DAMUS: Adaptively Updating Hardware Performance Counter Based Malware Detector Under System Resource Competition

Yanfei Hu[1,2], Shuai Li[1,2], Shuailou Li[1,2], Boyang Zhang[1], and Yu Wen[1,3]
[1] *Institute of Information Engineering*, *Chinese Academy of Sciences*, Beijing, China
[2]*School of Cyber Security*, *University of Chinese Academy of Sciences*, Beijing, China
[3]*Corresponding author, Email:wenyu@iie.ac.cn*
{huyanfei, lishuai, lishuailou, zhangboyang}@iie.ac.cn;

***Abstract*—Hardware performance counter based malware detection (HMD) model that learns HPC-level behavior by using machine learning or deep learning algorithms has been widely researched in various application scenarios. However, the program's HPC-level behavior is easily affected due to system resource competition, which leaves counter based malware detection out-of-date. Unfortunately, current research could not adaptively update HMD model. In this paper, we propose DAMUS, a distribution-aware model updating strategy to adaptively update counter based malware detection model. Specifically, we first design an autoencoder with contrastive learning to map existing samples into a low-dimensional space for better calculating distributions. Second, in the low-dimensional space, the distribution characteristics are calculated for further judging the drift of testing samples. Finally, based on the total determined drifts of testing samples and a threshold, a decision could be given on whether the counter based malware detection model needs to be updated. We evaluate DAMUS by testing HMD model on datasets collected under benchmark application environment and actual server environment with different resource types or pressure levels. The experimental results show the advantages of DAMUS over existing updating strategies in promoting model updating. We also demonstrate its overhead spent on the task of malware detection.**

***Index Terms*—Hardware Performance Counter, Malware Detection, Resource Competition, Model Updating**

I. Introduction

Hardware performance counter (HPC) based malware detection (HMD) has already been widely researched in different scenarios [1]–[4] such as embedding system or computer system because of its low overhead. They detect malware using the model that learns the distribution of HPC-level behavior of benign or malicious softwares. HPC-level behavior is the time-series value of hardware event (e.g., cache-miss or instructions-retired) that HPC records during the execution of a program. However, due to the uncertainty of the system, the HPC-level behavior might be different at each run even for the same program [2], [5]. The deviated HPC-level behavior weakens HMD models' performance [5] and leaves HMD model out-of-date. As a result, there are plenty of false classifications, which makes subsequent task more complicated. Hence, adaptively updating HMD models before model outputs a decision is necessary.

Existing techniques that update HMD model could be classified into four types. The first is updating model periodically [6]–[8]. It reconstructs new model or updates model incrementally based on dataset within a certain period or a mount of data. The second updating strategy is by evaluating the performance decline [9], [10], such as the decline of probability on the testing sample. When the value is smaller than a threshold, updating begins. The third is lifelong updating [11], [12]. This method determine the updates based on model's misclassifications. If there are misclassifications, model unlearns these instances that have already learned to adapt to the real scenario. The last updating method is based on statistics analysis [13], [14] such as the uniformity measure between testing samples and existing samples.

However, these updating techniques do not apply to HMD model. The reasons are the following: First, determining updating period is difficult. A short period accelerates model updating, which increases resource cost and wastes much calculation resources; If the update period is too long, there will be more false classifications once the system model is out-of-date. Second, it is easy to cause inappropriate update. For example, lifelong updating method determines model update based on model's decision. A false misclassification will bring about undesired update of the model and weaken model's detection performance. Third, it is hard to complete the updating. For example, methods that use uniformity measure are prone to fail to find a reasonable function, which makes HMD model cannot be updated.

In this paper, we propose DAMUS, a distribution-aware model updating strategy that perceives distribution change of testing sample to realize HMD model's updating adaptively. First, the autoencoder with contrastive learning is used for finding a reasonable transformation function that transforms samples from high-dimensional space to a low-dimensional space; Second, the transformation function maps samples in high-dimensional space into their low-dimensional representations, and their distribution characteristics are calculated in low-dimensional space; Third, transforming each testing sample into low-dimensional space and evaluating its similarity to other samples within each class. This enables us to determine testing sample's deviation from existing distribution. Fourth, set the threshold and when the total number of deviated samples is higher than the threshold, the model will be updated adaptively.

We evaluate DAMUS on two datasets collected under benchmark application environment and actual server environment with different resource types or pressure levels using HMD model. The experimental results show the advantages of

DAMUS over existing updating strategies in promoting model updating. We also demonstrate its overhead spent on the task of malware detection.

To conclude, we make the following contributions:

- We propose DAMUS, a distribution-aware model updating strategy to update HMD model in resource competition environment to tackle the out-of-date problem faced by HMD models.
- We introduce a series of new designs to make DAMUS adaptively judge the update of HMD model.
- We test DAMUS on two different datasets with different types or levels of resource competition and demonstrate the effectiveness of DAMUS in updating HMD models.

II. Background

HPC is a register in performance monitoring unit (PMU). It is originally used for performance analysis and optimization [15], high performance computing [16] or debugging [17] and so on. It monitors events occurred on architecture or micro-architecture inside CPU when a program is running. The value of HPC can be read either with polling mode at the end of a running program or with sampling mode periodically in its running. Obtaining accurate and reliable HPC values can be difficult. First, runtime environment may vary each time program runs [2]. Second, performance counters may overcount certain events on some processors [18]. For instance, the instruction retired event may be overcounted on Pentium D processors. Third, many sources of non-determinism such as hardware interrupts from periodic timers can be hard to predict [18]. Fourth, HPC collection tool is implemented in different ways (e.g., thread or process level or the point when HPC are read).

III. Insights & Possible Directions

By carefully investigating the problems caused by existing updating methods, we draw the following insights. The first is a strong assumption that the testing samples and existing training samples share the same distribution. For example, the update method based on performance decline holds this assumption. However, it is likely that the testing samples deviate from samples of existing classes, which is very common for HPC-level behavior in resource competition environment. Similarly, the lifelong model updating method based on unlearning holds this assumption, too. The second is changing model's distribution but suffering from the limitation of the first assumption. For example, the lifelong model updating strategy based on unlearning, which changes model's distribution by forgetting mis-classified instances. However, the unlearning is still based on model's decision. If there is a false mis-classification, there will be unnecessary updates. The third is comparing sample's statistical characteristics. This is a better direction even though the reasonable functions for evaluating uniformity measure are hard to define. For example, the fourth type of updating methods determine model updates based on sample statistical characteristics rather than model decision results.

Based on above analysis, we aim to update HMD model by judging testing sample's deviation from the distribution of existing samples.

Possible Directions. A feasible direction is to directly analyze the fitness of testing samples to samples in a given class. That is to evaluate the testing sample is more suitable for a class than other samples in this class. For example, the authors introduce a system called Transcend [14]. It defines the measurement of uniformity as the fitness evaluation, and uses a reliable value to quantitatively measure how much a testing sample is similar to other samples within the same class. Although this metric can accurately locate deviated samples, this system highly depends on a good definition of "dissimilarity" function [19].

Hence, we decide from the perspective of distance to quantitatively measure how much a testing sample is similar to other samples within the same class rather than the definition of "dissimilarity". However, for high-dimensional data, due to sparse dimensions, the meaning of distance is no longer valid. We have demonstrated this in Section V-B. Therefore, we decide to compress samples from high-dimensional space to low-dimensional space. The low-dimensional space makes distance measure between testing samples and existing samples continue to be effective.

Challenges and Solutions. It is a great challenge to find an appropriate means for space transformation. For example, the autoencoder could compress samples from high-dimensional space to low-dimensional space according to reconstructed error, which enables us to use distance again to evaluate sample distribution. However, as an unsupervised method, when sample's label is ignored, the transformed samples lack effective constraint in the low-dimensional space. For example, the distance between two samples in low-dimensional space becomes very small after compression, while it is large in original space. Such types of variation makes sample distribution in low-dimensional space different from that in original space.

To solve above challenge, we design an autoencoder with contrastive learning for space transformation. The autoencoder transforms samples from high-dimensional space to low-dimensional space. Contrastive learning is used to constrain sample's distribution in low-dimensional space. In this way, we could learn a better function for space transformation, namely encoder function, and also ensure the reasonable distribution of samples in low-dimensional space. Then, we can decide whether a testing sample is deviated from the distribution of existing samples in low-dimensional space, and HMD model needs to be updated.

IV. Methodology of DAMUS

We design DAMUS to update HMD model adaptively. DAMUS is short for "**D**istribution-**A**ware **M**odel **U**pdating **S**trategy through outlier detection". Next, we first illustrate the overview, followed by technical details.

A. *Overview of* DAMUS

DAMUS consists of three steps: First, sample space transforming. The training samples are converted from the original

space to a low-dimensional space by using autoencoder network with contrastive learning. This allows the distribution of existing samples easier to measure. Second, the calculation of samples' distribution. In low-dimensional space, the distribution characteristics of existing samples are calculated for providing a basis on judging the deviation of testing samples. Third, the judgement on model updating. For each testing sample, DAMUS judges whether it is an out-of-distribution sample and wheter the number of drifts satisfies the conditions (e.g., a threshold) of model update. DAMUS could be integrated into HMD model. This allows us to check whether the model needs to be updated before using the model to make a decision rather than updating HMD model based on its decisions. The adaptive counter based malware detection process based on DAMUS is shown in Figure 1, where $CENT$, $Z-CENT$, and MAD represent partial distribution charateristics of existing samples.

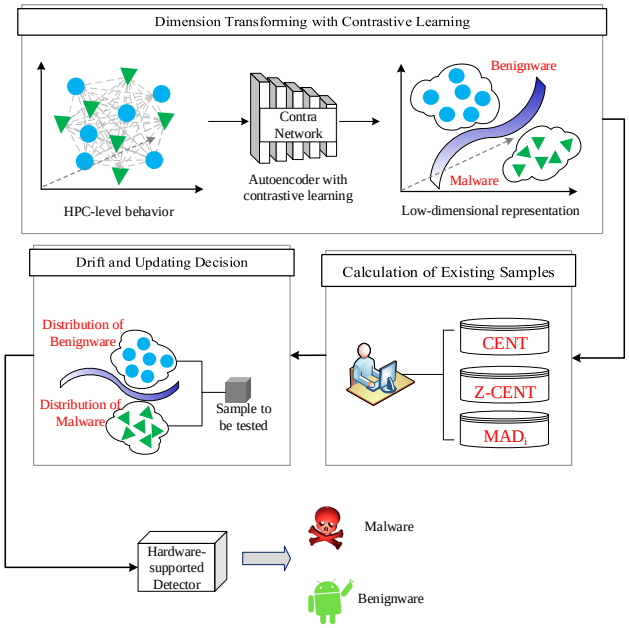

Fig. 1: Counter based malware detection using DAMUS.

B. Technical Details

Dimension Transforming. Dimension transforming maps high-dimensional samples into low-dimensional representations. To this end, we use an autoencoder model trained with contrastive learning. Autoencoder is used for outputting low-dimensional representation of a given input instance. The contrastive learning aims to constrain them in low-dimensional space. Concretely, samples in the same class have a smaller distance in low-dimensional space, while samples from different classes have a larger distance. In this way, the distance between representations in low-dimensional space could reflect the difference of each pair of samples. The testing sample with a large distance from all existing classes is probably a deviated sample.

Based on existing samples, we want to learn this extended autoencoder. Formally, for any sample from the training dataset, $x \in \mathcal{R}^{q\times 1}$, we train an autoencoder including encoder e and decoder de, where e is parameterized by θ, and de is parameterized by ϕ. The loss function is the following:

$$min_{\theta,\phi} E_x \|x - \hat{x}\|_2^2 + \lambda E_{x_i,x_j}[(1-y_{ij})d_{ij}^2 + y_{ij}(m-d_{ij})_+^2] \quad (1)$$

Here, the first term is the reconstruction loss from autoencoder. Concretely, the goal of encoder e is to learn the representation of original input in low-dimensional space. For any input x, encoder e transforms original input x into a low-dimensional representation $z = e(x;\theta)$. The decoder ensures the representation z could be decoded with minimal loss. Here, $\hat{x}$ stands for reconstructed samples, i.e., $\hat{x} = de(z,\phi)$. The loss is the mean squared error between x and $\hat{x}$.

The second term in above equation represents contrastive loss. Its input is a pair of sample (x_i, x_j) and their relationship y_{ij}. If two samples come from same class, then $y_{ij} = 0$; otherwise, $y_{ij} = 1$. $(\cdot)_+$ is short for function $max(0,\cdot)$. d_{ij} is the Euclidean distance between representation $z_i = e(x_i;\theta)$ and representation $z_j = e(x_j;\theta)$, where $z \in R^{d\times 1}(d<<q)$. This term allows minimal distance between samples x_i and x_j if they are from same class. It also maximizes samples' distance up to a radius m if they are from different classes, and ensures the distance contributes to the loss only when it is within the radius. λ is a hyper-parameter determining the weight of contrastive loss.

Once completing training, encoder e could map each input into low-dimensional space where the distance for evaluating sample's deviation could be effective.

Distribution Calculation. After obtaining autoencoder with contrastive learning, we can map existing samples into low-dimensional space, and calculate their distributions. The algorithm about calculating the distribution of existing samples is shown in Algorithm 1. Assume there are N classes within training dataset, and each class has n_i training samples, where $i = 1, 2, ..., N$. We first use the encoder to map all training samples into low-dimensional space (line 2-4). For each class, we calculate its centroid (line 5) by calculating the mean value of each dimension among all representations in Euclidean space.

To better determine the match degree between testing samples and existing samples, a main challenge is that the samples of different classes may have different compactness, so different distance thresholds are required. Instead of manually setting absolute distance thresholds for each class, a method called Median Absolute Deviation (MAD) is used. That is to estimate the sample distribution of each class by calculating MAD_i (line 6-10), which is the median of the difference of distance $d_i^{(j)}(j = 1, 2, ..., n_i)$ and these distance's median value. Here, $d_i^{(j)}(j = 1, 2, ..., n_i)$ describes a low-dimensional distance from each sample in class i to class centroid. Using the distance to centroid allows us to judge the distribution of testing sample relative to other samples in the same class. Note

that the distance is from each sample to the centroid rather than to the nearest sample. This is because the distance to the nearest sample is easily disturbed by the outliers in training dataset. n_i is the number of samples in class i (line 5). Thus, each class has its own distance threshold to determine whether the testing sample is a deviated sample.

Algorithm 1 Algorithm on calculating the distribution of existing training samples.

Require: input sample $x_i^{(j)}$, $i = \{1, 2, ..., N\}$, $j = \{1, 2, ..., n_i\}$,N is the number of classes , n_i is the number of training samples in class i; encoder e, constant b.

Ensure: centroid of each class c_i and its MAD_i.

1: **for** class $i = 1 \rightarrow N$ **do** //N means class number
2: **for** $j = 1, 2, ..., n_i$ **do** //n_i stands for the number of training samples in class i.
3: $z_i^{(j)} = e(x_i^{(j)}; \theta)$
4: **end for**
5: $c_i = \frac{1}{n_i} \sum_{j=1}^{n_i} z_i^{(j)}$
6: **for** $j = 1 \rightarrow n_i$ **do**
7: $d_i^{(j)} = \|z_i^{(j)} - c_i\|_2$
8: **end for**
9: $\bar{d}_i$ = median $(d_i^{(j)}), j = 1, 2, ..., n_i$
10: $MAD_i = b * median(|d_i^{(j)} - \bar{d}_i|)$, $j = 1, 2, ..., n_i$
11: **end for**

Deviation Judgement and Updating Decision. The algorithm on judging deviated samples and model updating is shown in Algorithm 2. Given a testing sample $x_t^{(k)}$, we still use the encoder to map it into low-dimensional representation (line 2). Then, we calculate the Euclidean distance between testing sample and class centroid, $d_i^{(k)} = \|z_t^{(k)} - c_i\|_2$ (line 4). Finally, based on MAD_i and T_{MAD}, we can determine whether the distance $d_i^{(k)}$ is large enough to make the testing sample deviate from the distribution of class i (line 3-16). If the testing sample deviates from all classes, it is considered as a deviated sample. Otherwise, it could be imported into detector for classifying directly. An important step in model updating is the threshold of total deviated samples. We use the percentage of the number of testing samples to determine the update threshold.

Training Strategy and Hyper-parameters. We apply adam optimizer to minimize the loss function in Equation (1) and set its learning rate as 0.001. The technical details of this optimization technique can refer to [20].

The contrastive autoencoder network has five hidden layers, and each layer is a two-layer building block. We apply Rectified Linear Unit as the activation function in building block, and each block is made up of two convolution operations. Besides, the batch size is 128 and the training epoch is 250. The dimension that the autoencoder outputs is 128. Here, we set $\tau = \lambda_d * K$(line 11), where λ_d is the scale factor of allowed deviated samples and set to 3% manually. Like [19], T_{MAD} = 3.5 and b = 1.4826.

Algorithm 2 Distribution-aware model updating strategy

Require: testing sample $x_t^{(k)}$,t testing dataset, $k = 1, 2, ...K$, K total number of testing samples; encoder e; centroid for class c_i, MAD_i, T_{MAD} and threshold τ.

Ensure: the sum of drifting samples S_d and warning for updating.

1: **for** $k = 1 \rightarrow K$ **do**
2: $z(k)_t = e(x_t^{(k)}; \theta)$
3: **for** class $i = 1 \rightarrow N$ **do**
4: $d_i^{(k)} = \|z_t^{(k)} - c_i\|_2$
5: $A^{(k)} = \frac{|d_i^{(k)} - \bar{d}_i|}{MAD_i}$
6: **end for**
7: $A^{(k)} = min(A_i^{(k)}), i = 1, 2, ..., N$
8: **if** $A^{(k)} > T_{MAD}$ **then**
9: $x_t^{(k)}$ is a potential drifting sample.
10: $S_d = S_d + 1$
11: **if** $S_d \geq \tau$ **then**
12: **return** warning //output warning for model updating
13: **end if**
14: **else**
15: $x_t^{(k)}$ is a non-drifting sample.
16: **end if**
17: **end for**
18: **return** S_d

V. Evaluation

We evaluate the effectiveness of DAMUS using HPC based malware detection. We mainly focus on three aspects:

(a) the design of DAMUS,

(b) the effectiveness brought by DAMUS, i.e., the performance comparison of model's detection using DAMUS and existing model updating mothods, and

(c) the cost on detection. Specifically:

As DAMUS mainly contains autoencoder trained with contrastive learning for transforming samples and calculating distribution characteristics. To verify the reasonability of autoencoder, we first evaluate the distance before and after transforming sample space. Then, to evaluate the effects brought by contrastive learning, we compare the performance on deviated sample judgement with contrast learning and without it. To evaluate the effect of the model update strategy, we compare the performance of HMD model that updates using periodical methods and DAMUS, respectively.To evaluate the overhead of DAMUS, we mainly measure its running time and analyze the running time proportion in the whole detection task.

Based on above analysis, we plan to answer the following questions:

$Q1_a$: What are the distribution characteristics of existing samples before and after transforming with autoencoder?

$Q2_a$: How about the effectiveness introduced by contrastive learning?

$Q3_b$: What is the performance improvement after using the model update strategy?

$Q4_c$: What is the overhead of DAMUS, and what is the proportion of time spent in the whole detection task?

A. Experimental Setup

Program Sample. Like [21], [22], we utilize an approximately equal number of benignwares and malwares to eliminate classification bias. The program sample includes 656 benignwares and 660 malwares. Benignwares are from two aspects, the first is the stress-ng of 184 programs [23] and the second is downloaded or extracted from fresh OS installations of 472 executable programs. Malwares are downloaded from virusshare [24]. We choose 660 (out of 5284 downloaded) malwares that have more HPC-level behavior traces than other malwares to mitigate sample bias. All the benignwares and malwares are executable binary version for linux system. We assume all the downloaded malwares have been verified by expert, thus label these malwares as positive and benignwares as negative.

Dataset. We prepare two different datasets including Ds-A and Ds-B to meet each evaluation aspect. Ds-A is a dataset of program sample's HPC-level behavior collected under 40 environments where different types and pressures of benchmark programs run as the background processes in each environment. Ds-B is a dataset of program sample's HPC-level behavior collected in the actual environments where multiple service processes are running together on the server system with different pressure levels. The more information about Ds-A and Ds-B could refer to [5]. In order to highlight the effectiveness of DAMUS, we divide Ds-A into two parts: Ds-Train and Ds-Test. The former is 30% of the total data in Ds-A, and the rest is Ds-Test. Each item in datasets is a vector of 4×6000.

Baseline Models. We utilize HMD models that use HPC-level behavior as features. These models almost account for recent ten years' works. Considering model's performance may be affected by algorithm's characteristic, we utilize a serious of algorithms including Support Vector Machine (SVM) [25], Logic Regression (LR) [22], [26], Random Forest (RF) [27], and Multiple Layer Perceptron (MLP) [22], [26] to eliminate model algorithm's effects. For evaluating DAMUS, we compare it with periodic model updating methods where we set the amount of testing data as the period.

Metrics. We use the proportion of deviated samples determined by DAMUS to the testing samples as an index. In addition, we follow evaluation index that commonly used to evaluate HMD model, including precision, recall and F1-score.

B. Experimental Design and Results

$Q1_a$: What are the distribution characteristics of existing samples before and after transforming with autoencoder?

To show the distribution characteristics of existing samples, we calculate the mean value, maximum and minimum value of each sample in Ds-Train to their class centroids, since the centroid plays an important role in controlling the distribution to some extent. The results are shown in Table I.

It can be seen that the distribution parameters of benign samples and malicious samples are significantly different between original space and transformed space. Specifically, the distance parameters of benignwares are in the same order of magnitude; On the contrary, the parameters of malwares vary greatly, and there are multiple orders of magnitude between the maximum distance and average distance. This is mainly due to two reasons: First, benignwares are high-dimensional and not sparse. Second, malwares are high-dimensional and sparse. However, in the encoded space, we can see that the sample distance of both benign samples and malware samples is in the same dimension, and their distance difference is not obvious. These results show that it is necessary to use the autoencoder to convert high-dimensional samples into low-dimensional space.

TABLE I: Distribution comparison of various samples in each class in Ds-Train calculated before and after encoding.

Phase	Distance	Benignware	Malware
Before encoding	d_{max}	747	$159 * e13$
	d_{min}	0	0
	d_{ave}	162	$144 * e8$
After encoding	d_{max}	**37**	**36**
	d_{min}	0	0
	d_{ave}	**25**	**20**

$Q2_a$: How about the effectiveness introduced by contrastive learning?

To verify the effectiveness introduced by contrastive learning, we trained the autoencoder with contrastive learning loss and without this loss on Ds-Train, respectively. The difference between them is whether contrastive learning loss is added or not. Then, we use the obtained transformed function to perform space tranforming and detect drift samples. The detection results on Ds-Test are shown in Table II.

Overall, we can see from Table II that the proportion of deviated samples detected in the testing samples has increased after using contrastive learning. Concretely, the proportion of drifted benign samples detected ranges from 3% to 5%, with a small increase. The proportion of detected malware samples increases from 7% to 12%. This is mainly because the distance between sample pairs becomes more separated after space transformation. That is to say, the autoencoder with contrastive learning consciously fine-tunes the distance between intra-class and inter-class samples. The data results show that the autoencoder with contrastive learning is helpful to detect deviated samples.

Note that since we have used the same samples in the experimental setup to collect their HPC-level behavior under different system environments, we think the drifts are caused only by resource competition. As the ground-truth of deviated HPC-level behavior is difficult to determine manually after the interference of resource competition, we use percentage of the drifts in total testing samples calculated as a comparison.

TABLE II: The comparison of drifting sample ratio detected using contrastive learning or not.

Dimension Transforming	Ds-Test	
	Benignware	Malware
Autoencoder	3%	7%
Contrastive Autoencoder	**5%**	**12%**

$Q3_b$: How is the performance improvement introduced by model update strategy?

To verify the performance improvement introduced by DAMUS, we take Ds-Train as the training dataset for training original model, and then give the detection results on Ds-B. Then, we add DAMUS before malware detection, when the deviation of the testing sample in Ds-B reaches the threshold value, we will update the model by fusing new HPC-level behavior into existing training dataset. The performance comparison of original model and updated model (short for DAMUS model) on Ds-B is shown in Table III.

At the high-level, we can see from Table III that the F1 score of DAMUS model is significantly improved. Specifically, the performance of each model has been improved by different degrees except DAMUS SVM. For SVM model, its performance improvement is relatively low. It is likely that the default kernel function of SVM model does not fit well on these datasets. For model built with MLP, the improvement is more obvious and up by 27.1%. In addition, the recall value is also increased, in other words, more malwares could be detected. The data results show that DAMUS is effective to HMD model updates under system resource competition environments.

TABLE III: Performance comparison of models with different updating strategy on real-world systems.

Strategy	Model	Ds-B		
		Precision	Recall	F1-score
Period	SVM [25]	0.79	0.79	0.79
DAMUS		0.79	0.82	**0.81**
Period	LR [26]	0.85	0.74	0.79
DAMUS		0.95	0.91	**0.93**
Period	RF [27]	0.95	0.79	0.86
DAMUS		0.98	0.98	**0.98**
Period	MLP [22]	0.54	0.47	0.50
DAMUS		0.82	0.97	**0.89**

$Q4_c$: What is the overhead of DAMUS, and what is the proportion of time spent in the whole detection task?

To evaluate the overhead of DAMUS, we calculate its execution time. Even though the model updating strategy needs to train the autoencoder model using contrastive learning, we do not consider the time spent on training since the model is offline training and the training time could be compensated by large number of data. The measured execution time of DAMUS is 0.11ms.

Further, we integrate DAMUS into original detection model and measure the time spent in the whole detection task. Each model's running time is shown in Table IV. Here, the row containing "/" represents original models that do not use DAMUS, and DAMUS model represents existing malware detection models that are extended with DAMUS in real-time. We can see that the running time of DAMUS model has increased to a certain extent compared with that of original models, especially for model using LR algorithm. In addition, the proportion of DAMUS in the whole detection task of each DAMUS model (DAMUS SVM, LR, RF, MLP) is 1.2%, 91.7%, 39.3%, 28.2%, respectively, which is inversely proportional to the detection time of original models. The overhead is worthy compared to the mis-classifications caused by model's out-of-date.

TABLE IV: Average detection time of model with each framework.

Framework	Aver Detection Time (Unit:ms)			
	SVM	LR	RF	MLP
/	8.96	0.01	0.17	0.28
DAMUS	**9.07**	**0.12**	**0.28**	**0.39**

VI. Discussion

Lifelong Detection Model. The lifelong detection model [11], [12] updates model actively to unlearn the distribution of mis-classified samples. Applying DAMUS to lifelong detection model helps to update model correctly. For example, lifelong detection model should unlearn mis-classified instances that are really deviated from existing distribution. DAMUS helps the model omit updates caused by mis-classifications that are really in-class samples or not deviated samples. In this way, DAMUS further improves the performance of unlearning based lifelong detection model.

Limitations and Future Work. DAMUS has several limitations. First, DAMUS is vulnerable and easy to be attacked. The main part of DAMUS is an encoder model trained using contrastive learning, which is composed by multi-layer network. Any attack (such as poisoning attack by integrating contaminated data, model inference attack) that puts the model in danger will affect model's performance and the function of DAMUS. This could be enhanced by relevant model defense methods [28]. Second, it is affected by experts' experience. The judgment of model updating conditions is based on a threshold, and different thresholds will affect the updating decision. Future work could solve such kind of defect by combining with lifelong malware detection model [11], that is implementing unlearning based on DAMUS's judgement on drift samples.

VII. Related work

Outlier Detection. Outlier detection method, for example, probability based detection [10] or anomaly based detection [29], detect the outlier that is out-of-distribution. These detection methods build on an assumption that the distribution of testing sample conforms to that of existing samples. However, we do not hold this assumption anymore. A more

related work to ours is CADE [19] that detect drifts from intra class samples or out-of-class samples. However, we have two obvious differences compared to CADE. First, we detect the drift caused by resource competition which is different from that of intra class. Second, we aim at developing model updating strategy based on the drift detection.

HPC based Malware Detection. Such types of detection techniques build their detection models by using machine learning algorithms or deep neural networks [5], [22], [25], [26], [30]. These models are data-driven and may be out-of-date due to the changes of system resource competition environments. DAMUS could be used to update these models directly before model's decision.

VIII. Conclusions

This paper presents DAMUS, a distribution-aware model updating strategy. First, by designing an autencoder with contrastive learning, we transfer existing training samples and testing samples to low-dimensional space; Second, in the low-dimensional space, the distribution characteristic are calculated for further deviation judgement. Finally, based on the total drifts determined and a threshold, a decision could be given on whether the detection model needs to be updated. Using the dataset collected under benchmark application environment and the actual server environment with different resource types or pressure levels, we demonstrate the advantages and practicability of DAMUS in promoting model updating.

References

[1] Z. Pan, J. Sheldon, and P. Mishra, "Hardware-assisted malware detection and localization using explainable machine learning," *IEEE Transactions on Computers*, 2022.

[2] 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)*, pp. 20–38, IEEE, 2019.

[3] T. Zhang, Y. Zhang, and R. B. Lee, "Cloudradar: A real-time side-channel attack detection system in clouds," in *International Symposium on Research in Attacks, Intrusions, and Defenses*, pp. 118–140, Springer, 2016.

[4] B. A. Ahmad, "Real time detection of spectre and meltdown attacks using machine learning," *arXiv preprint arXiv:2006.01442*, 2020.

[5] Y. Hu, Y. Wen, S. Liang, M. Li, T. Xue, and B. Zhang, "CARE: Enabling hardware performance counter based malware detection resilient to system resource competition," in *2022 International Conference on High Performance Computing & Communications (HPCC)*, pp. 586–594, 2022.

[6] C. Sima, Y. Fu, M.-K. Sit, L. Guo, X. Gong, F. Lin, J. Wu, Y. Li, H. Rong, P.-L. Aublin, *et al.*, "Ekko: A {Large-Scale} deep learning recommender system with {Low-Latency} model update," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pp. 821–839, 2022.

[7] E. Oldridge, J. Perez, B. Frederickson, N. Koumchatzky, M. Lee, Z. Wang, L. Wu, F. Yu, R. Zamora, O. Yilmaz, *et al.*, "Merlin: a gpu accelerated recommendation framework," in *Proceedings of IRS*, 2020.

[8] A. Eisenman, K. K. Matam, S. Ingram, D. Mudigere, R. Krishnamoorthi, K. Nair, M. Smelyanskiy, and M. Annavaram, "Check-n-run: a checkpointing system for training deep learning recommendation models," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pp. 929–943, 2022.

[9] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov, "Learning and classification of malware behavior," in *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pp. 108–125, Springer, 2008.

[10] A. Deo, S. K. Dash, G. Suarez-Tangil, V. Vovk, and L. Cavallaro, "Prescience: Probabilistic guidance on the retraining conundrum for malware detection," in *Proceedings of the 2016 ACM workshop on artificial intelligence and security*, pp. 71–82, 2016.

[11] B. Wang, Y. Yao, S. Shan, H. Li, B. Viswanath, H. Zheng, and B. Y. Zhao, "Neural cleanse: Identifying and mitigating backdoor attacks in neural networks," in *2019 IEEE Symposium on Security and Privacy (SP)*, pp. 707–723, IEEE, 2019.

[12] M. Du, Z. Chen, C. Liu, R. Oak, and D. Song, "Lifelong anomaly detection through unlearning," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1283–1297, 2019.

[13] G. Shafer and V. Vovk, "A tutorial on conformal prediction.," *Journal of Machine Learning Research*, vol. 9, no. 3, 2008.

[14] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, and L. Cavallaro, "Transcend: Detecting concept drift in malware classification models," in *PROCEEDINGS OF THE 26TH USENIX SECURITY SYMPOSIUM (USENIX SECURITY'17)*, pp. 625–642, USENIX Association, 2017.

[15] D. Chen, N. Vachharajani, R. Hundt, X. Li, S. Eranian, W. Chen, and W. Zheng, "Taming hardware event samples for precise and versatile feedback directed optimizations," *IEEE Transactions on Computers*, vol. 62, no. 2, pp. 376–389, 2011.

[16] X. Zhou, K. Lu, X. Wang, and X. Li, "Exploiting parallelism in deterministic shared memory multiprocessing," *Journal of Parallel and Distributed Computing*, vol. 72, no. 5, pp. 716–727, 2012.

[17] R. O'Callahan, C. Jones, N. Froyd, K. Huey, A. Noll, and N. Partush, "Engineering record and replay for deployability," in *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, 2017, pp. 377–389.

[18] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *2013 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013, pp. 215–224.

[19] L. Yang, W. Guo, Q. Hao, A. Ciptadi, A. Ahmadzadeh, X. Xing, and G. Wang, "{CADE}: Detecting and explaining concept drift samples for security applications," in *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[20] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[21] M. Ozsoy, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Malware-aware processors: A framework for efficient online malware detection," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pp. 651–661, IEEE, 2015.

[22] N. Patel, A. Sasan, and H. Homayoun, "Analyzing hardware based malware detectors," in *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, 2017.

[23] C. I. King, "Stress-ng," *URL: http://kernel. ubuntu. com/git/cking/stressng. git/(visited on 28/03/2018)*, 2017.

[24] Establisher, "torrents." https://virusshare.com/(visited on 30/04/2020), 2020.

[25] B. Singh, D. Evtyushkin, J. Elwell, R. Riley, and I. Cervesato, "On the detection of kernel-level rootkits using hardware performance counters," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pp. 483–493, 2017.

[26] M. Ozsoy, K. N. Khasawneh, C. Donovick, I. Gorelik, N. Abu-Ghazaleh, and D. Ponomarev, "Hardware-based malware detection using low-level architectural features," *IEEE Transactions on Computers*, vol. 65, no. 11, pp. 3332–3344, 2016.

[27] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo, "On the feasibility of online malware detection with performance counters," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 3, pp. 559–570, 2013.

[28] J. Steinhardt, P. W. W. Koh, and P. S. Liang, "Certified defenses for data poisoning attacks," *Advances in neural information processing systems*, vol. 30, 2017.

[29] M. Masana, I. Ruiz, J. Serrat, J. van de Weijer, and A. M. Lopez, "Metric learning for novelty and anomaly detection," *arXiv preprint arXiv:1808.05492*, 2018.

[30] S. Das, B. Chen, M. Chandramohan, Y. Liu, and W. Zhang, "Ropsentry: Runtime defense against rop attacks using hardware performance counters," *Computers & Security*, vol. 73, pp. 374–388, 2018.