DINAR: Enabling Distribution Agnostic Noise Injection in Machine Learning Hardware

Karthik Ganesan
karthik.ganesan@mail.utoronto.ca
University of Toronto
Toronto, ON, Canada

Viktor Karyofyllis
viktor.karyofyllis@mail.utoronto.ca
University of Toronto
Toronto, ON, Canada

Julianne Attai*
julianne.attai@mail.utoronto.ca
University of Toronto
Toronto, ON, Canada

Ahmed Hamoda*
a.hamoda@mail.utoronto.ca
University of Toronto
Toronto, ON, Canada

Natalie Enright Jerger
enright@ece.utoronto.ca
University of Toronto
Toronto, ON, Canada

Abstract

Machine learning (ML) has seen a major rise in popularity on edge devices in recent years, ranging from IoT devices to self-driving cars. Security in a critical consideration on these platforms. State-of-the-art security-centric ML algorithms (e.g., differentially private ML, adversarial robustness) require noise sampled from Laplace or Gaussian distributions. Edge accelerators lack CPUs [15, 25, 36, 50] to add such noise. Existing hardware approaches to generate noise on-the-fly incur high overheads and leak side-channel information that can undermine security [34, 47]. To remedy this, we propose DINAR,[1] lightweight hardware that enables noise addition from arbitrary distributions. For differentially private ML, DINAR enables noise addition while incurring 23× lower area and 40× lower energy compared to producing noise directly on-chip.

CCS Concepts

• **Computer systems organization** → *Embedded hardware*; **Neural networks**; • **Security and privacy** → **Embedded systems security**.

Keywords

Machine learning security, Neural network accelerators, adversarial attacks, differential privacy

ACM Reference Format:
Karthik Ganesan, Viktor Karyofyllis, Julianne Attai, Ahmed Hamoda, and Natalie Enright Jerger. 2023. DINAR: Enabling Distribution Agnostic Noise Injection in Machine Learning Hardware. In *Hardware and Architectural Support for Security and Privacy 2023 (HASP '23), October 29, 2023, Toronto, Canada.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3623652.3623665

1 Introduction

Given their tremendous advances in recent years, machine learning (ML) applications now run on devices from simple edge platforms to large, cloud-based accelerators [11]. However, due to tight energy, area and cost constraints, edge ML accelerators typically lack a CPU [15, 25, 36, 50]. As a result, the accelerator is responsible not only for performance but also for security. As prior work shows, security is a first class concern on edge devices [4].

In this paper, we identify a critical gap in current ML accelerators that prevents them from running security-centric ML algorithms. Namely, these algorithms require *random values*, sampled from specific distributions (e.g., Gaussian or Laplace). Examples of such algorithms include: differentially private ML (DP-ML), adversarial robustness and secure split inference.

We focus on DP-ML due to its increasing popularity for privacy-preserving ML in academia [10] and industry [5]. We explore adversarial robustness in Sec. 6 and secure split inference in Sec. 7. Differential Privacy (DP) is a widely used statistical technique to operate on large collections of private data, while protecting the privacy of each individual [3]. DP-ML extends the privacy-preserving guarantees of DP to ML models. Prior work shows that models trained on private user data can 'memorize' specific examples and leak sensitive information [22]. To ensure the integrity of ML models trained on private data, DP-ML adds Gaussian or Laplace random values to the activations of each layer during training.

As they lack CPUs, current edge ML accelerators cannot generate random values in hardware. One option is to include dedicated hardware for producing random values; however, existing approaches to do so suffer from a number of drawbacks: 1) They impose high overheads (Sec. 3.1) which are untenable in resource constrained edge devices. 2) Different distributions (e.g., Gaussian or Laplace) require different hardware, which further increases overheads. 3) Generating random values directly in hardware leaks side-channel information [47], making them susceptible to attacks. 4) Successful timing side-channel attacks have been shown against methods that produce both Gaussian- and Laplace-sampled values [34].

We observe that for the algorithms listed above, each model samples from a specific distribution with a fixed variance. Leveraging this, we propose DINAR– lightweight hardware modifications – to pre-compute and store noise[2] with this variance ahead of time.

[*] Both authors contributed equally to this research.

[1] DINAR: **D**istribution **I**ndependent **N**oise **A**ddition for **R**obustness

[2] We use the terms 'noise' and 'random values' interchangeably throughout our paper.

HASP '23, October 29, 2023, Toronto, Canada

ACM ISBN 979-8-4007-1623-2/23/10…$15.00
https://doi.org/10.1145/3623652.3623665

During runtime, we *randomly sample* from these stored points to produce the noise we need. We show that by storing noise points in plentiful off-chip memory, DINAR adds significantly less overhead compared to dedicated on-chip hardware for noise generation. Furthermore, by not generating the points on-chip, DINAR *is not vulnerable to the side-channel attacks* that plague dedicated noise generation hardware. Thus, DINAR allows ML accelerators to support a wide range of ML algorithms that require noise sampled from a variety of distributions.

In summary, we make the following contributions:

- We identify that current ML accelerators are unable to generate noise from specific distributions and therefore cannot run many security-critical ML algorithms.
- We explain why current hardware-based noise generation approaches suffer from high overheads and are insecure.
- To efficiently and safely produce random values, we propose DINAR, lightweight hardware modifications to ML accelerators to enable noise generation. DINAR pre-computes and stores random values in off-chip memory and randomly samples from these stored values during runtime.
- DINAR enables crucial algorithms such as differentially private ML with $< 0.5\%$ area and energy overheads. Compared to our baseline, DINAR has 23× lower area and 40× lower energy, while avoiding security issues of dedicated noise-generation hardware.

2 Background

Here, we describe the DP-ML algorithm we use to demonstrate the efficacy of DINAR. We then provide an overview of the architecture of typical accelerators, to understand how we incorporate DINAR into these accelerators.

2.1 Differentially Private ML

Differential privacy (DP) leverages the idea that a user's privacy is guaranteed if their data is not in the dataset at all. DP gives each user the same privacy that they would get from having their data removed from the dataset. Dwork et al. [18] first showed that adding random noise to each user's data can mimic the effect of removing that user's data from the set. DP is used by companies such as Facebook [48] and even the US Census Bureau [2].

DP adds noise sampled from either a Laplace or Gaussian distribution [24]. The Laplace mechanism is preferred for smaller datasets, while Gaussian is preferred for larger datasets [3]. To keep DINAR as flexible as possible, we support both these mechanisms. The amount of noise added controls the trade-off between privacy and accuracy [40]. Thus, the added noise is tuned per model, depending on the specific use case [3].

Training Differentially-Private Models. The goal of DP-ML is to protect private user data, used during model training. DP-ML adds noise to the activations of each layer during training [1]. DP-ML uses a modified Stochastic Gradient Descent (i.e., DP-SGD) algorithm, where noise is added to the gradients of each input. After noise addition, the gradients are accumulated to perform the weight update for that layer, similar to regular SGD [1]. This crucial noise addition step extends the certifiable security guarantees of DP to ML models. We now detail current approaches for producing such noise directly in hardware.

2.2 Hardware for noise generation

We now describe the prior approaches to producing noise directly in hardware, that we use as our baseline for comparing against DINAR. First, we describe our notation for distributions used throughout our paper. Gaussian and Laplace distributions are typically characterized by their mean (μ) and scale (σ). In our work, we always consider distributions with a mean of 0 as none of the applications we study uses a non-zero mean. Therefore, we denote the Gaussian and Laplace distributions as G^{σ} and L^{σ}, respectively. While the Gaussian distribution is more commonly characterized using it's variance (σ^2), for consistency with the Laplace distribution, we use the scale for both.

We require a hardware random number generator (RNG), which produces a *uniform distribution* ($U \in [-2^{N-1}, 2^{N-1} - 1]$), where N is the number of random bits generated [9]. We can convert from a uniform to a different distribution by applying the appropriate 'transform'. Fig. 1a shows this, along with the equations used for these conversions.

For the Gaussian distribution, we choose the design proposed by Lee et al. [39]. This design implements the Box-Muller transform[13], considered the best approach that balances accuracy and hardware usage [44]. The Box-Muller transform converts two uniformly distributed random numbers U_0 and U_1 to two Gaussian random numbers G_0^1 and G_1^1. For the Laplace distribution, we use the formula from Choi et al. [16], which converts U_0 and U_1 to a single Laplace random number L_0^1.

Hardware implementation. Fig. 1b shows the block diagram of our implementation of the design of Lee et al. [39]. Blocks in blue calculate transcendental functions, while blocks in orange (with rounded corners), compute basic math operations (e.g., multiplication, subtraction). All transcendental functions (i.e., $ln(U_0)$, $sqrt$, $sin()$ and $cos()$) are implemented using lookup tables, where the coefficients are determined using Chebyshev series approximations [54]. To reduce the table size, this design employs range reduction to first transform the input to each table into a small range of values [54].

Generating Laplace noise. The hardware proposed by Lee et al. only produces Gaussian random values. However, as we explained in Sec. 2.1, DP-ML sometimes requires Laplace random values. To enable this, we make minor modifications to the hardware shown in Fig. 1b to also produce Laplace random values, based on the formula from Choi et al. [16]. Since many hardware blocks are shared between the equations shown in Fig. 1a (i.e., $ln(U_0)$ and a multiplier), we add muxes to switch the inputs to those blocks.

We use the *Mode* input to switch between generating Gaussian ($Mode = 0$) and Laplace ($Mode = 1$) noise. In Gaussian mode, we produce two Gaussian random numbers, while in Laplace mode, G_1 is ignored. To maximize throughout, we fully-pipeline our design to produce 1 random value per cycle, after an initial start-up latency of 16 cycles for Gaussian mode and 8 cycles for Laplace mode.

2.3 CNN accelerator architecture

To understand how our approach enables noise addition, we describe a typical ML accelerator. Weights for each layer are stored in off-chip (DRAM) memory. Accelerators run one layer at a time, computing the outputs using the processing elements (PEs). The

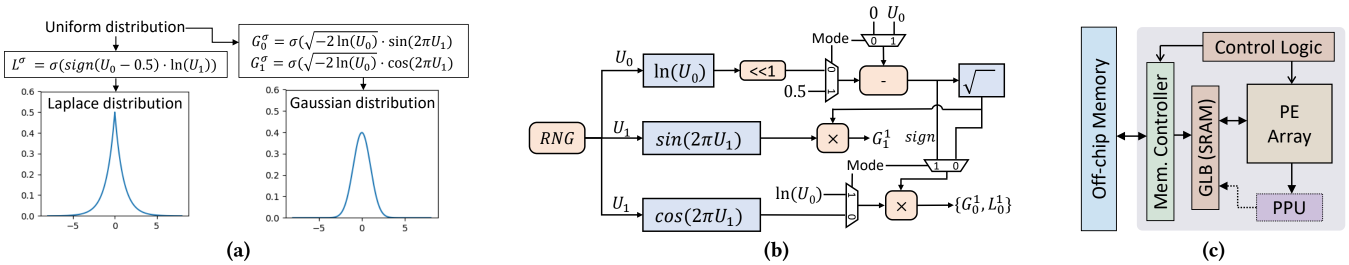

Figure 1: (a) Converting random numbers from uniform to Gaussian and Laplace distributions. (b) Hardware implementation of Box-Muller transform, to produce G^1 and L^1 random values. (c) Architecture of the *DiVa* CNN accelerator [50].

on-chip global buffer (GLB) is used to store the weights once they are read from DRAM and the activations between layers.

Baseline design. We use the *DiVa* accelerator as our baseline, as it is designed for DP-ML [50]. *DiVa* implements the DP-SGD algorithm, described in Sec. 2.1, To implement the gradient aggregation after adding noise, *DiVa* adds a post-processing unit (PPU) for parallel summation. Crucially for our work, *DiVa* does not detail any hardware for generating noise. Therefore, we compare DINAR against a version of *DiVa* which implements dedicated hardware for noise generation. Finally, while we use *DiVa*, DINAR is design-agnostic and can be easily incorporated into a variety of accelerators. For example, we demonstrate how DINAR can be used to add noise during inference for adversarial robustness using the Eyeriss design [15] in Sec. 6.

System integration. To meet strict area and energy constraints, edge ML accelerators – including *DiVa* – are typically deployed without a CPU [15, 25, 36, 50]. Since edge ML accelerators typically run a single network for long periods of time, they do not need the flexibility of a tightly-coupled CPU. They are instead configured, when required, using a scan-chain [15], a configuration bit-stream [25] or using on-chip control logic [36]. We therefore consider efficient noise addition in accelerators without CPUs.

3 DINAR

We motivate and describe DINAR, our technique for enabling efficient noise injection in ML accelerators. We begin with the challenges faced by existing approaches which generate noise directly in hardware. We then provide an overview of the hardware required for DINAR. Finally, we explain how DINAR overcomes the challenges of existing approaches.

3.1 Challenges

We now describe the difficulties associated with producing the noise required by security-centric ML algorithms, including the hardware shown in Fig. 1b.

ℂ1**: Altering the scale of the produced noise.** The hardware shown in Fig. 1b produces G^1 or L^1 noise. However, σ varies per network, requiring the hardware to generate noise of different scales [49]. To get such G^σ or L^σ noise, we need an additional multiplication operation, as shown in the equations in Fig. 1a.

ℂ2**: Floating-point noise.** Existing approaches (Fig. 1b) produce *fixed-point* values, while training requires floating-point values. Therefore, these fixed-point values must first be converted to floating point, which requires additional hardware.

ℂ3**: Continuous noise.** For certifiable privacy guarantees, DP requires the added noise to follow an ‘ideal’ distribution–every value across the entire range must be a possible output without any ‘gaps’. Assuming the uniform random numbers are evenly spread out – which is the case when using a true random number generator, the transformed distributions are then unevenly spaced, with more values around 0 and fewer values with larger magnitudes [47]. Prior work shows that this can be exploited by an attacker to undermine the guarantees offered by differential privacy [47].

ℂ4**: Timing side channel free noise.** Techniques to sample from Gaussian and Laplace distributions suffer from timing side channels [34]. The time taken to produce a sample leaks the magnitude of the noise. With a success rate of over 90%, this timing attack effectively subverts the security of differentially private systems.

3.2 High level overview

DINAR overcomes the challenges described above by **not producing noise directly in hardware**. As mentioned earlier, each model requires noise with a specific scale. Therefore, we pre-compute a large number of random points with this scale ahead of time and store them in plentiful off-chip DRAM memory. Thus, the noise values are loaded onto the accelerator along with the weights.

During runtime, DINAR simply reads a random value from this stored list to produce the required noise. DINAR only requires lightweight modifications to the memory controller and **imposes no other limitations on the architecture of the ML accelerator.** Specifically, DINAR modifies the address generation logic inside the memory controller (Fig. 1c).

Fig. 2a shows the hardware for address generation and the modifications made to support DINAR. The region labelled ② calculates the addresses sent to DRAM as a base plus an offset. The base is the starting address in memory of the item being read (e.g., the weights of a particular layer). As layer parameters – such as weights and inputs – are much larger than the size of the DRAM bus, we must perform multiple read operations to load all the values from DRAM. To support this, the offset is incremented by the stride value during each cycle to load the next set of values from DRAM ①.

As we store a list of pre-computed noise values in memory, we modify this hardware to randomly read a value from this list instead of incrementing by the stride every cycle. Our additions are shown in the region marked ②. We add a random number generator (RNG) to select a random instead of fixed value from memory. A 2 : 1 mux selects between the original offset and our randomly calculated offset. The `gen_rand_add` signal selects between these two modes. We then add the address where random values are stored in the chip's

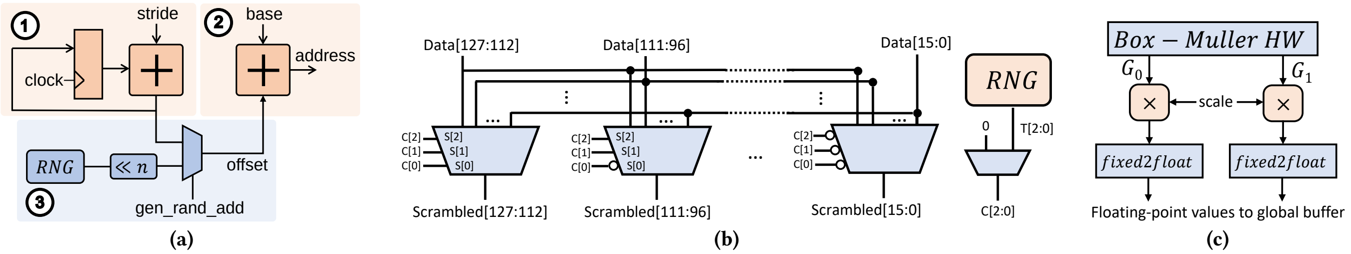

Figure 2: (a) Modifications to the address generation logic for DINAR. (b) DINAR hardware to scramble values read from DRAM. For clarity, some muxes and data lines are omitted. (c) NoiseGen hardware for producing G^{σ} and L^{σ} values.

control logic, so that the correct base address is provided for reading random values. For flexibility, we also add an n-bit shifter to allow for reading a contiguous set of noise points, which is greater than 128 bits. While n is set to 0 for DP-ML, we evaluate another application – adversarially robust ML – in Sec. 6, which requires this shifter.

3.3 Pre-fetching noise values from DRAM

As described in Sec. 2.1, DP-ML adds noise to the activations of every layer in the network. To seamlessly integrate with the scheduling schemes of existing accelerators, we frame noise addition as running a 'noise layer'. Consider the case of a noise layer being run between layers N and $N+1$ of a network. The weights for layer N have been read from DRAM and are stored in the GLB. Once the input for layer N is ready, that layer is run on the PE array. Since the weights have been dispatched, we can now pre-fetch the random values from DRAM and store them in the GLB. Reading random values while layer N is running allows us to hide DRAM read latency. Once layer N finishes running, we schedule our noise layer to run. The (noised) activations are then ready to be used as the inputs to layer $N + 1$. While the 'noise layer' is running, we pre-fetch the weights for layer $N + 1$. As we show in Sec. 5, for all the networks we evaluate, we always have sufficient space in the GLB to store noise points, along with the layer activations and weights.

3.4 Increasing randomness

For each individual DRAM read, a set of values is read in the same order each time. For the 128-bit DRAM bus and a 16-bit datatype we use in our evaluation (Sec. 5), we read 8 values, which are then used in the same order every time. This fixed ordering can potentially diminish the security of our approach. To prevent the chance of any information leakage from using points in-order every time, we add hardware to **randomly scramble** the order of values each time.

Scrambling hardware. Fig. 2b shows our scrambling hardware. We require $128/16 = 8$ multiplexers so that each 16-bit value can be placed in any position in the final 'scrambled' 128-bit output. We need a 3-bit RNG (i.e., $log_2(8)$) to select a random permutation each time. The RNG output is connected to the select signal of each mux, with varying negation (shown with the white circle on the mux select lines). This ensures that each mux selects a unique 16-bit value from the input. This scrambling produces $8! = 40,320$ possible scrambled orders. Coupled with the large number of overall values we store in DRAM (discussed in Sec. 5), this results in a very large number of possible random values. This makes it untenable for an attacker to learn the noise values that were added to activations to subvert the security of DINAR.

3.5 Benefits of DINAR

We now explain how DINAR addresses the challenges faced by prior approaches, described in Sec. 3.1.

- ℂ1: As DP-ML adds noise with a specific σ per network, DINAR simply stores values sampled from this distribution along with the network weights and avoids additional multiplications.
- ℂ2: DINAR can produce both fixed- and floating-point noise without additional conversion hardware.
- ℂ3: By producing noise points ahead of time, we obtain noise sampled from an 'ideal' distribution without missing any values and compromising the privacy guarantee offered by DP.
- ℂ4: As DINAR only reads from memory to get noise values, there is no variation in time based on the value being read. Therefore, we naturally obtain a constant time implementation, making DINAR immune to timing side-channel attacks.

4 Methodology

In this section, we present the methodolgy we use to evaluate DINAR. We begin by describing the two flavours of the `DiVa` design that we use in our evaluation. We then detail our evaluation setup.

4.1 Designs evaluated

We first describe `DiVa-DINAR`, which adds DINAR to *DiVa*. As *DiVa* does not describe how they generate noise in hardware, we also evaluate a version of *DiVa* incorporating noise generation hardware.

`DiVa-DINAR`. The first design we evaluate (`DiVa-DINAR`) implements DINAR, to read pre-computed values from DRAM. This includes the hardware to read random values from DRAM (Fig. 2a) and hardware to scramble these values (Fig. 2b). As we use a 128-bit DRAM bus and a 16-bit datatype, we read 8 values from DRAM per cycle. Thus, our scrambling hardware uses 8 muxes.

`DiVa-NoiseGen`. The second (`DiVa-NoiseGen`) generates noise directly on-chip, using the hardware shown in Fig. 1b. As described in Sec. 3.1, generating noise directly in hardware faces a number of challenges. To enable a fair comparison against DINAR, we augment the hardware in Fig. 1b to address two of these challenges. We refer to the final hardware, shown in Fig. 2c as `NoiseGen`, where the block labelled `Box-Muller` is the hardware in Fig. 1b.

ℂ1**: Supporting multiple scale factors.** Recall that the hardware in Figure 1b only produces G^1 or L^1 random values. To produce G^{σ} or L^{σ} noise, we need additional multipliers (equations in Fig. 1a). Therefore, we add the two additional multipliers in Fig. 2c.

ℂ2**: Floating-point noise.** `NoiseGen` produces fixed-point random values, which must be converted to floating point values using additional hardware (`fixed2float` in Fig. 2c). `fixed2float` is also pipelined and requires 16 cycles to convert a 16-bit fixed-point to a 16-bit floating-point value. Thus, the start-up latency of NoiseGen is 32 and 24 cycles, for Gaussian and Laplace modes, respectively. To save energy, we also power gate NoiseGen, when it is not being used. Finally, to match the operation of DINAR, we also store the noise values produced by NoiseGen in the GLB until needed.

4.2 Test setup

For training differentially private ML models, we use the Opacus [62] library, which adds DP-ML support to the PyTorch [51] framework. We evaluate three models using two datasets (Table 1), similar to prior work [23].

Modelling accelerators. We model our designs using Accelergy [59], which can model common components such as: multiply-accumulate units (MACs), SRAM for global memory, networks-on-chip and the interface to DRAM. We implement all additional hardware in Verilog, including: 1) the PPU for *DiVa*, 2) NoiseGen (Fig. 2c), 3) DINAR (Figs. 1b and 2b) We synthesize these using the Synopsys Design Compiler 2017.09 for the TSMC 65nm (nominal) process. We clock our design at 200MHz, matching prior work [15, 57].

Customization for edge ML. *DiVa* is designed for datacenters, while we target edge applications. Therefore, we opt for a smaller version of *DiVa*, suitable for low-power edge scenarios. *DiVa* uses a 128×128 PE array, while our designs use a 16×16 array. Similarly, our PPU can sum-reduce 16 values, in contrast to the original *DiVa* PPU, which operates on 128 values. We also proportionately scale the on-chip SRAM GLB from the 16MB used in *DiVa* to 256KB for our design. Finally, we assume a DRAM bus width of 128bits, similar to prior work [6, 21].

5 Evaluation

We now evaluate DiVa-DINAR and DiVa-NoiseGen for adding noise for DP-ML. We do not show accuracy results using noise produced by DINAR and NoiseGen, as they are within 0.5% of each other for all cases. Therefore, we focus on the overheads of both versions to the baseline *DiVa* accelerator. We first look at global memory and latency overheads which are similar for both our designs and then present area and energy overhead which differ by design.

Global memory. For both flavours, we store noise points in global memory until they are needed. This raises the possibility that we may not have enough space in the GLB to store the noise points, without evicting other data. We analyze this using Accelergy, which outputs the utilization of the GLB per layer. We see that there is no layer in any of the networks we evaluate where the GLB is more than 90.5% full. Thus, we always have at least 12 kB of space in the GLB for storing noise points. This allows us to store over 6000 noise points per layer. Thus, for the networks we evaluate, storing noise points does not lead to any global memory contention.

Latency. As both flavours read noise values while the previous layer is running, both incur a minimal latency overhead of just 0.49% for all the networks we evaluate. This is because both flavours are designed for low-latency operation. However, enabling low latency for DiVa-NoiseGen adds non-trivial area and energy overheads, which we explain next.

5.1 DiVa-DINAR overheads

We now present the DRAM footprint as well as the area and energy overheads of DiVa-DINAR.

DRAM. We examine the minimum number of noise points that must be stored to meet security requirements. As $\mathbb{C}3$ in Sec. 3.5 stated, we want to avoid 'gaps' when producing random values. Since we use the $bfloat16$ datatype, we must store one of each possible $65,536 (= 2^{16})$ points to avoid such gaps. With this minimum number of points in mind, we quantify the increase in memory footprint of DRAM. Table 2 shows the footprint of storing different numbers of points in DRAM. Even when storing over half a million points, we add just 1MB of storage. 1MB is less than 5% of the size of the smallest model we evaluate (PreActResNet-18). Thus DINAR only slightly increases the DRAM memory footprint.

On-chip area and energy. DINAR requires the use of an RNG for selecting random values from DRAM and for scrambling these values. We opt for a cryptographically secure RNG [9], which provides 9.4 Gbps of randomness. As we run our design at 200MHz, we obtain $9.4 \div 0.2 = 47$ random bits per cycle. For our scrambling hardware, we need 3 (i.e., $log_2(8)$) bits to scramble the 8 muxes we use in our design. We therefore have 44 bits of randomness available per cycle to read random values from DRAM. This allows us to randomly select from 2^{44} addresses in DRAM.

As each location in DRAM contains 8 values, DiVa-DINAR supports addressing up to 2^{44+3} (over 10^{13}) random values from memory. However, to address 1MB of stored noise points, we only require $log_2(524288) = 19$ bits from the TRNG. Thus, the TRNG we use supports significant headroom to address far more noise points in memory, if required. In total, DiVa-DINAR incurs only a 0.4% area and 0.2% energy overhead, compared to our baseline *DiVa* accelerator, which does not support any noise addition.

5.2 DiVa-NoiseGen overheads

As our training accelerators use a 16-bit datatype, we configure NoiseGen to produce 16-bit value every cycle. As shown in Fig. 1b, our implementation requires a random number generator (RNG) for its operation. The design by Lee et al. [39] – which we use as the basis for NoiseGen– requires 64 uniform random bits per cycle. However, using the RNG from above, we are only able to obtain 47 random bits per cycle. We therefore include two such RNGs to provide the 64 random bits needed for DiVa-NoiseGen.

On-chip area and energy. Table 3 shows the energy overhead of DiVa-NoiseGen for each model. We see that compared to the minimal 0.2% energy overhead imposed by DiVa-DINAR, DiVa-NoiseGen adds an average 8.15% energy overhead. Similarly, DiVa-NoiseGen adds 9.38% area overhead to the baseline accelerator. This increased overhead is due to: 1) the hardware to generate noise, including our augmentations described in Sec. 4.1 and 2) the extra RNG needed to produce enough random bits. These overheads are significant in the context of power and cost sensitive edge ML accelerators. This demonstrates the benefit of DINAR, compared to adding dedicated noise generation hardware. Note that Table 3 does not distinguish between CIFAR-10 and CIFAR-100; these models vary only in the number of output classes. This only impacts the size of the last layer which accounts for $< 1\%$ of the total energy in all cases.

Table 1: List of models evaluated.

Model	Dataset	Model size (MB)	
		int8	float16
PreActResNet-18 (P18)	CIFAR-10	11.18	22.36
	CIFAR-100	11.32	22.64
WideResNet-32 (W32)	CIFAR-10	46.18	92.36
	CIFAR-100	46.35	92.70
VGG-16 (V16)	CIFAR-10	14.75	29.50

Table 2: DRAM storage footprint for varying amounts of noise points stored.

No. of noise values in DRAM	Storage required (KB)	Footprint (%) over smallest model
65536	128	0.56
131072	256	1.11
262144	512	2.23
524288	1024	4.47

Table 3: Energy overhead of `DiVa-NoiseGen`.

Model	Energy overhead
P18	10.05%
W32	7.01%
V16	7.39%
Avg.	8.15%

6 Adversarial robustness

So far, our focus has been on enabling DP-ML on edge AI accelerators. We now demonstrate how DINAR can be used to add noise for robustness against *adversarial attacks*. Adversarial attacks subtly alter inputs to cause the ML model to mis-classify those inputs as a different class [41]. Such attacks predominantly target Convolutional Neural Networks (CNNs) used for image classification [14]. When used against networks powering safety critical systems, such attacks can have devastating results.

Due to the effectiveness of adversarial attacks, many prior works investigate making ML models more robust against such attacks [20, 28, 33, 37, 41, 49, 63]. In particular, several techniques add noise either to the inputs [37] or the layer activations [28] to improve robustness. The most commonly used noise is sampled from a univariate Gaussian distribution [28, 33, 37]. However, these techniques are ineffective against specialized attacks [7, 8]. Newer techniques add noise from a *multivariate* Gaussian distribution [20, 61].

We use Weight Covariance Alignment (WCA) [20] as our example, as it achieves state-of-the-art results for both robustness and classification accuracy. WCA injects *anisotropic multivariate* Gaussian noise before the final fully-connected layer of the network. During training, WCA uses a modified loss function, to 'align' the network weights with the distribution of the added noise. WCA outperforms techniques which add noise to the output of several layers [28, 33]. As we add *univariate* noise in DP-ML, each noise value can be treated independently. For WCA, we need *multivariate* noise, which consists of several values that must be stored and used contiguously and cannot be scrambled. WCA requires the dimensions of this distribution to be greater than the number of output classes. They recommend using 32 and 128 dimensions for CIFAR-10 and CIFAR-100, respectively.

6.1 Test methodology

We now describe our methodology for evaluating DINAR for adversarial robustness. We first establish some terminology required for our evaluation later.

Attack terminology. The strength of attacks is typically quantified using ϵ, which is the maximum change that can be made to any single pixel in the input. As pixels in our datasets are in the range $[0, 255]$, we set $\epsilon = 8/255$, matching prior work [55, 65]. We also distinguish between: *classification accuracy*, which is the accuracy for benign test inputs and *robustness accuracy*, the accuracy for adversarially attacked images.

Attacks. We evaluate robustness accuracy using two common attacks: 1) **Fast Gradient Sign Method** (FGSM) [26] calculates the gradient of the input and alters the input to maximize this gradient to cause a mis-classification. 2) **Projected Gradient Descent** (PGD) [42] improves on FGSM by iteratively making smaller alterations to the input image to increase the attack's success rate.

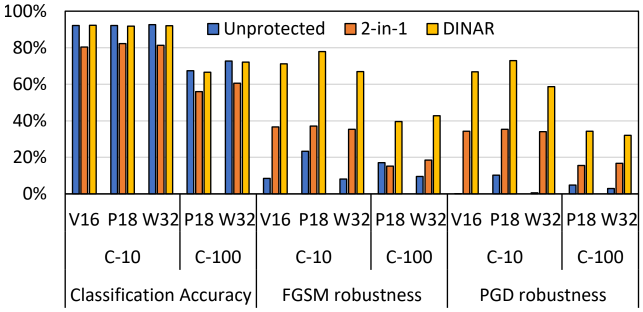

Figure 3: Classification and robustness accuracies for evaluated networks. We use the shortened names listed in Table 1

Models. We evaluate the same models as before (Table 1). We use PyTorch [51] v1.11 to run our models and use the TorchAttacks library for attacks [35]. Since we target inference, we use an 8-bit fixed-point datatype. We train all models using 32-bit floating point and perform post-training quantization using PyTorch to obtain int8 models. In all cases, we saw a <1% difference in classification accuracy between the 32-bit floating-point and int8 models.

Architecture description. Since WCA adds noise during inference, we evaluate a new accelerator design, based on Eyeriss [15]. `Eyeriss-DINAR` uses DINAR, while `Eyeriss-NoiseGen` incorporates `NoiseGen`. Again we use a DRAM bus-width of 128 bits and operate our design at 200MHz. As WCA requires points to be used in the same order, we do not add our scrambling hardware (Fig. 2b) to `Eyeriss-DINAR`.

Comparison against prior work. To demonstrate the efficacy of noise addition, we compare against another hardware technique for adversarial robustness. 2-in-1 accelerator (2-in-1) trains models at various bit-widths and randomly selects one model to run each time [23]. Using a random bit-width during inference makes it more difficult for the attack to infer gradients.

6.2 Evaluation

Accuracy. To evaluate the efficacy of noise addition for adversarial robustness, we compare WCA (using DINAR for producing noise) against the baseline unprotected models and 2-in-1 accelerator. Fig. 3 shows that the baseline networks achieve high classification accuracy but low robustness accuracy. This is expected as baseline models have no defences applied, either during training or inference, to improve their robustness. While 2-in-1 improves robustness accuracy compared to the baseline models, it suffers from low classification accuracy. We believe this is because models

Table 4: DRAM memory footprint required (compared to the smallest model we evaluate) for varying amounts of noise vectors stored.

Number of noise vectors stored	32 points per vector		256 points per vector	
	Storage (KB)	Footprint (%)	Storage (KB)	Footprint (%)
128	4	0.03	32	0.28
256	8	0.07	64	0.55
512	16	0.14	128	1.10
1024	32	0.28	256	2.21
2048	64	0.56	512	4.42

trained at various bit-widths do not significantly alter the model's loss landscape to prevent attacks. That is, even when switching bit-widths for each input, the different models are similar enough that attacks are still successful.

DINAR maintains high classification accuracy, with a <1% difference compared to the baseline models and also achieves the highest robustness accuracy. This demonstrates that noise injection is an effective method for improving the robustness of ML models, without sacrificing classification accuracy. Similar to DP-ML, we do not show separate results when using `NoiseGen` and DINAR, as they are identical. Thus, any noise generation technique achieves high robustness and classification accuracy. However, the overheads vary significantly between *Eyeriss-DINAR* and *Eyeriss-NoiseGen*, as we now show.

6.3 Overheads

As before, we first present the global memory and latency overheads for `Eyeriss-DINAR` and `DiVa-NoiseGen` which are similar. We then detail the area and energy overheads, which vary by design.

Global memory. Similar to DP-ML, we ensure there is sufficient space in global memory to store random noise vectors when needed. Since WCA only adds a single noise layer, we can fit the required noise in the GLB for the studied networks without contention.

Latency. Since WCA only requires reading values from DRAM for a single layer in the network, both *Eyeriss-DINAR* and *Eyeriss-NoiseGen* add minimal latency overhead.

6.3.1 Eyeriss-DINAR Unlike DP-ML, we do not have a theoretical lower limit for the number of vectors that must be stored in DRAM. Therefore, we opt to perform an empirical analysis to determine this lower limit. Recall that for WCA, we need vectors of 32 values for CIFAR-10 and 256 values for CIFAR-100. We experiment with storing k such vectors and sampling one each time. We see that accuracies are unaffected down to $k = 128$ stored vectors. Reducing k further results in a drop in both classification and robustness accuracy. Therefore, we conclude that storing a minimum of 128 vectors is necessary to maintain the benefits of WCA.

DRAM footprint. Table 4 shows the DRAM footprint for different number of stored vectors. Therefore, we show the footprint for vectors of size 32 and 256, compared to the smallest model we evaluate, PreActResNet-18 (P18). We sweep the number of stored vectors starting from the minimum value of 128, necessary for maintaining accuracy. While DINAR works with far fewer points, even storing 2048 vectors adds just 0.56% and 4.42% for the 32 and 256 sized noise vectors, respectively. DINAR only slightly increases the DRAM memory footprint. For `Eyeriss-DINAR`, we require 11 bits from the RNG to access a table of 2048 stored vectors. Using the same RNG as `DiVa-DINAR`, we have 47 bits from the RNG, providing us with plenty of headroom for storing many more vectors.

On-chip area and energy. `Eyeriss-DINAR` adds a $2:1$ mux and an RNG, similar to `DiVa-DINAR`. Together, these only add an additional 0.26% area compared to our baseline accelerator, which cannot add noise. However, as we only require the random values before a single layer, we use power gating to disable the RNG when it is not needed. This results in an energy overhead of $< 0.1\%$ for `Eyeriss-DINAR`.

6.3.2 Eyeriss-NoiseGen To match the 8-bit datatype we use for inference, we use a `NoiseGen` configuration which produces an 8-bit value every cycle, for `Eyeriss-NoiseGen`. This then reduces the number of uniform random bits required to 32. Thus, a single RNG is also sufficient to provide the required noise for `Eyeriss-NoiseGen`.

On-chip area and energy. `Eyeriss-NoiseGen` adds 8.05% area overhead to the baseline accelerator. Despite the smaller size of the 8-bit `NoiseGen` configuration and the exclusion of the extra hardware shown in Figure 2c, `Eyeriss-NoiseGen` still adds non-trivial area overhead. Due to the overall smaller size of the accelerator for int8 vs. bfloat16, even a small amount of hardware incurs a high area overhead. Similar to `DiVa-DINAR`, we employ power-gating to reduce leakage power in `Eyeriss-NoiseGen`, resulting in an average energy overhead of just 0.2%.

7 Securing split inference

We now describe another security-centric ML algorithm that requires random noise. When running large ML models on edge devices, one approach is to offload a portion of the network inference to the cloud. This requires transmitting partially processed data over a network, which can lead to privacy breaches. One approach to secure this data is to add noise before transmission, to reduce the mutual information between the original and transmitted data [45, 46, 56]. Titcombe et al. [56] add Laplace noise, while Cloak [46] adds Gaussian noise. In contrast, Shredder [45] adds custom noise distributions – learned during training – to the data before transmission. These works target inference using CPUs or GPUs, where the required noise is produced 'on-the-fly'. DINAR enables these algorithms to run on edge accelerators, which do not have CPUs or GPUs.

8 Related work

Generating noise in hardware. Various techniques have been proposed for generating Gaussian random numbers, in addition to the Box-Muller transform (Sec. 2.2). These include the rejection and inversion methods. *Rejection*: These techniques employ 'sample-and-reject' techniques, where they produce random numbers and sample them to see if they fit 'inside' the required Gaussian distribution. An example of this is the Ziggurat method, which is often used in software [19, 64]. However, this loop-based method is difficult to deploy in hardware, where constant latency is preferred. *Inversion*: This technique uses the inverse of the cumulative distribution function (ICDF) of the Gaussian distribution to generate random values. However, this ICDF does not have a closed form solution, leading to approaches implementing piecewise linear approximations [27, 38]. However, these methods are susceptible to attacks when used to generate noise for DP-ML [34]. Prior work produces Laplace random noise for differential privacy [16]. Using fixed-point can lead to a loss of privacy, due to quantization error; to counter this, the authors propose hardware to post-process the produced noise to maintain privacy. By pre-computing and storing noise points, DINAR allows us to perform any required post-processing ahead of time and avoid the latency cost of doing so at runtime.

Using analog noise. Prior work uses the noise inherent in analog components for robustness during inference. Examples include under-volting the ML accelerator to reduce the transistor error margins and produce random errors as noise [31, 32, 43], and tuning the noise of 6T and 8T SRAM cells, used to store model weights and activations [12]. Roy et al. show that ML accelerators which use crossbars in non-volatile memory to perform calculations can also provide robustness against adversarial attacks [52]. However, achieving robustness using analog noise without sacrificing performance requires careful tuning to account for hardware non-idealities [53]. In contrast, DINAR enables robustness without any hardware fine tuning.

ML accelerator security. There has been many works that aim to make ML accelerators more secure against a variety of attacks. Encryption [29] and Trusted Execution Environments [17] have been proposed to secure cloud-based ML accelerators. However, the overheads of these approaches would be infeasible on the resource constrained edge accelerators we consider. Other works look at side-channel attacks against ML accelerators such as recovering inputs via power side channels [58] or recovering the network structure by observing off-chip memory access patterns [30]. HuffDuff demonstrates an attack against accelerators that employ sparsity [60]. All these works rely on side channels, while our work focuses on enabling security-centric ML algorithms on edge ML accelerators.

9 Conclusion

We identify that current edge accelerators cannot generate noise, which is essential for security-critical ML algorithms. We present DINAR, which safely and efficiently enables noise addition on edge ML accelerators. Compared to generating noise directly in hardware, DINAR adds 23× lower area and 40× lower energy. Our work is the first to support efficiently adding noise to ML accelerators. Given the flexibility of DINAR, we hope to motivate architects to consider other security-centric applications which can benefit from random noise addition.

Acknowledgments

We thank the anonymous reviewers and the members of NEJ group for their valuable feedback. We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant RGPIN-2020-04179. This research was undertaken, in part, thanks to funding from the Canada Research Chairs program.

References

[1] Martin Abadi et al. 2016. Deep learning with differential privacy. In *Proceedings of the ACM SIGSAC conference on computer and communications security.*

[2] John M Abowd. 2018. The US Census Bureau adopts differential privacy. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining.* 2867–2867.

[3] Muhammad Aitsam. 2022. Differential privacy made easy. In *Proceedings of the International Conference on Emerging Trends in Electrical, Control, and Telecommunication Engineering (ETECTE).* IEEE.

[4] Abdulmalik Alwarafy et al. 2021. A Survey on Security and Privacy Issues in Edge-Computing-Assisted Internet of Things. *IEEE Internet of Things Journal* 8, 6 (2021).

[5] Apple. 2017. Learning with Privacy at Scale. https://docs-assets.developer.apple.com/ml-research/papers/learning-with-privacy-at-scale.pdf.

[6] Hadi Asghari-Moghaddam et al. 2016. Near-DRAM acceleration with single-ISA heterogeneous processing in standard memory modules. *IEEE Micro* 36, 1 (2016).

[7] Anish Athalye and Nicholas Carlini. 2018. On the Robustness of the CVPR 2018 White-Box Adversarial Example Defenses. https://arxiv.org/abs/1804.03286

[8] Anish Athalye, Nicholas Carlini, and David Wagner. 2018. Obfuscated Gradients Give a False Sense of Security: Circumventing Defenses to Adversarial Examples. https://arxiv.org/abs/1802.00420

[9] Mohammed Bakiri et al. 2018. A Hardware and Secure Pseudorandom Generator for Constrained Devices. *IEEE Transactions on Industrial Informatics* 14, 8 (2018).

[10] Samah Baraheem and Zhongmei Yao. 2022. A Survey on Differential Privacy with Machine Learning and Future Outlook. *arXiv preprint arXiv:2211.10708* (2022).

[11] Sathwika Bavikadi et al. 2022. A Survey on Machine Learning Accelerators and Evolutionary Hardware Platforms. *IEEE Design and Test* 39, 3 (2022).

[12] Abhiroop Bhattacharjee, Abhishek Moitra, and Priyadarshini Panda. 2021. Efficiency-driven Hardware Optimization for Adversarially Robust Neural Networks. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE).*

[13] George E. P. Box and Mervin E. Muller. 1958. A Note on the Generation of Random Normal Deviates. *Annals of Mathematical Statistics* 29 (1958), 610–611.

[14] Anirban Chakraborty et al. 2018. Adversarial Attacks and Defences: A Survey. https://arxiv.org/abs/1810.00069

[15] Yu-Hsin Chen et al. 2016. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits* 52, 1 (2016), 127–138.

[16] Woo-Seok Choi et al. 2018. Guaranteeing Local Differential Privacy on Ultra-Low-Power Systems. In *Proceedings of the International Symposium on Computer Architecture (ISCA).*

[17] Aritra Dhar, Supraja Sridhara, Shweta Shinde, Srdjan Capkun, and Renzo Andri. 2022. Empowering Data Centers for Next Generation Trusted Computing. *arXiv preprint arXiv:2211.00306* (2022).

[18] Cynthia Dwork. 2006. Differential privacy. In *Proceedings of the 33rd International Colloquium on Automata, Languages and Programming (ICALP).* Springer.

[19] Hassan Edrees et al. 2009. Hardware-Optimized Ziggurat Algorithm for High-Speed Gaussian Random Number Generators. In *ERSA.* 254–260.

[20] Panagiotis Eustratiadis, Henry Gouk, Da Li, and Timothy Hospedales. 2021. Weight-covariance alignment for adversarially robust neural networks. In *Proceedings of the 38th International Conference on Machine Learning.* PMLR.

[21] Amin Farmahini-Farahani et al. 2015. NDA: Near-DRAM acceleration architecture leveraging commodity DRAM devices and standard memory modules. In *Proceedings of the 21st IEEE International Symposium on High Performance Computer Architecture (HPCA).*

[22] Matthew Fredrikson et al. 2014. Privacy in Pharmacogenetics: An End-to-End Case Study of Personalized Warfarin Dosing. In *23rd USENIX Security Symposium.*

[23] Yonggan Fu et al. 2021. 2-in-1 Accelerator: Enabling Random Precision Switch for Winning Both Adversarial Robustness and Efficiency. In *Proceedings of the 54th annual IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[24] Quan Geng and Pramod Viswanath. 2015. Optimal noise adding mechanisms for approximate differential privacy. *IEEE Transactions on Information Theory* 62, 2 (2015), 952–969.

[25] Graham Gobieski et al. 2021. Snafu: An Ultra-Low-Power, Energy-Minimal CGRA-Generation Framework and Architecture. In *Proceedings of the 48th Annual ACM/IEEE International Symposium on Computer Architecture (ISCA).*

[26] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. 2014. Explaining and Harnessing Adversarial Examples. https://doi.org/10.48550/ARXIV.1412.6572

[27] Roberto Gutierrez, Vicente Torres, and Javier Valls. 2012. Hardware Architecture of a Gaussian Noise Generator Based on the Inversion Method. *IEEE Transactions on Circuits and Systems II: Express Briefs* 59, 8 (2012).

[28] Zhezhi He, Adnan Siraj Rakin, and Deliang Fan. 2019. Parametric noise injection: Trainable randomness to improve deep neural network robustness against adversarial attack. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 588–597.

[29] Weizhe Hua, Muhammad Umar, Zhiru Zhang, and G. Edward Suh. 2020. GuardNN: Secure DNN accelerator for privacy-preserving deep learning. *arXiv preprint*

arXiv:2008.11632 (2020).

[30] Weizhe Hua, Zhiru Zhang, and G. Edward Suh. 2018. Reverse Engineering Convolutional Neural Networks through Side-Channel Information Leaks. In *Proceedings of the 55th Annual Design Automation Conference.*

[31] Md Shohidul Islam, Behnam Omidi, Ihsen Alouani, and Khaled N. Khasawneh. 2023. VPP: Privacy Preserving Machine Learning via Undervolting. In *Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST).*

[32] Shohidul Islam, Ihsen Alouani, and Khaled N. Khasawneh. 2021. Lower Voltage for Higher Security: Using Voltage Overscaling to Secure Deep Neural Networks. In *Proceedings of the IEEE/ACM International Conference On Computer Aided Design (ICCAD).*

[33] Ahmadreza Jeddi et al. 2020. Learn2Perturb: An End-to-End Feature Perturbation Learning to Improve Adversarial Robustness. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).*

[34] Jiankai Jin et al. 2022. Are we there yet? timing and floating-point attacks on differential privacy systems. In *Proceedings of the IEEE Symposium on Security and Privacy (SP).* 473–488.

[35] Hoki Kim. 2020. Torchattacks: A pytorch repository for adversarial attacks. https://arxiv.org/abs/2010.01950

[36] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. 2018. MAERI: Enabling Flexible Dataflow Mapping over DNN Accelerators via Reconfigurable Interconnects. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems.*

[37] Mathias Lecuyer et al. 2019. Certified robustness to adversarial examples with differential privacy. In *IEEE Symposium on Security and Privacy (SP).* IEEE.

[38] Dong-U Lee, Ray C.C. Cheung, John D. Villasenor, and Wayne Luk. 2006. Inversion-based hardware gaussian random number generator: A case study of function evaluation via hierarchical segmentation. In *Proceedings of the IEEE International Conference on Field Programmable Technology.*

[39] D-U Lee, John D Villasenor, Wayne Luk, and Philip Heng Wai Leong. 2006. A hardware Gaussian noise generator using the Box-Muller method and its error analysis. *IEEE transactions on computers* 55, 6 (2006), 659–671.

[40] Jaewoo Lee and Chris Clifton. 2011. How much is enough? choosing ε for differential privacy. In *Proceedings of the 14th International Conference on Information Security (ISC).*

[41] Aleksander Madry et al. 2017. Towards Deep Learning Models Resistant to Adversarial Attacks. https://arxiv.org/abs/1706.06083

[42] Aleksander Madry et al. 2018. Towards Deep Learning Models Resistant to Adversarial Attacks. In *International Conference on Learning Representations.*

[43] Saikat Majumdar, Mohammad Hossein Samavatian, Kristin Barber, and Radu Teodorescu. 2021. Using Undervolting as an On-Device Defense Against Adversarial Machine Learning Attacks. In *Proceedings of the IEEE International Symposium on Hardware Oriented Security and Trust (HOST).*

[44] Jamshaid Sarwar Malik and Ahmed Hemani. 2016. Gaussian random number generation: A survey on hardware architectures. *ACM Computing Surveys (CSUR)* 49, 3 (2016).

[45] Fatemehsadat Mireshghallah et al. 2020. Shredder: Learning Noise Distributions to Protect Inference Privacy. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems.*

[46] Fatemehsadat Mireshghallah et al. 2021. Not all features are equal: Discovering essential features for preserving prediction privacy. In *Proceedings of the Web Conference 2021.*

[47] Ilya Mironov. 2012. On Significance of the Least Significant Bits for Differential Privacy. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security.*

[48] Chaya Nayak. 2020. New privacy-protected Facebook data for independent research on social media's impact on democracy. https://research.facebook.com/blog/2020/2/new-privacy-protected-facebook-data-for-independent-research-on-social-medias-impact-on-democracy/.

[49] Nicolas Papernot et al. 2016. Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks. In *2016 IEEE Symposium on Security and Privacy (SP).* 582–597.

[50] Beomsik Park et al. 2022. DiVa: An Accelerator for Differentially Private Machine Learning. In *Proceedings of the 55th IEEE/ACM International Symposium on Microarchitecture (MICRO).*

[51] Adam Paszke et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32.* 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[52] Deboleena Roy, Indranil Chakraborty, Timur Ibrayev, and Kaushik Roy. 2021. On the Intrinsic Robustness of NVM Crossbars Against Adversarial Attacks. In *2021 58th ACM/IEEE Design Automation Conference (DAC).*

[53] Deboleena Roy, Chun Tao, Indranil Chakraborty, and Kaushik Roy. 2021. On the Noise Stability and Robustness of Adversarially Trained Networks on NVM Crossbars. https://arxiv.org/abs/2109.09060

[54] Michael Schulte and Earl Swartzlander. 1994. Hardware designs for exactly rounded elementary functions. *IEEE Trans. Comput.* 43, 8 (1994).

[55] Liwei Song, Reza Shokri, and Prateek Mittal. 2019. Privacy Risks of Securing Machine Learning Models against Adversarial Examples. In *Proceedings of the Conference on Computer and Communications SecuritySIGSAC.*

[56] Tom Titcombe et al. 2021. Practical defences against model inversion attacks for split neural networks. In *Proceedings of the Workshop on Distributed and Private Machine Learning (DPML) co-located with ICLR.*

[57] Xingbin Wang et al. 2019. NPUFort: a secure architecture of DNN accelerator against model inversion attack. In *Proceedings of the 16th ACM International Conference on Computing Frontiers.*

[58] Lingxiao Wei et al. 2018. I Know What You See: Power Side-Channel Attack on Convolutional Neural Network Accelerators. In *Proceedings of the 34th Annual Computer Security Applications Conference.*

[59] Yannan N. Wu, Joel S. Emer, and Vivienne Sze. 2019. Accelergy: An Architecture-Level Energy Estimation Methodology for Accelerator Designs. In *IEEE/ACM International Conference On Computer Aided Design (ICCAD).*

[60] 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 (ASPLOS).*

[61] Hao Yang et al. 2022. Rethinking feature uncertainty in stochastic neural networks for adversarial robustness. *arXiv preprint arXiv:2201.00148* (2022).

[62] Ashkan Yousefpour et al. 2021. Opacus: User-Friendly Differential Privacy Library in PyTorch. *arXiv preprint arXiv:2109.12298* (2021).

[63] Xiaoyong Yuan, Pan He, Qile Zhu, and Xiaolin Li. 2017. Adversarial Examples: Attacks and Defenses for Deep Learning. https://doi.org/10.48550/ARXIV.1712.07107

[64] Guanglie Zhang et al. 2005. Ziggurat-based hardware Gaussian random number generator. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications.*

[65] Huan Zhang et al. 2019. The Limitations of Adversarial Training and the Blind-Spot Attack. In *Proceedings of the International Conference on Learning Representations.*