FriendlyFoe: Adversarial Machine Learning as a Practical Architectural Defense against Side Channel Attacks

Hyoungwook Nam
hn5@illinois.edu
University of Illinois at Urbana-Champaign, USA

Raghavendra Pradyumna Pothukuchi
raghav.pothukuchi@yale.edu
Yale University, USA

Bo Li
lbo@illinois.edu
University of Illinois at Urbana-Champaign, USA

Nam Sung Kim
nskim@illinois.edu
University of Illinois at Urbana-Champaign, USA

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

Abstract

Machine learning (ML)-based side channel attacks have become prominent threats to computer security. These attacks are often powerful, as ML models easily find patterns in signals. To address this problem, this paper proposes dynamically applying Adversarial Machine Learning (AML) to obfuscate side channels. The rationale is that it has been shown that intelligently injecting an adversarial perturbation can confuse ML classifiers. We call this approach *FriendlyFoe* and the neural network we introduce to perturb signals FriendlyFoe *Defender*.

FriendlyFoe is a practical, effective, and general architectural technique to obfuscate signals. We show a workflow to design Defenders with low overhead and information leakage, and to customize them for different environments. Defenders are transferable, i.e., they thwart attacker classifiers that are different from those used to train the Defenders. They also resist adaptive attacks, where attackers train using the obfuscated signals collected while the Defender is active. Finally, the approach is general enough to be applicable to different environments. We demonstrate FriendlyFoe against two side channel attacks: one based on memory contention and one on system power. The first example uses a hardware Defender with ns-level response time that, for the same level of security as a Pad-to-Constant scheme, has 27% and 64% lower performance overhead for single- and multi-threaded workloads, respectively. The second example uses a software Defender with ms-level response time that reduces leakage by 3.7× over a state-of-the-art scheme while reducing the energy overhead by 22.5%.

CCS Concepts

• **Security and privacy → Side-channel analysis and countermeasures**.

Keywords

Hardware security, Side-channel analysis, Machine learning

PACT '24, October 14–16, 2024, Long Beach, CA, USA

ACM ISBN 979-8-4007-0631-8/24/10
https://doi.org/10.1145/3656019.3676952

ACM Reference Format:
Hyoungwook Nam, Raghavendra Pradyumna Pothukuchi, Bo Li, Nam Sung Kim, and Josep Torrellas. 2024. FriendlyFoe: Adversarial Machine Learning as a Practical Architectural Defense against Side Channel Attacks. In *International Conference on Parallel Architectures and Compilation Techniques (PACT '24), October 14–16, 2024, Long Beach, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3656019.3676952

1 Introduction

Securing computers against information leakage has never been more challenging. Attackers can use complex analyses to exfiltrate sensitive information from a variety of side channels. Recently, attackers have begun using machine learning (ML) for side channel analysis—a move that greatly amplifies the risk of information leakage for many systems (e.g., [32, 30, 34, 39]).

ML-based side channel analysis is powerful for several reasons [39, 10, 32, 7, 29]. First, ML models learn information-carrying patterns from the dataset automatically, without domain-specific assumptions or feature engineering. In contrast, earlier approaches like template attacks assumed particular properties of the collected signals and targeted specific connections between the data [32]. As a result, they often fail to leverage all the correlations in the data, and become vulnerable to countermeasures. Second, ML networks can circumvent simple defenses like noise addition, masking, and shuffling. Finally, ML networks can work with raw data, lowering the efforts to construct leaky side channels and pre-process signals.

In contrast, most current approaches to obfuscate information against side channel analysis (e.g., [12, 60, 43]) are specific to particular attacks and circumstances. They are hard to generalize across different types of side channels. Further, the signal distortions they create often result in high performance overheads. Therefore, it is necessary to develop more intelligent defense strategies that are general and minimize performance overheads.

An intriguing approach is to use adversarial machine learning (AML) for defense. AML is a technique that intelligently perturbs the inputs of an ML classifier such that it causes a misclassification. AML has become popular in confusing image classifiers [56], and offers an opportunity to defeat ML-based side channel analyses.

There are two prior works that use AML as a countermeasure [41, 44]. These methods require the full trace of a signal to determine how it should be distorted. Such a trace is only available after the system execution. By the time these schemes start post-processing

the signal, the information has already leaked to an attacker. To protect real systems, we need a dynamic defense that produces the perturbations *on the fly*.

This paper shows, for the first time, that on-the-fly AML is an effective and general architectural technique to obfuscate side channel signals. We call the approach *FriendlyFoe* and the network to perturb signals FriendlyFoe *Defender*. We show the practicality, efficacy, and generality of FriendlyFoe against side channel attacks.

The core of our proposal is to train a FriendlyFoe Defender using a Generative Adversarial Network (GAN) [16, 56]. This approach trains three networks together: the Defender, which adds noise to a signal, and two attackers (classifier and discriminator) that take the noisy signal and try to make a correct guess. All networks are trained adversarially: the Defender to add noise to mislead the attackers; the attackers to identify the correct label. Through this, the Defender is trained to minimize information leakage with minimal noise addition, even against a strong adaptive attacker.

We propose a FriendlyFoe workflow to build Defenders for different environments. Such Defenders are *transferable*—i.e., they are effective against other types of attacker classifiers beyond those they trained with. Moreover, they also resist *adaptive* attacks, where attacker classifiers train using the obfuscated signals collected while the Defender is in operation. In addition, Defenders can exhibit inter-application transferability, where a Defender trained for one victim application can provide a reasonable level of protection to other victim applications. Finally, the approach is general, as it can be applied to different side channel analyses that collect time-series signals and apply pattern recognition to decode secret information.

We demonstrate FriendlyFoe by applying it to two different side channel attacks: one based on memory contention and one on system power. The first example uses a hardware Defender module with ns-level response time that, for the same level of security as a Pad-to-Constant scheme, has 27% and 64% lower performance overhead for single- and multi-threaded workloads, respectively. The second example uses a software Defender with ms-level response time that reduces leakage by 3.7× over a state-of-the-art scheme while reducing the energy overhead by 22.5%.

The contributions of this paper are:

- Showing the practicality, efficacy, and generality of on-the-fly AML as an architectural defense.
- Design of a FriendlyFoe Defender and a GAN structure to train it for architectural use.
- Workflow to design, implement, train, and deploy FriendlyFoe Defenders for different environments.
- Application of FriendlyFoe to thwart two side channel attacks based on memory contention and system power.

2 Background

2.1 Side Channel Analysis with ML

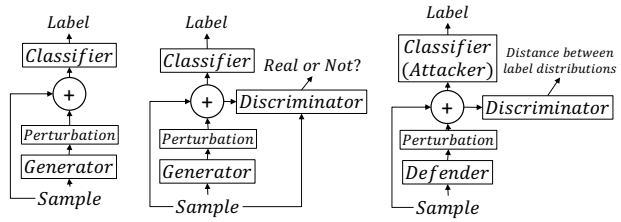
The goal of side channel analysis is to recover sensitive information from a victim's execution. Such analysis has been successfully applied to exfiltrate information from structures like caches [59, 1], branch predictors [13], and interconnects [34], as well as physical signals of computers like power [25, 43] and electromagnetic (EM) emanations [14].

Recently, attackers have used ML classifiers for signal analysis, leveraging their pattern recognition capabilities [22, 19, 21, 39, 54]. ML-based side channel analyses have recovered sensitive data from encryption modules [19], caches [57], power measurements [40, 27, 21, 45, 32], EM emissions [3] and on-chip interconnects [34]. These classifiers, particularly Deep Neural Networks (DNNs), can automatically identify information-carrying patterns, overcoming simple defenses like noise addition or signal misalignment [7, 39]. Hence, there is an urgent need to develop countermeasures that are effective against even the strongest ML-based attackers.

2.2 Adversarial Machine Learning (AML)

Recent studies have found that it is possible to generate *adversarial examples* [17] against an ML classifier which, with very small perturbations (imperceptible to the human eye), bring drastic changes to the classification outcomes. For example, adding an imperceptible noise to an image of a panda can cause a DNN to misclassify the image as that of a gibbon [17].

Figure 1a shows how Adversarial Machine Learning (AML) works. Given an input sample (e.g., an image), a *Generator* adds a small perturbation that causes a *Classifier* to misclassify the image to a wrong label. If both the generator and classifier are DNNs, they can be trained *adversarially* [56]—i.e., the generator is trained to create better perturbations, while the classifier is trained to predict the correct label despite the perturbation. In the best case, after the training, the generator's perturbations are *transferable* to other classifiers [36]. This means that the generator can induce misclassification in other classifiers that have a different structure and parameters than the original classifier.



(a) Adversarial ML (b) Adversarial GAN (c) Training the Defender

Figure 1: From adversarial ML (AML) to training a FriendlyFoe Defender. In our design, we call the generator *Defender*.

A Generative Adversarial Network (GAN) [16] is a structure that trains a generator adversarially against another neural network called *Discriminator*. The generator creates samples in a certain domain (e.g., images) and the discriminator tries to distinguish whether a sample is from the generator (fake) or from the real dataset (real). A common practice is for the discriminator to estimate the differences between the fake and real sample distributions using the Wasserstein distance [2]. By GAN training, the generator learns to create realistic outputs that can deceive the discriminator.

The Adversarial GAN (AdvGAN) structure introduced by Xiao et al. [56] combines the previous two concepts for maximum generator effectiveness. This structure, shown in Figure 1b, includes a generator, a classifier, and a discriminator network that are trained

adversarially. The classifier, generator, and discriminator can be any type of DNN, where the choice is often based on the data type. For images, a Convolutional Neural Network (CNN) [26] is often used, while to detect sequential patterns, a recurrent neural network (RNN) such as a Long Short-Term Memory (LSTM) [20] or a Gated Recurrent Unit (GRU) [9] is used.

3 FriendlyFoe

This paper shows, for the first time, that on-the-fly AML can be a generic, practical, and effective architectural technique to obfuscate side channel signals. Our approach, called *FriendlyFoe*, uses a DNN called FriendlyFoe *Defender* to obfuscate a signal coming out of an architecture side channel. Such signal is composed of a set of samples $\{x_1, x_2, \ldots\}$. Successful obfuscation involves adding a perturbation (i.e., noise) p to each of the samples such that, with the resulting signal $\{x_1 + p_1, x_2 + p_2, ...\}$, even advanced ML classifiers that have been trained using noisy signals are unable to extract information from the signal.

FriendlyFoe is a general technique. It can be applied to a side channel if the side channel leaks information through a signal measurable over time, an attacker applies pattern recognition to decode the secret in the signal, and there exists a method for the Defender to actuate on the signal. In this paper, we demonstrate FriendlyFoe in two side channels: memory contention and system power. There are many other side channels that could be used, such as network traffic that enables website fingerprinting [8] or PCIe latency that allows keystroke detection [51].

3.1 How to Apply FriendlyFoe

FriendlyFoe can be applied in several ways. Table 1 lists some of the attributes that determine how to apply FriendlyFoe, and whether they are affected by the side channel, the victim application, or the system constraints. We consider each in turn.

Table 1: Attributes that affect how to apply FriendlyFoe.

Attribute		Affected by		
		Side Channel	Victim Appl.	System
General	Signal recorded	✔		
	Actuation on signal	✔		✔
	Area, power, latency			✔
Signal	Sampling period	✔		
	Length (in # samples)	✔	✔	
Other	Actuation grain size	✔		✔
	Actuation timing	✔		
	Number of classes	✔	✔	

General Environment. Important attributes that determine how to apply FriendlyFoe are the type of recorded signal, the actuation capability to perturb the signal, and the hardware cost allowed for the FriendlyFoe module (area, power, latency).

Signals. Two attributes of the signal that affect a FriendlyFoe Defender design are the sampling period (the time between each sample), and the signal length in number of samples. The sampling period is determined by the side channel, and impacts the expected response time of the Defender. Typically, the Defender takes the set of most recent samples and generates the noise to add to a few upcoming samples before the attacker can observe the next sample. Consequently, if the period is short, the Defender has to make decisions quickly. The signal length is a characteristic of the victim application and the side channel. It determines the capacity of the Defender: long signals often require large DNNs to manage the long-term pattern, increasing the hardware cost.

Other Attributes. As shown in Table 1, there are other attributes that determine how to apply FriendlyFoe. For example, one may have to sub-sample the signal, adding noise to only one out of N samples. We call this approach *coarse-grain* actuation. Additionally, if responding before the next sample is outright impossible, one may generate noise to be applied to a future sample, delaying the actuation. We call this attribute actuation timing. Finally, some signals may carry a binary secret, while other signals carry a multi-label secret.

3.2 Defender Architecture and Training

We want a Defender that minimizes both the information leakage and the perturbation level on the side channel signal. Our general design for the Defender is shown in Figure 2. It has two modules: a Gaussian noise shaper that takes as inputs the mean μ and standard deviation σ, and a neural network that periodically generates new μ and σ pairs for the noise shaper. Given a μ and σ pair, the noise shaper generates N samples from the resulting Gaussian distribution that will become the next N samples of the *Target* signal.

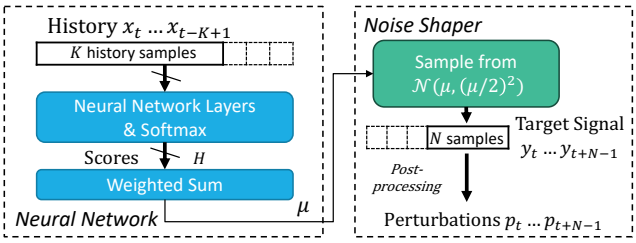


Figure 2: FriendlyFoe Defender architecture.

The noise shaper generates target signal values and not just perturbation values because generating perturbations using a Gaussian distribution leads to a signal that is easily compromised. Also, this target signal is not the final signal. If y_t is the value of the target signal at time t and x_t is the measured value of the system at time t, then a post-processing step computes the perturbation value p_t that must be added to the measured signal value to obtain the final signal value as as $p_t = y_t - x_t$. If a side channel signal cannot allow negative perturbations (e.g., a timing side channel), only positive or zero perturbations are added to the measured signal value.

After the N samples are generated, the neural network determines a new μ and σ pair. In our experiments, we set $\sigma = \mu/2$ because we empirically find that it is hard for neural networks to learn to generate good σ values. The likely reason for this is that the effect of the standard deviation is zero on average. Therefore, it becomes hard to learn good values of the standard deviation in large-batch DNN training. Overall, the neural network only learns

to generate sequences of μ values that minimize both the information leakage and the perturbation level of the side channel signal.

To pick a μ value, the neural network takes the history of K past samples of the signal. This vector is passed through a series of hidden neural network layers, followed by a softmax layer. Assume that the softmax layer has H outputs $w_1 \ldots w_H$. The neural network computes μ as $A \sum_{i=1}^{H} (\frac{i}{H} w_i)$, where A is a configurable hyperparameter to adjust the amplitude. The μ value obtained from the neural network is then applied to the noise shaper to generate the next N target signals. This process repeats every N samples.

The designer chooses the values of the subsampling window size N, the number of past samples read K, and the number of hidden neurons H to meet the appropriate area, power, and latency constraints. A smaller N enables finer control of the noise pattern, allowing the pattern to change more frequently. Increasing H increases the capacity of the model to handle more patterns. A higher K gives more visibility to the network to protect long-term patterns. However, these changes also increase the compute costs.

We recommend setting N first, based on the relative speed of the Defender module and the side channel signal. For example, if the Defender module is implemented on an FPGA, whose frequency is lower than the frequency of the memory system we are trying to defend, then N is adjusted to be at least the ratio of the two frequencies. Then, the designer can configure H and K to match the target implementation cost. Finally, to choose neural network layers, we recommend simple feed-forward layers when the signal patterns are short and we desire a low hardware cost. For long patterns, we may need to use RNNs [20, 9], 1-D CNNs [33], or attention-based Transformers [53].

To train the Defender, we propose the GAN structure shown in Figure 1c, inspired by Adversarial GAN [56]. In this design, the generator is the Defender and the classifier is the attacker. The Defender adds a perturbation to a signal and passes the noisy signal to the discriminator and classifier networks. The classifier tries to make a correct guess. The discriminator tries to estimate the Wasserstein distances [2] between the different label distributions, acting as another type of classifier. The networks are trained adversarially: the Defender is trained to perturb signal so that the classifier produces a wrong label; the classifier is trained to identify the correct label; and the discriminator is trained to recognize the difference between the different labels. The classifier and discriminator can be any type of DNN. They can be large and sophisticated, as they are used only for the training phase; only the Defender is deployed to the system.

3.3 Resisting Adaptive Attacks

The Defender is trained using a particular classifier and input data set. The operation is shown in Figure 3a, where classifier and data set are called *Classifier 1* and *Train 1*, respectively. However, the Defender must also be effective against a variety of other classifiers that the attacker may use at runtime. Specifically, the attacker may use a *deployed* Defender to train a potentially stronger classifier (*Classifier 2*) with a new input data set (*Train 2*) (Figure 3b). The attacker's goal is to become more effective against the Defender. Then, the attacker will use the trained *Classifier 2* to attack deployment runs with real (i.e., *Test*) input data sets and perform accurate predictions of the secret data (Figure 3c).

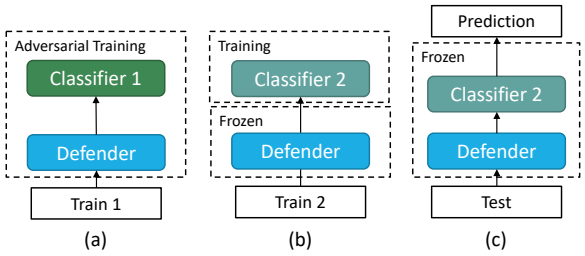


Figure 3: Training and deploying a FriendlyFoe Defender.

A Defender needs to be effective against classifiers which it has not been trained for. This ability is called *transferability*. In Figure 3, while the Defender has been trained only against Classifier 1, it must be effective against Classifier 2. For example, Classifier 1 may be a CNN and Classifier 2 a newly-trained CNN, or a different ML model such as an RNN or a Support Vector Machine (SVM) [55].

4 Two Target Side Channel Attacks

To show the effectiveness of FriendlyFoe, we examine attacks using two different side channels: memory contention and system power. Their characteristics are shown in Table 2. In this section, we outline the two use cases and, in Section 5, describe our proposed FriendlyFoe workflow, emphasizing the differences between the two use cases.

Table 2: Two FriendlyFoe use cases.

Attribute	Use Case 1	Use Case 2
Side channel	Memory contentn	System power
Perturbation	Introduce stalls	Add compute threads & idle time. Change freq.
Victim application	Crypto	PARSEC
Implementation	Hardware	Software
Time between samples	A few ns	20 ms
Signal length	28–75 samples	500 samples
Actuation timing	Now	Later
Classes	Binary	Multi-label

4.1 Threat Model

Memory-contention side channel. We instantiate an attack based on the work of Paccagnella et al. [34]. We consider a chip multiprocessor where the victim runs on one core and the attacker on another. The victim executes code whose memory accesses depend on secret bits; the attacker repeatedly accesses the main memory and measures the latency of its own accesses. The latency observed by the attacker's accesses is affected by main-memory contention induced by the victim. Depending on the pattern of memory access latencies observed, the attacker can deduce the secret bits.

A good target for this attack is the loop inside a crypto application where, in each iteration, the memory accesses depend on one bit of an encryption key. Following the work of Paccagnella et al., we assume that the administrator has configured the system to clean the victim's cache footprint on context switches, and that the attacker can interrupt the victim using preemptive scheduling techniques. In this case, one can time the attacker to interrupt the victim at the beginning of each iteration. Then, we simulate cache cleaning by calling `clflush` on the victim's cache lines, which evicts them from all cache levels. Then, as the victim executes the loop iteration with a clean cache and is forced to make main-memory accesses, the attacker accesses main memory and measures the load latency. With this design, the attacker can deduce the value of the bit in the iteration from the memory access latency measurements.

In our implementation, we run two crypto applications: the RSA [47] and EDDSA [4] algorithms. The trace of memory access latencies seen by the attacker during one iteration is the side channel signal, and the secret bit of the iteration is the target label. This is a binary classification problem.

We do not consider cache attacks. Caches can be secured by existing methods. For example, the last-level cache (LLC) can be partitioned into different security domains, so that the attacker cannot observe the victim's LLC access. Also, simultaneous multithreading can be disabled, so that private caches are not shared by multiple threads. Main memory remains shared and vulnerable.

Application power side channel. This attack is based on the work of Pothukuchi et al. [43]. It considers a chip multiprocessor running one of several victim applications. The attacker periodically measures the chip power consumption using the Intel RAPL interface [35]. Based on the measurements over time, the attacker tries to deduce which application is running. We run PARSEC 3.0 [5] applications. We assume the attacker knows when a new application starts its execution.

4.2 Defender for Memory Side Channel

The envisioned Defender is in a hardware unit in the memory controller (MC) that records the time when a load arrives at the MC and the time when the main memory produces the requested data. To obfuscate the latency of the attacker load access, the Defender may stall the returning data for a certain time period. We assume that every main-memory load transaction must pass through the MC and that its access latency is accurately measured in cycles.

It would appear that the time between samples is the time between loads from a core arriving at the MC. However, since there may be multiple attacker cores, the Defender has to process every single load arriving at the MC—irrespective of the source core. As a result, as shown in Table 2, the time between samples can be a few ns. The signal length is 28–75 samples per iteration of the algorithm. The Defender dynamically generates a delay to be added to the current memory access—not to a future one.

4.3 Defender for Power Side Channel

The envisioned Defender is a software process running in privileged mode in one or more cores. We assume that the OS and hardware power and performance measurements are all trusted. The Defender measures the power consumption using RAPL, and spawns compute-intensive threads, adds idle time, or changes the frequency to distort power. The set-up is based on the one described in Maya [43]. Maya used a randomized mask generator to distort power; in this paper, we replace it with a FriendlyFoe Defender.

As shown in Table 2, the sampling period is 20 ms, the default period of Maya. For the applications considered (Section 6), the signal length is about 500 samples. Because the Defender makes a decision only after measuring the current power, it can only affect future power samples. Finally, in this attack, the classifier picks one of multiple labels, which corresponds to the application run.

5 FriendlyFoe Workflow

In this section, we describe the workflow for FriendlyFoe, which we apply to the two use cases.

5.1 Designing the Defender Network

We design the Defender network based on the architecture of Section 3.2. For the memory side channel, we assume an FPGA implementation. We set the subsampling window N to 8, as determined by the speed of FPGA DNN accelerators (200Mhz [58]) and the frequency of DDR4-3200 memory (1600Mhz). We set the history length K and the hidden neuron size H so that the area and power of the Defender are less than 1% of the area and TDP, respectively, of the multicore chip. The actual values used are K=16 and H=16. The Defender's perturbation values must be positive numbers of cycles to add to the latency of memory accesses. Hence, negative values become zero. Note that we could use a custom circuit to implement a Defender with a smaller N and a higher frequency. However, such a Defender would consume more power and area.

The Defender computes new target samples at every $N = 8$ memory controller accesses. To support this actuation frequency, the Defender is implemented in hardware and has relatively few hidden neurons. To reduce the hardware cost, we quantize the network parameters (weights and biases) to 16-bit floating point (FP16) numbers. While 8-bit integer (INT8) is the standard for quantization, we find that an FP16 model with fewer neurons has better security than an INT8 model with more neurons. Further, our Defender has two layers of 16 neurons each. As a result, it has 528 parameters in total: 512 for the two 16×16 weight matrices of the two layers, and 16 for the bias addition in between.

The Defender for the power side channel measures and actuates on power every 20 ms. Hence, we use $N = 1$ because we have plenty of time between samples. In addition, since the response time is not very critical, we implement it in software and use $K = 32$ and $H = 64$. This makes the network compute latency less than 1 ms, which is small compared to the 20ms sampling period. The outputs can be positive or negative numbers, which correspond to increasing or decreasing the power consumption, respectively. This Defender does not need any quantization since the latency is already low even with an FP32 software implementation. It has 6208 parameters in total: 6144 for 32×64 and 64×64 weight matrices, and 64 for the bias addition in between.

A summary of the Defender configurations is shown in Table 3. The implementations are detailed in Sections 5.4.1 and 5.4.2.

Table 3: Defender configurations for two side channels.

Parameter	Memory contention side channel	Application power side channel
Window size N	8	1
History length K	16	32
Hidden neurons H	16	64
Parameter precision	FP16	FP32
Neural layer type	Feed-forward	Feed-forward
Parameter count	528	6208

5.2 Training the Defender

Once we have designed the Defender and collected labeled signals from the side channel, we train the Defender using a GAN network. As shown in Figure 1c, training is performed with three networks: the Defender, the classifier, and the discriminator.

Figure 4 shows how we train the networks. The Defender takes K samples of history as input and produces the perturbation values to be applied to N future samples. Specifically, at step t, the Defender reads the most recent K samples $\{x_{t-K+1}, \ldots, x_t\}$ and produces perturbations for the next window, which has N samples, i.e., $p_t, \ldots, p_{t+N-1}$. At the beginning of the next signal window, i.e., at time $t+N$, the Defender is fed K samples once more $\{x_{t+N-K+1}$, ..., $x_{t+N}\}$ and produces perturbations for the next window after that, i.e., $p_{t+N}, \ldots, p_{t+2N-1}$. After processing all the samples in x in this way, the entire signal and generated perturbations are added—eliminating negative perturbations in the memory side channel—and passed to the classifier and discriminator. Then, the classifier and discriminator produce their predictions, and the three networks compute their loss functions and proceed to backpropagation. Then, the whole process repeats for another input signal.

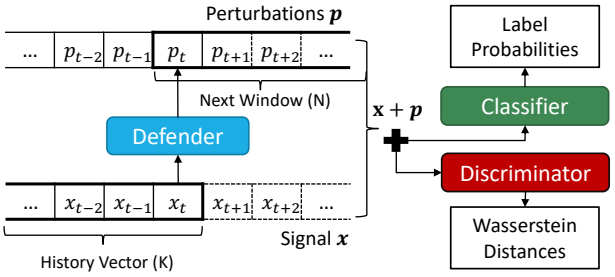


Figure 4: Training the networks in the GAN structure.

The three networks are adversarially trained simultaneously, with different loss functions. The classifier is trained by the cross-entropy loss between its prediction probabilities and the true label. The discriminator is optimized by Wasserstein loss [2], which tries to maximize the Wasserstein distances between distributions.

The Defender is trained with three losses: (1) the Kullback-Leibler divergence [28] between the classifier output probabilities and the uniform probability, (2) the negated Wasserstein loss of the discriminator, and (3) the L2-norm of the perturbation. The first loss optimizes the Defender to confuse the classifier, while the second loss optimizes it for perturbed outputs that are indistinguishable to the discriminator. Finally, the L2-norm loss minimizes the perturbation level to lower the overhead caused by the Defender's perturbations. The loss can be hinged [15] by a constant to adjust the allowed perturbation level.

5.3 Assessing Defender Transferability

Recall from Section 3.3 that, to be useful, a Defender trained with a certain classifier must be transferable to other classifiers. Consequently, after we train a Defender, we select various types of DNN networks for the classifier and, for each of them, perform the two steps shown in Figures 3b and c. Then, we report the highest classification accuracy attained by any classifier. This is the worst case for the Defender.

Theoretically, it is impossible to verify the minimal security guarantees provided by a Defender because one cannot evaluate it against every possible classifier. However, this transferability analysis for a variety of classifier architectures often provides empirical assurance that the Defender is transferable.

5.4 Architecting the Overall System

While the previous steps of the workflow are common for different uses of FriendlyFoe, the next step, which involves architecting the whole system, is use-specific. Depending on the side channel addressed, applications used, or systems targeted, the implementation is different. For this reason, in this section, we consider the implementations of our two FriendlyFoe examples separately.

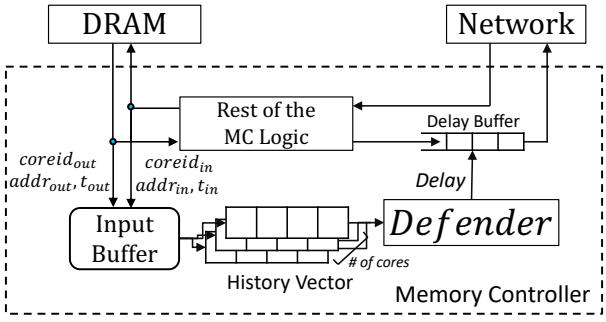


Figure 5: Defender for the memory contention side channel.

5.4.1 Defender for the Memory Contention Side Channel. We implement the Defender in hardware in the memory controller (MC). As shown in Figure 5, when a load request arriving at the MC from the network is sent to the DRAM module, the hardware stores its core ID ($coreid_{in}$), its address ($addr_{in}$), and the current time (t_{in}) in an Input Buffer. When the memory returns a response at t_{out} with an address ($addr_{out}$) and core ID ($coreid_{out}$), two operations occur in parallel: 1) the response is deposited in a Delay Buffer where it is delayed by a precomputed perturbation value, and 2) the response checks the Input Buffer for a matching entry and, on finding it, the hardware computes $\delta = t_{out} - t_{in}$ and pushes δ to a K-entry FIFO History Vector for that core. Every time that the History Vector for a core has taken in N entries, it sends its K entries to the Defender,

which computes the perturbation values $(p_1, \ldots, p_N)$ to delay the next N responses to that core. From then on, the next N responses to that core arriving at the Delay Buffer are delayed by $p_1, \ldots, p_N$. During this time, new entries continue to be pushed into the History Vector. As before, after N insertions, the History Vector sends its K entries to the Defender, starting the process again.

Most of the hardware cost of the Defender comes from computing matrix multiplications. Since the Defender has two layers of 16 neurons each, it needs hardware for 512 FP16 FMAs (Fused Multiply Adds). We estimate their cost based on Johnson's analysis [23] for TSMC 28nm, scaled down [49] to 7nm. Assuming a single MC with dual-channel DDR4-3200 memory, we conservatively estimate that our added hardware consumes a maximum dynamic power of 0.11W and uses an area of $0.4mm^2$. These are less than 1% of the TDP (105W) and area ($74mm^2$) of a contemporary 7nm desktop processor like the AMD 5800x [52]. We estimate that the critical path of the Defender can be pipelined in about ten stages: two stages for each of the two feed-forward layers (one for an FMA and one for a bias addition), three stages for softmax, and three stages for the final perturbation computation. In each stage, many operations are performed in parallel. Given an MC frequency of 1.6GHz, the Defender operation takes less than 7ns. Most of this latency can be overlapped with the memory controller's routing and flow-control logic—shown in Figure 5 as "rest of the MC logic". Since the Defender can process requests at the same rate as the MC, the memory throughput remains the same.

5.4.2 Defender for the System Power Side Channel. We implement the Defender in software as a privileged process running on one of the cores. We build on the framework of Maya [43], a technique used to obfuscate power signals. Figure 6a shows the Maya framework. Maya consists of two parts. The Mask Generator creates randomized target power shapes; the Robust Controller uses control theory techniques to control the system inputs (e.g., frequency) so that the system produces the target power shapes. As shown in Figure 6b, our Defender is plugged into the Maya framework, by replacing a Mask Generator with a FriendlyFoe Defender.

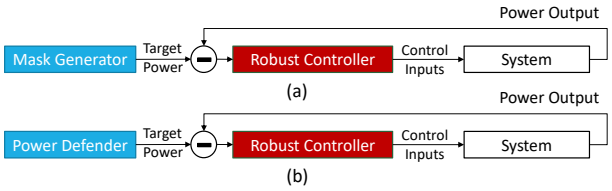


Figure 6: Maya [43] framework (a) and FriendlyFoe power side channel Defender plugged into the same framework (b).

5.5 Defending Other Victim Applications

A Defender is often effective for other victim applications beyond those for which it has been trained. We evaluate this case in Section 7.1.3.

However, in the general case, defending other types of victim applications may require the system designer to collect data for the new applications, train a Defender using the data, and deploy the new Defender. Note that the Defender architecture does not change; only the weight and bias parameters need to be updated based on the applications. Therefore, as we change the victim, we re-use the Defender module and re-program its parameters.

We can support this form of reconfigurability by placing the Defender parameters in registers or special memories, and providing a way for the software to update them across victim applications. For the system power side channel Defender, we can easily update the parameters since we deploy it via software. On the other hand, for the memory side channel Defender, we need a software-hardware interface to update the parameters.

This interface to reconfigure the Defender can use model-specific registers (MSR), microcode, or firmware. For example, if an MSR interface is used, the software can write to an MSR the ID of the parameter it wants to change and its new value. For our Defender, one MSR is sufficient to perform a full reconfiguration. This is because the Defender has 528 parameters (weights and biases) as shown in Table 3, and each of them is an FP16 value. Hence, the MSR takes 10 bits to specify the parameter ID and 16 bits to specify the new value. If one MSR write takes about 1μs, reprogramming all the parameters of the Defender takes less than 1 ms.

With this support, one can also disable the Defender, if needed, by writing zeros to all parameters. Thus, to selectively invoke the Defender only for a security-critical application, one can write the application-specific Defender parameters at the beginning of the application and clear them at program exit.

6 Experimental Methodology

To evaluate FriendlyFoe for our two side channels, we proceed in two steps. First, we run the victim and the attacker on a real computer, collect traces, and train a FriendlyFoe Defender. Second, we evaluate the security provided by the FriendlyFoe Defender and its execution and/or power overhead.

For the memory contention side channel Defender, we evaluate the security provided by applying the perturbations to the traces; we evaluate its performance overhead on benign applications through simulations of the architecture. Since we deploy the power side channel Defender on a real system, we evaluate its security and overheads with real system measurements.

6.1 Memory Contention Side Channel

We base our attack code on the ring side channel code of Paccagnella et al. [34]. We collect signal traces from a desktop with an Intel i5-7400 processor when running one victim crypto application and an attacker that repeatedly measures its memory access latency. We have two victim applications: the RSA decryption algorithm in *libgcrypt* 1.5.2, and the EDDSA algorithm in the *libgcrypt* 1.6.3 [24].

The execution traces consist of the latencies of the attacker accesses. The measurements are segmented into multiple signals. Each signal corresponds to one loop iteration of the victim application, which processes a single secret bit. Each signal contains 28 and 75 samples for RSA and EDDSA, respectively. The sample values are normalized by subtracting the median and dividing by the standard deviation of all the samples in all the signals. Each signal is labeled with the secret bit it corresponds to. For each application, we collect 12000 signals and split them into a 6:3:1 ratio to create Train 1, Train

2, and Test datasets (Figure 3). We use PyTorch [37] to train and evaluate the different ML networks.

We then apply four obfuscation methods: FriendlyFoe Defender, *Gaussian*, *Gaussian Sinusoid*, and *Padding to Constant*. The last three are techniques from previous work [43]. They add perturbations to create target signals of gaussian shape, gaussian + sinusoidal shape, and constant value, respectively. Recall that when the Defender produces a negative perturbation for a sample, we set it to zero.

We also evaluate whether the defense is robust against interference from concurrent threads. Hence, we conduct experiments in noisy environments for both attack and defense. Specifically, while collecting execution traces, we run random PARSEC [5] applications on the two remaining cores, filling up the four cores of i5-7400. The Defender is trained using traces from clean environments, but it is tested on the traces collected from this noisy environment.

We evaluate the performance impact of the FriendlyFoe Defender on benign applications in two environments: single- and multi-threaded. For this, we configure the Gem5 [31] simulator to model the hardware that generated the signals and run SPEC 2017 applications [6]. We compare the execution overhead induced by the FriendlyFoe Defender, Padding to Constant, and the state-of-the-art DAGguise defense [12].

To model the DAGguise defense, we mostly follow DAGguise's setup. Specifically, in single-threaded evaluations, we run one of DAGguise's victim applications (DocDist) and one SPEC application, for a total of two busy cores. In multi-threaded evaluations, we run four DAGguise victims (2 DocDist and 2 DNA) with four SPEC applications, for a total of 8 busy cores. We use this setup because the DAGguise defense is highly specific to the victim application that is running. Then, for fairness, we must also run the same DAGguise applications when evaluating the FriendlyFoe Defender. Since the Defender is not entirely specific to the victim and, instead, has transferability to defend other victim applications, we deploy the Defender trained to protect the RSA victim. In our experiments, we run the simulations up to one billion instructions per thread and compare the performance of the SPEC application threads.

6.2 System Power Side Channel

For this attack, we use the code base from Maya [42]. We collect power traces on a desktop with an Intel Xeon W-2245 processor using RAPL [35]. As victim applications, we use 6 PARSEC (blackscholes, bodytrack, canneal, freqmine, vips, streamcluster) and 4 Splash2x (radiosity, volrend, water_nsquared, and water_spatial) applications from the PARSEC 3.0 suite [5] with simlarge inputs.

The traces of the victim applications collected in an unprotected environment are the Train 1 dataset (Figure 3), which we use to train the FriendlyFoe Defender. Train 1 has 6000 signals. Then, we collect traces with two defense environments: a *Gaussian Sinusoid* mask generator from Maya [43], and FriendlyFoe Defender. We collect 4000 signals for each defense method. These signals are split 3:1 to the Train 2 and Test datasets. All signals are 500-samples long to record the longest workload (i.e., 10 seconds). The power values are normalized by subtracting the idle power (30W) and dividing by the power range (160-30=130W). Each signal is labeled with the label of its dataset. Hence, the attack becomes a 10-label classification problem. The networks are trained using PyTorch and deployed to the same machine using the Maya code base and the C++ API of PyTorch.

To evaluate inter-application transferability, we split the 10 applications into two sets: the 6 original PARSEC and the 4 Splash2x applications. We train one Defender on one set and the other on the other set, and evaluate whether the Defenders can protect applications on which they have not been trained.

6.3 Machine Learning Models

The neural network in the Defender consists of a 2-layer MLP, a softmax layer, and a weighted sum (Figure 2). For the classifier in Figure 4, we use a 16-layer 1-D CNN, which has the best attack accuracy. For the discriminator, we use an MLP, producing one real number for binary discrimination and N numbers for the N-class cases. We consider seven classifiers to evaluate network transferability (i.e., Classifier 2 in Figure 3). Five of them are DNNs implemented using PyTorch: the original 16-layer 1-D CNN (CNN), a similar CNN with 25 layers (CNN-D for deep), a 16-layer CNN with double hidden neurons (CNN-W for wide), a GRU-based network (RNN), and an attention-based network (Att). The other two are non-DNNs implemented with scikit-learn [38]: an SVM and a K-Nearest Neighbor (KNN).

7 Evaluation

Since the two side channels are different, we evaluate each in turn.

7.1 Memory Contention Side Channel Defense

7.1.1 Security Provided. We take the unperturbed *Test* signals (Figure 3c) for the RSA and EDDSA victims, perturb them with our defenses, and then use the 7 different classifiers described in Section 6.3 (trained with Train 2) to see if they can break the defenses. We consider the 4 defenses of Section 6.1 (FriendlyFoe, Gaussian, Gaussian Sinusoid and Padding to Constant) and, for each, vary the level of noise—i.e., the average added memory latency. This is accomplished by adjusting the amplitude A of the noise generators.

Figure 7 shows the information leakage in bits for RSA and EDDSA with different types of defenses and levels of noise—showing only the best attack results of the 7 classifiers. Information leakage is the mutual information [48] between the secret bits and the attacker's predictions. If one defense has M times less leakage than another, the attacker must do M times more repetitions to obtain the same information as for the other. As the attack tries to retrieve a single bit, the leakage is 1 bit when the accuracy is 100%, and the leakage is zero when the accuracy is 50%.

Consider RSA (Figure 7 left). The simplest defense, Pad to Constant, is effective only when the average noise added is large (120-160 cycles). For fewer cycles added to the memory requests, e.g., 40 to 110 cycles, Pad to Constant leaks more information than FriendlyFoe. For example, for an information leakage of 0.0026 bits, Pad to Constant induces about 38% more latency than FriendlyFoe (76 cycles for FriendlyFoe vs 105 cycles for Pad to Constant). Alternatively, for a fixed extra latency overhead of 76 cycles, Pad to Constant leaks 8.2x more information (0.0214 bits) than FriendlyFoe (0.0026 bits). The randomized defenses (Gaussian and Gaussian Sinusoid) are also ineffective. The main reason is that part of their

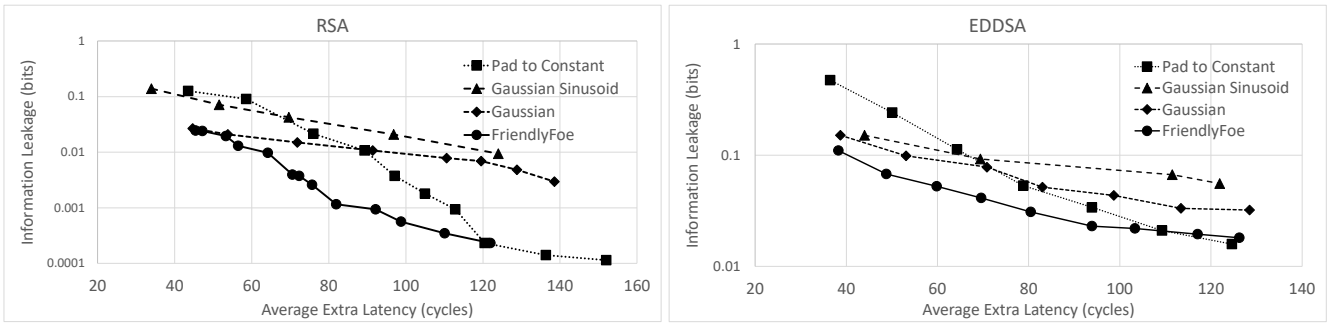


Figure 7: Trade-off between security and overhead of the defense methods for RSA (Left) and EDDSA (Right). The x-axis is the average extra latency a defense adds, while the y-axis is the information leakage with the best attacker in bits.

perturbation samples end up being zero because the target sample values were lower than the measurement values. On the other hand, the neural network in the FriendlyFoe Defender dynamically adjusts the noise level to reduce both the information leakage and the execution overhead. Hence, it is the best design.

Next, consider EDDSA (Figure 7 right). Compared to RSA, the leakage is higher in all cases. This is expected, as the EDDSA signals are longer than the RSA signals (75 vs 28 samples), delivering more information per signal. Nonetheless, FriendlyFoe remains a better choice over other defenses. For example, for an average extra latency of 80 cycles, the leakage with the FriendlyFoe Defender is 58% of that with the next best scheme.

7.1.2 Transferability to Other Classifiers. Our FriendlyFoe Defender was trained with a CNN classifier and an MLP discriminator. We now assess the transferability of the Defender to the other classifiers. Table 4 shows the classification accuracy of the 7 different attacker classifiers of Section 6.3 for both the RSA and EDDSA victims. Recall that, in binary classification, an accuracy of 0.5 is zero leakage, while 1.0 is full 1-bit leakage. The highest accuracies are in bold, which represent the worst-case information leakage with the Defender.

Table 4: Classification accuracy of 7 attacker classifiers as they attack our Defender trained with the CNN classifier.

Victim Appl.	Classifier Accuracy						
	Att	RNN	CNN	CNN-D	CNN-W	SVM	KNN
RSA	0.508	0.505	0.533	0.530	**0.537**	0.519	0.506
EDDSA	0.560	0.549	**0.603**	0.601	0.598	0.501	0.499

The range of accuracy numbers in the table is 0.499-0.603. Compared to the CNN classifier, the other networks do not show higher classification accuracy, even though the Defender was not trained with them. This means that the perturbations generated after training our FriendlyFoe Defender with a CNN classifier are transferable to various other types of ML attackers. We also see that CNNs with more layers (CNN-D) or neurons (CNN-W) do not show noticeably higher accuracy than the original CNN. This observation suggests that using a bigger DNN is unlikely to break the defense.

7.1.3 Different Victim Application. In this section, we examine the *inter-application* transferability of the FriendlyFoe Defender—i.e., the security of one victim when the Defender is trained for another victim. We consider Defenders trained for the RSA or the EDDSA victims. The resulting Defenders have the same model structure; they only differ in the neural network weights and biases.

Table 5 shows the attacker accuracies of the most effective classifiers against our FriendlyFoe Defender trained on one victim application (*Train*) but used to defend another victim (*Target*). We also show the average added cycles per sample. The Defenders are configured to have similar noise levels by setting the same noise amplitude A.

Table 5: Attacker accuracies against a FriendlyFoe Defender trained on one victim (Train) but used to defend another victim (Target), along with their average added cycles.

Target Victim	Train Victim			
	RSA		EDDSA	
	Attack Accur.	Added Cycles	Attack Accur.	Added Cycles
RSA	0.537	76.0	0.542	77.3
EDDSA	0.603	81.9	0.603	80.5

In the table, we compare the data within the same row. For example, for the EDDSA row, we target EDDSA with either a Defender trained with RSA or with EDDSA. We see that, in both cases, the accuracy is the same (0.603). Further, the number of cycles only increases by 1.4 as we go from an EDDSA-trained Defender to an RSA-trained one. When targeting RSA, the attacker accuracy only increases slightly from 0.537 to 0.542 as we go from an RSA-trained Defender to an EDDSA-trained Defender, and the number of cycles only increases by 1.3. Therefore, for these applications, the FriendlyFoe Defender shows inter-application transferability. One can avoid the effort of collecting data and re-training the Defender for the new application. Note that inter-application transferability is not guaranteed and it may not apply to some victim applications. In such cases, we must re-train a new Defender and deploy the Defender using the methods discussed in Section 5.5.

7.1.4 Effect of Environmental Noise. Since our target side channel is shared among all cores, other applications may inject noise to the channel. This noise alters the shapes of the signals observed by both attackers and the Defender. Table 6 shows the attacker accuracies against the FriendlyFoe Defender trained on a clean environment and used to defend a victim in a clean or noisy environment. The average added cycles is also shown. The noise is injected by PARSEC applications in the background (see Section 6.1).

Table 6: Attacker accuracies against a FriendlyFoe Defender trained on a clean environment and used to defend a clean or noisy environment, along with the average added cycles.

Target Victim	Target Environment			
	Clean		Noisy	
	Attack Accur.	Added Cycles	Attack Accur.	Added Cycles
RSA	0.537	76.0	0.533	79.8
EDDSA	0.603	80.5	0.539	81.6

We see that the attacker accuracy is lower in the noisy environment than in the clean one. The reason is that interference provides extra obfuscation to block information leakage. The number of added cycles in the noisy environment is only slightly higher.

7.1.5 Performance Overhead. We evaluate the performance overhead of FriendlyFoe on benign applications in two environments discussed in Section 6.1: single- and multi-threaded workloads. We compare three defenses: FriendlyFoe, DAGguise [12], and Pad to Constant. Due to space limitations, we only show the RSA evaluation; the EDDSA evaluation shows similar results. Recall from Table 5 that, with FriendlyFoe, the attacker accuracy is 0.537 and the added cycles are 76.0. We configure Pad to Constant to have a similar level of leakage as the FriendlyFoe Defender. It was discussed in Section 7.1.1 that this corresponds to a design that adds 105 extra cycles. Indeed, in Figure 7, the configuration in the FriendlyFoe Defender line with 76.0 cycles and the one in the Pad to Constant line with 105 cycles are at the same Y coordinate. The DAGguise configuration is the default one in their code base [11].

Figure 8 shows the execution overhead of the defense schemes on benign single-threaded workloads, namely running one SPEC application (plus the victim application). In this environment, minimizing memory latency is important for performance. DAGguise adds memory contention, which only indirectly increases the latency of the benign application. FriendlyFoe and, especially, Pad to Constant, directly add latency to the benign application. For these reasons, on average, DAGguise only adds 6.4% overhead, while FriendlyFoe and Pad to Constant add 8.1% and 11.1%, respectively.

Figure 9 shows the execution overhead of the defenses on benign multi-threaded workloads. In this case, there are four instances of the same SPEC application running (plus four victims). In this environment, the memory throughput is critical to their performance. FriendlyFoe and Pad to Constant do not add extra memory contention. However, DAGguise's fake memory requests introduce substantial traffic, which results in memory contention and higher latencies. On average, FriendlyFoe only adds 2.7% overhead, while Pad to Constant and DAGguise add 7.4% and 20.7%, respectively.

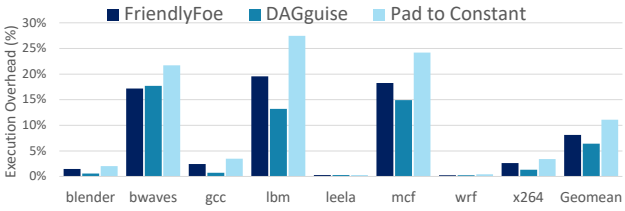


Figure 8: Execution overhead of defense schemes on benign applications in a single-threaded environment.

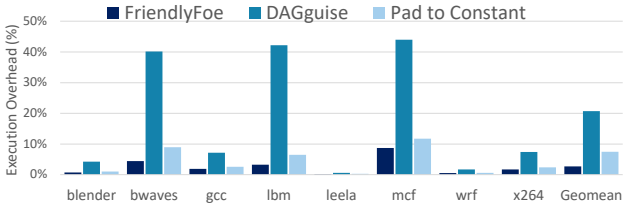


Figure 9: Execution overhead of defense schemes on benign applications in a multi-threaded environment.

Overall, FriendlyFoe is the most competitive scheme: it has the lowest overhead in multi-threaded environments (which are the most common ones) and modest overhead in single-threaded ones. Compared to Pad to constant, its execution overhead is 27% and 64% lower for single- and multi-threaded workloads, respectively.

7.2 Power Side Channel Defense

7.2.1 Security and Power. To evaluate the security versus power tradeoff, we compare three mask generators under the framework of Figure 6: None (i.e., no mask generator and therefore no defense), Maya (i.e., the state of the art gaussian sinusoid mask proposed in the Maya paper [43]), and FriendlyFoe Defender. For each of them, Table 7 shows the accuracy of different classifiers as they classify 10 PARSEC 3.0 applications. Since there are 10 applications, an attack accuracy of 10% corresponds to a perfect defense. The table only shows a few representative classifiers, including an MLP, which is used in the Maya paper [43], and a CNN, which consistently shows the highest accuracy. The table also shows the average application power consumption and the average application energy ($power \times time_{exec}$). The latter is relative to None, to compare the efficiency of the systems.

Table 7: Accuracy of different classifiers for 10 PARSEC 3.0 applications under three different defense schemes. The table also shows the application power and relative energy.

Defense	Classifier Accuracy			Avg. Appl.	Rel. Appl.
	SVM	MLP	CNN	Power (W)	Energy
None	0.729	0.653	0.917	85.9	1
Maya	0.113	0.114	0.235	95.4	1.89
FriendlyFoe	0.105	0.113	0.169	91.7	1.69

Table 7 shows that FriendlyFoe provides better security than the state-of-the-art approach represented by Maya. Although Maya is effective against the simpler ML networks (SVM and MLP), it is less effective against the more powerful CNN model. This is because Maya's gaussian sinusoid simply adds random noise and does not take advantage of the application's dynamic behavior. In contrast, our FriendlyFoe Defender shapes the noise based on the application's behavior to maximize obfuscation.

Compared to Maya, and against the strongest attacker (i.e., CNN), which is the one that matters, FriendlyFoe reduces the attacker accuracy from 0.235 to 0.169. Converting accuracy to information leakage, it can be shown that this corresponds to leaking 0.11 and 0.03 bits per observation, respectively. Consequently, FriendlyFoe reduces the leakage of Maya by 3.7×. Since this is a 10-class classification problem, each label has $log_2(10) = 3.32$ bits of information. If we assume that the attacker can force repeated re-execution of the victim, the attacker under Maya requires 3.32/0.11 = 30.2 repeated observations on average to retrieve the application label. The attacker under FriendlyFoe requires 3.32/0.03 = 110.7 observations on average. In many cases, however, the attacker cannot force repeated re-execution of the victim.

The controller in both FriendlyFoe and Maya (Figure 6) distorts the system power by modulating CPU frequencies, injecting CPU idle times, and adding dummy compute threads. However, compared to Maya, FriendlyFoe is optimized to minimize the distortion of the power signal while still obfuscating it. Less distortion implies less inefficiency added to the system. As a result, applications consume less energy with FriendlyFoe than with Maya. This is shown in Table 7, where FriendlyFoe and Maya consume 69% and 89% more energy than None, respectively. In other words, FriendlyFoe reduces the energy overhead of Maya by 22.5%.

The impact of FriendlyFoe on the applications' average power and average execution time relative to Maya depends on application behavior. The data, however, shows that, on average, applications consume less average power and take less time to execute with FriendlyFoe than with Maya. Table 7 shows that FriendlyFoe and Maya increase the average power consumption over None by 5.8W and 9.5W, respectively.

Figure 10 shows the execution time of the applications when the system is defended with FriendlyFoe and with Maya, relative to the system without protection. We see that, while there is variation across applications, applications tend to have longer execution times with Maya than with FriendlyFoe. On average and compared to None, applications take 70% longer with Maya and 58% longer with FriendlyFoe.

7.2.2 Inter-application Transferability. To verify if our FriendlyFoe Defender is transferable to other victim applications, we split the 10 PARSEC 3.0 applications in two sets (6 original PARSEC and 4 Splash2x) and train two Defenders separately on the two sets. Then, we measure the CNN attacker accuracy when targeting one set using the Defender trained with either the same set or the other set. Table 8 shows the attacker accuracies. Comparing the numbers within the same row, we see that the attacker accuracy on a set of applications increases only slightly if the Defender has been trained on another set of applications. It can be shown that, in both cases, the application power differences are very small. Consequently, we conclude that our defense has inter-application transferability.

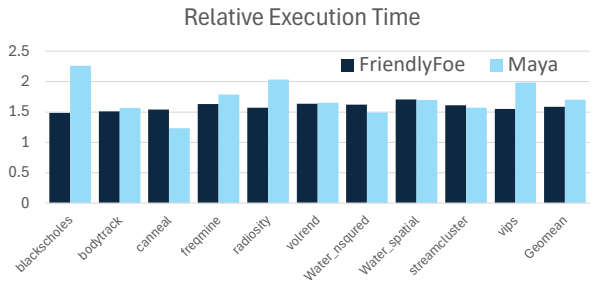


Figure 10: PARSEC 3.0 application execution times on a system defended with FriendlyFoe and Maya, relative to the system without protection.

Table 8: CNN attacker accuracies against a FriendlyFoe Defender trained on one set of applications (Train) and used to defend either the same set or another set (Target).

Target	Train Victim	
Victim	PARSEC	Splash2x
PARSEC	0.191	0.195
Splash2x	0.289	0.279

8 Related Work

There are two prior works that use AML as a countermeasure against ML analysis of signals [41, 44]. These schemes need to know the full trace of a signal to produce the perturbations for the signal. Such a trace is only available after the system execution. By the time these schemes start post-processing the signal, the information has already leaked to an attacker. To protect real systems, we need a dynamic defense like our proposed scheme, which uses AML to produce the perturbations on the fly.

In addition, these prior works have limitations that our technique solves. Specifically, Picek et al. [41] showed that adding small adversarial modifications can mislead ML-based power side channel attackers. However, they only use classifiers trained with *unprotected* signals, which is unrealistic. Our methodology targets attackers that are trained with signals *already perturbed* by the Defender. The work by Rahman et al. [44] showed the use of AML against website fingerprinting attacks through traffic measurement. Their defense is not very effective, in that attackers can still achieve 38–42% accuracy for a 100-class classification.

There are two works where ML is used not to obfuscate a signal, but to help other techniques to do so. Gu et al. [18] designed a countermeasure against a power side channel attack by combining the one-pixel attack [50] with the insertion of noise instructions by the compiler. This is a compiler-based method that requires access to the source code of the victim application, while ours does not. Rijsdijk et al. [46] applied reinforcement learning (RL) to help counter ML-based attacks. The work assumes multiple existing

countermeasures and uses RL to choose a right method combination dynamically. Our method directly obfuscates the signal and does not depend on multiple existing solutions.

There are many non-ML techniques to obfuscate information in particular side channels. Two relevant ones are Maya [43] and DAGguise [12]. Maya [43] re-shapes power signals into gaussian sinusoid signals with the help of control theory; our approach has been shown to improve Maya by replacing the signal generator.

DAGguise [12] is a defense technique that re-shapes the main memory accesses of a victim to obfuscate its behavior. Specifically, it may delay memory responses to the victim or create extra memory accesses for the victim. FriendlyFoe is different in multiple ways. First and foremost, FriendlyFoe is a general defense for various side channels (including power), while DAGguise in applicable only to contention side channels. Second, FriendlyFoe uses ML methods, while DAGguise does not. Third, focusing on the memory contention side channel, FriendlyFoe distorts the attacker's access latencies (and therefore its measurements); DAGguise distorts the victim's access patterns by delaying the victim's accesses and adding fake requests. Finally, as shown in Section 7.1.5, FriendlyFoe incurs substantially less performance overhead than DAGguise in multi-threaded workloads and only slightly more than DAGguise in the less common single-threaded workloads.

9 Conclusion and Future Work

This paper showed the practicality, efficacy, and generality of on-the-fly AML as an architectural defense to obfuscate signals from architectural side channels. We called our approach FriendlyFoe. We showed a workflow to design, implement, train, and deploy transferable FriendlyFoe Defenders for different environments. We successfully applied FriendlyFoe to thwart two very different side channel attacks: one based on memory contention and one on system power. We are now studying the use of FriendlyFoe for defending against other attacks such as PCIe contention and frequency side channel attacks.

Acknowledgments

This work was supported in part by NSF under grants CNS 1956007 and CCF 2107470; by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA; and by a gift from Intel.

References

[1] Onur Aciiçmez. 2007. Yet another microarchitectural attack: Exploiting I-cache. In *Proceedings of the 2007 ACM workshop on Computer security architecture*, 11–18.

[2] Martin Arjovsky, Soumith Chintala, and Léon Bottou. 2017. Wasserstein generative adversarial networks. In *International conference on machine learning*. PMLR, 214–223.

[3] Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli, and Cécile Dumas. 2018. Study of deep learning techniques for side-channel analysis and introduction to ASCAD database. *ANSSI, France & CEA, LETI, MINATEC Campus, France*, 22, 2018.

[4] Daniel J Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, and Bo-Yin Yang. 2012. High-speed high-security signatures. *Journal of cryptographic engineering*, 2, 2, 77–89.

[5] Christian Bienia, Sanjeev Kumar, Jaswinder Pal Singh, and Kai Li. 2008. The PARSEC Benchmark Suite: Characterization and Architectural Implications. In *International Conference on Parallel Architectures and Compilation Techniques*. (Oct. 2008).

[6] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. 2018. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, 41–42.

[7] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. 2017. Convolutional neural networks with data augmentation against jitter-based countermeasures. In *International Conference on Cryptographic Hardware and Embedded Systems*. Springer, 45–68.

[8] Giovanni Cherubin, Rob Jansen, and Carmela Troncoso. 2022. Online website fingerprinting: Evaluating website fingerprinting attacks on Tor in the real world. In *31st USENIX Security Symposium (USENIX Security 22)*, 753–770.

[9] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1724–1734.

[10] Debayan Das, Anupam Golder, Josef Danial, Santosh Ghosh, Arijit Raychowdhury, and Shreyas Sen. 2019. X-DeepSCA: Cross-Device Deep Learning Side Channel Attack. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 1–6.

[11] Peter W Deutsch. 2022. Github repository for DAGguise. https://github.com/CSAIL-Arch-Sec/DAGguise.

[12] Peter W. Deutsch, Yuheng Yang, Thomas Bourgeat, Jules Drean, Joel S. Emer, and Mengjia Yan. 2022. DAGguise: Mitigating Memory Timing Side Channels. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. Association for Computing Machinery, New York, NY, USA, 329–343. ISBN: 9781450392051. https://doi.org/10.1145/3503222.3507747.

[13] Dmitry Evtyushkin, Ryan Riley, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2018. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems* (ASPLOS '18). Association for Computing Machinery, Williamsburg, VA, USA, 693–707. ISBN: 9781450349116. DOI: 10.1145/3173162.3173204.

[14] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. 2016. ECDSA key extraction from mobile devices via nonintrusive physical side channels. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 1626–1638.

[15] Claudio Gentile and Manfred KK Warmuth. 1998. Linear hinge loss and average margin. *Advances in neural information processing systems*, 11.

[16] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2014. Generative adversarial nets. *Advances in neural information processing systems*, 27.

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

[18] Ruizhe Gu, Ping Wang, Mengce Zheng, Honggang Hu, and Nenghai Yu. 2020. Adversarial attack based countermeasures against deep learning side-channel attacks. *arXiv preprint arXiv:2009.10568*.

[19] Benjamin Hettwer, Stefan Gehrer, and Tim Güneysu. 2019. Applications of machine learning techniques in side-channel attacks: A survey. *Journal of Cryptographic Engineering*, 1–28.

[20] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation*, 9, 8, 1735–1780.

[21] Gabriel Hospodar, Benedikt Gierlichs, Elke De Mulder, Ingrid Verbauwhede, and Joos Vandewalle. 2011. Machine learning in side-channel analysis: A first study. *Journal of Cryptographic Engineering*, 1, 4, 293.

[22] Sunghyun Jin, Suhri Kim, HeeSeok Kim, and Seokhie Hong. 2020. Recent advances in deep learning-based side-channel analysis. *ETRI Journal*, 42, 2, 292–304.

[23] Jeff Johnson. 2018. Rethinking floating point for deep learning. *NIPS Systems for ML Workshop*.

[24] Werner Koch and Moritz Schulte. 2005. The libgcrypt reference manual. *Free Software Foundation Inc*, 1–47.

[25] Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. 2011. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1, 1, (Apr. 2011), 5–27. DOI: 10.1007/s13389-011-0006-y.

[26] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25, 1097–1105.

[27] Takaya Kubota, Kota Yoshida, Mitsuru Shiozaki, and Takeshi Fujino. 2020. Deep learning side-channel attack against hardware implementations of AES. *Microprocessors and Microsystems*, 103383.

[28] Solomon Kullback and Richard A Leibler. 1951. On information and sufficiency. *The annals of mathematical statistics*, 22, 1, 79–86.

[29] Liran Lerman, Romain Poussier, Gianluca Bontempi, Olivier Markowitch, and François-Xavier Standaert. 2015. Template attacks vs. machine learning revisited (and the curse of dimensionality in side-channel analysis). In *International Workshop on Constructive Side-Channel Analysis and Secure Design*. Springer, 20–33.

[30] Pavel Lifshits, Roni Forte, Yedid Hoshen, Matt Halpern, Manuel Philipose, Mohit Tiwari, and Mark Silberstein. 2018. Power to peep-all: Inference attacks by malicious batteries on mobile devices. *Proceedings on Privacy Enhancing Technologies*.

[31] Jason Lowe-Power et al. 2020. The gem5 simulator: version 20.0+. *arXiv preprint arXiv:2007.03152*.

[32] Houssem Maghrebi, Thibault Portigliatti, and Emmanuel Prouff. 2016. Breaking cryptographic implementations using deep learning techniques. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer, 3–26.

[33] Aaron van den Oord, Sander Dieleman, Heiga Zen, Karen Simonyan, Oriol Vinyals, Alex Graves, Nal Kalchbrenner, Andrew Senior, and Koray Kavukcuoglu. 2016. Wavenet: A generative model for raw audio. *arXiv preprint arXiv:1609.03499*.

[34] Riccardo Paccagnella, Licheng Luo, and Christopher W Fletcher. 2021. Lord of the Ring (s): Side Channel Attacks on the CPU On-Chip Ring Interconnect Are Practical. *USENIX Security Symposium*.

[35] Srinivas Pandruvada. 2014. Running Average Power Limit – RAPL. Retrieved June, 2014 from https://01.org/blogs/2014/running-average-power-limit--rapl.

[36] Nicolas Papernot, Patrick McDaniel, and Ian Goodfellow. 2016. Transferability in machine learning: From phenomena to black-box attacks using adversarial samples. *arXiv preprint arXiv:1605.07277*.

[37] Adam Paszke et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32.

[38] Fabian Pedregosa et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research*, 12, 2825–2830.

[39] Guilherme Perin, Baris Ege, and Jasper van Woudenberg. 2018. Lowering the bar: Deep learning for side channel analysis. *BlackHat USA, Las Vegas, NV, USA, Tech. Rep.*

[40] Stjepan Picek, Annelie Heuser, Alan Jovic, Simone A Ludwig, Sylvain Guilley, Domagoj Jakobovic, and Nele Mentens. 2017. Side-channel analysis and machine learning: A practical perspective. In *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 4095–4102.

[41] Stjepan Picek, Dirmanto Jap, and Shivam Bhasin. 2019. Poster: When Adversary Becomes the Guardian–Towards Side-channel Security With Adversarial Attacks. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2673–2675.

[42] Raghavendra Pradyumna Pothukuchi. 2021. Github repository for Maya: Obfuscating Power Side Channels with Formal Control. https://github.com/maya defense/maya.

[43] Raghavendra Pradyumna Pothukuchi, Sweta Yamini Pothukuchi, Petros G Voulgaris, Alexander Schwing, and Josep Torrellas. 2021. Maya: Using formal control to obfuscate power side channels. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 888–901.

[44] Mohammad Saidur Rahman, Mohsen Imani, Nate Mathews, and Matthew Wright. 2020. Mockingbird: Defending Against Deep-Learning-Based Website Fingerprinting Attacks With Adversarial Traces. *IEEE Transactions on Information Forensics and Security*, PP, (Nov. 2020), 1–1. DOI: 10.1109/TIFS.2020.3039691.

[45] Keyvan Ramezanpour, Paul Ampadu, and William Diehl. 2020. SCAUL: Power side-channel analysis with unsupervised learning. *IEEE Transactions on Computers*, 69, 11, 1626–1638.

[46] Jorai Rijsdijk, Lichao Wu, and Guilherme Perin. 2021. Reinforcement learning-based design of side-channel countermeasures. In *International Conference on Security, Privacy, and Applied Cryptography Engineering*. Springer, 168–187.

[47] Ronald L Rivest, Adi Shamir, and Leonard Adleman. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21, 2, 120–126.

[48] Claude E Shannon. 1948. A mathematical theory of communication. *The Bell system technical journal*, 27, 3, 379–423.

[49] Aaron Stillmaker and Bevan Baas. 2017. Scaling equations for the accurate prediction of CMOS device performance from 180 nm to 7 nm. *Integration*, 58, 74–81.

[50] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. 2019. One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*.

[51] Mingtian Tan, Junpeng Wan, Zhe Zhou, and Zhou Li. 2021. Invisible probe: Timing attacks with PCIe congestion side-channel. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 322–338.

[52] TechPowerUp. 2020. AMD Ryzen 7 5800X. https://www.techpowerup.com/cpu-specs/ryzen-7-5800x.c2362.

[53] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

[54] R. Vinaykumar, KP. Soman, M. Alazab, S. Sriram, and K. Simran. 2020. A comprehensive tutorial and survey of applications of deep learning for cyber security. *TechRxiv*. DOI: 10.36227/techrxiv.11473377.v1.

[55] Lipo Wang. 2005. *Support vector machines: theory and applications*. Vol. 177. Springer Science & Business Media.

[56] Chaowei Xiao, Bo Li, Jun-Yan Zhu, Warren He, Mingyan Liu, and Dawn Song. 2018. Generating adversarial examples with adversarial networks. *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*.

[57] Tianwei Zhang, Yinqian Zhang, and Ruby B. Lee. 2018. Analyzing cache side channels using deep neural networks. In *Proceedings of the 34th Annual Computer Security Applications Conference, ACSAC 2018, San Juan, PR, USA, December 03-07, 2018*. ACM, 174–186. DOI: 10.1145/3274694.3274715.

[58] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. 2018. DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, 1–8.

[59] Yinqian Zhang. 2017. Cache Side Channels: State of the Art and Research Opportunities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (CCS '17). Association for Computing Machinery, Dallas, Texas, USA, 2617–2619. ISBN: 9781450349468. DOI: 10.1145/3133956.3136064.

[60] Yanqi Zhou, Sameer Wagh, Prateek Mittal, and David Wentzlaff. 2017. Camouflage: Memory Traffic Shaping to Mitigate Timing Attacks. In *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 337–348. DOI: 10.1109/HPCA.2017.36.