HuffDuff: Stealing Pruned DNNs from Sparse Accelerators

Dingqing Yang
University of British Columbia
Vancouver, BC, Canada
dingqingy@ece.ubc.ca

Prashant J. Nair
University of British Columbia
Vancouver, BC, Canada
prashantnair@ece.ubc.ca

Mieszko Lis
University of British Columbia
Vancouver, BC, Canada
mieszko@ece.ubc.ca

ABSTRACT

Deep learning models are a valuable "secret sauce" that confers a significant competitive advantage. Many models are never visible to the user and even publicly known state-of-the-art models are either completely proprietary or only accessible via access-controlled APIs. Increasingly, these models run *directly on the edge*, often using a low-power DNN accelerator. This makes models particularly vulnerable, as an attacker with physical access can exploit side channels like off-chip memory access volumes. Indeed, prior work has shown that this channel can be used to steal dense DNNs from edge devices by correlating data transfer volumes with layer geometry.

Unfortunately, prior techniques become intractable when the model is *sparse* in either weights or activations because off-chip transfers no longer correspond exactly to layer dimensions. Could it be that the many mobile-class sparse accelerators are inherently safe from this style of attack?

In this paper, we show that it is feasible to steal a pruned DNN model architecture from a mobile-class sparse accelerator using the DRAM access volume channel. We describe HuffDuff, an attack scheme with two novel techniques that leverage (i) the boundary effect present in CONV layers, and (ii) the timing side channel of on-the-fly activation compression. Together, these techniques dramatically reduce the space of possible model architectures up to *94 orders of magnitude*, resulting in fewer than 100 candidate models — a number that can be feasibly tested. Finally, we sample network instances from our solution space and show that (i) our solutions reach the victim accuracy under the iso-footprint constraint, and (ii) significantly improve black-box targeted attack success rates.

CCS CONCEPTS

• **Security and privacy** → **Side-channel analysis and countermeasures**; • **Computer systems organization** → **Neural networks**.

KEYWORDS

Side-channel attacks, Sparse DNN accelerators

ACM Reference Format:
Dingqing Yang, Prashant J. Nair, and Mieszko Lis. 2023. HuffDuff: Stealing Pruned DNNs from Sparse Accelerators. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3575693.3575738

1 INTRODUCTION

Commercial products that rely on deep learning (DNN) have become common in both the cloud and mobile spaces. Thus, their machine learning (ML) models and datasets used in DNN training have become valuable intellectual property, vital for maintaining a company's competitive advantage. Consequently, key models are often not published, and even non-profit institutions avoid releasing their ML models, gate them behind invitation-only APIs, or delay their release, ostensibly due to concerns about misuse [67–70].

Why steal DNN models: Most importantly, knowing the model architecture enables **followup attacks** on DNN inference engines, such as adversarial example generation [22], stealing weights [87], or membership inference attacks [80]. Moreover, the desire to "adapt" a competitor's DNN model is perhaps unsurprising, as improvements on tasks like ImageNet [13] largely come from new model architectures [29, 32, 82]. The alternative, such as developing and training an in-house model, tends to be far more expensive. However, if the adversary can reverse-engineer an unpublished model's architectural parameters (layer dimensions, pruning factors, etc.), their effort to develop an efficient model is dramatically reduced [33].

DNN inference vulnerability — edge vs. datacentre: Our paper focuses on models that are deployed on edge devices in the field, as opposed to those in a datacentre. In this scenario, the attacker can relatively easily obtain physical access to the device. Many such devices do not offload computation (such as DNN inference) to the cloud for privacy concerns and transmission power limitations, instead performing their computation locally on-device. In addition, such devices often have relatively limited compute capabilities and low power envelopes [9, 10, 72, etc] compared to cloud accelerators [45, etc], which encourages using pruned DNN models.

Many DNN edge deployment scenarios exist, including safety-critical applications like autonomous driving [85] and robotics [47], as well as wearable health applications [60] where privacy matters. When DNNs are used for applications where safety and privacy are paramount, they are often IP-protected, and deploy on devices with extra security protections like memory encryption [25] or even light-weight SGX-like secure enclaves [34, 54, 81] to protect the DNN model from theft.

Edge-deployed DNNs invoke new security concerns as compared to datacentre-deployed DNNs. Various side-channel attacks have been showcased for CPUs [92] and GPUs [62, 89] in datacentres, but these attacks are less applicable on edge devices. Attacks in a datacentre [62, 89, 92] typically leverage shared resource contention between the adversary and the victim to compromise a multi-user system. In contrast, edge devices typically lack support for virtualization and have only a single user [40]. Thus, attacks and threat models designed for datacentres are generally not applicable to edge devices.

As edge devices are typically deployed in the field, they can be physically accessible to an adversary (attacker) and are vulnerable to a broad range of attacks [40]. For instance, a number of attacks have been demonstrated on accelerators targeting Dynamic Random Access Memory (DRAM) bus snooping [33, 35], coldboot attacks [90], physical side channels like electromagnetic (EM) or power signatures [4, 91]. Moreover, physical access to devices enables invasive attacks such as decapsulation [5] and microprobing [88]. For example, DeepLaser [5] decapsulates the chip and uses a laser to cause bit flips to violate output integrity. Performing such attacks requires a specialized lab and is usually destructive to the device. Overall, as compared to the cloud (e.g., the attacker could be a datacentre employee), physical access is typically easier in edge devices, as attested by prior work [26, 33, 35, 37, 90].

In this paper, we focus on the threat model that places the *fewest* limits on the attacker. Our threat model *only requires physical access* to the device to monitor the DRAM bus. This is realistic, as typical accelerator designs [2, 10, 20, 30, 46, 48, 53, 56, 72, 84, 93, 95, 96] consist of an on-chip accelerator SoC and external off-chip DRAM, with the off-chip DRAM either in a socket (e.g., via a DIMM) or directly mounted on the same PCB. For the former, the Hybrid Memory Trace Tool (HMTT) [39] can be used to probe; for the latter, tools like [86] are able to perform measurements in the currently dominant Surface Mount Technology (SMT) [74]. Edge devices that fall in this category include Raspberry Pi 4, Google Nexus One, etc.

Limitations of prior attacks: (Un)fortunately, prior model stealing attacks [33, 35] do not work on *sparse* accelerators that execute *pruned* models (i.e., which skip zero weights and/or activations). Pruning, however, is common in edge devices as it can dramatically reduce the model size, with 90% or more of the weights set to zero [17, 28]. A sparse accelerator can leverage pruning to significantly reduce the energy and latency of inference [10, 20, 27, 72].

Irregular sparsity also makes reverse-engineering DNN architectures significantly more difficult. This is because the volume of data transferred for each tensor is compressed (to eliminate the zeros) and no longer directly corresponds to tensor dimensions; thus, their memory-related side-channel information is obfuscated.

Table 1 illustrates the magnitude of the problem. We first apply ReverseCNN [35], the state-of-the-art DRAM volume side-channel attack, to an Eyeriss-like [9] dense accelerator running ResNet-18 [29]. This attack yields only 8 possible solutions. Then, we straightforwardly extend this approach to attack sparse models on a variant of the Eyeriss that accommodates weight and activation sparsity. This yields a whopping 4×10^{96} solutions — a number that is clearly impossible to train and evaluate.

Key insights: This paper overcomes this problem by using a novel attack that leverages two key insights:

- We observe that Convolutional (CONV) layers exhibit a *boundary* effect [23, 24, 41, 79]. This means that features at the *edges* are *not* translationally equivariant with the shift operations as opposed to the features elsewhere. Therefore, probing the accelerator with multiple carefully constructed images collectively allows us to detect boundary effects across many layers. This helps determine filter dimensions, stride factors, and pooling parameters based on different boundary responses.
- A sparse accelerator's post-processing unit performs on-the-fly encoding that compresses the dense partial sums into sparse output feature maps. This means that a timing side channel can be used to reveal the ratio between the sizes of dense partial sums across different layers.

Table 1: Solution space and resources required to reverse engineer dense ResNet-18 using ReverseCNN [35] and sparse ResNet-18 (pruned by 10×) using the Lottery Ticket Hypothesis [17].

	Number of solutions	Resources required
Dense	8	16 GPU hours
Sparse	4×10^{96}	9.1×10^{92} GPU years

These generic insights apply to all inference accelerators with irregular sparsity that we know of [2, 10, 20, 30, 46, 48, 53, 56, 72, 84, 93, 95], as well as a vast range of pruned DNN architectures.

Contributions: In developing our attack, HuffDuff, we make the following four contributions:

- We identify patterns of inputs that can be fed to a layer to predictably trigger different off-chip traffic volumes. This allows us to determine the filter dimensions.
- We show how to construct inputs that create such patterns many layers downstream, revealing geometries of layers for which we cannot directly provide inputs.
- We describe how to collectively use multiple probes to overcome unobservable boundary effects.
- We identify a compression-time side channel that reveals the ratio between partial sum footprint across all layers, which further reveals their channel counts. Since the boundary effect is agnostic to channel counts, this fills up the missing component that the prober cannot identify.

HuffDuff can reverse-engineer pruned modern deep CNNs within hours, typically yielding less than a hundred possibilities that can be trained and tested in a reasonable time by an attacker. Our evaluation shows that HuffDuff solutions reach the victim accuracy and raise the black-box targeted attack success rate to a semi-white-box level (in which the attacker knows the correct architecture).

2 THREAT MODEL

Our threat model reflects a situation where an *unstructured pruned* DNN model is executing inference tasks locally on an *edge* device. The DNN accelerator supports *2-sided* unstructured sparsity with off-chip DRAM. We assume that the attacker has *device access* and can provide inputs (e.g., via a camera) and monitor *chip↔DRAM transfer volume* (but not contents).

This threat model, illustrated in Fig. 1, is the same as that of prior work [35], except that we allow the DNN accelerator to support

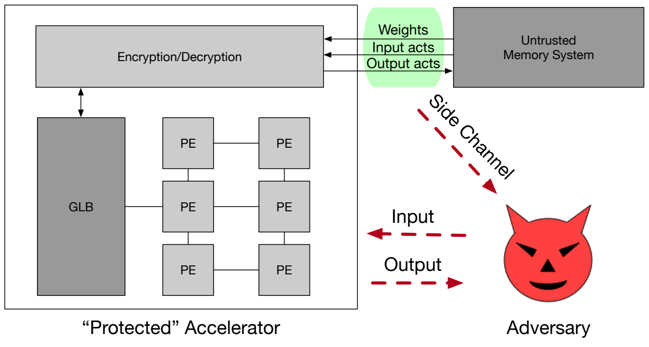


Figure 1: The threat model of HuffDuff (inspired by [35]). It consists of a trusted sparse DNN accelerator and an untrusted external memory system.

sparse execution. We describe our threat model in more detail below.

Attacker's objective: The attacker aims to reverse-engineer the DNN model architecture being used for inference such that the attacker can (a) re-train to obtain a model with on-par or better accuracy and efficiency, or (b) use the knowledge gained to mount follow-up attacks, such as generating adversarial examples [22], model extraction [87], or membership inference [80].

The aim is to determine all network architectural hyperparameters including (a) the *layer geometry* (input size, output dimensions, filter dimensions) of each layer in the DNN, (b) the *dataflow graph* among the layers, and (c) the *weight sparsity factors* for each layer. Once these hyperparameters are determined, the attacker can retrain the model on their own data, obtaining models with a similar level of accuracy and resource efficiency (e.g., sparse footprint).

Next, we explain how reverse-engineering these models can help mount follow-up attacks using adversarial example generation [22] as an example.

Adversarial example generation is a process of turning a benign input sample into a malicious sample by adding small perturbations that are indistinguishable to human eyes. There are well-established adversarial example generation algorithms like FGSM [22] for the white box setting where the attacker has access to the victim model; FGSM leverages gradient information to find perturbations that maximize the loss versus the true label under the infinite-norm constraint.

However, in a black box setting like our threat model, the attacker usually has no access to any model information like model architecture and parameters. Sometimes, the attacker can rely on *transferability* between models [71]: adversarial examples generated from performing a white-box attack on a random surrogate can compromise the black-box victim models. Such transferability is, however, limited to simple attacks [59] on small-scale datasets, and non-targeted attacks where any misprediction counts as success. In reality, targeted attacks are usually more threatening, with severe consequences that are crucial to mitigate.

To boost the targeted attack success rate, prior works [59, 65] have identified that network architecture similarity between the surrogate and the victim plays an important role. Intuitively, this makes sense, as an architecturally similar surrogate provides more accurate gradient information than a random surrogate. Indeed, Deepsniffer [33] demonstrates that targeted attack success rates increase if the surrogate is from the same model family as the victim, and shows significant improvement in targeted attack success rates with reverse-engineered surrogates (as compared to random surrogates from a model zoo). Therefore, reverse-engineering the victim architecture is crucial to improve (and even enable) follow-up attacks.

Attacker's capabilities: The attacker has physical access to the device and can monitor the signals while the device is executing through a DRAM tracing tool like HMTT [39] or other probes [86]. Specifically, we assume the attacker can observe distinct DRAM accesses with addresses and operation types (read or write) for each access; this is the same assumption made by prior attacks [33, 35]. The attacker can also construct bespoke inputs (e.g., images) to be processed by the accelerator [35] (e.g., by faking camera inputs).

In contrast, we assume that the attacker is *unable* to observe or manipulate the *data* being read or written from DRAM (e.g., due to data encryption), and cannot observe internal on-chip states.

Workload: We assume that our victim is a Convolution Neural Network (CNN) that is statically pruned in an unstructured manner [17, 28] for maximum compression. Structured pruning, while also in use, is a simpler case, so we focus on unstructured pruning here[1]. We assume that the victim uses ReLU activation where negative values are clamped to zero; this enables accelerators to transfer compressed activations to save energy. Additional optimization such as magnitude-based dynamic activation pruning [57] can be viewed as ReLU generalized to a non-zero cut-off. In contrast, dynamic activation pruning is rarely used compared to weight pruning; unlike weights, ineffectual activations cannot be statically pruned and detecting them adds runtime overhead. Overall, ReLU generates a decent amount of zeros, and further dynamic pruning only provides marginal savings.

Execution environment: We assume that the model executes on a dedicated edge DNN accelerator comprising (a) a systolic-array-like accelerator chip and (b) off-chip memory (Fig. 1 "protected" accelerator). We allow the accelerator to support both *weight* and *activation* sparsity, with zero-skipping during execution. It also supports compressing weight and activation tensors during off-chip memory communication. We also assume the accelerator performs layerwise execution so that the entire footprint of all data types is present in the external DRAM memory system at least once. This corresponds to a vast range of edge-class DNN accelerators proposed in the literature [10, 20, 30, 46, 48, 53, 56, 72, 84, 93, 95].

We allow encrypted tensor data to be transferred off-chip. Memory encryption techniques have been studied to protect physical memory attacks from mobile devices [25], and commercial products like ZeroPoint Secure memory [94] have been deployed. SGX-like secure enclaves [11, 12, 97] that further provide integrity and freshness guarantees also encrypt data transferred off-chip through a Memory Encryption Engine (MEE). Although full SGX support might be overkill and create huge performance overhead for DNN

[1]See also *Broader Application* below.

accelerators, light-weight SGX techniques have been used in accelerators like GuardNN [34], TNPU [54], and Seculator [81], which provide confidentiality, integrity, and freshness guarantees at coarser (tile/layer) granularity. Meanwhile, we do not require memory addresses to be contiguous in DRAM. However, we assume that a written DRAM address maintains its value (unlike in ORAM [19]) and traffic volumes are not obfuscated by injecting random requests (as they would be in ORAM). We believe ORAM-like measures that inject faux requests and autonomously move data in DRAM are far too energetically expensive to be practical in an edge accelerator.

Finally, as common practice, we assume that operations such as batch normalization, ReLU, accumulator quantization, and on-the-fly activation compression are handled within the post-processing module on-chip [9, 10, 45, 63, 72].

Excluded configurations: We exclude SRAM-only accelerators [6, 15, 27][2], which do not have off-chip memory accesses. We also exclude accelerators that execute multiple layers on-chip [3, 18, 38]; indeed, we are unaware of any *sparse* DNN accelerators that do that.[3]

Broader application: Our choice of workloads and execution environments correspond to the most challenging case where all available side-channel information is blurred. Memory volume of all data types (weights, input, and output activation) cannot directly correspond to layer geometries due to the unknown amount of pruned or compressed zeros. Execution time also cannot correspond to layer geometries we have an unknown amount of skipped zeros.

Our techniques apply to a broader range of workloads and environments than the ones specified above. In fact, relaxing some of these assumptions makes the problem easier to solve: for example, executing pre-activation batch-norm layers separately means that additional side-channel information on the exact activation tensor volumes (since partial sums are typically dense) is also revealed. Similarly, accelerators with structured sparsity [14, 36, 64] can be attacked by existing techniques for dense execution [33, 35], since the transfer sizes do not vary with data content.

3 DENSE-CASE: APPROACH AND SOLUTIONS

We first formulate the task for the simpler case where the DNN model is dense (not pruned) and the accelerator does not support sparse execution. We then review the analytical solution approach employed by prior work [35], which we refer to as ReverseCNN.

3.1 Problem Formulation

Recall from Section 2 that the attacker aims to determine the model's architectural parameters, shown in Table 2. This includes (1) input activation tensor dimensions X, Y, C; (2) output activation tensor dimensions P, Q, K; (3) kernel dimensions R, S, C, K; (4) convolution stride $STRIDE_X$ and $STRIDE_Y$; and (5) pooling layer factors $POOL_X$ and $POOL_Y$. The attacker can observe the **type, address, and transfer sizes** to/from off-chip memory, but cannot decipher the data.

Table 2: Symbols for Input, Output, and Weight tensors: uppercase = actual; lowercase = unknown.

I and O	input/output activation tensor transfer sizes
W	weight tensor transfer size
C and K	number of input and output channels
$X \times Y \times C$	output activation map dimensions
$P \times Q \times K$	input activation map dimensions
$R \times S \times C \times K$	weight tensor dimensions
$STRIDE_X$, $STRIDE_Y$	width and height convolution stride
$POOL_X$, $POOL_Y$	width and height pooling factors

We use the convention that uppercase symbols indicate the *actual* (possibly unknown) dimensions, while lowercase letters indicate the corresponding *variables* in constraint equations.

3.2 Prior Solution: ReverseCNN

ReverseCNN [35] finds the hyperparameters of interest by formulating constraint equations that relate the observed off-chip memory traffic volumes to layer dimensions.

Their key observation is that the **read-after-write (RAW) dependency between layers** must be preserved independent of any micro-architectural details or mapping/scheduling choices. Thus, the output feature map of one layer becomes the input feature map of one or more layers downstream. Because the attacker is able to monitor the DRAM addresses, we can identify these dependencies regardless of how the tensors are laid out in the address space.[4]

From this, one can also determine the memory footprint of the input and output activation tensors (I and O) as follows. For the first layer, the size of I is known, as the attacker controls the inputs to the accelerator (e.g., by spoofing camera outputs) [35]. For each subsequent layer, each activation layer O is first written to some memory addresses, and then the same addresses are later read (possibly more than once) as the input I of another layer. This yields the footprint of I and O, as well as the boundaries between processing different layers. Weights are not modified during inference, so the footprint of tensor W for the layer can be determined by identifying read-only addresses accessed during the layer's processing.

Once the traffic volumes for I, O, and W are known, ReverseCNN formulates the following set of equations for each layer to determine the channel counts c and k, the activation and dimensions x, y, p, and q, the filter dimensions r and s, convolution stride $stride_x$ and $stride_y$, and pooling factors $pool_x$ and $pool_y$:

$$x \times y \times c = \text{size}(I) \tag{1}$$

$$\frac{p \times q \times k}{pool_x \times pool_y} = \text{size}(O) \tag{2}$$

$$r \times s \times c \times k = \text{size}(W) \tag{3}$$

$$x = stride_x \times p + r - stride_x \tag{4}$$

$$y = stride_y \times q + s - stride_y \tag{5}$$

$$r = s;\ \ x = y;\ \ stride_x = stride_y;\ \ pool_x = pool_y \tag{6}$$

[2]Cerebras wafer-scale accelerators (WSE-2) [6] are primarily for training large DNNs, and they do have external DRAM called MemoryX [58] from which weights are streamed into the accelerator. Nevertheless, WSE-2 has a huge 40 GB on-chip SRAM that might fit the entire model during inference.

[3][66] is a sparse accelerator that fuses bottleneck blocks, but they are then executed as if they were single layers.

[4]Note that this holds even if the memory is reused: each write generates a new name or "version" for the address as is typically done when converting code to Static Single Assignment (SSA) form [77].

Following ReverseCNN [35], we assume that activations, filters, strides, and pooling layers are symmetric (Eq. 6); CNNs for vision typically fit these assumptions [61].

This helps reverse-engineer a single layer. ReverseCNN [35] relies on induction to extend this to multiple layers. First, recall that x, y, and c for the first layer are known because the attacker can observe (and, indeed, craft) inputs to the chip. Then, any layer reading the O tensor from the first layer will have the following dimensions for its I tensor (due to inter-layer RAW data dependency):

$$x_{next} = \frac{p}{pool_x},\ y_{next} = \frac{q}{pool_y},\ \text{and } c_{next} = k \tag{7}$$

This allows ReverseCNN [35] to again apply Eqs. 2–6 and recursively solve for the geometry of all layers.

In this way, ReverseCNN [35] can reverse-engineer most dense DNNs: as shown in Table 1, it yields only 8 possible solutions for dense ResNet-18.

4 TACKLING SPARSE MODELS

The problem becomes significantly more complicated when pruned models run on a sparse DNN accelerator [10, 20, 72] or a DNN accelerator that compresses tensors when they are transferred to and from off-chip memory [9].

4.1 Challenges

If we employ a sparse accelerator, the data blocks that are transferred to/from off-chip DRAM **no longer correspond directly to the relevant tensor dimensions**. This is because sparse accelerators compress tensors for both evaluation and transfer by eliding zeros. This is the case for both weight tensors (where the attacker does not know the pruning factor) and activation tensors (which depend both on weight values and input activation values). Because of this, ReverseCNN's Eqs. 1–3 no longer hold. Instead, we have the following three *inequalities*:

$$x \times y \times c \geq \text{size}(I) \tag{8}$$

$$\frac{p \times q \times k}{pool_x \times pool_y} \geq \text{size}(O) \tag{9}$$

$$r \times s \times c \times k \geq \text{size}(W) \tag{10}$$

In other words, these inequalities state that the size of any tensor is *at least* as large as the corresponding DRAM transfer volume observed by the attacker. However, as this is only a *lower bound*, these tensors could be much larger depending on the pruning factor or activation sparsity. Note that there are *no upper bounds* for any of the unknowns, so solving this system of inequalities will yield an **infinite number of solutions** even for a single layer.

4.2 Naïvely Handling Sparsity

One might think that solving Eqs. 8–10 is simply a matter of establishing an upper bound for the expected sparsity — perhaps by profiling many different models for a related task — and obtaining a finite number of solutions. To understand this, let's write α to mean this maximum sparsity for the weight tensor, where $\alpha = 0.9$ means that 90% of the weights have been pruned away. This gives us an additional equation that is denoted as follows:

$$r \times s \times c \times k \leq \frac{\text{size}(W)}{1-\alpha}. \tag{11}$$

Unfortunately, sparsity levels can vary significantly among layers, even for some optimally pruned nets like a 10× compressed ResNet-18 [29] — for example, the first and final layers are typically hard to prune whereas intermediate layers can be quite sparse. This means that the upper bound is likely to be a very high sparsity factor (e.g., $\alpha = 0.999$); indeed, the Conv5_3 layer of our pruned version of VGG-S has 3627 out of 2359296 weights that are retained, corresponding to $\alpha = 0.9985$ with no loss of accuracy. So the question is, how well does such a bound constrain the solution space?

To better understand this, let us consider solving for a layer's output channel count k using Eqs. 10 and 11. Let us assume that we have already solved the prior layer, so we know the actual value of c from the data dependency constraint on the previous layer's output activations (Eq. 7). Again, we will denote actual values with uppercase letters and constraint variables as lowercase, so here we know that $c = C$. Let's for the moment imagine that we also know that $r = R$ and $s = S$ so that only k is unknown. We will denote the *actual* weight sparsity as β, and the *assumed* upper bound on the sparsity as α. Substituting Eq. 11 into Eq. 10 yields:

$$\text{size}(W) \leq R \times S \times C \times k \leq \frac{\text{size}(W)}{1-\alpha} \tag{12}$$

Rewriting the observed weight footprint W in terms of the actual sparsity β then gives us the following equations.

$$(1-\beta)RSCK \leq R \times S \times C \times k \leq \frac{(1-\beta)RSCK}{1-\alpha} \tag{13}$$

$$(1-\beta)K \leq k \leq \frac{(1-\beta)K}{1-\alpha} \tag{14}$$

Observe that the tightness of this bound is determined by (a) the actual weight sparsity β, and (b) how close the upper bound α is to β. While we now have a finite number of solutions, typical ranges for β can be around 50% up to 99.9%, yielding very loose bounds. For example, for VGG-S [82] and ResNet-18 [29], we have 2.6×10^{74} and 4×10^{96} of possible solutions for the whole network, a number of possible geometries that is infeasible to train and evaluate.

In the next three sections, we show how to reduce the number of solutions to a manageable level by (i) actively probing the accelerator with carefully constructed input patterns and (ii) exploiting architectural insights about DNN accelerators.

5 LEARNING VIA ACTIVE PROBING

5.1 Exploiting the Boundary Effect

Convolutional layers in modern CNNs are not fully translationally equivariant, a phenomenon known as the *boundary effect* [23, 24, 41, 79]. This effect arises on the edges of an input feature map, where the part of the convolution filter that is outside of the feature map does not contribute to the output activation. Typically CNNs perform zero padding in this case [61]. We will take advantage of this to determine the dimensions of the convolutional filters in the model under attack.

To understand this effect, let us first examine how a single-channel 1D convolution is affected by different inputs. Fig. 2 shows a 3×1 filter $[3, 4, 5]$ on three 5×1 inputs, $[1, 0, 0, 0, 0]$, $[0, 1, 0, 0, 0]$, and $[0, 0, 1, 0, 0]$, to get a 5×1 output. Note that, for the first input (Fig. 2a), the leftmost element of the filter (1) is always out of bounds, and never involved in the convolution or reflected in the output. However, for the second and third inputs (Fig. 2b and Fig. 2c), all of the filter elements make it to the output vector.

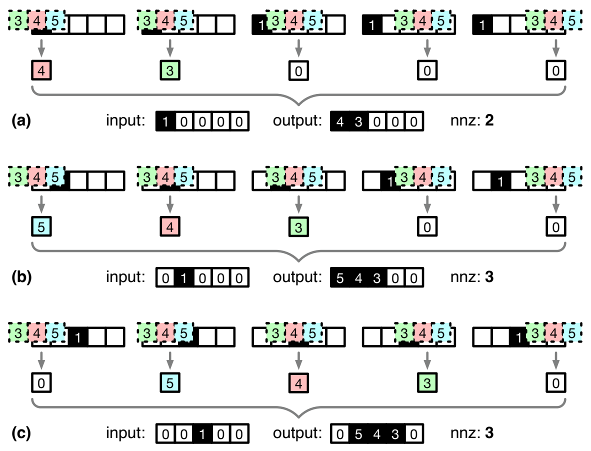


Figure 2: Boundary effect in 1D convolution and its outcome on different inputs. nnz = # of non-zero elements in the output. ReLU activation is omitted for clarity.

Now, note that the *number of non-zero elements* (nnz) is the same in panes (b) and (c), but different in pane (a). This difference tells us something about the size of the filter. Specifically, the difference in the nnz between (a) and (b) tells us that there is at least one filter element to the left of the filter center, and the fact that the nnz is the same for (b) and (c) tells us that there is at most one. Applying the same reasoning on the right input boundary (not shown) allows us to conclude that the filter is 3×1 and centered around the second element. If the filter were 1×1, all cases would have the same nnz, and if the filter were 5×1, all cases would have different nnzs.

How can we take advantage of this? Our threat model allows the attacker to craft inputs to the accelerator, so we can certainly probe at least the first layer of the CNN this way. In a dense accelerator, we cannot observe this difference, because an attacker can only observe the *volume* of memory transfers can be observed, not the actual values (e.g., on account of encryption). But in a *sparse* accelerator that compresses activations for storage [9, 10, 20, 72], the size of the tensor being transferred will be different because the zeros will be elided from the output activation tensor.

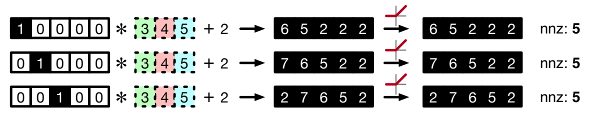
This gives us an intuition for determining the convolutional filter dimensions r and s: we will probe the accelerator with inputs crafted to determine the filter size (using the 2D equivalent of Fig. 2), measure the transferred activation size to find the number of non-zeros, and compare the different cases to determine the filter size.

However, two difficulties arise in practice. First, the attacker has direct control over the input queries for only the first layer, and cannot directly craft any of the intermediate feature maps. The inputs to the second layer will have passed through the first layer's convolutional filters, so there is no reliable way to create the pattern of single activations surrounded by zeros shown in Fig. 2. Stride and pooling further obfuscate this.

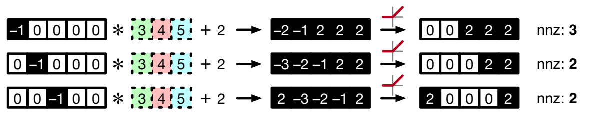
Second, in practice convolution layers are affine, i.e., they either have an additive bias or are followed by a batch normalization layer. This means that the input zeros may not propagate to the output: for example, if there is a bias of +1 in Fig. 2, the output in all three cases will have five non-zeros. We show how to address these difficulties in the next sub-section.

5.2 Handling Bias and Batch Normalization

Bias and the additive term in batch normalization can render the technique above ineffective. For example, if the convolution includes a bias term of +2, the cases in Fig. 2 become:


1 0 0 0 0 ∗ 3 4 5 + 2 → 6 5 2 2 2 → 6 5 2 2 2 nnz: 5
0 1 0 0 0 ∗ 3 4 5 + 2 → 7 6 5 2 2 → 7 6 5 2 2 nnz: 5
0 0 1 0 0 ∗ 3 4 5 + 2 → 2 7 6 5 2 → 2 7 6 5 2 nnz: 5

Now, these cases can no longer be distinguished, because the number of non-zeros is the same for all three. To mitigate this, we make two observations: (i) the probe inputs can be any number, not just 1, and (ii) the ReLU activation function will make all negative values zero in the output feature map. For example, if the probe vector contains −1 instead of 1, the filter footprint will be negative (and thus zero post-ReLU), while the bias terms will be non-zero:


−1 0 0 0 0 ∗ 3 4 5 + 2 → −2 −1 2 2 2 → 0 0 2 2 2 nnz: 3
0 −1 0 0 0 ∗ 3 4 5 + 2 → −3 −2 −1 2 2 → 0 0 0 2 2 nnz: 2
0 0 −1 0 0 ∗ 3 4 5 + 2 → 2 −3 −2 −1 2 → 2 0 0 0 2 nnz: 2

Note that, while the nnz for the edge and non-edge cases is the opposite from Fig. 2 (the edge case has more zeros, not fewer), the two are still observably different, and this is enough to determine the filter size.

With all the examples discussed so far, we start to formalize a single-layer attack. Let's define the conv_layer operation as the composition of CONV, BatchNorm, and ReLU. For features that do not reside on the edge, conv_layer is equivariant to shift operations:

$$\text{conv\_layer}(\text{shift}(x)) = \text{shift}(\text{conv\_layer}(x))$$

Counting the nnz elements on both sides reveals that the number of non-zero element responses is also invariant to shift operations:

$$\begin{aligned}\text{nnz}(\text{conv\_layer}(\text{shift}(x))) &= \text{nnz}(\text{shift}(\text{conv\_layer}(x)))\\ &= \text{nnz}(\text{conv\_layer}(x))\end{aligned}$$

Neither of these holds for features x_e that reside on the edge:

$$\text{conv\_layer}(\text{shift}(x_e)) \neq \text{shift}(\text{conv\_layer}(x_e))$$

While we can't observe the activations themselves (which might be encrypted), we can measure nnz. Different activation tensors are likely to have nnz, which allows us to observe the boundary effect.

Rarely, the boundary effect is *obscured*, and different activation tensors can end up having the same nnz even if their values are different. Essentially, the boundary effect always exists (due to non-equivariance at the edge), but can either be *observable* (different nnz count) or *unobservable* (same nnz count).

In practice, we find that this is not a problem, and boundary effects are usually observable. This is because there are many CONV kernels, and the boundary effect will be obscured only if all of the kernels are unobservable, or their total nnz differences cancel out exactly — both unlikely situations. To estimate the likelihood that the boundary effect is observable, we randomly sampled kernels in pruned models and applied random half-Gaussian inputs; the boundary effect was observable in 77% of the cases.

More importantly, the "unobservability" issue can only cause *false negatives*, and never false positives, as it is impossible to observe a boundary effect when one does not exist. Therefore, we can simply make multiple independent random probes to amplify the probability of observing the boundary effect.

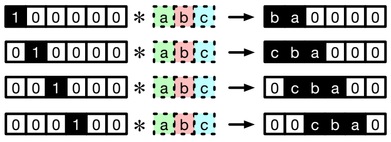
5.3 Probing Downstream Layers

In the bias discussion above, we used "probe vectors" with 0 in the "inactive" positions, and either 1 or -1 in the "active" position where we wish to place the convolution kernel. However, the inputs do not have to be 0, 1, or -1: they can be any values as long as the inactive and active positions have different values. The boundary effect (where some of the filter entries are unused) still occurs, and from the previous section, we already know how to separate any two values in the output activation tensor by taking advantage of ReLU.

This observation gives us a way to probe layers downstream even if we cannot directly inject inputs into those layers. The intuition is that the boundary effect can survive multiple layers, but the footprint of the probe impulse (the "active" position) will get progressively blurred over a larger area as it passes through more layers.

To develop some intuition, let us continue the running example, this time propagating it through two 1D convolutional layers without bias, and generalizing it to arbitrary weight values. For clarity, we will omit ReLU here, but in general, ReLU can be used to distinguish values as in the previous section.

After the first layer, with filter weights $[a, b, c]$, the output activations become:

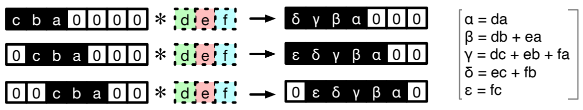


1 0 0 0 0 0 * a b c → b a 0 0 0 0
0 1 0 0 0 0 * a b c → c b a 0 0 0
0 0 1 0 0 0 * a b c → 0 c b a 0 0
0 0 0 1 0 0 * a b c → 0 0 c b a 0

Recall that we can observe the memory traffic for each layer, so we can determine that filter in the first layer is 3×1.

Note that in the first row, only a part of the filter survives in the output, which will cause problems downstream: we will not be able to distinguish whether any observed boundary effect comes from the first or second layer. We therefore drop the first row and proceed with the remaining rows.

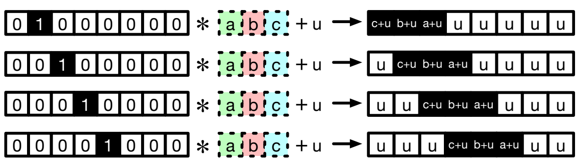
Now, let us apply another CONV layer, with filter $[d, e, f]$:



c b a 0 0 0 0 * d e f → δ γ β α 0 0 0
0 c b a 0 0 0 * d e f → ε δ γ β α 0 0
0 0 c b a 0 0 * d e f → 0 ε δ γ β α 0
α = da
β = db + ea
γ = dc + eb + fa
δ = ec + fb
ε = fc

Observe that our probe impulse is still visible in the output as $[\varepsilon, \delta, \gamma, \beta, \alpha]$ surrounded by zeros. Also, it is still located where the original impulse was.

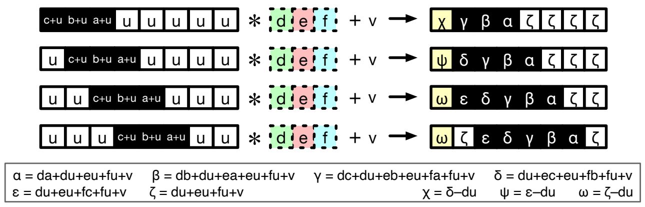
Finally, let us reintroduce non-zero bias into the problem. Probing the first layer (filter $[a, b, c]$, bias u) yields:



0 1 0 0 0 0 0 0 * a b c + u → c+u b+u a+u u u u u u
0 0 1 0 0 0 0 0 * a b c + u → u c+u b+u a+u u u u u
0 0 0 1 0 0 0 0 * a b c + u → u u c+u b+u a+u u u u
0 0 0 0 1 0 0 0 * a b c + u → u u u c+u b+u a+u u u

This is exactly what we saw after the first layer above, except that now we have added u everywhere.

The analysis is a bit more involved after the second layer (filter $[d, e, f]$, bias v):



c+u b+u a+u u u u u u * d e f + v → χ γ β α ζ ζ ζ ζ
u c+u b+u a+u u u u u * d e f + v → ψ δ γ β α ζ ζ ζ
u u c+u b+u a+u u u u * d e f + v → ω ε δ γ β α ζ ζ
u u u c+u b+u a+u u u * d e f + v → ω ζ ε δ γ β α ζ
α = da+du+eu+fu+v β = db+du+ea+eu+fu+v γ = dc+du+eb+eu+fa+fu+v δ = du+ec+eu+fb+fu+v
ε = du+eu+fc+fu+v ζ = du+eu+fu+v χ = δ–du ψ = ε–du ω = ζ–du

This pattern is *almost* the same as the no-bias case above: the two-layer impulse response $[\varepsilon, \delta, \gamma, \beta, \alpha]$ surrounded by ζ, the second layer's filter response to the first layer's bias (this was 0 with no bias). Also, as the first layer's bias u is distinguished from the implicit padding of 0, the first element in each vector (shaded yellow) differs from its later corresponding occurrence later on – because the latter includes the bias response du.

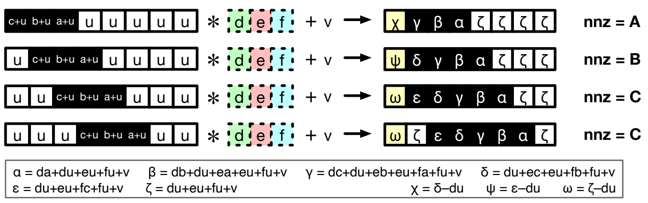
Once we distinguish the various values (see Section 5.2 above), we will conclude that the filter size is 3×1. If we wish to probe the third layer, we again discard the rows where the filter response is partial; in this case, we would discard the first two rows.

Although the running example here is a 1D convolution, exactly the same analysis applies to 2D convolutions, except with more edge cases to distinguish. Other layer types (e.g., pooling) and effects such as stride are also amenable to this kind of analysis; we omit the details for these layers here because of space limitations.

5.4 Handling "Errors" in Downstream Layers

Section 5.3 shows boundary effects on downstream layers with a running example. As discussed in Section 5.2, since we only have partial observability by measuring nnz, some downstream layers could have "errors" (i.e., unobservable boundary effects). We find that it is difficult to find a random probe that has directly observable boundary effects in all layers.

To mitigate this, let us first consider a concrete example of "success", that is, of an unobscured boundary effect with all relevant nnzs different:



c+u b+u a+u u u u u u * d e f + v → χ γ β α ζ ζ ζ ζ nnz = A
u c+u b+u a+u u u u u * d e f + v → ψ δ γ β α ζ ζ ζ nnz = B
u u c+u b+u a+u u u u * d e f + v → ω ε δ γ β α ζ ζ nnz = C
u u u c+u b+u a+u u u * d e f + v → ω ζ ε δ γ β α ζ nnz = C
α = da+du+eu+fu+v β = db+du+ea+eu+fu+v γ = dc+du+eb+eu+fa+fu+v δ = du+ec+eu+fb+fu+v
ε = du+eu+fc+fu+v ζ = du+eu+fu+v χ = δ–du ψ = ε–du ω = ζ–du

Observe that the nnz form the pattern *ABCC*, where *A*, *B*, and *C* are different nnz values. In this case, we have full observability, since the content of the first row and the second row differs from the third or fourth row.

Now, non-observability occurs when an nnz for one row is equal to that of another even if the values are different; here, this could be *ABBB* (partial observability) or *AAAA* (no observability). The last case (*AAAA*) is particularly troublesome because it is the correct output for pointwise layers.

Luckily, the error is one-sided again, because nnz cannot change once the filter has cleared the edge; e.g., it's impossible to observe *ABCD* in this example. Therefore, we only need to look for the longest non-convergent pattern among multiple random probes (e.g., choose *ABCC* over *ABBB* and *AAAA*). With repeated probes, the probability of failure on a layer (i.e., that *none* of the probes demonstrate observability) decreases exponentially with the number of independent random probes.

Next, we will generalize this intuition from this section and present the complete attack scheme.

6 AUTOMATING THE ATTACK

6.1 Generalized Input Pattern

Recall that the inputs observed by any layer in the model will contain a "feature" segment that combines all of the previous layers' filters ($[\varepsilon, \delta, \gamma, \beta, \alpha]$ in the examples in the previous section), surrounded by zero or more responses to the bias term (ζ above). In addition, the initial columns (one column in the example above) contain constants generated by the edge effect applied to the bias term (ω above).

We can generalize this pattern as $\mathcal{A}(m, n)$, where n is the feature length, and m is the number of the initial column constants:

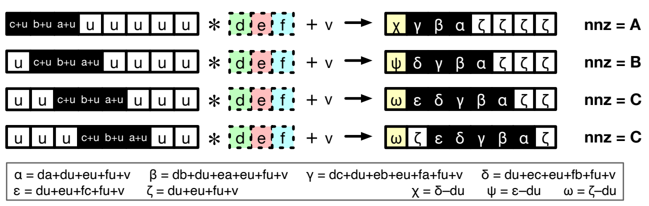
$$\mathcal{A}(m,n) = \{x_i\}_{i=1}^{q}, \text{ where } q = \text{\# of query patterns, and}$$
$$x_1 = s_1, s_2, \ldots, s_m, f_1, f_2, \ldots, f_n, b, b, b, \ldots,$$
$$x_2 = s_1, s_2, \ldots, s_m, b, f_1, f_2, \ldots, f_n, b, b, \ldots,$$
$$x_3 = s_1, s_2, \ldots, s_m, b, b, f_1, f_2, \ldots f_n, b, \ldots, \text{ etc.}$$

For example, our initial input sequence in the previous section can be denoted by $\mathcal{A}(0, 1)$.

6.2 Symbolic Convolution Engine

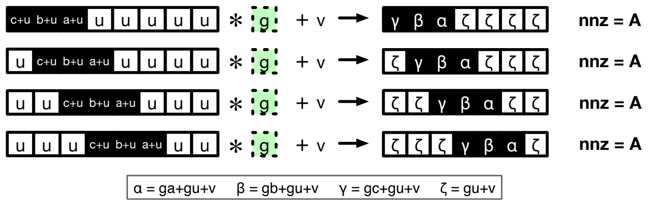
To assist in probing multi-layer networks, we developed an engine that evaluates convolutions *symbolically*. That is, rather than adding and multiplying numbers, it constructs an algebraic expression for the result of the convolution after layer l given (a) a specific input pattern $\mathcal{A}(m, n)$ for layer 1, and (b) a hypothesis for the geometry of layer l (e.g., 3×3 kernel, stride 1, pooling factor 2).

For example, consider again the second layer from the running example, and let us hypothesize that the convolution is 3×1 with no pooling. Having analyzed the first layer, we know that the second layer will receive the probe pattern $\mathcal{A}(0, 3)$. The symbolic convolution engine will yield:


c+u b+u a+u u u u u u * d e f + v → χ γ β α ζ ζ ζ ζ nnz = A
u c+u b+u a+u u u u u * d e f + v → ψ δ γ β α ζ ζ ζ nnz = B
u u c+u b+u a+u u u u * d e f + v → ω ε δ γ β α ζ ζ nnz = C
u u u c+u b+u a+u u u * d e f + v → ω ζ ε δ γ β α ζ nnz = C
α = da+du+eu+fu+v β = db+du+ea+eu+fu+v γ = dc+du+eb+eu+fa+fu+v δ = du+ec+eu+fb+fu+v
ε = du+eu+fc+fu+v ζ = du+eu+fu+v χ = δ–du ψ = ε–du ω = ζ–du

From this, the engine will produce the pattern of the expected number of non-zeros (nnz) observed for each input once ζ has been distinguished from the other values (see Section 5.2). In this case, we have three possible nnz counts, with the last one repeating as the filter leaves the edge; we write this pattern as *ABCC*...

On the other hand, let us hypothesize that the second layer is a pointwise 1×1 convolution. The engine will yield:


c+u b+u a+u u u u u u * g + v → γ β α ζ ζ ζ ζ nnz = A
u c+u b+u a+u u u u u * g + v → ζ γ β α ζ ζ ζ nnz = A
u u c+u b+u a+u u u u * g + v → ζ ζ γ β α ζ ζ nnz = A
u u u c+u b+u a+u u u * g + v → ζ ζ ζ γ β α ζ nnz = A
α = ga+gu+v β = gb+gu+v γ = gc+gu+v ζ = gu+v

which gives the nnz pattern *AAAA*... The reader is invited to verify that for a 3×1 convolution followed by 2×1 max pooling, the expected pattern would be *ABCDCD*....

These nnz patterns allow us to distinguish different types of layers. To do this, the symbolic convolution engine (1) generates expected nnz patterns for each geometry hypothesis *for the current layer*, (2) feeds the first-layer inputs to the accelerator, and (3) compares the output nnzs obtained by snooping on the off-chip memory traffic to determine which layer geometry is correct.

6.3 The Probing Algorithm

Algorithm 1 shows the pseudocode for the probing algorithm.

```
Input: $T$ i.i.d. random probes where each corresponds to input queries
    $\{x_i\}_{i=1}^{l} \in \mathcal{A}(m,n)$, $\#layers$
Output: reverse engineered layer geometry in $result$
for $i \leftarrow 1$ to $l$ do
    for $t \leftarrow 1$ to $T$ do
        nnz[$i$][$t$][$j$] ← Inference ($x$)
    end
end
$result \leftarrow List()$
for $j \leftarrow 1$ to $\#layers$ do
    select a probe $t$ with the longest nnz converging pattern.
    select the valid subset of nnz[:][$t$][$j$] to form $test\_nnzs$
    $patterns$ ← SymbolicConv ($m, n$)
    foreach $k \in$ all possible layer configs do
        if $test\_nnzs$ matches $patterns[k]$ then
            $result[j] \leftarrow k$
            $m, n$ ← DecodeOutPattern ($patterns[k]$)
            break
        end
    end
end
```

Algorithm 1: The HuffDuff probing attack.

SymbolicConv(m,n) symbolically evaluates a given layer for the sequence $\mathcal{A}(m, n)$, and *DecodeOutPattern(...)* determines m', n' for which the layer output matches $\mathcal{A}(m', n')$, so that this can be used in analyzing the next layer.

6.4 Limitations of the Probing Attack

The HuffDuff probing attack is quite powerful in practice: for example, it works across **all** layers evaluated on VGG-S [82] and ResNet-18 [29]. It is able to find the correct information about kernel size, stride, and pooling within 2048 random probes. However, the boundary effect appears agnostic to the number of channels, and therefore the HuffDuff probing attack on its own cannot determine k.

7 USING ARCHITECTURAL PROPERTIES

To reverse-engineer the final piece of the puzzle — the channel (k) information — we rely on observations about (a) the dense nature of partial sums, and (b) how partial sums are transferred between an accelerator chip and DRAM.

7.1 Dense PSUMs

We use psums to provide an additional architectural timing channel and deduce the value of k. We observe that psums — as opposed to output activations — are extremely unlikely to contain zeros and therefore are held dense during accumulation. The zeros in psum can only arise (i) if all weights in the kernel or all input activations within the kernel footprint are 0 (very unlikely), (ii) the Multiply and Accumulate (MAC) operations exactly cancel each other out (even more unlikely).

In addition, it is unsafe to clamp negative psums and create sparsity *before* the accumulation process ends, because these values might turn positive again before accumulation is complete. Therefore, the psum tile is held in the Global Buffer (GLB) in a dense fashion during this process[5]. Only after the psum calculations are completed, will they be sent to the post-processing unit. This unit clamps, quantizes, and compresses the final values into sparse output feature maps and sends them back to DRAM.

As the dense psums are stored on-chip during accumulation to exploit reuse, they do not need to be transferred back to DRAM. However, after the accumulation phase, the post-processing unit exploits sparsity, compressing psums before sending them to off-chip DRAM. This allows us to create a timing side channel.

7.2 The Timing Side-Channel

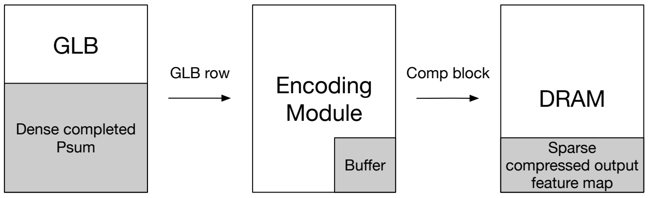


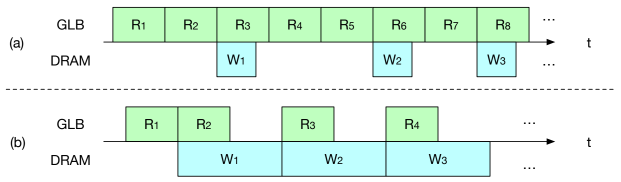
Figure 3: The flow of psums into DRAM, depicting the on-the-fly encoding for output activations.

Fig. 3 shows how dense psum values in GLB are compressed to a sparse output feature map. First, a GLB row that contains multiple psum words is sent to the encoding module[6] where negative values are clamped and the compressed content is stored in its local buffers. Once there is enough data in the buffer, a compressed sparse block will be written back to DRAM. This continues until all dense psum are processed.

[5]We exclude ReLU prediction type of accelerators [1, 83] that exploit output activation sparsity based on additional predictions. To the best of our knowledge, no sparse accelerators proposed to date employ this technique.

[6]More precisely, it is the post-processing module where other post-processing operations such as quantizing the accumulator to the actual activation width are performed. We omit this for simplicity.

The execution time of the encoding process can be bounded by either the GLB side the DRAM side, as shown below.



Panel(a) depicts a case where the encoding process is GLB-bound. R_i represents distinct GLB rows being read, and W_i stands for distinct DRAM transfers. Multiple packed GLB rows generate a single compressed block, and the DRAM has sufficient bandwidth to quickly transfer those compressed blocks. The total encoding time is proportional to the number of GLB rows that are read, and thus corresponds to dense psum size. We approximate the total time as the difference between the last DRAM transfer time and the first DRAM transfer time.

Panel (b) depicts a case where the encoding process is DRAM-bound. This could be because the GLB row is wide enough to construct a large compressed block and the DRAM has only limited bandwidth to write these blocks. In this case, the GLB row reads will stall as the buffer within the encoding module will quickly become full. Here, the total processing time is proportional to the number of DRAM transfers, which corresponds to the size of the sparse output feature map.

Fortunately, in practice, the encoding process tends to be GLB-bound (see Section 8). This is because the accumulators for psum typically have a larger bitwidth to avoid overflow. For example, Eyeriss v2 [10] and SCNN [72] use 20 bits and 24 bits respectively, whereas their activation width is only 8 bits. Additionally, since psums are dense, they naturally have more elements than output feature maps, which are sparse: overall, the size of dense psums tends to be 5×–6× larger than sparse output feature maps. Thus, we can obtain the psum size ratio between different layers, and, with known values of P, Q (from the prober), use this to reverse engineer the ratio of k between those layers.

8 EVALUATION

8.1 Methods

To obtain the sparse victim model, we used the Lottery Ticket Hypothesis [17], pruning VGG-S [82] by a factor of 10× and ResNet-18 [29] also by a factor of 10×, each trained on CIFAR-10 [50]. We instrumented the PyTorch [73] code for each model to generate the responses to the probing component of the HuffDuff attack.

We build a custom analytic simulator for the on-the-fly encoding process discussed in Section 7.2 based on runtime activation snapshot captured from Pytorch [73] models. We used the available psum GLB bandwidth from Eyeriss v2 [10], an state-of-the-art 2-sided sparse accelerator, with LPDDR3 [42], LPDDR4 [44], and LPDDR4X [43] memory.

To evaluate the effectiveness of our attack, we sample models from the solution space and evaluate our accuracy as well as the black box targeted adversarial attack success rate. We generate adversarial examples using BIM [52] method based on implementation from TorchAttacks [49].

8.2 Effectiveness of HuffDuff Components

Prober: Section 5 discusses how to reverse engineer filter size, stride, and pooling based on different probing responses. Although "errors" (i.e., unobservable edge effects) could occur on any downstream layers, the failure probability can be successfully decreased based on probability amplification via multiple independent random trials: we keep increasing the number of random trials until the identified geometry converges. We find that 2048 random trials are sufficient to correctly reveal all of the layer geometry such as filter size, stride, and pooling. In practice, we are able to obtain all layer geometry information except output channel counts in less than 10 minutes on an NVIDIA 2080Ti GPU.

Encoding the timing side-channel: Section 7 approximates dense psum ratios between different layers based on the ratio of the timing difference between the first and last DRAM transfers. Our simulation is based Eyeriss v2 [10], which has 8 psum GLB banks, where each bank is 3 words wide (20-bit accumulator, running at 200MHz. The evaluated DRAM includes both the single-channel and dual-channel versions of LPDDR3 [42], LPDDR4 [44], and LPDDR4X [43]. Our evaluation is based on profiled output activations from PyTorch [73]. We observe that the system is GLB-bound even with the lowest-bandwidth DRAM (i.e., single-channel LPDDR3).

To further determine the limits of leveraging the GLB-bound property, we also evaluated how much more GLB bandwidth is required for Eyeriss v2 [10] for it to begin experiencing some DRAM-bound layers for VGG-S and ResNet18; the results are shown below (*s* for single channel and *d* for dual channel).

LPDDR4	3-*s*	3-*d*	4-*s*	4-*d*	4X-*s*	4X-*d*
VGG-S	2×	4×	2.3×	4.6×	2.7×	5.3×
ResNet18	1.8×	3.5×	2×	4.1×	2.3×	4.7×

While the accelerator designer can increase the available GLB bandwidth by creating more banks, the bottleneck will only shift to the encoder as it is challenging to encode a large number of words within a single cycle.

Although we have validated that encoding is GLB-bound, the side channel information collected is only an approximation, because the time between the first GLB row read and the first DRAM transfer is unknown. We found this small inaccuracy to be acceptable in practice without the need for additional denoising.

Finalizing the solution space: Since the encoding timing channel only provides ratios between different output channel counts (k), we would like to identify the channel count range for at least one layer. We find the first layer is a good candidate as its weight is much denser compared to other layers. First, first-layer filters directly process the input images, and an aggressive pruning on the first layer weight compromises the accuracy more comparing pruning other weights [28]. Second, first-layer weights are typically tiny, and we find that pruning algorithms are more likely to prune aggressively on layers with large weights. Empirically, we find that first-layer sparsity is rarely beyond 60%, so we use this to establish empirical sparsity bound, which gives an output channel count range $[30, 73]$ and $[58, 123]$ for ResNet18 [29] and VGG-S [82] respectively. Combining the encoder timing channel info, we get the 44 and 66 solutions for ResNet18 [29] and VGG-S [82] respectively.

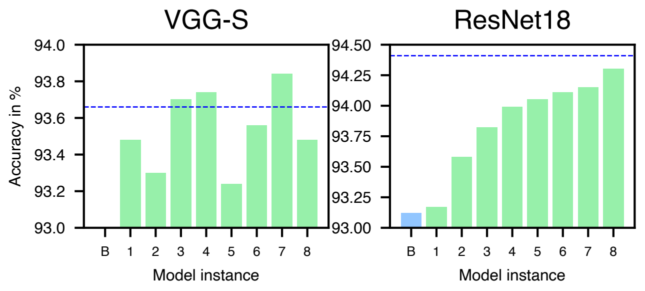


Figure 4: Accuracy for sampled instances on VGG-S (left) and ResNet-18 (right). Baseline accuracy is in the blue shaded bars whereas the accuracy of 8 HuffDuff sampled instances are in green dotted bars. The original victim accuracy is depicted in a blue dashed line. VGG baseline accuracy is 75.8% (not shown in the figure).

8.3 Quality of Reverse-Engineered Models

A good reverse-engineered model should have the following properties: (i) high classification accuracy under the same model footprint such that the attacker can use it, and (ii) usefulness in mounting follow-up attacks, such as generating adversarial examples that compromise the victim system. Therefore, we evaluated the quality of the reverse-engineered models using two metrics: accuracy and targeted adversarial attack success rate. We performed uniform sampling from the solution space, sampling 8 candidates each for VGG-S [82] and ResNet18 [29].

Accuracy: We compare the accuracy of 8 sampled candidates (model instance id sorted with respect to unpruned size in Fig. 4. Our baselines are constructed as selecting a model from a prior generation in the model zoo and pruned to the same footprint. We choose a model from a prior generation because it does not make sense for the attacker to steal a "worse" model if the goal is to match iso-footprint accuracy. Thus, to compare with candidates obtained from reverse-engineering VGG-S [82], we selected AlexNet [51] as the baseline, and we used VGG-S [82] as the baseline for ResNet18 [29] candidates.

Our candidates for the VGG-S victim significantly outperform the baseline (75.8%, not shown in Fig. 4), with some even exceeding the original victim. All candidates for ResNet18 exceed the baseline and the best-performing model is within 0.1% of the victim.

Adversarial success rate: We examine the black box targeted success rate among our sampled candidates and the baselines. We follow prior works [33, 59] where baselines are chosen from random surrogates in the model zoo. Unlike the prior dense case, we further prune them to different sparsity levels for a more thorough comparison. For our VGG-S victims (Fig. 5 left), we select ResNet18 [29] and MobileNetV2 [78], each pruned 2× and 5× (B1 to B4 in Fig. 5 left). For Fig. 5 left on ResNet18 victim, we include four baselines: VGG-S and MobileNetV2, also pruned to 2× and 5×.

In terms of the target selection heuristic, prior works [33, 59] pick a random label as the transfer target that might not be challenging enough if the randomly chosen target is similar to the original label. Since transferability is harder to achieve with a more challenging target, we choose the most challenging target selection heuristic:

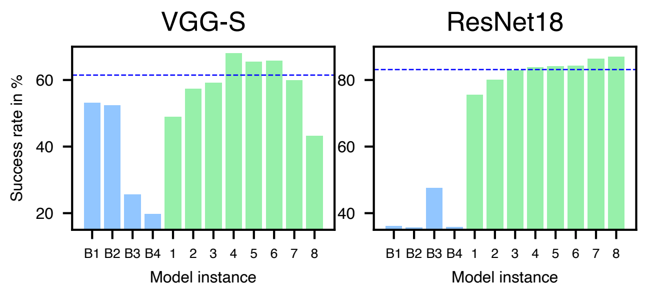


Figure 5: Black box targeted success rate (where max per pixel disturbance ε bounded by 32) for sampled instances on VGG-S (left) and ResNet-18 (right). Instance B1 to B4 on the left (shaded light blue bars) corresponds to 4 baselines ResNet18 pruned 2x, ResNet18 pruned 5×, MobileNetV2 pruned 2×, MobileNetV2 5× respectively. B1 to B4 on the right corresponds to ResNet18 pruned 2×, ResNet18 pruned 5×, MobileNetV2 pruned 2×, MobileNetV2 5× respectively. Instances 1 to 8 (dotted light green bars) correspond to 8 sampled instances using HuffDuff. The success rate for the one with identical architecture is in a blue dashed line.

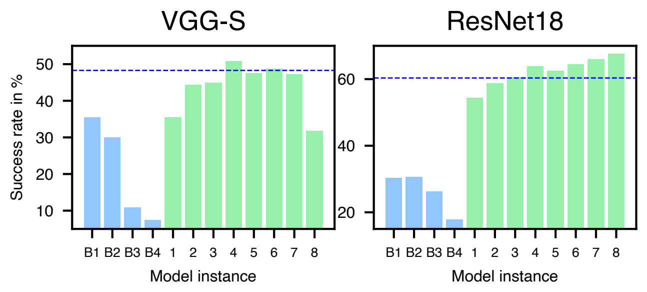


Figure 6: Success rate over different model instances. We use a similar setting to Fig. 5 except that we reduced the per pixel disturbance ε to 16.

the least likely label. In this target, perturbation is added to trick the model to predict the least likely label in the original prediction (i.e. tricking the model to perform the worst prediction).

Notice here that measuring transferability on the victim model using the victim itself is equivalent to the white-box attack, so instead, we compare with a model having the oracle structure with the victim but trained with a different random seed. We first demonstrate the targeted attack success rate with allowed per pixel disturbance (ε) to be 32 (follow to [59]) in Fig. 5. Our candidate models transfer significantly better than most baselines, and many even outperform an idealized setting where the model architecture is known to the attacker. Moreover, we evaluate the attack with $\varepsilon = 16$ where such disturbance is considered imperceptible [52] in Fig. 6 and observe a similar trend.

9 DISCUSSION AND FUTURE DIRECTIONS

9.1 HuffDuff Limitations

In this subsection, we summarize the scenarios where HuffDuff does not work well. Those scenarios are not the common case, and HuffDuff is sufficient for most cases.

In terms of the target accelerator, we exclude the SRAM-only ones like EIE [27] and ShiDianNao [15], as well as Cerebras WSE-2 [6] with its gigantic 40GB SRAM. These accelerators may not leave visible footprints in DRAM, but they are prohibitively expensive in terms of silicon area. We also exclude accelerators that perform layer-fusion [3, 18, 38]. This is because they do not leave the entire footprint visible in DRAM, and therefore HuffDuff or other DRAM-snooping-based attacks [33, 35] are not effective. Finally, we exclude accelerators that are not sparse [7, 8, 45].

The HuffDuff prober does not provide any insight on convolutions that do not exhibit the boundary effect, so it does not work on the transposed convolutions used in UNet [76] and GANs [21]. Nevertheless, the prober applies to all padding modes (“valid”, “same”, and “full” [16]) and strategies (“constant”, “reflect”, “replicate”, and “circular”, following PyTorch [73] terminology), as they do create boundary effects. However, the HuffDuff prober does not distinguish among them, and our implementation assumes the “same” padding mode and the zero padding as the most common case in TorchVision [61].

9.2 Potential Defence Strategies

Fully effective defences, like ORAM [19], SRAM-only accelerators [6, 15, 27], are prohibitively expensive in silicon area, especially for edge accelerators. In theory, fused-layer accelerators [3, 18, 38] would expand the search space, but no such sparse accelerators exist, and dense accelerators are easy to crack [35]. We leave these outside our threat model as unrealistic.

Hardware defences that might appear cheaper are also non-trivial. There are two widely adapted defence strategies that could apply: (i) blocking the source of the leak and (ii) obfuscating the detection of the leak. For example, the victim could leave “sensitive” pixels (i.e., positions that might reveal the boundary effect compared with other probes) uncompressed. This leverages the first approach that avoids the boundary effect from being observed in DRAM: for example, an *ABCC* pattern shown in Section 5.4 will appear to be *AAAA*, and no filter size information will be obtainable. However, such a scheme is non-trivial because the “sensitive” position is different for different attack patterns, and therefore would require additional *dynamic* hardware support. Following the second approach, the victim could randomly leave zeros uncompressed; in this case, an *ABCC* pattern shown in Section 5.4 might become *ABCD* to obfuscate the detection of filter size. However, this may still not provide enough security guarantees, as this kind of noise could be overcome with repeated trials. We believe a serious defence study would be a paper unto itself, and therefore we leave this to future work.

10 RELATED WORK

Model-stealing and corruption attacks: Prior work has investigated reverse engineering network architectures with IP protections. With the reverse-engineered model, one can stage other types of attacks. For instance, Tramer et al. [75] try to duplicate the ML model by exploiting the features of ML-as-a-Service. Rather than using features of the service model, HuffDuff exploits the features of the hardware to tailor specific inputs to reverse engineer the model.

Oh et al. [65] proposed a software-based attack that investigates techniques to infer parameters and characteristics from a black-box model. To enable this, they use multiple input queries to characterize the decision boundary of the victim model and then employ a "meta-model" to try to predict certain hyper-parameters. Unfortunately, these black-box approaches have limited efficacy as they do not have access to any information related to side channels or hardware architectures. On the other hand, as HuffDuff utilizes the information of the underlying architecture, it can be relatively more powerful.

Rather than stealing the model, prior work has also proposed degrading the model using fault injection attacks that flip a small number of bits to significantly degrade the accuracy of the model [75]. These attacks can be orchestrated on DRAM modules using the Rowhammer vulnerability and can cause integrity violations [22, 31, 55, 75]. Such attacks are orthogonal to HuffDuff.

Hardware-based attacks: Yan et al. [92] use the insight that the DNNs executed on CPUs heavily rely on blocked GEMM operations. They then use a cache side-channel attack to extract this information about GEMM. They can use this attack to infer the number of GEMM calls and the size of matrices that GEMM operates on. They show that this can be used to reveal the DNN architecture. However, this attack works very well with dense networks and unlike HuffDuff it is ineffective with sparse networks. LeakyDNN [89] exploits GPU context-switch side channels to steal DNNs. Other GPU side channels [62] can also be potentially exploited. Several prior works have also explored attacks using physical probing of the hardware [33, 35]. These works do not translate to sparse accelerators.

In a similar vein, Deepsniffer [33] tries to steal dense models on GPUs. They use the insight that one can use an end-to-end learning-based approach to handle a lot of system and architectural noises. While Deepsniffer has access to the entire software stack to create training data to extract useful side-channel information, HuffDuff does not require this. Furthermore, Deepsniffer is an expensive approach as they need to re-collect the training set and re-train the model when they use different GPUs and runtimes. In contrast, for HuffDuff the noise on side channel info (tensor sparsity) is also part of the secret, and we do not need to construct the training set.

11 CONCLUSIONS

There is an increasing trend of using sparse DNN models to run directly and locally on edge devices using custom sparse DNN accelerators. These devices are controlled by the users and can be disassembled and monitored to measure off-chip access volume.

In this paper, we show that this physical access is sufficient to enable the theft of the DNN models within. We demonstrate a novel attack scheme, called HuffDuff, which leverages (i) the boundary effect present in CONV layers, and (ii) the timing side channel created by on-the-fly activations compression.

Together, these techniques offer a practical method to dramatically reduce the space of possible model architectures by up to *94 orders-of-magnitude*, often yielding fewer than a hundred solutions. Our evaluation shows that candidate models sampled from the HuffDuff solution space reach the accuracy of the victim, and raise black-box targeted attack success rates to a semi-white-box level (where the attacker knows the correct architecture) while remaining black-box techniques.

ACKNOWLEDGMENTS

The authors are grateful to the anonymous reviewers and the shepherd for insightful feedback and helpful suggestions.

This material is based on research sponsored by Air Force Research Laboratory (AFRL) and Defense Advanced Research Project Agency (DARPA) under agreement number FA8650-20-2-7007, and by the Natural Sciences and Engineering Research Council of Canada (NSERC) under award number NETGP 485577-15. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of Air Force Research Laboratory (AFRL), Defense Advanced Research Project Agency (DARPA), the U.S. Government, the Natural Sciences and Engineering Research Council of Canada (NSERC), or the Government of Canada.

REFERENCES

[1] Vahideh Akhlaghi, Amir Yazdanbakhsh, Kambiz Samadi, Rajesh K. Gupta, and Hadi Esmaeilzadeh. 2018. SnaPEA: Predictive Early Activation for Reducing Computation in Deep Convolutional Neural Networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 662–673. https://doi.org/10.1109/ISCA.2018.00061

[2] Jorge Albericio, Patrick Judd, Tayler Hetherington, Tor Aamodt, Natalie Enright Jerger, and Andreas Moshovos. 2016. Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 1–13. https://doi.org/10.1109/ISCA.2016.11

[3] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. 2016. Fused-layer CNN accelerators. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783725

[4] Lejla Batina, Shivam Bhasin, Dirmanto Jap, and Stjepan Picek. 2019. CSI NN: Reverse Engineering of Neural Network Architectures through Electromagnetic Side Channel. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 515–532.

[5] Jakub Breier, Xiaolu Hou, Dirmanto Jap, Lei Ma, Shivam Bhasin, and Yang Liu. 2018. Practical Fault Attack on Deep Neural Networks. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2204–2206. https://doi.org/10.1145/3243734.3278519

[6] Cerebras. 2021. Wafer-Scale Engine: The Largest Chip Ever Built. https://f.hubspotusercontent30.net/hubfs/8968533/WSE-2%20Datasheet.pdf

[7] Tianshi Chen, Zidong Du, Ninghui Sun, Jia Wang, Chengyong Wu, Yunji Chen, and Olivier Temam. 2014. DianNao: a small-footprint high-throughput accelerator for ubiquitous machine-learning. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2014, Salt Lake City, UT, USA, March 1-5, 2014*, Rajeev Balasubramonian, Al Davis, and Sarita V. Adve (Eds.). ACM, 269–284. https://doi.org/10.1145/2541940.2541967

[8] Yunji Chen, Tao Luo, Shaoli Liu, Shijin Zhang, Liqiang He, Jia Wang, Ling Li, Tianshi Chen, Zhiwei Xu, Ninghui Sun, and Olivier Temam. 2014. DaDianNao: A Machine-Learning Supercomputer. In *2014 47th Annual IEEE/ACM International*

Symposium on Microarchitecture. 609–622. https://doi.org/10.1109/MICRO.2014.58

[9] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. 2016. Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 367–379. https://doi.org/10.1109/ISCA.2016.40

[10] Yu-Hsin Chen, Tien-Ju Yang, Joel Emer, and Vivienne Sze. 2019. Eyeriss v2: A Flexible Accelerator for Emerging Deep Neural Networks on Mobile Devices. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 9, 2 (2019), 292–308. https://doi.org/10.1109/JETCAS.2019.2910232

[11] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. *IACR Cryptol. ePrint Arch.* 2016, 86 (2016), 1–118.

[12] Tom Woller David Kaplan, Jeremy Powell. 2016. AMD Memory Encryption – A White Paper. https://developer.amd.com/wordpress/media/2013/12/AMD_Memory_Encryption_Whitepaper_v7-Public.pdf. [Online; accessed 7-July-2022].

[13] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*. IEEE Computer Society, 248–255. https://doi.org/10.1109/CVPR.2009.5206848

[14] Caiwen Ding, Siyu Liao, Yanzhi Wang, Zhe Li, Ning Liu, Youwei Zhuo, Chao Wang, Xuehai Qian, Yu Bai, Geng Yuan, Xiaolong Ma, Yipeng Zhang, Jian Tang, Qinru Qiu, Xue Lin, and Bo Yuan. 2017. CirCNN: Accelerating and Compressing Deep Neural Networks Using Block-Circulant Weight Matrices. In *2017 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 395–408.

[15] Zidong Du, Robert Fasthuber, Tianshi Chen, Paolo Ienne, Ling Li, Tao Luo, Xiaobing Feng, Yunji Chen, and Olivier Temam. 2015. ShiDianNao: Shifting vision processing closer to the sensor. In *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*. 92–104. https://doi.org/10.1145/2749469.2750389

[16] Vincent Dumoulin and Francesco Visin. 2016. A guide to convolution arithmetic for deep learning. *CoRR* abs/1603.07285 (2016). arXiv:1603.07285 http://arxiv.org/abs/1603.07285

[17] Jonathan Frankle and Michael Carbin. 2019. The Lottery Ticket Hypothesis: Finding Sparse, Trainable Neural Networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=rJl-b3RcF7

[18] Mingyu Gao, Xuan Yang, Jing Pu, Mark Horowitz, and Christos Kozyrakis. 2019. TANGRAM: Optimized Coarse-Grained Dataflow for Scalable NN Accelerators. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019, Providence, RI, USA, April 13-17, 2019*, Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck (Eds.). ACM, 807–820. https://doi.org/10.1145/3297858.3304014

[19] Oded Goldreich and Rafail Ostrovsky. 1996. Software Protection and Simulation on Oblivious RAMs. *J. ACM* 43, 3 (1996), 431–473. https://doi.org/10.1145/233551.233553

[20] Ashish Gondimalla, Noah Chesnut, Mithuna Thottethodi, and T. N. Vijaykumar. 2019. SparTen: A Sparse Tensor Accelerator for Convolutional Neural Networks. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO '52)*. Association for Computing Machinery, New York, NY, USA, 151–165. https://doi.org/10.1145/3352460.3358291

[21] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron C. Courville, and Yoshua Bengio. 2014. Generative Adversarial Nets. In *Advances in Neural Information Processing Systems 27: Annual Conference on Neural Information Processing Systems 2014, December 8-13 2014, Montreal, Quebec, Canada*, Zoubin Ghahramani, Max Welling, Corinna Cortes, Neil D. Lawrence, and Kilian Q. Weinberger (Eds.). 2672–2680. https://proceedings.neurips.cc/paper/2014/hash/5ca3e9b122f61f8f06494c97b1afccf3-Abstract.html

[22] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2015. Explaining and Harnessing Adversarial Examples. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1412.6572

[23] D Griffith and Carl Amrhein. 1983. An Evaluation of Correction Techniques for Boundary Effects in Spatial Statistical Analysis: Traditional Methods. *Geographical Analysis* 15, 4 (1983), 352.

[24] Daniel A Griffith. 1983. The Boundary Value Problem in Spatial Statistical Analysis. *Journal of Regional Science* 23, 3 (1983), 377–387.

[25] Le Guan, Chen Cao, Sencun Zhu, Jingqiang Lin, Peng Liu, Yubin Xia, and Bo Luo. 2019. Protecting Mobile Devices from Physical Memory Attacks with Targeted Encryption. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks* (Miami, Florida) *(WiSec '19)*. Association for Computing Machinery, New York, NY, USA, 34–44. https://doi.org/10.1145/3317549.3319721

[26] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. 2009. Lest We Remember: Cold-Boot Attacks on Encryption Keys. *Commun. ACM* 52, 5 (may 2009), 91–98. https://doi.org/10.1145/1506409.1506429

[27] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A. Horowitz, and William J. Dally. 2016. EIE: Efficient Inference Engine on Compressed Deep Neural Network. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. 243–254. https://doi.org/10.1109/ISCA.2016.30

[28] Song Han, Huizi Mao, and William J. Dally. 2016. Deep Compression: Compressing Deep Neural Network with Pruning, Trained Quantization and Huffman Coding. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1510.00149

[29] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 770–778. https://doi.org/10.1109/CVPR.2016.90

[30] Kartik Hegde, Hadi Asghari Moghaddam, Michael Pellauer, Neal Clayton Crago, Aamer Jaleel, Edgar Solomonik, Joel S. Emer, and Christopher W. Fletcher. 2019. ExTensor: An Accelerator for Sparse Tensor Algebra. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 319–333. https://doi.org/10.1145/3352460.3358275

[31] Sanghyun Hong, Pietro Frigo, Yiğitcan Kaya, Cristiano Giuffrida, and Tudor Dumitraş. 2019. Terminal Brain Damage: Exposing the Graceless Degradation in Deep Neural Networks under Hardware Fault Attacks. In *Proceedings of the 28th USENIX Conference on Security Symposium* (Santa Clara, CA, USA) *(SEC'19)*. USENIX Association, USA, 497–514.

[32] Jie Hu, Li Shen, and Gang Sun. 2018. Squeeze-and-Excitation Networks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*. Computer Vision Foundation / IEEE Computer Society, 7132–7141. https://doi.org/10.1109/CVPR.2018.00745

[33] Xing Hu, Ling Liang, Shuangchen Li, Lei Deng, Pengfei Zuo, Yu Ji, Xinfeng Xie, Yufei Ding, Chang Liu, Timothy Sherwood, and Yuan Xie. 2020. DeepSniffer: A DNN Model Extraction Framework Based on Learning Architectural Hints. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 385–399. https://doi.org/10.1145/3373376.3378460

[34] Weizhe Hua, Muhammad Umar, Zhiru Zhang, and G. Edward Suh. 2022. GuardNN: Secure Accelerator Architecture for Privacy-Preserving Deep Learning. In *Proceedings of the 59th ACM/IEEE Design Automation Conference* (San Francisco, California) *(DAC '22)*. Association for Computing Machinery, New York, NY, USA, 349–354. https://doi.org/10.1145/3489517.3530439

[35] Weizhe Hua, Zhiru Zhang, and G. Edward Suh. 2018. Reverse Engineering Convolutional Neural Networks Through Side-channel Information Leaks. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. 1–6. https://doi.org/10.1109/DAC.2018.8465773

[36] Weizhe Hua, Yuan Zhou, Christopher De Sa, Zhiru Zhang, and G. Edward Suh. 2019. Boosting the Performance of CNN Accelerators with Dynamic Fine-Grained Channel Gating. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 139–150. https://doi.org/10.1145/3352460.3358283

[37] Andrew Huang. 2002. Hacking the Xbox: An Introduction to Reverse Engineering. (2002).

[38] Chao-Tsung Huang, Yu-Chun Ding, Huan-Ching Wang, Chi-Wen Weng, Kai-Ping Lin, Li-Wei Wang, and Li-De Chen. 2019. eCNN: A Block-Based and Highly-Parallel CNN Accelerator for Edge Inference. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2019, Columbus, OH, USA, October 12-16, 2019*. ACM, 182–195. https://doi.org/10.1145/3352460.3358263

[39] Yongbing Huang, Licheng Chen, Zehan Cui, Yuan Ruan, Yungang Bao, Mingyu Chen, and Ninghui Sun. 2014. HMTT: A Hybrid Hardware/Software Tracing System for Bridging the DRAM Access Trace's Semantic Gap. *ACM Trans. Archit. Code Optim.* 11, 1, Article 7 (feb 2014), 25 pages. https://doi.org/10.1145/2579668

[40] Mihailo Isakov, Vijay Gadepally, Karen M. Gettings, and Michel A. Kinsy. 2019. Survey of Attacks and Defenses on Edge-Deployed Neural Networks. In *2019 IEEE High Performance Extreme Computing Conference (HPEC)*. 1–8. https://doi.org/10.1109/HPEC.2019.8916519

[41] Bernd Jahne. 2004. *Practical Handbook on Image Processing for Scientific and Technical Applications*. CRC Press.

[42] JEDEC Standard. 2015. Lpw Power Double Data Rate 3 SDRAM (LPDDR3). In *JESD209-3C*.

[43] JEDEC Standard. 2021. Addendum No. 1 to JESD209-4, Low Power Double Data Rate 4X (LPDDR4X). In *JESD209-4-1A*.

[44] JEDEC Standard. 2021. Low Power Double Data Rate 4 (LPDDR4). In *JESD209-4D*.

[45] Norman P. Jouppi, Cliff Young, Nishant Patil, David A. Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu,

Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture, ISCA 2017, Toronto, ON, Canada, June 24-28, 2017*. ACM, 1–12. https://doi.org/10.1145/3079856.3080246

[46] Patrick Judd, Alberto Delmas Lascorz, Sayeh Sharify, and Andreas Moshovos. 2017. Cnvlutin2: Ineffectual-Activation-and-Weight-Free Deep Neural Network Computing. *CoRR* abs/1705.00125 (2017). arXiv:1705.00125 http://arxiv.org/abs/1705.00125

[47] Changhyeon Kim, Sanghoon Kang, Dongjoo Shin, Sungpill Choi, Youngwoo Kim, and Hoi-Jun Yoo. 2019. A 2.1TFLOPS/W Mobile Deep RL Accelerator with Transposable PE Array and Experience Compression. In *2019 IEEE International Solid- State Circuits Conference - (ISSCC)*. 136–138. https://doi.org/10.1109/ISSCC.2019.8662447

[48] Dongyoung Kim, Junwhan Ahn, and Sungjoo Yoo. 2018. ZeNA: Zero-Aware Neural Network Accelerator. *IEEE Design & Test* 35, 1 (2018), 39–46. https://doi.org/10.1109/MDAT.2017.2741463

[49] Hoki Kim. 2020. Torchattacks : A Pytorch Repository for Adversarial Attacks. *CoRR* abs/2010.01950 (2020). arXiv:2010.01950 https://arxiv.org/abs/2010.01950

[50] Alex Krizhevsky. 2009. *Learning Multiple Layers of Features from Tiny Images*. Master's thesis. University of Toronto.

[51] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*, Peter L. Bartlett, Fernando C. N. Pereira, Christopher J. C. Burges, Léon Bottou, and Kilian Q. Weinberger (Eds.). 1106–1114. https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html

[52] Alexey Kurakin, Ian J. Goodfellow, and Samy Bengio. 2017. Adversarial examples in the physical world. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=HJGU3Rodl

[53] Ching-En Lee, Yakun Sophia Shao, Jie-Fang Zhang, Angshuman Parashar, Joel Emer, Stephen W Keckler, and Zhengya Zhang. 2018. Stitch-X: An Accelerator Architecture for Exploiting Unstructured Sparsity in Deep Neural Networks. In *SysML Conference*, Vol. 120.

[54] Sunho Lee, Jungwoo Kim, Seonjin Na, Jongse Park, and Jaehyuk Huh. 2022. TNPU: Supporting Trusted Execution with Tree-less Integrity Protection for Neural Processing Unit. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. 229–243. https://doi.org/10.1109/HPCA53966.2022.00025

[55] Guanpeng Li, Siva Kumar Sastry Hari, Michael Sullivan, Timothy Tsai, Karthik Pattabiraman, Joel Emer, and Stephen W. Keckler. 2017. Understanding Error Propagation in Deep Learning Neural Network (DNN) Accelerators and Applications. In *SC17: International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–12.

[56] Jiajun Li, Shuhao Jiang, Shijun Gong, Jingya Wu, Junchao Yan, Guihai Yan, and Xiaowei Li. 2019. SqueezeFlow: A Sparse CNN Accelerator Exploiting Concise Convolution Rules. *IEEE Trans. Comput.* 68, 11 (2019), 1663–1677. https://doi.org/10.1109/TC.2019.2924215

[57] Tailin Liang, Lei Wang, Shaobo Shi, and John Glossner. 2018. Dynamic runtime feature map pruning. *arXiv preprint arXiv:1812.09922* (2018).

[58] Sean Lie. 2022. Cerebras Architecture Deep Dive: First Look Inside the HW/SW Co-Design for Deep Learning : Cerebras Systems. In *2022 IEEE Hot Chips 34 Symposium (HCS)*. 1–34. https://doi.org/10.1109/HCS55958.2022.9895479

[59] Yanpei Liu, Xinyun Chen, Chang Liu, and Dawn Song. 2017. Delving into Transferable Adversarial Examples and Black-box Attacks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=Sys6GJqxl

[60] Johnson Loh, Jianan Wen, and Tobias Gemmeke. 2020. Low-Cost DNN Hardware Accelerator for Wearable, High-Quality Cardiac Arrythmia Detection. In *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. 213–216. https://doi.org/10.1109/ASAP49362.2020.00042

[61] Sébastien Marcel and Yann Rodriguez. 2010. Torchvision the Machine-Vision Package of Torch. In *Proceedings of the 18th ACM International Conference on Multimedia* (Firenze, Italy) *(MM '10)*. Association for Computing Machinery, New York, NY, USA, 1485–1488. https://doi.org/10.1145/1873951.1874254

[62] Hoda Naghibijouybari, Ajaya Neupane, Zhiyun Qian, and Nael Abu-Ghazaleh. 2018. Rendered Insecure: GPU Side Channel Attacks Are Practical. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 2139–2153. https://doi.org/10.1145/3243734.3243831

[63] NVIDIA. 2017. NVIDIA Deep Learning Accelerator (NVDLA). http://nvdla.org/

[64] NVIDIA. 2020. NVIDIA A100 Tensor Core GPU Architecture. https://images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf.

[65] Seong Joon Oh, Max Augustin, Mario Fritz, and Bernt Schiele. 2018. Towards Reverse-Engineering Black-Box Neural Networks. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net. https://openreview.net/forum?id=BydjJte0-

[66] MohammadHossein Olyaiy, Christopher Ng, and Mieszko Lis. 2021. Accelerating DNNs inference with predictive layer fusion. In *ICS '21: 2021 International Conference on Supercomputing, Virtual Event, USA, June 14-17, 2021*, Huiyang Zhou, Jose Moreira, Frank Mueller, and Yoav Etsion (Eds.). ACM, 291–303. https://doi.org/10.1145/3447818.3460378

[67] OpenAI. 2019. Better Language Models and Their Implications. https://openai.com/blog/better-language-models/.

[68] OpenAI. 2019. GPT-2: 1.5B Release. https://openai.com/blog/gpt-2-1-5b-release/.

[69] OpenAI. 2019. GPT-2: 6-Month Follow-Up. https://openai.com/blog/gpt-2-6-month-follow-up/.

[70] OpenAI. 2020. OpenAI API. https://openai.com/blog/openai-api/.

[71] Nicolas Papernot, Patrick McDaniel, Ian Goodfellow, Somesh Jha, Z. Berkay Celik, and Ananthram Swami. 2017. Practical Black-Box Attacks against Machine Learning. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (Abu Dhabi, United Arab Emirates) *(ASIA CCS '17)*. Association for Computing Machinery, New York, NY, USA, 506–519. https://doi.org/10.1145/3052973.3053009

[72] Angshuman Parashar, Minsoo Rhu, Anurag Mukkara, Antonio Puglielli, Rangharajan Venkatesan, Brucek Khailany, Joel Emer, Stephen W. Keckler, and William J. Dally. 2017. SCNN: An accelerator for compressed-sparse convolutional neural networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*. 27–40. https://doi.org/10.1145/3079856.3080254

[73] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, Edward Z. Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035. https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html

[74] Ray Prasad. 2013. *Surface Mount Technology: Principles and Practice*. Springer Science & Business Media.

[75] Adnan Siraj Rakin, Zhezhi He, and Deliang Fan. 2019. Bit-Flip Attack: Crushing Neural Network With Progressive Bit Search. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. 1211–1220. https://doi.org/10.1109/ICCV.2019.00130

[76] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-Net: Convolutional Networks for Biomedical Image Segmentation. In *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, Nassir Navab, Joachim Hornegger, William M. Wells, and Alejandro F. Frangi (Eds.). Springer International Publishing, Cham, 234–241.

[77] B. K. Rosen, M. N. Wegman, and F. K. Zadeck. 1988. Global Value Numbers and Redundant Computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '88)*. Association for Computing Machinery, New York, NY, USA, 12–27. https://doi.org/10.1145/73560.73562

[78] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2018. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 4510–4520. https://doi.org/10.1109/CVPR.2018.00474

[79] Osman Semih Kayhan and Jan C. van Gemert. 2020. On Translation Invariance in CNNs: Convolutional Layers Can Exploit Absolute Spatial Location. In *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 14262–14273. https://doi.org/10.1109/CVPR42600.2020.01428

[80] Reza Shokri, Marco Stronati, Congzheng Song, and Vitaly Shmatikov. 2017. Membership Inference Attacks Against Machine Learning Models. In *2017 IEEE Symposium on Security and Privacy (SP)*. 3–18. https://doi.org/10.1109/SP.2017.41

[81] Nivedita Shrivastava and Smruti R. Sarangi. 2022. Seculator: A Fast and Secure Neural Processing Unit. *CoRR* abs/2204.08951 (2022). https://doi.org/10.48550/arXiv.2204.08951 arXiv:2204.08951

[82] Karen Simonyan and Andrew Zisserman. 2015. 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*, Yoshua Bengio and Yann LeCun (Eds.). http://arxiv.org/abs/1409.1556

[83] Mingcong Song, Jiechen Zhao, Yang Hu, Jiaqi Zhang, and Tao Li. 2018. Prediction Based Execution on Deep Neural Networks. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. 752–763. https://doi.org/10.1109/ISCA.2018.00068

[84] Rastislav Struharik, Bogdan Vukobratović, Andrea Erdeljan, and Damjan Rakanović. 2018. CoNNA – Compressed CNN Hardware Accelerator. In *2018 21st Euromicro Conference on Digital System Design (DSD)*. 365–372. https://doi.org/10.1109/DSD.2018.00070

[85] Hsin-Hsuan Sung, Yuanchao Xu, Jiexiong Guan, Wei Niu, Bin Ren, Yanzhi Wang, Shaoshan Liu, and Xipeng Shen. 2022. Brief Industry Paper: Enabling Level-4 Autonomous Driving on a Single $1k Off-the-Shelf Card. In *28th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2022, Milano, Italy, May 4-6, 2022*. IEEE, 297–300. https://doi.org/10.1109/RTAS54340.2022.00032

[86] Keysight Technologies. 2014. W2637A, W2638A and W2639A LPDDR BGA Probes for Logic Analyzers and Oscilloscopes. Retrieved November 3, 2022 from https://www.keysight.com/us/en/assets/7018-02123/data-sheets/5990-3892.pdf

[87] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. 2016. Stealing Machine Learning Models via Prediction APIs. In *Proceedings of the 25th USENIX Conference on Security Symposium* (Austin, TX, USA) *(SEC'16)*. USENIX Association, USA, 601–618.

[88] Assia Tria and Hamid Choukri. 2011. Invasive Attacks. In *Encyclopedia of Cryptography and Security, 2nd Ed*, Henk C. A. van Tilborg and Sushil Jajodia (Eds.). Springer, 623–629. https://doi.org/10.1007/978-1-4419-5906-5_511

[89] Junyi Wei, Yicheng Zhang, Zhe Zhou, Zhou Li, and Mohammad Abdullah Al Faruque. 2020. Leaky DNN: Stealing Deep-Learning Model Secret with GPU Context-Switching Side-Channel. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 125–137. https://doi.org/10.1109/DSN48063.2020.00031

[90] Yoo-Seung Won, Soham Chatterjee, Dirmanto Jap, Arindam Basu, and Shivam Bhasin. 2021. DeepFreeze: Cold Boot Attacks and High Fidelity Model Recovery on Commercial EdgeML Device. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. 1–9. https://doi.org/10.1109/ICCAD51958.2021.9643512

[91] Yun Xiang, Zhuangzhi Chen, Zuohui Chen, Zebin Fang, Haiyang Hao, Jinyin Chen, Yi Liu, Zhefu Wu, Qi Xuan, and Xiaoniu Yang. 2020. Open DNN Box by Power Side-Channel Attack. *IEEE Transactions on Circuits and Systems II: Express Briefs* 67, 11 (2020), 2717–2721. https://doi.org/10.1109/TCSII.2020.2973007

[92] Mengjia Yan, Christopher W. Fletcher, and Josep Torrellas. 2020. Cache Telepathy: Leveraging Shared Resource Attacks to Learn DNN Architectures. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 113, 18 pages.

[93] Zhe Yuan, Jinshan Yue, Huanrui Yang, Zhibo Wang, Jinyang Li, Yixiong Yang, Qingwei Guo, Xueqing Li, Meng-Fan Chang, Huazhong Yang, and Yongpan Liu. 2018. Sticker: A 0.41-62.1 TOPS/W 8Bit Neural Network Processor with Multi-Sparsity Compatible Convolution Arrays and Online Tuning Acceleration for Fully Connected Layers. In *2018 IEEE Symposium on VLSI Circuits*. 33–34. https://doi.org/10.1109/VLSIC.2018.8502404

[94] ZeroPoint. 2022. ZeroPoint Technologies Signs Memory Encryption Contract. https://www.zeropoint-tech.com/news/zeropoint-technologies-signs-memory-encryption-contract.

[95] Jie-Fang Zhang, Ching-En Lee, Chester Liu, Yakun Sophia Shao, Stephen W. Keckler, and Zhengya Zhang. 2019. SNAP: A 1.67 — 21.55TOPS/W Sparse Neural Acceleration Processor for Unstructured Sparse Deep Neural Network Inference in 16nm CMOS. In *2019 Symposium on VLSI Circuits*. C306–C307. https://doi.org/10.23919/VLSIC.2019.8778193

[96] Shijin Zhang, Zidong Du, Lei Zhang, Huiying Lan, Shaoli Liu, Ling Li, Qi Guo, Tianshi Chen, and Yunji Chen. 2016. Cambricon-X: An accelerator for sparse neural networks. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783723

[97] Pengfei Zuo, Yu Hua, Ling Liang, Xinfeng Xie, Xing Hu, and Yuan Xie. 2021. SEALing Neural Network Models in Encrypted Deep Learning Accelerators. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 1255–1260. https://doi.org/10.1109/DAC18074.2021.9586199

Received 2022-07-07; accepted 2022-09-22