience many cache misses and run slowly.

We build a fast Flush+Flush-based cache monitor. It has been observed that there is a slight, but distinct difference between the time taken to flush a valid cache line versus an invalid one (already flushed before), about 9 to 12 cycles [12]. The timing difference varies significantly with processor architectures, timers, and fencing instructions used around timer readings. Carré's work [7] uses a Flush+Flush monitor on ECDSA algorithm. Figure 1 shows the setup for our persistent cache monitoring, where the spy runs consecutive `CLFLUSH` and times each one. On our platform and experimental setup, the two timings our monitor reads are about 320 cycles (victim accesses the monitored address) and 180 cycles (no-access) respectively, a very strong timing side-channel.

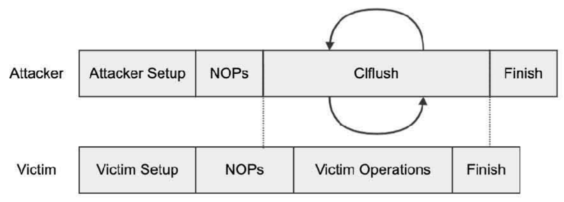

Figure 1: Persistent Cache Monitoring with F+F

b. Victim - DNN Model Operation: The victim in our attack is the general matrix-matrix multiplication (GEMM) function of OpenBLAS, one core operation for DNN. We can monitor the cache states of a running GEMM function to extract the dimensions of input matrices, a hyperparameter of the convolutional kernel [37].

In modern BLAS libraries, blocked matrix-matrix multiplication is applied to get better performance with cache locality. GEMM function takes two matrices A ($M \times K$) and B ($K \times N$) as input, and calculates $A \cdot B$ to generate the output matrix C ($M \times N$). To fit different level caches, OpenBLAS splits input matrix A into blocks of size $P \times Q$, and B into blocks of the size $Q \times R$, where P, Q and R are preset parameters. The block matrix-matrix multiplication algorithm involves several recursive loops, and the number of the iterations for these loops are related to M, K, N through known P, Q, R, and $UNROLL$. With our monitor, we can obtain the loop iterations and therefore derive the matrix dimensions. The OpenBLAS source code gives out the preset parameters, which on our test machines are *P=512, Q=256, R=16384* and *UNROLL=8*.

2.1.3 *Experimental Results.*

We conduct some preliminary experiment and launch our persistent cache monitoring on a GEMM function. We focus on monitoring two loops, one with iterations $iter_2 = (N \bmod R)/3UNROLL$ where each iteration calls a function oncopy, and the other with iterations $iter_1 - 1 = M/P - 1$ where each iteration calls a function `itcopy`, as shown in Figure 2.

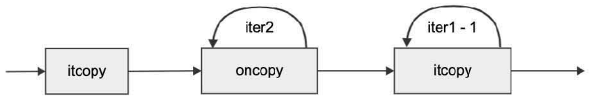

Figure 2: Flow of a loop in GEMM function[37]

In our spy program, we monitor two instruction addresses, `target1` from the function `itcopy`, and `target2` from the function oncopy, consecutively, as shown in Algorithm 1. In each monitoring iteration two time stamps will be collected (`rdtsc`) and stored in an array, and the differential between every two consecutive time stamps will be the time the `CLFLUSH` in between takes.

Algorithm 1: Monitor program for GEMM

```
for $i \leftarrow 1$ to $ITER$ do
    CLFLUSH target1;
    Tarray[2i-1] = rdtsc();
    CLFLUSH target2;
    Tarray[2i] = rdtsc();
```

In the victim program, we vary the size of input matrices (M, N and K) to get different $iter_1$ and $iter_2$. Figure 3 shows a trace segment for the GEMM execution in the case $M = 2048, N = 120, K = 1$, with correspondingly $iter_1 = M/P = 4$ and $iter_2 = (N \bmod R)/3UNROLL = 5$. We mark timings for `target1` *itcopy* as blue points, and timings for `target2` *oncopy* red points.

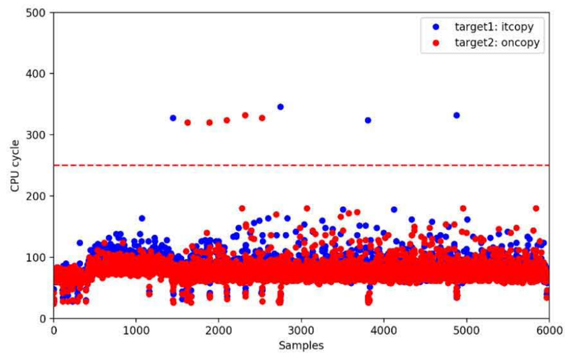

Figure 3: Monitor result for Matrix dimension: M=2048, N=120, K=1

In Figure 3, we have 5 red points followed by 3 blue points (above the threshold of 250 cycles), which means *oncopy*() is accessed by the victim 5 times and *itcopy*() is accessed 3 times, and therefore $iter_2 = 5$ and $iter_1 = 4$. Such information can help derive the size of M, N, and K.

2.1.4 *Discussion.*

To mitigate the Flush+flush based cache attack, we can add more noise to the cache state, i.e., contention on the shared cache by other concurrent processes. Disabling Hyperthreading on x86 processors can also reduce the resolution of the CLFLUSH-based monitor to reduce the cache information leakage.

Our persistent cache monitoring attack is fast and reliable. We have applied it to break an AES encryption (from the OpenSSL library) and recover a round key with less than 2000 executions and monitoring traces. It is more effective than the traditional

Flush+Reload attack, which requires strong synchronization between the victim and the spy. We will further investigate our persistent cache attack on DNN execution to recover more hyperparameters and parameters including weights and biases.

2.2 Reverse-Engineering Model Parameters via Floating-Point Side Channels

This section describes a finer-grained passive attack for model parameters retrieval.

2.2.1 Attack Model. The side-channel being exploited is the operand-dependent timing for floating-point computations. According to the IEEE-754 floating-point (FP) number standard [15], a single-precision floating-point number is represented as a 32-bit string, consisting of a single-bit *sign* (S), an 8-bit *exponent* (E), and a 23-bit *mantissa* (M). We target this format in this work, but the attack is otherwise agnostic to the FP format.

A *normal* floating-point number is representable with a mantissa starting with 1, and an exponent in some predefined range. In contrast, a *subnormal* (or *denormal*) number has a magnitude between 0 and the smallest normal number and thus requires leading zeros in the mantissa. Commercial CPUs typically have dedicated floating-point arithmetic units (FPUs) and registers for normal floating-point operations. However, since subnormal floating-point numbers are less frequent, there is no dedicated hardware support for them on modern processors. Instead, processors may have the hardware to *detect* subnormal operands, but implement operations on them in software (i.e., dispatch them onto microcode executions). This can make such operations much slower than normal operations [27], opening up a timing channel.

FP Multiplication Timing Model: Consider a floating-point multiplication $a \cdot b = c$, where a, b and c are non-zero. In most cases, if one of a, b, or c is a subnormal floating-point number, this operation will feature abnormally long timing. However, if either operand or the result is zero, we will not observe abnormal timing. We developed a suite of microbenchmarks to characterize the timing model of x86 floating-point multiplications, shown in Table 1. All experiments are performed on a workstation with Intel i7-7700 quad-core processor and 2×8GB Dual-channel DDR4 memory. We found an average extra timing of 114 cycles for abnormal operations, which we denote as σ.

Table 1: Timing model for floating-point mutiplications

Case	Operation	CPU cycles
1	$normal \cdot normal = normal$	10
2	$normal \cdot normal = subnormal$	124
3	$subnormal \cdot normal = normal$	124
4	$subnormal \cdot normal = subnormal$	124
5	$subnormal \cdot subnormal = 0$	10
6	$subnormal \cdot 0 = 0$	10

FP Addition Timing Model: A floating-point addition $a + b = c$ will feature abnormal timing when $|a| \in (\min_n, 6\text{e}{-33})$ and $|c| \in (1\text{e}{-43}, \max_{sn})$ (as observed on our experimental platform), where $\max_{sn}$ is the largest (single-precision) subnormal number ($\approx 1.1754942\text{e}{-38}$), and $\min_n$ is the smallest normal number ($\approx 1.1754944\text{e}{-38}$). We ran microbenchmarks to characterize the timing model of FP addition and found that the average extra timing σ is again about 114 cycles. Previous work that utilizes floating-point timing side channels mainly focuses on multiplications and divisions. In this work, we take advantage of the timing leakage of additions too (which are frequent in DNN inference). In the following subsections, we will show how we leverage these two timing models to reverse-engineer the weights and biases.

2.2.2 Attack Details. We attack the model in a layer-by-layer fashion. We focus on recovering the first layer of an MLP model.

The algebraic representation of the first layer is

$$\mathbf{l}_1 = Activation(\mathbf{W}_1 \cdot \mathbf{l}_0 + \mathbf{b}_1)$$

Our goal of attacking this layer is to recover all the elements of $\mathbf{W}_1$ and $\mathbf{b}_1$, by only varying the layer input $\mathbf{l}_0$ and observing the timing. In this paper, we assume the activation function to be a rectified linear unit (ReLU), one of the most effective and widely adopted activation functions.

Our approach proceeds in three steps: 1) recover the *set* of absolute values of each column of the weight matrix, i.e. without knowing the order within the column; 2) arrange the weights to figure out weights belonging to the same row and find their relative signs; and 3) recover the bias vector and the actual signs of all parameters in the first layer.

1) Column Absolute Values: This attack utilizes the first timing model presented in 2.2.1. We utilize case 2 in Table 1, where the product is subnormal, and the inputs are normal numbers within the range of $[min_n, 1]$. For the first DNN layer, each neuron computes a scalar product of the input vector and a weight row. In software implementations without parallelism, these neuron computations are carried out in sequence, and all contribute to the total timing. To focus on the first column, we set $l_0[1] = a$, $l_0[2 : m] = 0$, for some value a, where m is the length of the input vector $\mathbf{l}_0$. With n neurons, the observed first-layer computation time is the sum of the times for n multiplications with the fixed value of a. The total timing model for the first layer is thus:

$$T_{layer1}(a) = \sum_{i=1}^{n} T(a \cdot \mathbf{W}_1[i, 1]) + T_{others} ,$$

where T_{others} summarizes other timing components and can be considered constant. Our attack consists of two steps. First, find a vector $\mathbf{A} = (a_1, a_2, ...a_n)$ with the n values in decreasing order, such that $T_{layer1}(a_i) = c + i \cdot \sigma$. We envision that in the range $[min_n, 1]$, there exist n such values, namely $A_0[i] = max_{sn}/V[i], i \in [1, n]$, where $V[i]$ are the n weight values. We treat these n values as reference points, which divide the range of $[min_n, 1]$ into $n + 1$ segments for the value of a, with $T_{layer1}(a)$ for each segment decreasing from $c+n\cdot\sigma$ to c, from left to right. We are finding a vector $\mathbf{A}$ such that its n values partition the range of a into $n + 1$ intervals, where each of the intervals contains one such reference value $A_0[i]$. By tracing this value with the interval known, we can recover the weight. We employ a binary search to reduce the interval to the precision desired. The precision threshold to terminate the binary search in this algorithm is denoted by ϵ.

2): **Weights with Relative Signs in Each Row** The first step has recovered all the weights in each column of $\mathbf{W}_1$, but we do not know their order. Recovering all the locations of weights together is hard because we can only control the input and observe the timing. We adopt an iterative technique to accomplish this task. To create a reference point for each row, we pick the first column of the weight matrix and sort its values. Then, for each element of the first column, we identify which element in each of the remaining columns belongs to the same row, i.e., we recover a weight row vector. The input is constructed so that only if the target element is in the same row as the reference point, an addition abnormal timing will be triggered. We repeat this step for all elements in the first weight column and recover all the n weight rows.

After the previous two steps, we have recovered $\mathbf{W}_1$ except its actual signs. For the final step, we exploit the definition of the ReLU function, which is to reduce any negative input to 0. We construct the value of one neuron to be of large magnitude and with the same relative sign. If the output of the neural network does not change, the actual sign is negative, and vice versa. The bias vector can be recovered in a similar way.

Once the first layer is recovered, similar steps can be applied to follow-on layers and they will all be recovered in a sequence.

2.2.3 Experimental Results. The experimental platform is as discussed in Section 2.2.1: the total execution time of the layer is measured, in CPU cycles, for a hundred times repetitively; the most frequent ones are averaged.

The model we recovered is a four-layer MLP, although the method also applies to CNN as well. The input layer flattens the MNIST dataset with a size of $28 \times 28 = 784$. The second and third layers both have a size of 50. The last layer is the output layer before the softmax function, which has a size of 10. All the activation functions are ReLU. The model is trained using stochastic gradient descent (SGD) with a learning rate of 1e−2, a momentum of 5e−1, and a batch size 64 for 5 epochs. The testing loss and accuracy are 1.342e−1 and 96.04%, respectively. Our entire reverse-engineering attack takes less than one hour for the selected MLP model on our testing workstation.

We define the accuracy of parameter recovery as follows: $\rho_p = 1 - |p - p'|/p$, where p is the actual parameter, and p' is the recovered parameter. We evaluate the accuracy of all recovered first-layer parameters and take their average.

We also evaluate the effect of adjusting the precision parameter ϵ in the algorithm sketched in Section 2.2.2; Table 2 shows the results. When ϵ is below 1e−39, the accuracy is close to 1. We can use even smaller values for ϵ in our deployed algorithm.

Table 2: First-layer Parameter Accuracy with Different ϵ

$-\log\epsilon$	37	38	39	40
ρ_p	0.838 ± 0.118	0.987 ± 0.011	0.998 ± 0.001	$0.999 \pm$ 1e−4

We plug in the recovered model for testing with the MNIST dataset, and evaluate the model accuracy. Table 3 shows that the recovered model reaches the original testing accuracy when ϵ is below 1e−39.

2.2.4 Countermeasures. The timing side channel considered here relies on longer execution times for certain operations involving subnormal numbers, so eliminating these numbers—for instance by flushing all subnormal results to zero—eliminates this side channel. Incidentally, this can speed up the computation. The downsides are that it can decrease the computation accuracy, and it is platform- and compiler-dependent.

Table 3: Model Accuracy in classifying MNIST for different ϵ

$-\log\epsilon$	37	38	39	40
ρ_{model}	0.9193	0.9598	0.9604	0.9604

Even subnormal-free floating-point arithmetic contains many (fine-grained) timing dependencies, for instance due to exceptions being raised on rounding, overflows, etc. Ultimately, these can only be eliminated using a constant-time numeric library, e.g. based on fixed-point arithmetic, or customizable floating-point arithmetic [1]. In addition to the likely performance degradation, such approaches no longer benefit from the trade-off between precision and range.

3 ACTIVE ATTACKS - ON DNN EXECUTION

As deep learning is used in many safety-critical applications including autonomous driving, medical disease diagnosis, and machine-learning-as-a-service (MLaaS) in the cloud, the integrity of model execution is crucial. This section presents two active attacks, with their goal to generate adversarial inputs and introduce faults during model execution, respectively.

3.1 Adversarial Attacks

Adversarial examples have been proved to successfully deceive deep learning systems and become a serious threat [11]. With slight, but carefully crafted, noise imposed on the input sample, the deep learning system misclassifies the adversarial example to a targeted wrong class or any class which is different from the correct one. Figure 4 demonstrates three adversarial examples we have generated based on a publicly available COVID-19 dataset of X-ray images [41]. The COVID-19-CXR-Dataset contains 6,354 CXR images, for both training and testing, divided into three categories: COVID-19, Normal and Pneumonia. Misclassifying COVID-19 images to normal, i.e., false negative, would result in life loss; while misclassifying normal/pneumonia images to COVID-19, i.e., false positive, would place tremendous stress on the patients and also drain the already stringent hospital medical resources.

We first investigate effective algorithms to generate adversarial examples to test the vulnerability of deep learning models, with the ultimate goal to enhance the robustness of deep learning models under such adversarial attacks.

3.1.1 Attack Model. We extend two previous attack methods for generating adversarial examples to our COVID-19 disease detection system, and demonstrate the vulnerability of the system. We first develop a baseline model to classify COVID-19 disease from chest X-ray images. Irregular, patchy, hazy, reticular, and widespread ground-glass opacities shown in the chest X-ray image are considered as the symptoms of COVID-19. In Figure 4, the first column shows three original chest X-ray images of three patients, Normal, Pneumonia and COVID-19, respectively. ResNet-18 [13] is adopted as our backbone model since it is easier to optimize, and can substantially reduce the number of parameters through a residual learning framework.

3.1.2 Attack Details. We implement both fast gradient sign method (FGSM) and projected gradient descent (PGD) adversarial attacks.

FGSM [10] can find adversarial examples reliably. For a neural network N, let x be the input of the model, y is the targeted output associated with x, and ω is the set of trained parameters of the model. During the model training, the loss function is denoted as $L(\omega, x, y)$. The loss function can be linearized to obtain an optimal max-norm constrained perturbation:

$$\eta = \epsilon sign(\nabla_x L(\omega, x, y)), \tag{1}$$

where ϵ is a hyperparameter that controls the maximum permitted L_∞ norm of the perturbation and the gradient sign determines the polarity of the perturbation. In general, a larger ϵ allows a more noticeable perturbation on the original image and leads to a more effective attack on the DNN model.

Based on FGSM, PGD attack proposed by [24] is also a L_∞ attack and utilizes the gradient of the loss function. It is the standard method for large-scale constrained optimization. Instead of simple one-step in FGSM, PGD extends to the multi-step variant:

$$x^{t+1} = \prod_{x+S} (x^t + \epsilon sgn(\nabla_x L(\omega, x, y))), \tag{2}$$

where S denotes a set of allowed perturbations that formalizes the manipulative power of the adversary.

3.1.3 Experimental Results. Our experiments are conducted on a server with 4× NVIDIA RTX2080 GPU by using PyTorch API. We train and test our model on COVID-19-CXR-Dataset with 5614 images for training, and the other 740 images for testing.

Our baseline model is trained from scratch using 100 training epochs with the training batch size of 64. The SGD optimizer and cosine learning rate scheduler is used with the starting learning rate value of 0.01. We generate 740 adversarial examples based on the entire testing dataset to test the vulnerability of our COVID-19 disease model.

The results of our experiments are shown in Figure 4 and Figure 5. Our baseline model has achieved good performance with 94.7% accuracy. For the FGSM attack, targeting the incorrect answer in every case, we swipe the hyperparameter ϵ from 0.01 to 0.06. It is shown that it brings the effective accuracy down to below 1%. Figure 4 illustrates the adversarial examples by the FGSM attack, all with tiny noise that human eyes cannot notice. The PGD attack has reduced the accuracy to 0% after 15 iterations with hyperparameter $\epsilon = 0.007$. Both attacks are effective and successful against our COVID-19 disease model.

Our experiments indicate that adversarial attacks can easily upset the deep learning system in classification, even for extremely accurate classifiers. This vulnerability of the medical deep learning system should raise urgent attention, calling for approaches to enhance the adversarial robustness of DNN when developing the medical deep learning system.

3.1.4 Countermeasures. We outline several methods towards improving the robustness of deep learning systems against adversarial examples.

Algorithmic defenses: Adversarial defense schemes have been studied extensively for deep learning algorithms and promising

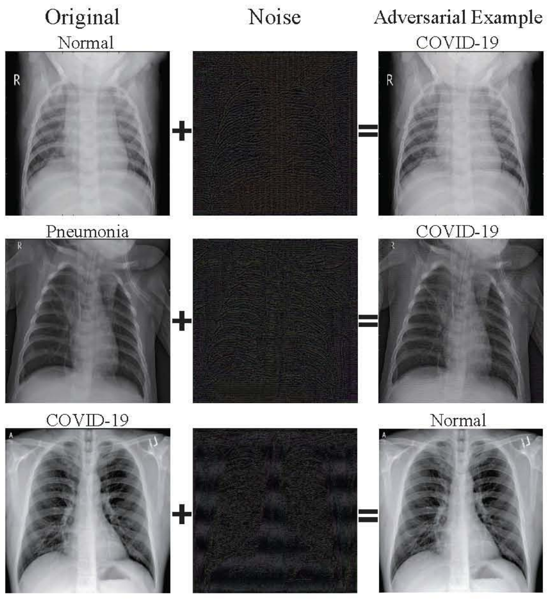

Figure 4: Adversarial examples generated by the FGSM attack with hyperparameter ϵ = 0.06.

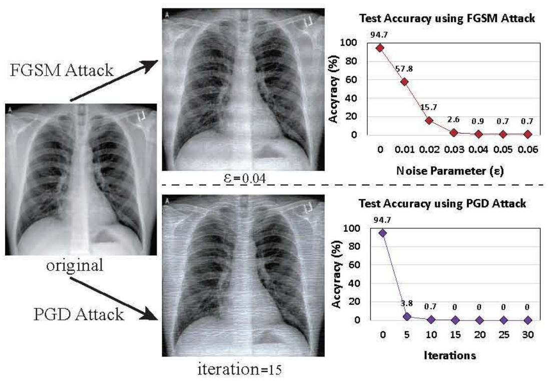

Figure 5: Accuracy performance of FGSM attack and PGD attack.

approaches are proposed such as gradient masking [2], robust optimization[24], and adversary detection [36].

System defenses: Different stages of deep learning systems can all be examined to guard against adversarial attacks. Particularly in medical deep learning systems, medical images should be well protected, both for privacy consideration and for integrity, i.e., disallowing any modifications on the inputs to happen.

3.2 GPU Overdrive Fault Attacks on CNNs

Graphics processor units (GPUs) are widely used to accelerate a variety of computational intensive applications, including deep learning-based image classification in medical disease diagnosis. Using GPUs in such life-critical settings necessitates a thorough study of their security properties, including their resistance against fault attacks. In this section, we evaluate the fault resiliency of the same deep learning model presented in Section 3.1, ResNet-18, when used as an inference kernel on GPUs for automatic recognition of COVID-19 from the chest X-ray (CXR) images. We leverage a non-intrusive software-controlled fault injection method on GPUs, called *overdrive* fault attack from our prior DAC 2020 work [28]. In this paper, we extend this attack to deep learning execution.

3.2.1 Attack Model. The GPU overdrive fault attack exploits the voltage-frequency scaling (VFS) feature of power management units on GPUs. VFS is a low-power feature which trades off computation performance with energy consumption on GPUs by dynamically changing their operating voltage and frequency (i.e., operating performance point or OPP). Prior work [28] discovers that we can use the vendor-provided software interfaces on the host CPU to force the GPU to operate under an out-of-specification, and unsafe, OPP. This causes the GPU instructions to experience silent data corruption (SDC) due to timing violations in the execution engine. SDCs are undetected errors in the data which are transient and do not cause permanent damage to the device.

Figure 6 provides an overview of the GPU overdrive attack model where an adversary on the CPU sends a series of overdrive commands to the GPU, forcing it to operate under an out-of-spec OPP while some critical kernels are running on it. If the kernels are cryptographic implementation, the adversary collects the faulty outputs of the victim and conducts differential analysis to infer the secret key.

If the kernels are deep learning implementation, the SDCs may propagate through the network and lead to misclassficiation or malfunction of the DNN execution.

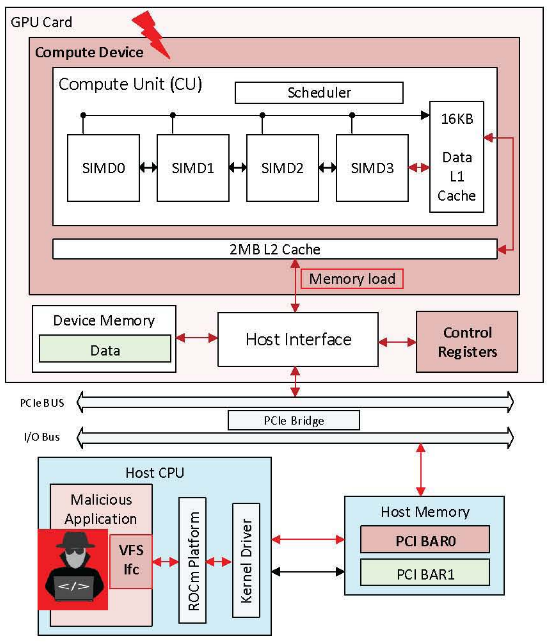

Figure 6: The overdrive fault attack model, with a malicious application on the host CPU and the victim kernel running on the GPU.

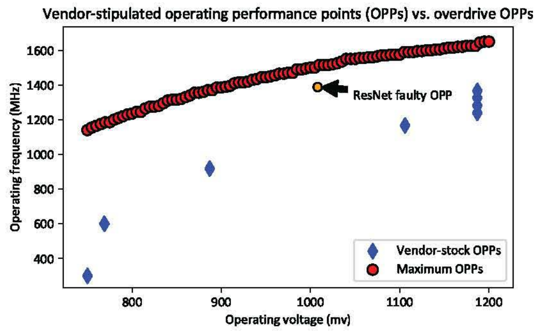

Figure 7: Overdrive characterization for AMD Readon RX 580 GPU.

Table 4: Typical outcome of kernels with different GCN instructions under overdrive fault injection.

Category	Examples	Faulty outcome
Arithmetic/logical	S_MUL_I32, S_LSHR_B64, S_XOR_B32, V_MAC_F32	Hang/Crash
Control/branch	S_NOP, S_GETPC, S_BRANCH	Hang/Crash
Store	S_STORE_DWORD, FLAT_STORE_UBYTE,	Hang/Crash
Load	S_LOAD_DWORD, FLAT_LOAD_UBYTE, FLAT_LOAD_DWORD	SDC/Hang/Crash

3.2.2 Attack Details. We address several salient issues in our fault injection attacks.

Faulty OPPs characterization: We target an AMD Readon RX 580 GPU and identify its vulnerable OPPs for different kernels. We sweep the operating voltage and frequency of our target GPU in small increments and find when the GPU instructions produce SDCs. Figure 7 shows vendor-specified OPPs (normal ones), the maximum OPPs before which the GPU become unstable (hang/crash) (silicon limit), and the faulty OPP for the ResNet-18 inference engine.

Vulnerable instructions: We construct several kernels with different categories of GPU instructions, including arithmetic/logical, branch/control, store, and load. As CNN models are both computation and data intensive, V_MAC_F32 instruction from arithmetic and FLAT_LOAD_DWORD instruction from load category are frequently-used. We find out that GPU kernels containing *memory loads* experience SDC under overdrive OPP, while others cause the system to hang or crash, as shown in Table 4.

Overdrive attack timing control: The CNN models for image recognition are composed of two main stages, the feature extraction and the classification. The feature extraction stage has a series of convolutional (CONV) layers with batch normalization (BN) or pooling (POOL) layers in between, while the classification stage has a series of fully-connected (FC) layers producing the output predictions. We find through experiments that the CONV layers are resilient against fault injection, and the BN and POOL layers often "absorb" the SDC fault injections, with the capability to mask them from propagating to the next layers and affecting the final output. In contrast, injecting faults on the later classification stage produces more misclassifications, indicating that FC layers are more sensitive to fault injection.

As shown in Figure 8, there are few parameters available in the VFS software interface to control the overdrive timings, including the rising time before reaching the target OPP (r). We use $r = 32ms$ to push the fault injection toward the classification stage of CNNs for misclassifications. We conduct an end-to-end attack on a deep learning model for COVID-19 recognition as detailed in the next section.

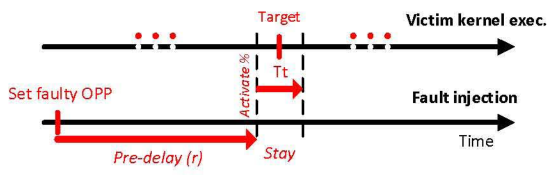

Figure 8: Overdrive timing parameters.

3.2.3 Experimental Results. Our training and test data sets are downloaded from a public dataset collection [40]. We port the ResNet-18 model trained in Section 3.1 to OpenCL for the AMD GPU. The accuracy of the original trained model is 94.73% for inference.

We then set the OPP to (1008mV, 1388Mhz) and the rising time to 32ms (pre-delay), and launch an overdrive fault attack during the ResNet-18 image inference. With these settings, we could observe 86 more misclassified images, reducing the inference accuracy to 83.13% significantly. Figure 9 shows four examples of misclassified images produced by the ResNet-18 inference engine under the overdrive attack. We could change the CXR image classes from Normal to Pneumonia or COVID-19, and from Pneumonia to Normal or COVID-19, i.e., overall false positives. However, we have not observed false negative misclassifications, i.e., from COVID-19 to Normal or Pneumonia.

We conclude that the GPU overdrive attack can pose a serious threat to deep learning based recognition engines that are deployed for large-scale and automatic medical image classification.

3.2.4 Countermeasures. Overall, SDCs are hard to detect by software or reliability measures on GPUs.

We outline both software and hardware countermeasures against GPU overdrive fault injections.

- **Software:** As the culprit lies in the faulty OPPs, the GPU kernel and firmware can be augmented to block suspicious OPPs, with extensive characterization of the target GPU. Considering effects such as temperature variations, aging, and process variations, this procedure can be calibrated regularly.
- **Hardware:** The underlying VFS mechanism can be improved to prevent faulty OPPs from being taken. Hardware approaches include adding hardware guardbands in the GPU's power management circuitry to rollback unsafe voltage and frequency settings to the safe vendor stipulated OPPs promptly. Specifically the clock stretcher circuits will be modified and voltage regulators can respond more quickly.

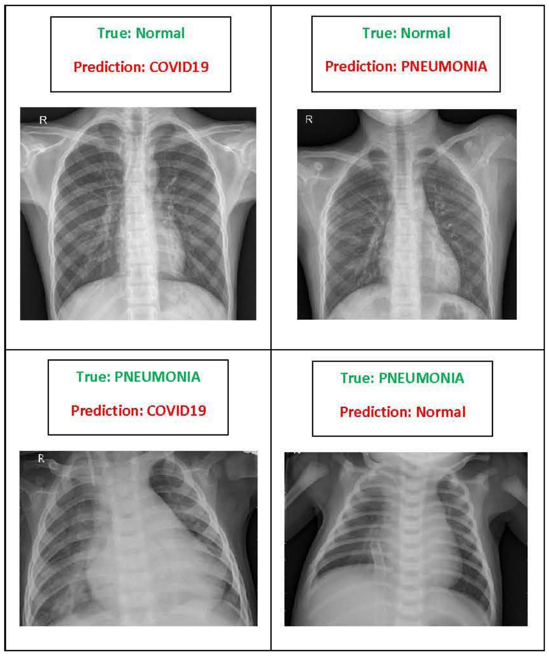

Figure 9: Four examples of the misclassifications introduced by the overdrive fault attack.

4 CONCLUSIONS AND FUTURE WORK

Confidentiality, integrity and reliability are paramount in much of data-intensive computing, but especially so for medical applications, which typically process highly sensitive customer data. DNN deployed in medical devices open up a new attack surface. In this work, we presented four attacks targeting this new surface. Two passive attacks can steal the valuable IP of the DNN models. The threat of active attacks in medical applications is demonstrated by an adversarial attack and a fault injection attack. Individual countermeasures are discussed for each attack. Applying all these countermeasures on top of each other to protect a DNN inference engine against all attacks is not practical. For future work, we propose to build a comprehensive threat model for DNN applications, to enable the design of countermeasures that defend against multiple types of attacks.

Our new attacks are first-of-their-kind, revealing an under-explored attack surface in modern applications, i.e., DNN inference engines. Insights gained during attack exploration will provide valuable guidance to effectively protect against reverse engineering and integrity violations.

ACKNOWLEDGMENTS

We would like to acknowledge the funding support of National Science Foundation with grants SaTC-1929300, TWC-1563697, and IUCRC 1916762 (Center for Hardware and Embedded Systems Security and Trust - CHEST).

REFERENCES

[1] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. 2015. On Subnormal Floating Point and Abnormal Timing. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015.* IEEE Computer Society, 623–639.

[2] Anish Athalye, Nicholas Carlini, and David Wagner. 2018. Obfuscated gradients give a false sense of security: Circumventing defenses to adversarial examples. *arXiv preprint arXiv:1802.00420* (2018).

[3] Alessandro Barenghi, Luca Breveglieri, Israel Koren, and David Naccache. 2012. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proc. IEEE* 100, 11 (2012), 3056–3076.

[4] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. 2019. CSI NN: Reverse Engineering of Neural Network Architectures Through Electromagnetic Side Channel. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 515–532. https://www.usenix.org/conference/usenixsecurity19/presentation/batina

[5] Jakub Breier, Xiaolu Hou, and et.al. 2018. Practical fault attack on deep neural networks. In *CCS*. 2204––2206.

[6] Eric Brier, Christophe Clavier, and Francis Olivier. 2004. Correlation power analysis with a leakage model. In *International workshop on cryptographic hardware and embedded systems*. Springer, 16–29.

[7] Sebastien Carr\'e, Victor Dyseryn, Adrien Facon, Sylvain Guilley, and Thomas Perianin. 2019. End-to-end automated cache-timing attack driven by Machine Learning. In *Proceedings of 8th International Workshop on Security Proofs for Embedded Systems (Kalpa Publications in Computing, Vol. 11)*, Karine Heydemann, Ulrich K\"uhne, and Letitia Li (Eds.). EasyChair, 1–16. https://doi.org/10.29007/nwj8

[8] Daniel Genkin, Adi Shamir, and Eran Tromer. 2014. RSA key extraction via low-bandwidth acoustic cryptanalysis. In *Annual Cryptology Conference*. Springer, 444–461.

[9] Cheng Gongye, Yunsi Fei, and Thomas Wahl. 2020. Reverse-Engineering Deep Neural Networks Using Floating-Point Timing Side-Channels. In *Proc. Design Automation Conference.*

[10] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572* (2014).

[11] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and harnessing adversarial examples. In *International Conference on Learning Representations (ICLR).*

[12] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. 2016. Flush+Flush: A Fast and Stealthy Cache Attack. *Lecture Notes in Computer Science* (2016), 279–299. https://doi.org/10.1007/978-3-319-40667-1_14

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[14] Weizhe Hua, Zhiru Zhang, and G. Suh. 2018. Reverse engineering convolutional neural networks through side-channel information leaks. 1–6. https://doi.org/10.1145/3195970.3196105

[15] IEEE Standard Association. 2008. IEEE Standard for Floating-Point Arithmetic. , 58 pages. http://grouper.ieee.org/groups/754/.

[16] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. 2017. SGX-Bomb: Locking down the processor via Rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*. 1–6.

[17] Tahar Jarboui, Jean Arlat, Yves Crouzet, and Karama Kanoun. 2002. Experimental analysis of the errors induced into linux by three fault injection techniques. In *Proceedings International Conference on Dependable Systems and Networks*. IEEE, 331–336.

[18] Timo Kasper, David Oswald, and Christof Paar. 2009. EM side-channel attacks on commercial contactless smartcards using low-cost equipment. In *International Workshop on Information Security Applications*. Springer, 79–93.

[19] Mingyu Kim, Jihye Yun, Yongwon Cho, Keewon Shin, Ryoungwoo Jang, Hyun-jin Bae, and Namkug Kim. 2019. Deep learning in medical imaging. *Neurospine* 16, 4 (2019), 657.

[20] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. 2014. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. *ACM SIGARCH Computer Architecture News* 42, 3 (2014), 361–372.

[21] June-Goo Lee, Sanghoon Jun, Young-Won Cho, Hyunna Lee, Guk Bae Kim, Joon Beom Seo, and Namkug Kim. 2017. Deep learning in medical imaging: general overview. *Korean journal of radiology* 18, 4 (2017), 570–584.

[22] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks Are Practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, USA, 605–622. https://doi.org/10.1109/SP.2015.43

[23] Yannan Liu, Lingxiao Wei, Bo Luo, and Qiang Xu. 2017. Fault injection attack on deep neural network. 131—-138.

[24] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2017. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083* (2017).

[25] Philippe Maurine. 2012. Techniques for em fault injection: equipments and experimental results. In *2012 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 3–4.

[26] Pranav Rajpurkar, Jeremy Irvin, Kaylie Zhu, Brandon Yang, Hershel Mehta, Tony Duan, Daisy Ding, Aarti Bagul, Curtis Langlotz, Katie Shpanskaya, et al. 2017. Chexnet: Radiologist-level pneumonia detection on chest x-rays with deep learning. *arXiv preprint arXiv:1711.05225* (2017).

[27] Ashay Rane, Calvin Lin, and Mohit Tiwari. 2016. Secure, Precise, and Fast Floating-Point Operations on x86 Processors. In *USENIX Security Symp.* 71–86.

[28] Majid Sabbagh, Yunsi Fei, and David Kaeli. 2020. A Novel GPU Overdrive Fault Attack. In *Proc. Design Automation Conference.*

[29] M. Sabbagh, C. Gongye, Y. Fei, and Y. Wang. 2019. Evaluating Fault Resiliency of Compressed Deep Neural Networks. In *IEEE Int. Conf. on Embedded Software & Systems (ICESS).*

[30] Hoo-Chang Shin, Kirk Roberts, Le Lu, Dina Demner-Fushman, Jianhua Yao, and Ronald M Summers. 2016. Learning to read chest x-rays: Recurrent neural cascade model for automated image annotation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2497–2506.

[31] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2017. CLKSCREW: Exposing the Perils of Security-Oblivious Energy Management. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 1057–1074. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/tang

[32] Niek Timmers and Cristofaro Mune. 2017. Escalating privileges in linux using voltage fault injection. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*. IEEE, 1–8.

[33] Jasper GJ Van Woudenberg, Marc F Witteman, and Federico Menarini. 2011. Practical optical fault injection on secure microcontrollers. In *2011 Workshop on Fault Diagnosis and Tolerance in Cryptography*. IEEE, 91–99.

[34] Xiaosong Wang, Yifan Peng, Le Lu, Zhiyong Lu, and Ronald M Summers. 2018. Tienet: Text-image embedding network for common thorax disease classification and reporting in chest x-rays. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 9049–9058.

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

[36] Weilin Xu, David Evans, and Yanjun Qi. 2017. Feature squeezing: Detecting adversarial examples in deep neural networks. *arXiv preprint arXiv:1704.01155* (2017).

[37] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. 2020. Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures. , 2003–2020 pages. https://www.usenix.org/conference/usenixsecurity20/presentation/yan

[38] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[39] Pu Zhao, Siyue Wang, Cheng Gongye, Yanzhi Wang, Yunsi Fei, and Xue Lin. 2019. Fault sneaking attack: a stealthy framework for misleading deep neural networks. In *Proc. Design Automation Conf.*

[40] Yi Zhong. 2020. COVID-19-CXR: A Special And Simple DCNN Models. https://github.com/ZY-ZRY/COVID19-CXR.

[41] Yi Zhong. 2020. Using Deep Convolutional Neural Networks to Diagnose COVID-19 From Chest X-Ray Images. *arXiv preprint arXiv:2007.09695* (2020).