Automated Detection of Spectre and Meltdown Attacks using Explainable Machine Learning

Zhixin Pan and Prabhat Mishra
Department of Computer & Information Science & Engineering
University of Florida, Gainesville, Florida, USA

***Abstract*—Spectre and Meltdown attacks exploit security vulnerabilities of advanced architectural features to access inherently concealed memory data without authorization. Existing defense mechanisms have three major drawbacks: (i) they can be fooled by obfuscation techniques, (ii) the lack of transparency severely limits their applicability, and (iii) it can introduce unacceptable performance degradation. In this paper, we propose a novel detection scheme based on explainable machine learning to address these fundamental challenges. Specifically, this paper makes three important contributions. (1) Our work is the first attempt in applying explainable machine learning for Spectre and Meltdown attack detection. (2) Our proposed method utilizes the temporal differences of hardware events in sequential timestamps instead of overall statistics, which contributes to the robustness of ML models against evasive attacks. (3) Extensive experimental evaluation demonstrates that our approach can significantly improve detection efficiency (38.4% on average) compared to state-of-the-art techniques.**

***Index Terms*—Spectre, Meltdown, Hardware Security, Explainable Machine Learning, Side-Channel Analysis**

I. Introduction

Processing speed of computing devices has been significantly boosted by speculative execution properties such as branch prediction and out-of-order execution. As depicted in Figure 1, processors are able to perform parallel processing of predicted tasks with excess system resources by utilizing speculative execution. Unfortunately, these performance enhancement techniques introduces security vulnerabilities that are exploited by Spectre and Meltdown attacks. Spectre attack [1] can successfully break devices' memory isolation capabilities by abusing the 'branch prediction' capability. Similarly, Meltdown attack [2] exploits the vulnerability arising from the 'out-of-order execution' feature. Both attacks enable a malicious process to access memory locations without authorization. According to the study in [2], the Meltdown attack is able to dump kernel memory at a speed of 503 KB/s! Therefore, it is critical to detect Spectre and Meltdown attacks in order to enable trustworthy computing.

This work was partially supported by the NSF grant CCF-1908131.

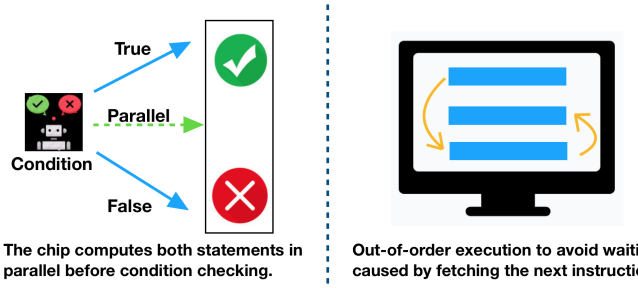


Fig. 1: Typical strategies applied in speculative execution

There are a large number of existing efforts for defending against Spectre and Meltdown attacks. Existing approaches focus on mitigation techniques [3]–[6], such as enforcing the processor to empty branch target buffer during task switching, or occasionally shutting down speculative execution. However, these techniques can lead to unacceptable performance degradation. A recent approach utilizes machine learning for detection of Spectre and Meltdown attacks [7]. While it provides promising results, it has two inherent weaknesses. (1) It cannot detect Spectre and Meltdown attacks in the presence of obfuscation techniques or other deviation capabilities [8]. (2) Due to the black-box nature of machine learning models, the results cannot be interpreted in a meaningful way.

A. Threat Model

In this paper, we make the following assumptions that are consistent with the related efforts in Section II.

Goals: The attacker's goal is to reveal values in concealed memory locations to cause information leakage.

Knowledge: We assume that the adversary has sophisticated knowledge in both Spectre and Meltdown vulnerabilities. For the target device, we assume that the adversary possess information about the operating

system and hardware architecture. We also assume that the attacker is aware of specific patterns that are utilized by existing detection techniques.

Attack Modes: The adversary will attack the target device with well-crafted Spectre/Meltdown attack codes. The attack starts with raising exceptions during program execution, which is followed by a side-channel attack to achieve the adversary's goal. We assume that the adversary is able to measure the system's reaction time towards memory fetching operations. We also assume that the evasive Spectre and evasive Meltdown attacks can exploit the knowledge of specific patterns used by detection techniques to devise obfuscation methods.

B. Motivation

There are two major problems that affect the performance of existing detection efforts [7], [9]–[12]: high overhead and poor robustness.

High Overhead: Passive prevention of Spectre attacks [9], [10] relies on selectively turning off speculative execution to prevent possible attacks, which inevitably leads to significant reduction in performance. The study in [13] highlights that software patches to mitigate Meltdown (such as KPTI [14]) can introduce unacceptable overhead. Similarly, architectural alteration [11], [12] increases the burden on the pipeline. Moreover, it takes considerable time for the detection to complete before normal execution can proceed.

Poor Robustness: Recent efforts rely on hardware-based detection methods due to their low latency compared to software-based solutions. A majority of these approaches utilize *Hardware Performance Counter* (HPC) values. HPCs are components in microprocessors to monitor hardware events, such as cache misses and branch misprediction. Since Spectre and Meltdown attacks rely on triggering exception and cache access measurements, they leave noticeable traces in HPCs. An ML classifier would be able to detect malicious attacks by observing specific patterns in HPC values. However, this type of detection is vulnerable towards obfuscation techniques [8]. The basic idea is to invoke benign functions between malignant payloads and inserting instructions that increase the specific HPC values (e.g., number of branch mispredictions) to fool the detectors. Results in [8] revealed that the state-of-the-art classifier can provide less than 60% detection rate, which is comparable to a random guess. Our proposed approach effectively fulfils these requirements as outlined in Section III.

C. Major Contributions

In this paper, we propose a hardware-assisted Spectre and Meltdown detection framework that takes advantage of explainable machine learning. Specifically, this paper makes the following four important contributions.

1) To the best of our knowledge, our approach is the first attempt in hardware-assisted Spectre and Meltdown detection using explainable machine learning.
2) We provide theoretical analysis on the close relationship between hardware events and inherent features of Spectre and Meltdown attacks, which enables the attack detection with high credibility and robustness against obfuscation.
3) We utilize hardware events as time sequential inputs to mitigate the misprediction induced by obfuscation techniques, making the trained machine learning model resistant against evasive attacks.
4) Experimental results demonstrate that our proposed approach can provide significant improvement in detection accuracy and robustness compared to the state-of-the-art methods.

The rest of this paper is organized as follows. Section II surveys related efforts to motivate the need for this work. Section III describes our proposed attack detection technique. Section IV presents the experimental results. Finally, Section V concludes the paper.

II. Background and Related Work

Given the importance of trustworthy computing, there are many research efforts in efficient detection and mitigation of security vulnerabilities [15]–[29]. This section provides background on Spectre and Meltdown attacks. It also provides an overview of explainable machine learning.

A. Spectre and Meltdown Attacks

The operating system has one of the most fundamental security requirements – it must prevent user programs from accessing the memory locations of the kernel or any other programs. Once a user program tries to perform illegal access, the CPU will detect the permission violation during the execution, and throw an exception leading to the termination of current program. However, during this permission checking and scene clearing process, the information about accessing target is retained in the cache. These are inherent vulnerabilities in most of modern chips, which can be exploited by attackers to reveal kernel memory information. Specifically, a simple template of Meltdown attack code is shown in Listing 1.

```
mov rax byte[x]    // illegal access
shl rax 0xC        // page alignment
mov rcx rbx[rax]   // probe data
```

Listing 1: Example Meltdown Attack

In this example, 'byte[x]' is a private memory location, illegal access to this location shall raise exception during execution. Ideally, 'rax' should be cleared before executing the subsequent instructions. However, due to the speculative execution property, the second and third instructions will be partially executed before the exception handling takes effect. Also, according to the modern cache designs, if 'rax' is not in the cache, the CPU will bring it into the cache to hide the latency of subsequent accesses. Although 'rax' will be cleared by exception handling, the cache will not be flushed immediately. Therefore, the information of the latest illegal access is temporarily stored in the cache. An attacker can restore this address by a *cache-based side channel attack* as shown in Figure 2. This is achieved by simply traversing the entire array headed by 'rbx' and measuring the access time – the page with the shortest access time is the one addressed by 'rax', thereby this kernel value is obtained.

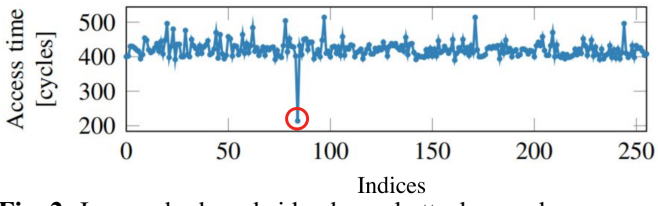


Fig. 2: In a cache-based side-channel attack, an adversary can identify the data corresponding to the index with the shortest access time since it is likely to be pre-stored in the cache [2].

Spectre attack is very similar but it exploits branch prediction instead, which is shown in Listing 2. Obviously, if index 'x' is out of range, the second line should not be executed. Due to branch prediction scheme, it will still be "pre-executed". Once this pre-execution occurs, it will inevitably leave traces in the cache, where the same cache-based side-channel attack can be used. Compared to Meltdown, Spectre is more dangerous since it has a wider attack range [1].

```
if(x < arr1_size);      //boundary check
y = arr2[arr1[x]*4096]; //array access
```

Listing 2: Example Spectre Attack

B. Explainable Machine Learning

Machine Learning (ML) has shown its potential in security domain for tasks like malware detection and post-sillicon validation [30], which makes it a promising choice for detecting Spectre and Meltdown attacks. Due to the black-box nature of ML models, no additional information apart from detection result is available. Most importantly, security practitioners gain no clue from incorrect predictions. This lack of transparency hinders its adoption in many safety-critical domains. Moreover, to detect evasive attacks, the ML model needs to reveal why false negatives evade the detection. We address this challenge by a data augmentation method utilizing explainable machine learning.

In general, explainable ML seeks to provide interpretable explanation for the results of ML model. Specifically, given an input instance $\mathbf{x}$ and a traditional ML model, the classifier will generate a corresponding output $\mathbf{y}$ for $\mathbf{x}$ during the testing time. Explanation techniques then aim to illustrate why instance $\mathbf{x}$ is classified into $\mathbf{y}$. This often involves identifying a set of important features inside $\mathbf{x}$ that make key contributions to the classification result. If the selected features are interpretable by human analysts, these features can offer an "explanation".

Gradient-based method [31] is a successful attempt in enabling interpretable machine learning. It computes an image-specific class *saliency map* corresponding to the gradient of output neurons. *Integrated Gradients* [32] is a variation of saliency map where integral methods are adopted to improve the information acquisition. Various explainable ML techniques [33]–[35] utilize 'deconvolution' concept to inverse and visualize the features learning in *convolution neural networks* (CNNs). DeepLIFT [36] is another popular algorithm that was designed to observe the activation effects of each neuron, and assign contribution scores to each of them.

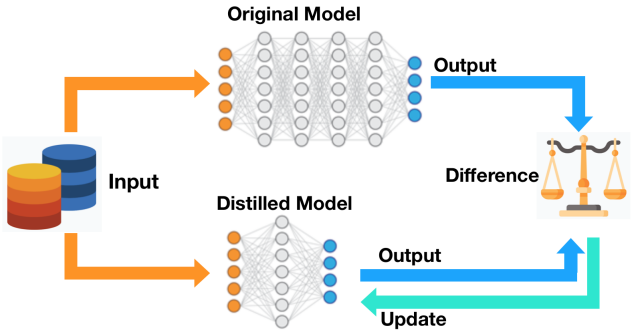


Fig. 3: The goal of model distillation is to minimize the difference of input-output mapping behaviors. Interpreting the distilled model can provide insights into the internal representation to explain how it makes a decision.

We apply "model distillation" [37] method as the EML technique. The basic idea of model distillation is that it develops a separate model called as "distilled model" to be an approximation of the input-output behavior of the target machine learning model. This distilled model

is inherently explainable, which helps a user to identify the decision rules or input features influencing the final decision, as depicted in Figure 3. Therefore, lightweight structures are preferred, such as linear regression [38], decision tree [39] or object graph [40].

Explainability is an important property for our proposed work since it provides two major advantages compared to traditional ML-based detection of Spectre and Meltdown attacks.

- Our proposed method uses explainability to help identify critical features. In other words, only a small set of crafted samples satisfies the need for adversarial training. In the absence of explainability, one would need extensive training with a large number of samples that can be expensive in terms of both space and time. For example, Figure 8 shows how explainability helps in crafting synthesized samples with similar patterns to evasive samples.
- Explainability also helps in interpreting the prediction in a human understandable way that can be used to handle incorrect classification results.

Due to the importance of explainability in security analysis, explainable machine learning has received significant attention in recent years. However, the existing approaches face two fundamental challenges:

- Existing approaches consider input data that are discrete values. In security domain, we need to handle input data that are time-sequential records.
- Existing approaches focus on computer vision tasks using CNN. However, CNN model is not suitable for security applications with time-sequential data.

There are recent attempts in applying explainable ML for malware detection [41]. To the best of our knowledge, our proposed approach is the first attempt in applying explainable machine learning for detecting Spectre and Meltdown attacks.

III. Hardware-assisted Detection of Spectre and Meltdown Attacks

This section is organized as follows. First, we provide an overview of our approach that consists of four major tasks. Next, we describe these four tasks in detail.

A. Overview

Figure 4 shows an overview of our proposed detection mechanism using explainable machine learning that satisfies the following two requirements.

- *Design Overhead:* Efficient utilization of hardware features with minimal impact on overhead.
- *Detection Robustness:* Effective countermeasures to protect against obfuscation techniques.

Our proposed detection framework is an iterative process comprising four major tasks.

Data Collection: We first run both benign and malicious programs to collect hardware event records using HPCs (Section III-B). These records are considered as the initial sample pool. Next, we select several important events as critical features to be fed into a machine learning training process.

Model Training: Our ML model structure is based on *recurrent neural network*. The model is trained with *stochastic gradient descent*. Section III-B provides details on model training using selected features.

Result Interpretation: The trained model is tested through sufficient number of test samples to produce classification results. These results are utilized by *result interpretation* task as outlined in Section III-D. This task provides crucial information regarding the input features that are most relevant (or misleading) for classification.

Adversarial Training: Based on the analysis obtained from result interpretation, we craft adversarial samples and mix them into the original pool of training samples to retrain the model (Section III-E). The model continuously improves itself until the convergence of the testing accuracy. This well-trained model is utilized for automatic attack detection.

B. Data Collection

The first step for training of ML model is to collect and determine the format of model inputs. According to the discussion in Section I-B, it is promising to utilize hardware events. Considering the fact that there are a wide variety of hardware events monitored by a large number of hardware performance counters (HPC), it is a major challenge to select a small set of HPCs that are beneficial for ML-based attack detection. For both type of attacks, we need to record the total number of instructions, which can provide system-wide information of the current process. The out-of-order memory lookup in Meltdown attack generates significantly high number of page faults which can be used as an effective indicator. For Spectre attack, due to its abuse of branch prediction property, we record the total number of branch instructions and mispredictions to compute branch miss rate. Moreover, since both attacks rely on a cache-based side channel attack, we collect the total number of low-level cache (LLC) references and misses to detect suspicious cache events. Based on these observations, we have selected six critical features as shown in Table I.

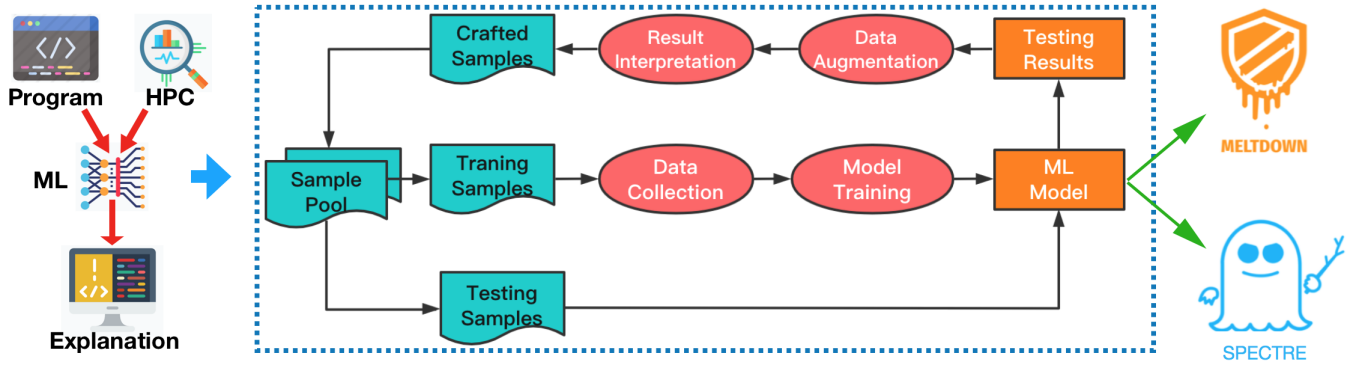
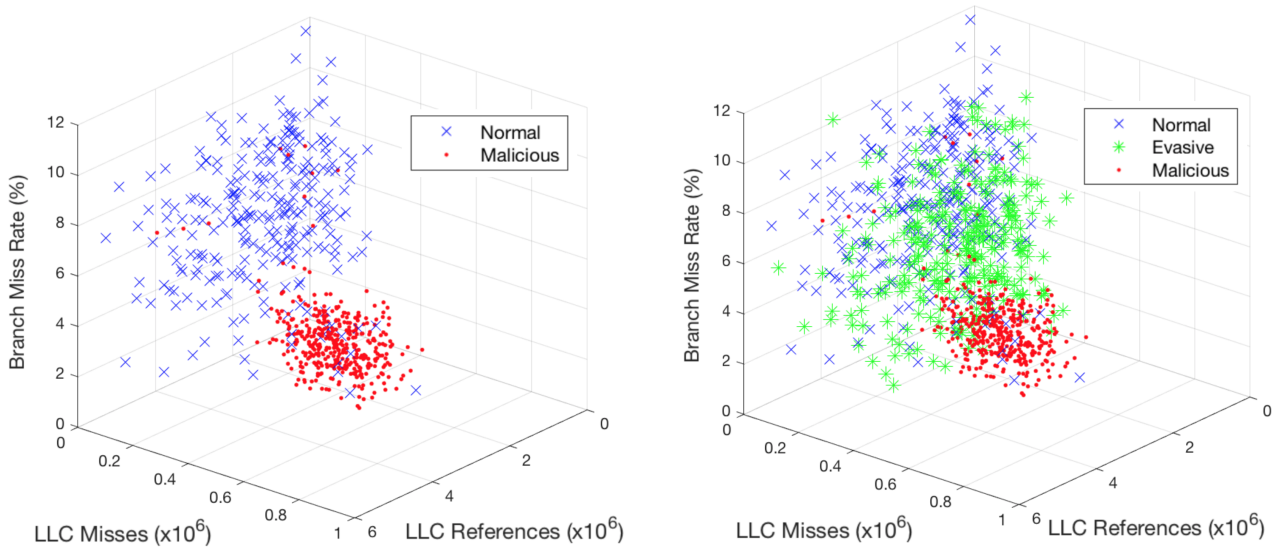


Fig. 4: Overview of our detection framework that utilizes hardware events to predict potential attacks. The training process of the ML model consists of four major activities: data collection, model training, result interpretation and data augmentation.



(a) Malicious vs Normal

(b) Malicious vs Evasive vs Normal

Fig. 5: Distribution of LLC references, LLC misses and branch miss rate for different types of samples.

TABLE I: Selected hardware performance counters for detection of Spectre and Meltdown attacks

Hardware events	Event ID	Spectre	Meltdown
Total number of instructions	INS	✓	✓
Total page faults	PGF	✗	✓
Total branch instruction	BRC	✓	✗
Branch Miss-Predictions	BMP	✓	✗
Low-Level cache references	LLCR	✓	✓
Low-Level cache misses	LLCM	✓	✓

A straightforward way of formatting these events would be crafting vectors composed of the above selected features. Interestingly, this naive strategy works in reality, and is adopted by state-of-the-art works [7], [42]. An illustrative example is shown in Figure 5a, where we plot the distribution of normal and malicious Spectre attack samples with LLC references, LLC misses and branch miss rate. As we can see, the cluster of malicious samples are clearly distinguishable from that of normal ones, both the regions and boundary between two classes are obvious. This observation validates that fact that the chosen features are indeed helpful in detecting attacks. However, this naive approach fails against evasive attacks. In evasive attacks, the adversary can simply add redundant non-profitable loops or cache-access statements, which enables the malicious program to mimic the pattern collected from benign programs, making the overall statistics indistinguishable from normal programs. This is depicted in Figure 5b, where the distribution of evasive Spectre samples are also plotted. This time the cluster of evasive samples mingled with normal ones, and there is no clear boundary to distinguish them. As demonstrated in Section IV, evasive attacks drastically reduces the performance of state-of-the-art methods.

To address this, two strategies were applied. First, the data format in our approach is designed as an *event-*

TABLE II: An example record of event trace differences

Events \ Time	x_0	x_1	x_2	x_3
Δ BMR	0.5	1.1	- 0.1	...
Δ LLCR	59	83	46	...
Δ LLCM	11	26	16	...
. . .				

tracing table as shown in Table II. The key idea is that instead of directly utilizing the overall statistics, we use HPC to sample hardware events in multiple timestamps and record their differences. Each row represents a specific selected hardware event, and we use 'Δ' symbol to denote that each entry represents the increase of corresponding event compared to previous timestamp. Since we consider the hardware events in sequential timestamps instead of overall statistics, a natural advantage of this strategy is that it grants the model with potential information concealed in consecutive adjacent inputs. The second strategy is applying *data augmentation* in ML training process, which will be discussed in Section III-E.

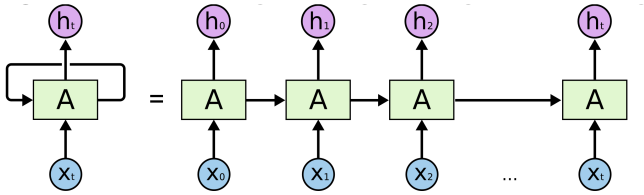


Fig. 6: Recurrent Neural Network (RNN)

C. Model Training

Since we are providing time-sequential data, the major structure of our model is selected as *Recurrent Neural Network (RNN)*. The neural network is represented by **A**, where $\boldsymbol{x}_0, \boldsymbol{x}_1, \boldsymbol{x}_2, \ldots$ means the time series inputs (columns of event-tracing table in our case), and h_is are the outputs of hidden layers. Instead of finishing the input-output mapping in one forward pass, the RNN accepts sequential inputs. For each single input $\boldsymbol{x}_i$, RNN not only provides immediate response h_i but also information corresponding to the previous step will be fed into the architecture to supply extra information. This mapping can be understood by unrolling the RNN structure as shown in Figure 6. To avoid the *vanishing gradient* and the *exploding gradient* problems detailed in Bengio et al. [43], we choose to implement RNN as *long-short term memory* (LSTM). LSTM is a special type of RNN that utilizes special units in addition to basic configuration. LSTM adopts a 'memory cell' that can maintain information in memory for long periods of time. At the same time, a set of gates is used to control the information flow inside LSTM's structure. This architecture lets them learn longer-term dependencies.

The LSTM works as an auto-encoder to map the original data to the outputs of hidden features, yet we still need an extra structure to play the role of mapping the learned "distributed feature representation" to a more sophisticated feature space. Therefore in our design, the output of the last layer of LSTM passes through a *multi-layer perceptron* (MLP) neural network.

Finally, the output of the MLP is normalized by a *softmax* layer to generate binary prediction labels, where a *cross_entropy* function is applied to produce the training loss. The gradient of loss is fed into *stochastic gradient descent (sgd)* method to update model parameters. The overall structure of the proposed method is outlined in Figure 7, and the training process is described in Algorithm 1.

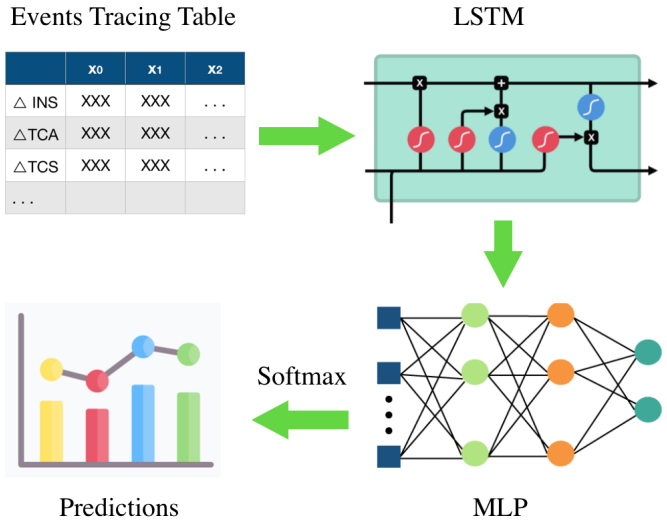


Fig. 7: The structure of the proposed ML model consists of two major components, LSTM and MLP. The output of the last hidden layer passes through softmax function to produce prediction labels for input samples.

D. Result Interpretation

The trained model from previous step can be directly applied for detection task, but it is still vulnerable towards obfuscation techniques as discussed in Section I-B. To address obfuscation challenge, we first apply explainable ML to interpret the detection outcome, which can be further utilized to synthesize evasive data samples. Next, these samples are merged into the pool of training set to retrain the ML model, thereby enhancing the robustness of model against known attacks. Intuitively, this process is similar to 'vaccine treatment'. It starts

with diagnosing the 'patient' to detect the pathogen, and the immunity towards a particular disease is enhanced by preventive vaccination.

To achieve result interpretation, we utilize model distillation technique which is composed of three major steps: model specification, model computation, and outcome explanation.

Algorithm 1: Model Training

Input : Model Inputs $\{\boldsymbol{x}_i\}$
Output: Trained Model $\boldsymbol{A}$

initialize($\boldsymbol{A}$)
$h_0 = \boldsymbol{A}(\boldsymbol{x}_0)$
repeat
  for $i = 1 \dots t$ **do**
    $h_i = \boldsymbol{A}(\boldsymbol{x}_i, h_{i-1})$
  res = softmax(h_t)
  $loss$ = cross_entropy($res, label$)
  $\boldsymbol{A}$ = sgd($\boldsymbol{A}, \nabla loss$)
until $converge$;
Return $\boldsymbol{A}$

Model Specification: First, the type of distilled model has to be specified. This often involves a trade-off between transparency and expression ability. A complex model can offer better performance in mimicking the behavior of the original model. However, increasing complexity also leads to the drop of model transparency, where the distilled model itself becomes hard to explain, and vice versa. In this task, due to demands for low-latency and high-transparency, we choose linear regression model for its simplicity and interpretability.

Model Computation: Once the type of distilled model (denoted by $\mathbf{A}^*$) is determined, the original model $\mathbf{A}$ has to be passed through test samples to produce sufficient number of input-output pairs. Next, the model computation task aims at searching for optimal parameters θ to minimize the difference between $\mathbf{A}$ and $\mathbf{A}^*$. Since we are applying linear regression, this task is in fact a least square problem and can be solved efficiently.

$$\theta = \arg\min_{\theta} \sum_{i=1}^{n} ||\mathbf{A}^*_\theta(\boldsymbol{x}_i) - \mathbf{A}(\boldsymbol{x}_i)||_2 \qquad (1)$$

where vector $\boldsymbol{x}_i = [x_{i1}, x_{i2}, ...]$ is the i-th input.

Outcome Explanation: Based on the computed distilled model, the explanation boils down to measuring the contribution of each input feature in producing the model output. In this case, linear regression model is applied which can always be expressed as a polynomial. Then by sorting the terms with amplitude of coefficients, we have access to information regarding the input features it found to be most discriminatory. For instance, assume we have $\mathbf{A}^*_\theta(\boldsymbol{x}_i) = a_1x_{i1} + a_2x_{i2} + a_3x_{i3} + ...$ after regression, then we sort the terms by absolute value of their coefficients. For example, if a_j is the largest coefficient in the term a_jx_{ij}, the most important contributor is x_{ij}.

In this task, by listing and sorting the contributions of the input features, the ML model is able to identify the most important elements of each input $\boldsymbol{x}_i$. In fact, it represents which entries of the i-th column from event tracing table are main contributors to the model output, i.e, the hardware events occurred in the i-th timestamp are considered by the ML model. This is crucial for model improvement especially in handling incorrect classification. It clearly points out the critical location that led to the misprediction. Next, we show how to craft synthesized samples based on these observations.

E. Data Augmentation

By interpreting the result, we obtain clues from incorrect predictions. The next step is improving the model to prevent similar mistakes in the future. This can be achieved by *data augmentation*, where new synthesized evasive samples are generated based on original malicious samples. The process of data synthesis consists of five major steps.

1) **Slicing:** With the help of explainable ML, it starts by marking and slicing codes corresponding to the most important timesteps from evasive samples that are incorrectly labeled by the classifier. These code slices are denoted as 'inducements'.
2) **Deleting:** For original sample, randomly delete non-important and irrelevant parts to prune the size.
3) **Padding:** Non-profitable code slices were randomly generated and augmented in the sample to mess-up the overall statistics.
4) **Inserting:** Bootstrapping method was applied to sample the pool of 'inducements' and to insert them into the target sample.
5) **Permuting:** Function blocks are permuted to reorder hardware events, making the fluctuation of HPC records different from original samples.

This process is demonstrated in Figure 8, and Algorithm 2 describes the entire data augmentation method along with result interpretation technique.

F. HPC Reliability Concerns

We have considered the pitfalls outlined in [44] since our proposed work is based on HPC values. Our pro-

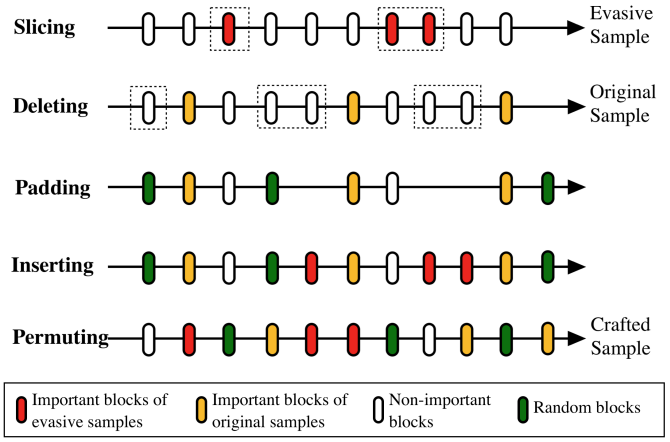


Fig. 8: Illustrative examples of data augmentation used in our work. Each row is a small fragment of the full execution log. Each box represents one basic function block. Crafting synthesized sample consists of five stages. First, important misleading activities were sliced from evasive attack codes. Next, non-important blocks from target sample are randomly deleted to prune the size, followed by padding of non-profitable blocks. Then, the misleading blocks are inserted into target sample. Finally, permutation of basic blocks completes the synthesis of new evasive samples.

Algorithm 2: Training with Data Augmentation

Input : Original model ($\mathbf{A}$), distilled model ($\mathbf{A}^*$), sample pool P, number of iterations (k)

Output: Optimized model, $\mathbf{A}'$

$i = 1$
repeat
  $P' = \{x \in P | \mathbf{A}(x) \neq lable(x)\}$
  $rank = sort(\mathbf{A}^*.\mathbf{coeff})$
  $P_{adv} = craft(P', rank)$ ▷ craft adversarial samples
  $P = P \cup P_{adv}$
  $\mathbf{A} = train(\mathbf{A}, P)$ ▷ retrain the model
  $\mathbf{A}^* = distill(\mathbf{A}, P)$
  i++
until $i \geq k$;
Return $\mathbf{A}$

posed algorithms have addressed the four pitfalls as follows.

- *External Sources:* The system status gets reset after each run to ensure that the measurements are independent across different runs.
- *Non-determinism:* To reduce the impact of non-determinism, we ran each program for 200 runs.
- Overcounting: We have selected important counters as shown in Table I. Figure 5 demonstrates that these counters avoid overcounting problem.
- *Implementation Variations:* Our proposed method is an online method which utilizes RNN to accept time-sequential data as input. Therefore, it avoids the data acquisition pitfall across runs.

IV. Experiments

This section demonstrates the effectiveness of our proposed framework for detection of Spectre and Meltdown attacks. First, we describe the experimental setup. Next, we present the detection results.

A. Experimental Setup

The experimental evaluation is performed on a host machine with Intel i7 3.70GHz CPU, 32 GB RAM and RTX 2080 256-bit GPU. We developed code using Python for model training. We used PyTorch as the machine learning library. To enable fair comparison with existing approaches, we deploy the experiments on the same benchmarks applied in [7] from SPEC integer benchmark [45]. During program execution, we extract performance counter values with 'perf' tool at a sampling rate of 100ms.

The machine learning model consists of a LSTM network and a MLP. The architecture of LSTM contains a one-hot encoding layer, a hidden layer with 32 nodes and a 50% dropout. The MLP is composed of 3 layers and 64 nodes. For input data, data samples consist of hardware performance counter values collected during the execution of both malicious (with implanted Spectre/Meltdown attack) and benign programs. The initial pool of evasive attack samples were manually crafted by the following obfuscation techniques introduced in [8]:

1) **Strategy 1:** Put attack into sleep between memory-flush.
2) **Strategy 2:** Insert redundant instructions for obfuscation.

The sampling happens at a rate of 100ms. For each program, the collected traces are further formatted into event-tracing table as illustrated in Table II. In terms of the size, our training data set contains data collected from 200 runs of malicious programs as well as 200 runs of benign programs. Average collected data size of each run is approximately 19.2 KB. Therefore, the total data size is 3840 (200 *19.2) KB for each program. We have considered a realistic scenario for data collection. The system status was reset after each run to ensure that the measurements were independent across different runs.

TABLE III: Comparison of Spectre attack detection by various approaches

	RDSM [7]			AT-RDSM			ODSA-MLP [42]			Proposed			
Methods	**DR** (%)	**FP** (%)	**FN** (%)	**DR** (%)	**FP** (%)	**FN** (%)	**DR** (%)	**FP** (%)	**FN** (%)	**DR** (%)	**FP** (%)	**FN** (%)	**improvement** over ODSA (%)
Spectre	89.1	7.7	3.2	94.1	4.3	1.6	99.2	0.8	0.0	96.2	2.4	1.4	-3.0
Evasive-Spectre	22.1	39.3	38.6	58.4	27.5	14.1	59.4	20.4	20.2	88.1	4.4	7.55	28.7
Average	**55.6**	**23.5**	**20.9**	**76.2**	**15.9**	**7.9**	**79.3**	**10.6**	**10.1**	**94.4**	**2.3**	**3.3**	**18.5**

Based on the above configuration, we compare performance in terms of detection accuracy and robustness between the following methods:

- **RDSM:** State-of-the-art detection framework for both Spectre and Meltdown attacks [7].
- **AT-RDSM:** We extended RDSM by training with adversarial samples to enable fair comparison.
- **ODSA:** State-of-the-art detection approach for Spectre attack using various implementations [42].
- **Proposed:** Our proposed detection technique using LSTM and explainable machine learning.

B. Detection of Spectre Attacks

ODSA [42] supports five implementations for Spectre detection. Table IV summarizes the effectiveness of these variations in detecting Spectre attacks. For each implementation, we provide detection rate (DR), false positive (FP) and false negative (FN) rates. Since MLP provided the best performance among ODSA implementations, we us the MLP implementation for subsequent comparison with our approach in Table III.

TABLE IV: Different ODSA implementations [42]

Implementations	**DR (%)**	**FP (%)**	**FN (%)**
LR	92.8	3.8	3.4
Tuned LR	96.4	1.1	2.5
SVM	96.8	0.7	2.5
Kernel SVM	98.3	0.7	1.0
MLP	99.2	0.8	0
Average	**96.7**	**1.4**	**1.9**

Table III compares the performance of our approach with state-of-the-art methods in detecting Spectre as well as evasive Spectre attacks. ODSA (MLP) performs better compared to both RDSM and AT-RDSM. Although ODSA (MLP) provides outstanding detection performance for normal Spectre attacks, its performance drops to about 50% facing evasive attacks, which is comparable a random guess. In other words, ODSA is not suitable for deployment due to its lack of robustness against evasive Spectre attacks. The failure of ODSA in the presence of evasive attacks is likely due to the fact that the features extracted from evasive samples are almost identical with that from the benign programs. This is consistent with the feature analysis discussed in Figure 5, when features are mingled in space, linear classifiers like LR, SVM or MLP are not expected to succeed. Our proposed approach significantly outperforms both RDSM and AT-RDSM for detecting Spectre attacks. While the performance of our approach (96.2% in Table V) is comparable with ODSA (96.7% average in Table IV) for detecting Spectre attacks, our approach significantly outperforms (28.7%) ODSA in detecting evasive Spectre attacks.

To demonstrate the effectiveness of our proposed explainable framework, we plot the distribution of incorrect classification for four categories. The category on false positive (FP) represents the misclassified benign programs. The false negative (FN) category represents the attacks that bypassed detection, and we further divide it into three subcategories: evasive attacks and two obfuscation strategies (Strategy 1 and Strategy 2 from Section IV-A). We observe that ODSA is extremely sensitive to Strategy 2, where appending redundant instructions into the original program often mislead ODSA to make false negative predictions. This is expected from the mechanism of ODSA, where two hardware events (LLC miss rate, branch miss rate) are treated as the dominant measurement for classification. Therefore, Strategy 1 makes little contribution to the misclassification since randomly putting program into sleep will not affect the above events. However, the branch selection and cache references induced by redundant execution are detrimental to ODSA.

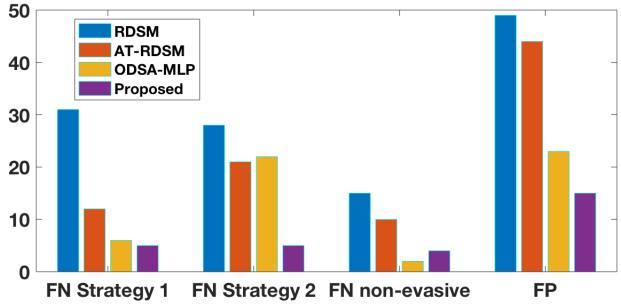


Fig. 9: Distribution of incorrect Spectre classification for both false positive (FP) and false negative (FN) results.

C. Detection of Meltdown Attacks

Table V compares our proposed method with RDSM and AT-RDSM for detection of Metldown attacks. Note that ODSA did not report any results for Meltdown

TABLE V: Comparison of Meltdown attack detection by various approaches

	RDSM [7]			AT-RDSM			Proposed			
Methods	**DR** (%)	**FP** (%)	**FN** (%)	**DR** (%)	**FP** (%)	**FN** (%)	**DR** (%)	**FP** (%)	**FN** (%)	**improvement** over AT-RDSM (%)
Meltdown	93.5	2.9	3.6	95.9	2.5	1.66	99.0	0	1.0	3.1
Evasive-Meltdown	19.2	25.6	55.2	55.6	22.8	21.6	94.5	2.3	3.2	38.9
Average	**56.4**	**14.2**	**29.4**	**75.9**	**12.7**	**11.4**	**96.7**	**1.2**	**2.1**	**20.8**

attacks. Since AT-RDSM outperforms RDSM in both categories, we compare the improvement with AT-RDSM in the last column. While AT-RDSM model provides 95.9% performance in detecting Meltdown attacks, its performance with evasive Meltdown drops to 55.6%, which is comparable to random guess. Such huge gap clearly indicates the instability of these methods with respect to obfuscation techniques. In contrast, our approach achieves more than 96% detection rate for non-evasive attacks. When we consider evasive ones, our approach still maintains a high detection accuracy. Most importantly, our approach provides superior performance for detecting both evasive Spectre (88.1%) and evasive Meltdown (94.5%) attacks.

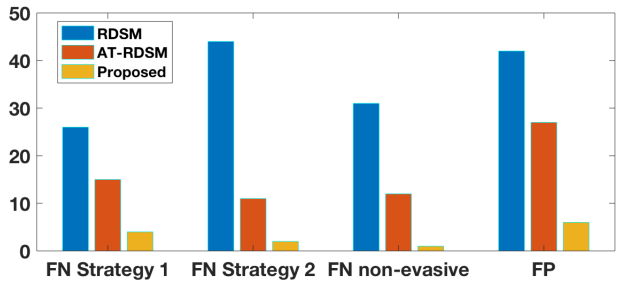


Fig. 10: Distribution of incorrect Meltdown classification results for both false positive and false negative results.

Table V also reveals the weakness of previous works in terms of high false positive results. Figure 10 shows the distribution of all misclassified inputs for four different categories. Clearly, RDSM is very vulnerable towards Strategy 2, since RDSM is based on the overall statistics from HPC. Inserting redundant instructions can hide the patterns of malicious behaviors such that a classifier cannot distinguish between malicious attacks and benign programs. While AT-RDSM improves the robustness against evasive attacks to some extent, it still has high false positive rate due to the lack of interpreting the reason for misclassification. Without the interpretation, users have no idea what is the exact reason for wrong prediction, and only blindly feed adversarial samples. This is expected to cause serious 'overfitting' problem - the model is likely to learn some benign features in these samples, and is likely to induce high false positive rate. We can observe two major reasons for our approach outperforming the state-of-the-art methods.

1) *Alternative Structure:* RNN handles time-sequential data so that it makes decisions utilizing potential information concealed in consecutive adjacent inputs, which provides more reliable classification results.
2) *Explainable Interpretation:* The critical difference between our method and AT-RDSM is that we perform outcome interpretation before adversarial training, so that we can carefully craft adversarial samples. This 'diagnose' before 'prescribe' guarantees the model robustness.

V. Conclusion

Spectre and Meltdown vulnerability coupled with the cache-based side channel attacks arise as serious threat to modern computer systems and have dramatically changed our perception of hardware security vulnerabilities. While existing defense mechanisms provide promising results, they have serious limitations including significant performance penalty and hardware overhead. Recently proposed machine learning based solutions are also not effective in the face of evasive attacks with obfuscation or other deviation capabilities. In this paper, we address these limitations by developing an explainable machine learning based detection framework. Our machine learning model is able to make decisions utilizing hardware events generated from hardware performance counters. Moreover, our proposed approach is also able to find the major contributors among all input features to help interpret the classification results, which is further utilized to defend against obfuscation techniques through adversarial training. Experimental results demonstrated that our approach provides comparable performance in detecting Spectre and Meltdown attacks, while it achieves drastic improvement (28.7% for evasive Spectre and 38.9% for evasive Meltdown) in defending evasive attacks compared to state-of-the-art approaches.

References

[1] P. Kocher, J. Horn *et al.*, "Spectre attacks: exploiting speculative execution," *Commun. ACM*, vol. 63, no. 7, pp. 93–101, 2020.
[2] M. Lipp *et al.*, "Meltdown: reading kernel memory from user space," *Commun. ACM*, vol. 63, no. 6, pp. 46–56, 2020.

[3] K. Khasawneh *et al.*, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *DAC*, 2019, p. 60.

[4] G. Wang *et al.*, "oo7: Low-overhead defense against spectre attacks via binary analysis," *CoRR*, vol. abs/1807.05843, 2018.

[5] M. Yan *et al.*, "Invisispec: Making speculative execution invisible in the cache hierarchy," in *MICRO*, 2019, p. 1076.

[6] V. Kiriansky *et al.*, "DAWG: A defense against cache timing attacks in speculative execution processors," in *MICRO*, 2018, pp. 974–987.

[7] B. Ahmad, "Real time detection of spectre and meltdown attacks using machine learning," *CoRR abs/2006.01442*, 2020.

[8] C. Li and J. Gaudiot, "Challenges in detecting an "evasive spectre"," *Com. Arch. Letters*, vol. 19, no. 1, pp. 18–21, 2020.

[9] P. Li *et al.*, "Conditional speculation: An effective approach to safeguard out-of-order execution against spectre attacks," in *HPCA*, 2019.

[10] T. M. Conte, E. P. DeBenedictis *et al.*, "Rebooting computers to avoid meltdown and spectre," *Computer*, vol. 51, no. 4, pp. 74–77, 2018.

[11] K. Khasawneh *et al.*, "Safespec: Banishing the spectre of a meltdown with leakage-free speculation," in *DAC*, 2019, p. 60.

[12] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss, "Context: A generic approach for mitigating spectre," in *NDSS*, 2020.

[13] M. Löw, "Overview of meltdown and spectre patches and their impacts," *Advanced Microkernel OS*, p. 53, 2018.

[14] L. Müller, "Kpti a mitigation method against meltdown," *Advanced Microkernel OS*, p. 41, 2018.

[15] Z. Pan and P. Mishra, "Automated test generation for hardware trojan detection using reinforcement learning," in *Asia and South Pacific Design Automation Conference (ASPDAC)*, 2021, pp. 408–413.

[16] Z. Pan, J. Sheldon, C. Sudusinghe, S. Charles, and P. Mishra, "Hardware-assisted malware detection using machine learning," in *Design Automation and Test in Europe (DATE)*, 2021.

[17] Z. Pan, J. Sheldon, and P. Mishra, "Test generation using reinforcement learning for delay-based side channel analysis," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2020.

[18] S. Charles, Y. Lyu, and P. Mishra, "Real-time detection and localization of distributed dos attacks in noc based socs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[19] S. Charles, M. Logan, and P. Mishra, "Lightweight Anonymous Routing in NoC based SoCs," in *Design Automation & Test in Europe (DATE)*, 2020.

[20] S. Charles and P. Mishra, "Reconfigurable network-on-chip security architecture," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 25, no. 6, pp. 1–25, 2020.

[21] ——, "A survey of network-on-chip security attacks and countermeasures," *ACM Computing Surveys (CSUR)*, vol. 54, no. 5, pp. 1–36, 2021.

[22] H. Witharana, Y. Lyu, and P. Mishra, "Directed test generation for activation of security assertions in rtl models," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 26, no. 4, pp. 1–28, 2021.

[23] F. Farahmandi, Y. Huang, and P. Mishra, *System-on-Chip Security: Validation and Verification*. Springer Nature, 2019.

[24] Y. Lyu and P. Mishra, "Scalable activation of rare triggers in hardware trojans by repeated maximal clique sampling," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2020.

[25] A. Ahmed, F. Farahmandi, Y. Iskander, and P. Mishra, "Scalable hardware trojan activation by interleaving concrete simulation and symbolic execution," in *2018 IEEE International Test Conference (ITC)*. IEEE, 2018, pp. 1–10.

[26] Y. Lyu and P. Mishra, "Maxsense: Side-channel sensitivity maximization for trojan detection using statistical test patterns," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 26, no. 3, pp. 1–21, 2021.

[27] Y. Huang, S. Bhunia, and P. Mishra, "Scalable test generation for trojan detection using side channel analysis," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 13, no. 11, pp. 2746–2760, 2018.

[28] ——, "Mers: statistical test generation for side-channel analysis based trojan detection," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 130–141.

[29] Y. Lyu and P. Mishra, "A survey of side-channel attacks on caches and countermeasures," *Journal of Hardware and Systems Security*, vol. 2, no. 1, pp. 33–50, 2018.

[30] R. Elnaggar and K. Chakrabarty, "Machine learning for hardware security: Opportunities and risks," *JETTA*, vol. 34(2), pp. 183–201, 2018.

[31] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep inside convolutional networks: Visualising image classification models and saliency maps," *arXiv preprint arXiv:1312.6034*, 2013.

[32] M. Sundararajan, A. Taly, and Q. Yan, "Axiomatic attribution for deep networks," in *ICML*, 2017, pp. 3319–3328.

[33] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," *arXiv preprint arXiv:1412.6806*, 2014.

[34] M. D. Zeiler and R. Fergus, "Visualizing and understanding convolutional networks," in *ECCV*, 2014, pp. 818–833.

[35] M. D. Zeiler, G. W. Taylor, and R. Fergus, "Adaptive deconvolutional networks for mid and high level feature learning," in *ICCV*, 2011, pp. 2018–2025.

[36] A. Shrikumar, P. Greenside, and A. Kundaje, "Learning important features through propagating activation differences," in *ICML*, 2017, pp. 3145–3153.

[37] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Security & Privacy*, 2017, pp. 39–57.

[38] M. Ribeiro *et al.*, "Why should I trust you?: Explaining the predictions of any classifier," in *SIGKDD*, 2016, pp. 1135–1144.

[39] Q. Zhang, Y. Yang, H. Ma, and Y. N. Wu, "Interpreting cnns via decision trees," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 6261–6270.

[40] Q. Zhang, R. Cao, Y. N. Wu, and S.-C. Zhu, "Growing interpretable part graphs on convnets via multi-shot learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 31, no. 1, 2017.

[41] Z. Pan, J. Sheldon, and P. Mishra, "Hardware-assisted malware detection using explainable machine learning," in *ICCD*, 2020, pp. 663–666.

[42] C. Li and J.-L. Gaudiot, "Online detection of spectre attacks using microarchitectural traces from performance counters," in *SBAC-PAD*, 2018, pp. 25–28.

[43] Y. Bengio, P. Simard, and P. Frasconi, "Learning long-term dependencies with gradient descent is difficult," *IEEE transactions on neural networks*, vol. 5, no. 2, pp. 157–166, 1994.

[44] S. Das, J. Werner, M. Antonakakis, M. Polychronakis, and F. Monrose, "Sok: The challenges, pitfalls, and perils of using hardware performance counters for security," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 20–38.

[45] "Spec integer benchmark." www.spec.org/benchmarks.html.