Revisiting Concept Drift in Windows Malware Detection: Adaptation to Real Drifted Malware with Minimal Samples

Adrian Shuai Li*, Arun Iyengar†, Ashish Kundu‡, Elisa Bertino*
*Purdue University, †Intelligent Data Management and Analytics, LLC, ‡Cisco Research
*{li3944, bertino}@purdue.edu, †aki@akiyengar.com, ‡ashkundu@cisco.com

***Abstract*—In applying deep learning for malware classification, it is crucial to account for the prevalence of malware evolution, which can cause trained classifiers to fail on drifted malware. Existing solutions to address concept drift use active learning. They select new samples for analysts to label and then retrain the classifier with the new labels. Our key finding is that the current retraining techniques do not achieve optimal results. These techniques overlook that updating the model with scarce drifted samples requires learning features that remain consistent across pre-drift and post-drift data. The model should thus be able to disregard specific features that, while beneficial for the classification of pre-drift data, are absent in post-drift data, thereby preventing prediction degradation. In this paper, we propose a new technique for detecting and classifying drifted malware that learns drift-invariant features in malware control flow graphs by leveraging graph neural networks with adversarial domain adaptation. We compare it with existing model retraining methods in active learning-based malware detection systems and other domain adaptation techniques from the vision domain. Our approach significantly improves drifted malware detection on publicly available benchmarks and real-world malware databases reported daily by security companies in 2024. We also tested our approach in predicting multiple malware families drifted over time. A thorough evaluation shows that our approach outperforms the state-of-the-art approaches.**

I. Introduction

Recent automatic malware classification methods use deep learning (DL) techniques with various feature types, including static features [1], [2], [7], [29], [33], [41], [43], [44], [49], [51] and dynamic features [16], [54], and their combinations [11]. DL has shown noticeably better performance than the traditional signature-based approach and machine learning (ML) methods. However, a major challenge in the use of DL is the concept drift, wherein the distribution of test data diverges from the one of the original training data. While this issue is not unique to malware—it is akin to the "out-of-distribution" problem in DL—the intricacies of the malware evolution make domain drift particularly complex and crucial. Malware continually evolves, adopting new paradigms to evade detection and maximize the damage. Attackers further complicate detection by creating adversarial samples through code mutations and injection of dummy code. With the development of generative AI, research has shown that the attack can even jailbreak large language models for malicious code generation [34], [48].

Traditional methods to handle concept drift in malware classification involve labeling new samples and retraining the model. However, this process is time-consuming, costly, and sometimes impractical due to analysts' limited capacity for daily sample labeling. Another approach involves using pseudo-labels to provide noisy labels for drifted malware, which is then used to update the model [18], [49]. However, this method relies heavily on the accuracy of the initial estimate and is prone to negative feedback loops if the process is not designed properly. The state-of-the-art solutions use active learning to adapt to concept drift. They deliberately select important labels crucial for learning new malware distribution and then retrain the classifier with those new labels. There are many schemes for selecting which samples to label [5], [9], [31], [32], [36], [53], with the goal of reducing the amount of manual labeling effort needed to achieve a good performance.

Past work has made progress in the detection of drifted malware. Nonetheless, past approaches have predominantly used basic retraining techniques for model updating. Chen et al. [9] were the first to distinguish between two prevalent model updating strategies in active learning. The first strategy, known as cold-start learning, involves training a fresh model each time new labels are introduced. The second strategy, referred to as warm-start learning, continues training an existing model with new samples. However, our experimental findings indicate that neither strategy yields optimal performance when only a few new samples are available.

We envision that addressing concept drift, particularly with scarce labeled samples, requires learning features that remain consistent before and after the drift. Malware detection models should thus be able to disregard features from the pre-drift data that may not be present in the post-drift data. Neural networks are prone to "cheating" where they resort to shortcuts during prediction [55], leading to prediction failures when tested with data devoid of such shortcut information [22]. This problem extends to malware prediction systems as well. For instance, some malware samples may primarily consist of a few define directives for reserving variable storage space due to packing and obfuscation. These samples predominantly contain instructions like db, dw, and dd, which allocate byte, word, and double word, respectively. If a neural network is trained with many of these malware samples, it might leverage this pattern

Network and Distributed System Security (NDSS) Symposium 2025
24-28 February 2025, San Diego, CA, USA
ISBN 979-8-9894372-8-3
https://dx.doi.org/10.14722/ndss.2025.240830
www.ndss-symposium.org

to predict an input as malicious upon encountering a series of these instructions. Consequently, it might incorrectly classify a malware sample with fewer of these operation codes but more API calls as benign. Likewise, a benign software that employs packing tools could be misclassified as malicious, resulting in a false positive due to the prevalence of define directive instructions in its code. Therefore, an adaptive model should avoid using domain-specific features and instead acquire knowledge of common characteristics in malware executables. Both the cold-start and warm-start learning techniques do not exhibit these qualities.

Our Method. In this paper, we introduce a novel graph-based adversarial domain adaptation (DA) method to address malware drift, coupled with a new method for graph-based clustering that identifies statistically distant malware clusters for evaluation. At a high level, warm-start learning enhances cold-start learning by using a model that has been previously trained on existing malware. Our approach learns a new model directly from the existing and drifted malware samples simultaneously. To facilitate effective knowledge transfer to new samples, our methodology focuses on learning an intermediate representation containing information that remains consistent before and after the drift while still being sufficient to make a good classification. We use Control Flow Graphs (CFGs) derived from malware assembly, as they are more comprehensive and contain significantly more nodes than function call graphs. Then, we use a pre-trained assembly model [24] to generate embeddings for instructions in CFGs for neural network training. We introduce a training task, *Label Prediction (LP)*, based on the Graph Isomorphism Network (GIN) [50], to learn graph representations from graph structures, code semantics, and labels. To address the shift between old and new malware samples, we introduce another learning task involving the training of two networks through minimax optimization to predict the input domain (pre-drift or post-drift). This training task referred to as *Adversarial Training (AT)*, draws inspiration from generative adversarial networks [15] and DA for images [14], [22]. Our model automatically learns domain-invariant representations through those two training tasks using CFGs as the input. The domain-invariant representations are indistinguishable regarding whether they originate from pre-drift or post-drift data. If this representation allows us to achieve strong classification performance on the pre-drift data, it will also improve generalization on the post-drift data.

We also emphasize the importance of proper evaluation of techniques tackling concept drift. Previous research has frequently overlooked the validation of actual drift occurrence (particularly on research datasets), potentially resulting in an overestimation of the algorithm's predictive accuracy [20], [28], [31], [32], [44], [46], [49]. Despite varying experiment designs, these studies rely on the labels of malware families and typically exclude one label to serve as the ground truth for the "unseen family". However, closely related malware families might enable the algorithm to predict one family exceptionally well, even when trained on another, an issue similar to the temporal bias issue studied in [36]. In one of the most used Big-15 benchmarks [37], the distance between malware samples from different families is strikingly small. In this paper, we advocate for assessing the extent to which malware characteristics have changed within the dataset. We do so in all our experiments and provide distinct cluster labels for datasets where the original labels are not well separated, along with a new graph-based clustering algorithm for generating these clusters. For the clustering algorithm, we use an ensemble of clustering predictors. The clusters are then generated through a weighted consensus process that takes into account the performance of each predictor. Several clustering performance indexes demonstrate that this approach improves cluster assignments, effectively separating distant samples and grouping close ones.

Evaluation. We conduct experiments to evaluate our approach and compare it with leading graph-based malware classification models in both cold-start and warm-start learning settings. We also explore other DA methods to determine which DA techniques are most efficient for malware detection. Our initial experiments are conducted on malware classification research benchmarks, with one family as the target and the rest as pre-drift data. We used original family labels in Big-15 [37] and then with our cluster labels. We find that the model trained with warm-start learning has a performance decline of at most 5.8% - the highest-upon evaluation with cluster labels of Big-15. Conversely, our method experiences only a minor 0.5% performance decline on this dataset and outperforms all the others. We evaluate our design choice of using CFGs for malware representation against content-based [1], [11] and image-based [7], [33], [41], [44] representations on Big-15 research benchmark. Our adaptation technique proves effective across all representations, and the best results are achieved when combined with our graph representation. We also find that in both cold-start and warm-start learning scenarios, overall the graph representation surpasses the other two representations.

We use the MB-24 malware dataset, which we collected from the MalwareBazaar daily feed between March and September 2024, for real-world malware evaluation. Data from March to May serves as pre-drift data. In July, we labeled a fixed set of samples for post-drift training, updating the model. This updated model is tested using August data. Then, we labeled a few samples in August for further updates and tested the new model with September data. On average, we matched the accuracy of the upper bound results obtained through supervised training for each testing month while reducing the labeling effort by 5X. In subsequent experiments using the MalwareDrift dataset, our method improved family-level classification by $9 - 14\%$ over the state-of-the-art with just 10 new samples per family. This was achieved in a closed-set DA scenario, where pre-drift and post-drift samples are from the same malware families. We further demonstrate that our approach effectively extends to open-set DA, where new malware families appear in the post-drift data. Our final experiment demonstrates that our DA method performs better by learning features that reduce the distribution divergence between pre-drift and post-drift data.

Contributions. This paper has three main contributions.

- We introduce a novel model training method based on CFG and DA to address the concept drift problem in Windows malware classification. Our method fits in both closed and open-set scenarios.
- We highlight the limitations of previous work that conducted evaluations on research datasets without verifying

actual drift occurrences. We propose a new graph-based clustering method that computes statistically distinct malware clusters to eliminate bias.
- We extensively evaluate many previously proposed methods for model updating to determine the most effective solution—encompassing both malware representation and training methods—against malware drift, an area that has not been studied before. The results demonstrate significant improvements over previous work. We have released the code and data to support future research, with details provided in the Artifact Appendix.

II. Methodology

We leverage the information from existing labeled malware samples (source data[1]) to aid in the classification of partially labeled new malware samples (target data[2]). In this section, we first formally define the research problem and introduce the notations used throughout the rest of the paper. Next, we demonstrate how our suggested approach effectively addresses the challenges outlined in Section I.

A. Problem definition

Given a binary, we use its representation as a control flow graph (CFG). A CFG is a directed graph where vertices represent sequences of assembly instructions, and edges represent the execution flow. Each vertex represents a basic block; in what follows, we use the terms "basic block" and "node" interchangeably. Figure 1 shows a binary code snippet alongside its corresponding CFG. Each instruction is converted to a vector. We average the instruction embeddings to obtain the node attribute.

The source data is represented as $G^s = \{G_i^s\} = \{(X_i^s, A_i^s, Y_i^s)\}$, where $X_i^s \in \mathbb{R}^{n_i^s \times m^s}$ is the node attribute matrix for G_i^s with n_i^s being the number of nodes and m^s being the number of node attributes in the source data. Additionally, $A_i^s \in \mathbb{R}^{n_i^s \times n_i^s}$ is the adjacency matrix with $A_i^s(p, q)$ denoting the number of edges between node p and q, and Y_i^s is the one-hot encoding of the classification label for G_i^s.

Similarly, the target data is represented as $G^t = \{G_i^t\} = \{(X_i^t, A_i^t, Y_i^t)\}$, where $X_i^t \in \mathbb{R}^{n_i^t \times m^t}$ is the the node attribute matrix for G_i^t with n_i^t being the number of nodes and m^t being the number of node attributes, $A_i^t \in \mathbb{R}^{n_i^t \times n_i^t}$ is the adjacency matrix, and Y_i^t is the one-hot encoding of the label for G_i^t. Furthermore, let $|G_i^s|$ represent the number of samples from the source domain and $|G_i^t|$ represent the sample size in the target domain. We assume that $|G_i^s| \gg |G_i^t|$.

The source and target data contain the same attributes, that is $m^s = m^t$. The number of attributes is adjustable and determined in Section II-D, where one can indicate the dimension of the node attribute matrix.

We now formally state our research problem as follows: *A divergence exists between the source and target malware data, yet the label space remains consistent ($Y^s = Y^t$). Our main objective is to develop a classifier that can effectively identify drifted malware in the target domain with very few labeled graphs from the target.* We formulate our research problem as a closed-set prediction task where the source and target have the same label space. In this work, we also show that our approach can be extended to fit in the open-set scenario where the target may have new malware classes not in the source (see Section VII-B for details).

[1]The terms "source domain" and "source data" are used interchangeably.
[2]The terms "target domain" and "target data" are used interchangeably.

B. Overview

Figure 1 illustrates the design of our approach, which is composed of three key components: CFG Construction from ASM Files, Vertex Feature Extraction, and Shift Adaptation. Initially, we disassemble the malware binary to extract the CFG from the assembly code. Each node in the graph represents a basic block in the assembly code, while edges represent jumps in the control flow. Moving to the second component, we employ a pre-trained language model to generate high-quality node embeddings. After that, we train our model with source data and limited target data (see Figure 2). The model comprises two main components: the domain prediction component, consisting of a generator and a discriminator, and the classification component, composed of a generator and a classifier. During the optimization process, the shared generator can learn features that combine class distinctiveness and domain invariance. This empowers the classifier to classify data in the target domain with the assistance of the source data and only a limited set of labeled target data. Once the model has been trained, we deploy it for prediction. In what follows, we introduce the approach for constructing CFGs, followed by the vertex feature extraction process. Finally, we present our method for training a graph neural network (GNN) with adversarial DA for drift malware adaptation.

C. CFG construction from disassembly

We first disassemble binary files using IDA Pro. We use the open-source code from MAGIC [51] for CFG extraction. The algorithm employs a two-pass traversal methodology. Initially, the file is processed to create a mapping from addresses to instructions. Instructions are associated with four tags, i.e., {`start`, `branchTo`, `fallThrough`, `return`}, which are used by the second pass. In the first pass, each instruction is visited and its associated tags are updated accordingly. The second pass is dedicated to creating basic blocks and edges between these blocks based on the assigned tags.

D. Vertex feature extraction

To apply deep learning to the CFGs, the first step is to extract feature vectors for the instructions, as GNNs cannot directly operate on raw instructions. Among the available approaches – directly feeding raw bytes, employing manually designed features, or automatically generating vector representations for individual instructions using a designated representation model – the preference is given to the third approach. This approach generally yields better-quality embeddings without requiring a manual selection of the features. PalmTree [24] is a pre-trained assembly language model based on BERT for general-purpose instruction representation learning. Experimental results [24] show its efficacy in generating high-quality instruction embeddings for various downstream binary analysis tasks. Hence, we use the PalmTree model with pre-trained parameters to generate instruction embeddings.

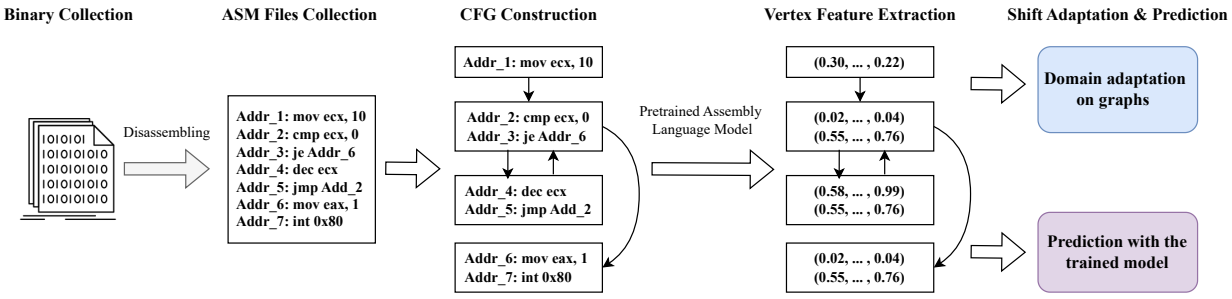

Fig. 1. Overview of our approach: we show the assembly code on the left and the corresponding control flow graph on the right.

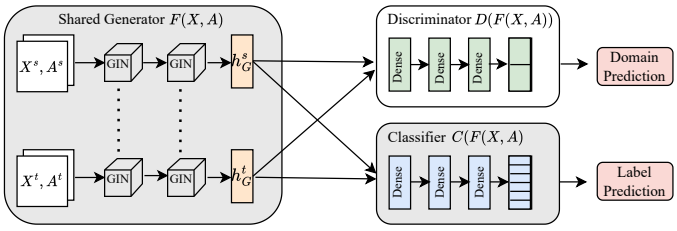

Fig. 2. The inputs are source and target graph data, represented as node attribute matrix ($X^{s/t}$) and adjacency matrix ($A^{s/t}$). We obtain those two matrices for each graph after the Vertex Feature Extraction step in Figure 1. The training process is modeled as a minimax game between the generator and the discriminator. h_G^s and h_G^t denote the graph-level representations corresponding to the source and target inputs, respectively. Following the training process, the discriminator fails to discern the domain distinction solely based on h_G^s and h_G^t. At the same time, they retain useful information crucial for achieving good classification in both domains.

1) Normalization: In NLP, the Out-of-Vocabulary (OOV) problem occurs when a token encountered during inference is absent in the vocabulary the model was trained on. The OOV issue in assembly code encoding is particularly challenging given the diverse nature of codebases and the potential for encountering unseen strings and constant numbers. To address this issue, our initial step involves normalizing the CFG: strings are substituted with a designated token `[str]`. Constants, which contain a minimum of five hexadecimal digits, are replaced by a specific token, `[addr]`. Smaller constants remain intact and are encoded as one-hot vectors. These strategies align with those utilized during PalmTree's training.

2) Node feature embedding: After normalization, we apply the PalmTree model to generate an embedding for each instruction. We still need to aggregate vectors to obtain a single feature vector to be associated with a vertex. We adopt an efficient yet effective solution based on an analysis of the related literature [30]. We aggregate all the instruction embeddings and compute an unweighted mean vector. This process is repeated for all basic blocks, yielding the final CFG. Each processed CFG is represented by $G = (X, A, Y)$, where $X \in \mathbb{R}^{n \times m}$ is the node attribute matrix for G with n being the number of nodes and m being the dimension of each node feature vector, $A \in \mathbb{R}^{n \times n_i}$ is the adjacency matrix, and Y is the one-hot encoding of the label for G.

E. Shift adaptation: model design

1) Representation learning: Learning with graphs requires effectively representing their structures and node attributes. GNNs are an effective framework for the representation learning of graphs. They generally adopt a recursive aggregation approach where each node combines the feature vectors of its neighbors to derive its updated feature vector. With k iterations of aggregation, a node is represented by a node feature vector, encapsulating the structural information within its k-hop neighborhood. The graph-level representation is achieved via a pooling function. Formally, the k-th iteration of a GNN is constructed as:

$$h_v^{(k)} = \text{COMBINE}\left(h_v^{(k-1)}, a_v^{(k)}\right) \quad (1)$$

$$a_v^{(k)} = \text{AGGREGATE}\left(\left\{h_u^{(k-1)}, \forall u \in \mathcal{N}(v)\right\}\right) \quad (2)$$

where $h_v^{(k)}$ is the feature vector of node v at the k-th iteration. We initialize $h_v^{(k)} = x_v$, where x_v is the feature vector for node v. $a_v^{(k)}$ denotes the aggregation of features vectors from v's neighbours at the $(k-1)$-th iteration. $\mathcal{N}(v)$ are the neighbours to v. The selection of AGGREGATE and COMBINE differs among GNN variants. In Graph Isomorphism Network (GIN), which we use in this work, these steps are integrated as follows:

$$h_v^{(k)} = \text{MLP}\left(\left(1 + \epsilon^{(k)}\right) h_v^{(k-1)} + \sum_{u \in N(v)} h_u^{(k-1)}\right) \quad (3)$$

where ϵ represents a learnable parameter. We designate ϵ as 0, a configuration referred to as GIN-0 in [50].

The graph representation h_G is obtained by aggregating node features from the final iteration K:

$$h_G = \text{READOUT}\left(\{h_v^{(K)} | v \in G\}\right) \quad (4)$$

READOUT can be either a summation function or a graph-level pooling function. It is important to note that relying solely on node representations from the final layer may limit performance, as features from earlier iterations can sometimes generalize better [50]. To address this, we adopt a method similar to Jumping Knowledge Networks, wherein we concatenate graph representations across all iterations:

$$h_G = \text{CONCAT}\left(\text{READOUT}\left(\{h_v^{(k)} | v \in G\}\right) | k = 0, 1, ..., K\right) \quad (5)$$

In the next subsection, we present the loss function designed to obtain graph representation h_G.

2) Loss functions: A main goal for the successful DA is to generate a domain-independent representation of the data from different domains, that is, graph representations (h_G) from the source domain that are similar to those from the target domain. This allows classifiers trained on graphs from the source domain to generalize to the target domain, as the inputs to the classifier are invariant with respect to the domain of origin. To generate such representation, we leverage the domain adversarial training approach [14], [21] that adversarially trains two neural networks to ensure the representation is domain invariant. Those two networks serve as a discriminator and a generator, respectively. The generator is trained adversarially to maximize the discriminator's loss. To learn distinctive graph representations for classification, the generator is also trained with a label predictor.

More specifically, the source inputs are given by $\{(X_i^s, A_i^s, Y_i^s, d_i^s)\}$, where each element represents node matrix, node adjacency matrix, class label, and domain label. The target is given by $\{(X_i^t, A_i^t, Y_i^t, d_i^t)\}$. The domain label $d_i \in \{0, 1\}$ is the ground truth domain label for sample i. Let $F(X, A; \theta_f)$ be the generator parameterized by θ_f, which maps a node adjacency matrix A and a node attribute matrix X to a hidden graph representation, h_G, representing graph features that are common across domains. Let $D(h_G; \theta_d)$ be the discriminator that maps a hidden representation h_G to the domain-specific prediction. Finally, $C(h_G; \theta_c)$ represents a classifier, parameterized by θ_c that maps from a hidden representation h_G to the task-specific prediction. The resulting model is shown in Figure 2.

Inference in our model is given by $\hat{Y} = C(F(X, A; \theta_f); \theta_c)$ and $\hat{d} = D(F(X, A; \theta_f); \theta_d)$ where $\hat{Y}$ is the label prediction and $\hat{d}$ is the domain prediction. The goal of training is to minimize the following loss with respect to parameters $\theta_f, \theta_c, \theta_d$:

$$min_{\theta_f, \theta_c}\{\gamma \mathcal{L}_g + \mathcal{L}_c\} \tag{6}$$

$$min_{\theta_d}\{\mathcal{L}_d\} \tag{7}$$

where γ is the weight that controls the $\mathcal{L}_g$ term. The classification loss $\mathcal{L}_c$ trains the model to predict the output labels. Because we assume that the target domain has limited labels, the loss applies to both domains, and it is defined as follows:

$$\mathcal{L}_c = -\sum_{i=1}^{N_s} Y_i^s \cdot log\hat{Y}_i^s - \lambda \sum_{i=1}^{N_t} Y_i^t \cdot log\hat{Y}_i^t \tag{8}$$

where N_s represents the number of samples from the source domain, $\hat{Y}_i^s$ is the softmax output of the model: $\hat{Y}_i^s = C(F(X_i^s, A_i^s))$. N_t represents the number of samples from the target domain, $\hat{Y}_i^t$ is the softmax output of the model: $\hat{Y}_i^t = C(F(X_i^t, A_i^t))$. We use λ as the penalty coefficient for the loss value obtained from target data.

The discriminator loss trains the discriminator to predict whether the output of F is generated from the source or the target domain. Let $\hat{d}_i^s = D(F(X_i^s, A_i^s))$ and $\hat{d}_i^t = D(F(X_i^t, A_i^t))$ be the domain predictions from samples of source and target, respectively. The discriminator loss is also applied to both domains, and it is defined as follows:

$$\mathcal{L}_d = -\sum_{i=1}^{N_s+N_t} \Big[d_i log\hat{d}_i + (1 - d_i) log(1 - \hat{d}_i) \Big] \tag{9}$$

Finally, the generator loss encourages the hidden graph representations h_G^s and h_G^t from the generator to be as similar as possible so the discriminator cannot predict the domain of the representation. This is achieved via adversarially training the generator so that parameters θ_f are optimized to reduce the domain classification accuracy. Essentially, we minimize the loss for the domain prediction task with respect to θ_d, while maximizing it with respect to θ_f. Hence, the generator loss is defined with inverted ground truth domain labels:

$$\mathcal{L}_g = -\sum_{i=1}^{N_s+N_t} \Big[(1 - d_i) log\hat{d}_i + d_i log(1 - \hat{d}_i) \Big] \tag{10}$$

3) Model training: Training F consists of optimizing $\mathcal{L}_g$ and $\mathcal{L}_c$, since we want to minimize the domain classification accuracy and maximize label classification accuracy. The discriminator is trained with $\mathcal{L}_d$ to maximize domain classification accuracy. The classifier is trained with $\mathcal{L}_c$ to maximize label prediction accuracy. The training algorithm follows the mini-batch gradient descent procedure. More specifically, the following steps are executed after creating the mini-batches. The generator updates its weight to minimize $\gamma \mathcal{L}_g + \mathcal{L}_c$. The classifier updates its weight to minimize classification loss. The discriminator weights remain frozen during this step. Then, the discriminator updates its weight to minimize discriminator loss. Upon the model's convergence, we can achieve graph representations that are both discriminative of the class and invariant to the domain. To classify graphs in the target, one can obtain prediction by running $C(F(X, A))$.

III. Generating Drifted Malware Clusters

Past approaches to malware drift were evaluated using existing malware research datasets. Despite varying experiment designs, they all rely on the original malware family labels for their analysis. In this paper, we do so as well. However, we discovered that despite being labeled as different families, malware samples exhibit highly similar characteristics in one of the most used benchmarks. This phenomenon is evidenced by the data in Table I, which shows that the distances between malware samples from different families are remarkably small. Evaluation relying on those family labels is likely to overestimate the accuracy of the prediction. Consequently, we developed and implemented a graph-based clustering algorithm to assign cluster labels to malware, thereby increasing inter-cluster distance and amplifying domain shift. We conducted our experimentation on Big-15 in two scenarios to demonstrate the effect of performance overestimation: one evaluation using the original labels (Section IV-A), and another using the labels obtained by using clustering (Section IV-B).

The graph-based clustering algorithm comprises two primary components: (1) the graph embedding component, responsible for learning a feature vector at the graph level, and (2) a weighted consensus clustering mechanism that operates on the learned graph embeddings using an ensemble of clustering predictors. The resulting clusters are generated via a weighted consensus method that takes into account the performance of every single predictor.

Graph Embedding. The goal is to learn a graph representation able to preserve the graph structure and code semantics. The representations learned by minimizing Equation 6 are

not suitable for this task since they only contain information that is helpful for classification and would filter out other information that is important for capturing the characteristics of the entire graph. Therefore, we consider solutions based on the graph autoencoder (GAE) [19], [40], which consists of an encoder that learns a hidden representation and a decoder that can reconstruct the entire graph from this representation. Our implementation of this model involves utilizing a graph convolutional network (GCN) as the encoder and a simple inner product as the decoder.

In particular, we calculate graph embedding Z and the reconstructed adjacency matrix $\hat{A}$ as follows:

$$\hat{A} = \sigma(ZZ^\top), \; Z = GCN(X, A) \tag{11}$$

where $\sigma(\cdot)$ is the sigmoid function. We use binary cross entropy loss for reconstruction loss, which is used to train both the encoder and decoder.

$$\mathcal{L}_{recon} = -\sum_{i,j} \Big[A_{ij} log(\hat{A}_{ij}) + (1 - A_{ij}) log(1 - \hat{A}_{ij}) \Big] \tag{12}$$

where A_{ij} is 1 if there is an edge between node i and j, and 0 otherwise. After training the graph autoencoder with $\mathcal{L}_{recon}$ on graph $G = (X, A)$, we derive the graph representation Z from the encoder: $Z = GCN(X, A)$, and then retrain the graph autoencoder from scratch for the next graph to get its graph representation.

Weighted Consensus Clustering. After obtaining an embedding for each graph, the next step is to assign a new label to each graph based on unsupervised clustering. In this paper, we introduce a novel weighted consensus clustering approach that leverages multiple clustering algorithms and assigns more weight to predictors that yield better clustering results.

First, we apply P clustering algorithms to the graph embeddings, resulting in P individual clustering solutions. Each solution assigns every graph to a specific cluster. In total, we have P label solutions plus the original label if available. Next, we initialize a consensus matrix CM with zeros. We iterate through all the solutions and update the consensus matrix on the fly. For every pair of graphs i and j, if i and j belong to the same cluster, we update the matrix CM using the formula:

$$CM[i][j] \mathrel{+}= 1 \times \frac{(s - (-1))}{2} \tag{13}$$

Here, s denotes the silhouette coefficient of the current clustering solution, which is the mean of the silhouette coefficient for all samples. The averaged silhouette coefficient ranges from -1 for incorrect clustering to $+1$ for well-separated clustering. A superior clustering solution, characterized by a higher silhouette score, exerts a higher coefficient (s), hence a greater impact on the CM matrix. Finally, we apply a clustering algorithm to the normalized matrix CM to derive the final clusters, assigning each graph a new cluster label.

IV. Evaluation: Research Malware Dataset

In this section, we evaluate our approach using a popular Windows malware benchmark: the Microsoft malware classification challenge (Big-15). We focus the evaluation on the detection of previously unseen malware families. Our assessment is conducted in two scenarios: one with the original labels and another using the labels generated by our graph cluster algorithm. We evaluate different malware representations and current retraining approaches for each representation in Section V. We also test our approach on a real-world malware dataset collected from MalwareBazaar's malware database in 2024 (see Section VI). Finally, we evaluate the family-level classification (see Section VII).

A. Evaluation based on original labels of Big-15

1) Dataset: The Microsoft Malware Classification Challenge (Big-15) [37] is one of the most used benchmarks for testing malware classification methods. In total, it has $21,741$ malware samples where $10,868$ samples are labeled. Those labeled samples are from nine different malware families, namely Ramnit, Lollipop, Kelihos_ver3, Vundo, Simda, Tracur, Kelihos_ver1, Obfuscator.ACY, Gatak. In this experiment, we created CFGs from $10,868$ samples.

For the collection of Windows PE benign samples, as there is no dataset of benign binaries available due to copyright issues, we adopt the commonly used collection method [26]. We utilized virtual machines with clean installations of Windows 11, 10, and 8, along with common Windows applications. The resulting PE files were collected as benign samples, yielding a total of $16,000$ samples.

2) Source and target datasets setup: We design our experiments using the "leave-one-out" method to simulate an unseen malware family. We pick one of the malware families as the previously unseen family (serving as the target malware data), whereas the remaining families serve as the source malware data. We ensure the general applicability of our results by repeating this process with Ramnit, Lolipop, and Kelihos_ver3, the dataset's top three malware families.

Both the source and the target datasets contain benign samples. We randomly split $8,000$ benign samples as the benign data for the source domain, while the target domain uses the rest of the $8,000$ benign samples. Ultimately, we successfully disassembled $6,510$ PE files from the source benign data, achieving an approximate malware-to-benign ratio of $1.2:1$ in the source dataset. The unseen family is combined with $5,768$ successfully disassembled benign Windows PE samples to form the target dataset, maintaining an approximate malware-to-benign data ratio of $0.35:1$.

Since the Big-15 dataset does not provide sample timestamps, we split the source and target data conventionally. The source and target datasets are randomly split into source training set (75%), source testing set (25%), target training set (50%) and target testing set (50%), respectively. To prevent spatial bias [36], the malware/benign class ratio is consistent across training and testing sets in all domains. Our model is trained using the labeled source training set and a subset of the labeled target training set, then evaluated on the target testing set, which is unavailable during training. In continuous learning, a labeling budget limits the number of samples analysts can label. We simulate this by randomly selecting $20, 50, 100, 200, 300, 500$ labels from the target training set and using them as the target training data. In all experiments, the fraction of selected training samples is at a maximum of 50% of the total target training set. We aim to detect

malware in the target testing set with minimal labels, focusing on comparing model updating methods rather than improving performance through advanced sample selection. We chose random sampling, aware of its inefficiency, to demonstrate that if our method performs well with random sampling, it will excel with more sophisticated sampling techniques.

3) Baselines: We evaluate our scheme against existing graph-based malware classification methods and several enhanced baselines, which are adapted from previously published work with our improvements. This allows us to provide evidence that our method outperforms all these approaches.

Baseline Methods. MAGIC [51]: Yan et al. [51] developed MAGIC, employing DGCNN, a type of GNN, to classify CFGs with basic blocks serving as nodes. MAGIC utilizes manually crafted token-level and block-level features to represent each basic block. Table V in the Appendix shows the features defined in MAGIC.

MCBG [47]: MCBG is another CFG-based malware classification model. It adopts a pre-trained BERT model to generate node embeddings instead of relying on handcrafted features. MCBG also adopts GIN as its classification model.

MCBG and MAGIC are designed for supervised learning, assuming sufficient labeled data samples. Therefore, we employ cold-start learning (i.e., we train a classifier from scratch with target training labeled samples), which is consistent with past work. We directly adopt their available public implementations with the default hyper-parameter. The details can be found in Appendix A-A2.

Improved Baseline Methods. Warm-start MAGIC and MCBG: We extend those approaches with warm-start learning. Rather than training a new model from scratch each time, we train the model with the source training data and continue training it with target training samples. The details of the implementation and retraining procedure are given in Appendix A-A2.

DAN [27]: DAN was designed to generalize to test images different from those in the training set. Specifically, it learns a domain-independent representation by reducing the discrepancy in domain distribution. This discrepancy is quantified using the Maximum Mean Discrepancy (MMD) loss. We adapt DAN to support graph-based malware classification models. It learns a domain-independent graph representation via a shared feature extractor, which is based on GIN. Subsequently, we implement a classifier that uses this graph representation as input to determine if it is malware. The feature extractor is trained to minimize both the classification loss and the MMD loss, while the classifier is trained to minimize the classification loss. The MMD loss is computed using the hidden graph representations h_G^s and h_G^t derived from the feature extractor. Following previous approaches, we opted for RBF as the kernel function. To ensure a fair comparison, the architecture of the feature extractor and classifier mirrors that of the generator and classifier components in our method. See the Appendix for the details on the implementation of DAN A-A2 and our approach A-A1. Both our approach and DAN aim to produce indistinguishable representations across domains. DAN uses MMD loss, a kernel-based distance function minimizing the disparity between the hidden representations of source and target samples, thereby achieving distribution matching. Our method is very different as it measures the disparity between distributions based on their separability by a neural network (discriminator). In image classification tasks, discriminator loss has shown superior performance compared to DAN [14].

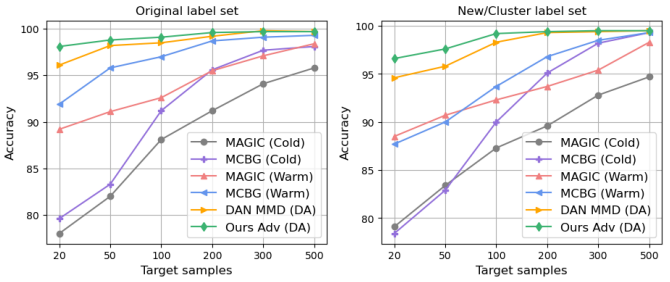

Fig. 3. Given a set of fixed target training labels, we compute the accuracy of the target testing data for different baseline techniques and our method. The left diagram reports the averaged accuracy based on the original label set of Big-15, and the right one reports results based on the cluster label assignment.

4) Results: We run each experiment five times and compute the average results across all five iterations. The accuracy of the three target families is shown in the top three graphs in Figure 12 in the Appendix. The F1 score can be found in Table VII in the Appendix. Here, we present the averaged results from the three target families in Figure 3. We observe that:

- Our approach has the best accuracy in all experiments. It achieves over 98% average accuracy when only trained with 20 labeled samples from the target, demonstrating that our adaptation approach can effectively adapt to the new malware family with very few labeled samples.
- DA methods (including DAN MMD and ours) perform better than the warm-start learning strategy, while the cold-start training yields the least effective results when trained with the same amount of target labels. In the malware-detection task, the discriminator loss demonstrates superior performance compared to DAN as well.
- MCBG (Warm) attains 91% accuracy with just 20 samples. This is primarily due to the lack of clear separation among different family labels, which we further verify by the experiments presented in Section IV-B.

In the next experiment, we initially demonstrate that the original labels of Big-15 are not well separated using common inter-label distance metrics, potentially resulting in an overestimation of the warm-start learning and DA methods. We create denser clusters for the same dataset using the graph-based clustering algorithm presented in Section III and evaluate all the methods on this more challenging adaptation task.

B. Evaluation based on cluster labels of Big-15

1) Inter-label distance: We refer to three common metrics to measure the distance among labels: Silhouette Coefficient [38], Calinski-Harabasz Index [8], and Davies-Bouldin Index [12]. These metrics assess whether clusters are dense and well-separated. The Calinski-Harabasz Index scores higher for dense, well-separated clusters, whereas the Davies-Bouldin Index suggests better partitions when it approaches zero. These metrics are not directly applicable to graphs, so we use the

TABLE I. EVALUATION OF THE ORIGINAL LABELS AND NEW CLUSTERS OF BIG-15 WITH DISTANCE-BASED METRICS. ALL METRICS CONFIRM THAT OUR NEW CLUSTERS BETTER SEPARATE DATA, WITH DISTANT SAMPLES IN DIFFERENT CLUSTERS AND CLOSE SAMPLES IN THE SAME CLUSTER.

Metric	Original labels	New clusters
Silhouette Coefficient	-0.112	-0.036
Calinski-Harabasz Index	0.540	61.150
Davies-Bouldin Index	40.210	29.511

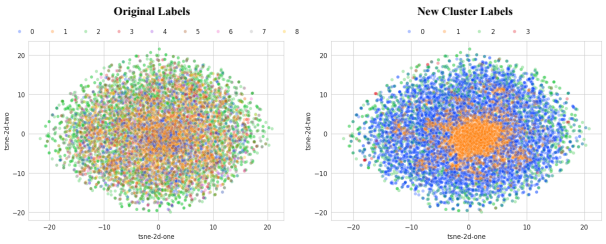

Fig. 4. Visualization of the graph feature vector with their labels. The left part of the figure shows the data with the original labels from Big 15 [37], and the right one shows the newly learned clusters. The legend represents the mapping between labels and colors.

method from Section III to learn graph embeddings, compute metrics from transformed vectors and original labels, and present results in Table I. All three metrics indicate poor separation of samples with different labels. We also give a visual insight into the poor data partition based on the original label. The visualization is shown in Figure 4.

2) Obtaining well-separated clusters: We implemented the graph clustering algorithm in Section III to obtain the new cluster labels for Big-15. See Appendix A-A3 for the implementation details. Figure 4 illustrates the new clusters, while Table I presents the metric values for the new labels.

3) Source and target datasets setup: We still employ the "leave-one-out" strategy and pick one of the cluster labels as the "unseen family" and the remaining clusters as the source malware data. We still use the same source benign data and target benign data. The train/test split ratio for each domain remains consistent with those detailed in Section IV-A2.

4) Results: We use the same baselines as in the previous experiments. Similarly, we conducted three adaptation tasks, each targeting a specific cluster (we have picked cluster 0, cluster 1 and cluster 2 as the target family). We randomly sampled $20, 50, 100, 200, 300, 500$ labels from the target training set for each task. The full accuracy result and F1 score are included in Figure 12 and Table VIII in the Appendix. We present the averaged results from those three cases in Figure 3. To understand the impact of the more accurate labels on different training strategies, we calculate the difference in the averaged accuracy of each approach based on the original label and cluster label and show the results in Table VI in the Appendix. The highlight results are:

- Our approach maintains the highest accuracy across all experiments and demonstrates robust performance even in this more challenging adaptation task. It achieves an average accuracy of 96% when trained with only 20 labeled samples from the target domain.
- In addition, the experiments show that the DA methods yield the highest performance, with warm-start learning trailing behind the DA methods and cold-start learning producing the lowest outcomes.
- MCBG (Warm) experienced the most substantial performance decline due to a more pronounced distribution shift between the source and target domains. When assessed on actual drifted labels, MCBG (Warm) saw an accuracy decrease of at most 5.8%. Both DAN and MAGIC (Warm) only experienced an average accuracy decrease of 0.8% and 0.9%, respectively. Our method experienced the smallest drop of 0.5% on average.

V. EVALUATION: IMPACT OF DIFFERENT REPRESENTATIONS

In this section, we conduct a comprehensive analysis of the effectiveness of various combinations of malware representation with diverse model update strategies. We consider the existing cold-start and warm-start learning approaches, as well as novel approaches grounded in DA. Our objective is to answer the following research questions:

Q1: How effective is our adaptation technique when dealing with different malware representations, such as those that are content-based or image-based? Is our method the best for updating the model using these two representations?

Q2: Does our graph representation contribute to the overall model's performance? If so, to what extent?

Q3: In terms of cold-start and warm-start learning, which form of malware representation is more effective?

To answer those questions, we use the Big-15 dataset and the Windows benign dataset, obtain three representations (content-based, image-based, graph-based) for each malware sample, and test with various baselines for each representation. For the malware data, we use the new cluster label as it poses a more challenging adaptation task. The configuration of the source and target datasets adheres to the same process outlined in IV-B3 and remains consistent across all representations. In what follows, we first describe the representation we consider. Then, we discuss the implemented baselines for each representation, followed by the results of our evaluation.

1) Content-based features and baselines: We select nine families of static analysis features that were curated in prior research works [1], [2]. In total, we extracted a total of 965 individual features from the assembly code sample in our datasets. A summary of the selected features is included in Appendix A-B.

Baseline Methods. SVM and MLP in both cold and warm settings: In continuous learning for malware detection, SVM and MLP are commonly used classifier models to learn from tabular features extracted from malware samples [9], [53]. Chen et al. [9] further enhanced this approach by incorporating warm-start training, which they found to be superior to cold-start for malware detection models. The details of each classifier can be found in Appendix A-C.

Improved Baseline Methods. DAN: We modify DAN to accommodate MLP-based models for malware classification. Both the feature extractor and the classifier are implemented

as multi-layered feedforward neural networks, and their combined structure mirrors the layer configuration of the baseline MLP. The feature extractor is trained to minimize both the classification and MMD losses, while the classifier's training focuses on minimizing the classification loss. The computation of the MMD loss involves the hidden representations obtained from the feature extractor. Consistent with prior research, we choose RBF for kernel functions.

For our adaptation method, to maintain a fair comparison, we designed the architecture of the generator and classifier to match the topology of the feature extractor and classifier modules in DAN. The key distinction is that our domain representation is acquired via adversarial learning loss in Equation 6 and 7, and not through MMD loss. See Appendix A-C for details on the implementation of DAN and our approach.

2) Image-based features and baselines: We adopt the same approach as in [6], [28], [44] to transform a malware binary into an image. This approach does not require any feature engineering and domain expert knowledge. It reads a given binary as a vector of 8-bit unsigned integers and then converts a vector into a 2D array. The height of the malware image is allowed to vary depending on the file size, and the width of the malware image is fixed. We apply this process to our datasets to obtain images of binaries.

Baseline Methods. Cold-start ResNet-50: We follow the previous works, which use several deep convolutional neural networks for malware prediction. We choose to implement the network components as ResNets-50 with short-cut connections since it shows the best prediction performance with images of malware [28]. In our first baseline, we train a ResNet-50 model from scratch using only target training samples, following approaches in [33], [41] where no adaptation is involved. The implementation and training details can be found in Appendix A-D.

Warm-start ResNet-50: This approach aligns with the predominant methodologies employed in classifying malware images with concept drift [7], [20], [28], [44]. Specifically, we utilize the same customized ResNet-50 architecture described earlier but initialize the model with weights from pre-training on ImageNet [13]. Subsequently, we retrain the model using both source and target image data. This strategy not only leverages the network's knowledge of general images but also integrates insights learned from the source malware data.

Improved Baseline Methods. DAN: Our implementation of DAN and our methodology utilize a simple multi-layered CNN without using any pre-trained models. This demonstrates that the DA techniques surpass both the cold-training and warm-training approaches, even when using smaller models trained from scratch. For an in-depth explanation of the implementation of DAN and our method, please refer to Appendix A-D.

3) Results: Each baseline is assessed across three adaptation tasks, with each task leaving one cluster out as the target, and then we report the average from three tasks. The averaged accuracy of all feature representations alongside their respective baselines is reported in Table II. The F1 score can be found in Table IX in the Appendix. We have omitted the cold and warm strategies from the table for the graph representations as they are shown in Figure 3. Instead, we have

TABLE II. AVERAGED ACCURACY OF BASELINES WITH VARIOUS MALWARE REPRESENTATIONS. FOR THE CONTENT AND IMAGE REPRESENTATION, WE REPORT THE PERCENTAGE CHANGE IN ACCURACY FROM OUR ADVERSARIAL (ADV) DA METHOD TO THE PEAK PERFORMANCE AMONG ALL BASELINES WITHIN THE SAME REPRESENTATION. ULTIMATELY, WE DEMONSTRATE THE IMPROVEMENT OF OUR FULL PIPELINE (GRAPH + ADV DA) COMPARED TO BOTH CONTENT + ADV DA AND IMAGE + ADV DA.

Malware Representation	Strategy	Method	Target samples 20	50	100	200	300	500
Content-based (CB)	Cold	SVM	67.2	71.4	75.4	79.3	83	85.6
		MLP	75.9	79.4	85.3	91.6	92.9	94.9
	Warm	SVM	70.2	74	79.1	83.2	86.7	90.5
		MLP	**91**	93.4	95.4	96.2	**97.4**	**97.9**
	DA	DAN (MMD) + MLP	90.8	**93.9**	**95.4**	**96.9**	97.1	97.5
		Ours (Adv) + MLP	**94.1** ↑3.1	**95.2** ↑1.3	**96.2** ↑0.8	**97.1** ↑0.2	**97.7** ↑0.3	**97.8** ↓0.1
Image-based (IB)	Cold	ResNet-50	60.6	60.6	70.6	74.3	77	84.4
	Warm	ResNet-50	**86.2**	**87.7**	89.5	**91.9**	**93.5**	**94.4**
	DA	DAN (MMD) + CNN	85.4	87.5	**89.9**	91.6	93.1	94
		Ours (Adv) + CNN	**88.6** ↑2.4	**90** ↑2.3	**92.2** ↑2.3	**93.1** ↑1.2	**94** ↑0.5	**94.6** ↑0.2
Graph-based (GB)	DA	Ours (Adv) + GIN	**96.6**	**97.6**	**99.2**	**99.4**	**99.5**	**99.5**
Improvement over CB: Ours (Adv)			↑2.5	↑2.4	↑3	↑2.3	↑1.8	↑1.7
Improvement over IB: Ours (Adv)			↑8	↑7.6	↑7	↑6.3	↑5.5	↑4.9

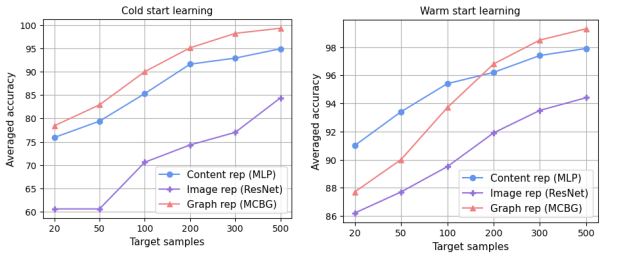

Fig. 5. Comparison of different representations under cold-start learning (left) and warm-start learning (right).

selected the top-performing method from each representation under both cold-training and warm-training settings for a direct comparison in Figure 5. The following are our observations in response to the initial questions:

- Our shift adaptation component consistently has the highest accuracy across all feature representations, demonstrating the versatility of our adversarial DA technique with different malware representations (**Q1**).
- All the key components in our pipeline contribute to the overall model's performance. Combining the adaptation approach with our graph representations achieves the best results. (**Q2**).
- In cold-start training, content-based and graph-based representations have similar performance, while the ResNet-50 image model using image features performs poorly, indicating the challenge posed by inadequate training data for large-scale neural network models. (**Q3**).
- In warm-start training, the image model shows the most significant improvement, and the graph representation surpasses the content-based features beyond 200 target samples (**Q3**).

VI. EVALUATION: REAL-WORLD MALWARE DATASET

In this section, we first show the evaluation of our method on a more recent real-world malware dataset with diverse

families using data split based on temporal information. Then, we show results on the effectiveness of our approach with obfuscated malware samples.

A. Detection performance on most recent malware samples

1) Dataset: The experiment aims to evaluate our method in a realistic setup using recent malware samples that reflect current trends. Each sample includes a timestamp to eliminate temporal bias [36]. Furthermore, rather than employing a "leave-one-out" approach, the drifting samples must be derived from diverse malware families.

Existing Windows malware datasets are outdated and do not meet our criteria. Unlike Android datasets like AndroZoo [3], there is no Windows malware dataset with original samples of both malware and goodware over the same period. The BODMAS dataset [52] provides only malware binaries from five years ago, and the EMBER dataset [4] only includes features from PE files, both malicious and benign, from seven years ago. EMBER's original samples are only accessible via VirusTotal and are not publicly available for download.

Therefore, we collected a new malware dataset from March 2024 to September 2024 using the MalwareBazaar daily feed[3]. After filtering non-PE samples and removing noisy samples with inconsistent labels, we created the MB-24 dataset. Detailed information, including the number of families and samples per month, is provided in Table III. We use the collected $16,000$ benign Windows PE files as the benign dataset.

TABLE III. SUMMARY OF THE MB-24 DATASET.

Summary	Mar	April	May	July	Aug	Sep
# Samples	1505	1080	1496	1618	1613	1337
# Malware Families	104	81	99	126	111	92

2) Source and target datasets setup: We follow the time-consistent data split used in [9] to simulate the update of malware detection models in practice. We use the data from March, April and May 2024 as the source malware set, the data in July 2024 as the target malware training set, and the data in August 2024 as the target testing set. In this way, we strictly follow the temporal constraint commonly used in the literature. We skipped June 2024 to ensure that malware distribution has drifted. This is demonstrated in the left part of Figure 6. We also tested another model update using the March-May 2024 data as the source malware set, the data in August 2024 as the target malware training set, and the data in September 2024 as the target testing set (right part of Figure 6).

We randomly split 8000 benign samples as the benign data for the source domain, while we evenly split the rest of 8000 samples evenly to July, August and September[4]. In the end, the source dataset has a malware-to-benign ratio of $0.6:1$, and the malware-to-benign data ratio in the target training and testing sets (July, August and September) is consistently about $0.8:1$ to prevent spatial bias.

[3]https://bazaar.abuse.ch/api/

[4]Since the benign PE files do not have timestamps, we randomly split the data conventionally. A total of $6,510$ PE files were successfully disassembled from the source benign data. Additionally, about $1,922$ benign PE files were successfully disassembled for each of July, August, and September.

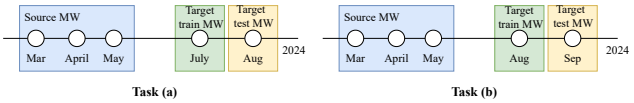

Fig. 6. Source and target malware datasets setup for the monthly model update in July 2024 (Task (a)) and August 2024 (Task (b)).

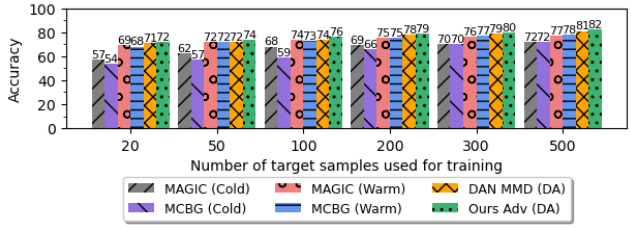

Fig. 7. The averaged accuracy on the target testing data using the experimental setup described in Figure 6.

3) Baselines: We use the same baselines of the previous experiments, with each baseline's implementation detailed in Appendix A-E1. For each task, $20, 50, 100, 200, 300, 500$ labels are randomly sampled from the target training set to serve as the target training data. The random sampling is restricted to the target training set, with July data used for testing in August and August data used for testing in September. The primary focus of this experiment is to compare different model updating methods, hence the use of random sampling. Should the proposed method perform well with random sampling, its performance will further improve with more sophisticated sampling techniques such as uncertainty sampling [9] or the application of a rejection-threshold [5], [53]. However, exploring these advanced techniques is beyond the scope of this work.

4) Results: We report the averaged accuracy of two adaptation tasks in Figure 7. The F1 score can be found in Table X in the Appendix. Our approach achieves the best results compared to all the baselines, followed by DAN and the warm-start training strategies. All methods demonstrate a performance decline on the real-world dataset compared to the research dataset due to the complexity of the dataset. We conduct experiments to find the upper bound of prediction so we can reasonably assess the performance of our approach. We train MCBG and MAGIC on August and September data separately, splitting each month's data into a training set (75%) and a testing set (25%). The models are trained on the training set and evaluated on the test set of the same month. MCBG achieves an average accuracy of 77% accuracy, while MAGIC achieves 79%. These results suggest that even if a human analyst labels 75% of the samples in August ($1209/1613$) and September ($1002/1337$), the prediction accuracy would be the same as labeling only 200 samples each month and using our approach. With the warm-start strategy, labeling 500 samples achieves the same result.

B. Impact of obfuscation

To demonstrate the effectiveness of our approach against obfuscated malware, we conduct a two-step experiment.

First step: We design the experiment using Task (b) in Figure 6 where the source malware data is from March to

May, the target training malware is from August, and the target testing malware is from September. We obfuscate all target testing malware samples with Hyperion[5], a runtime PE-Crypter. The source malware and target training malware samples remain unobfuscated. Neither the baselines nor our model are exposed to obfuscated samples during training. The goal is to compare the performance of each method trained with unobfuscated malware when testing on obfuscated malware.

Second step: We show that accuracy improves by adding a few obfuscated samples to the target training set. Specifically, the model is trained using the source data, 100 unobfuscated August samples, and either 10 or 20 obfuscated August samples, and then tested on the obfuscated September data.

The results of the first experiment are reported in Figure 13 in the Appendix, while the results of the second experiment are presented in Figure 8. The first experiment shows that our approach exhibits the lowest accuracy degradation when tested on obfuscated malware samples not encountered during training. In the second experiment, incorporating 20 obfuscated samples from August into the training set improves the accuracy of our approach by 11% in predicting the obfuscated samples in September. Although this experiment represents an initial exploration of testing the approach with obfuscated malware, it provides insights for real-world deployment. In practice, the robustness of the model can be enhanced by obfuscating existing samples using various obfuscation tools and incorporating them into the training process. The experiment indicates that this strategy can significantly enhance protection against obfuscated samples.

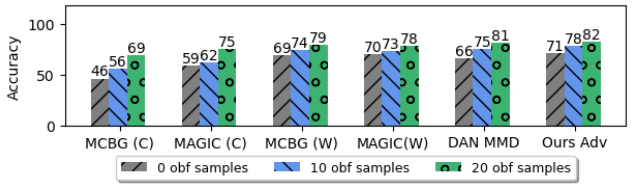

Fig. 8. Accuracy improvement of different methods tested on September data by adding 10 and 20 obfuscated samples from August to the 100 unobfuscated August target training data.

VII. Evaluation: Multi-Family Classification

In this section, we address malware classification at the family level. The classification process can be approached as a closed-set or an open-set DA problem. In the closed-set DA scenario, both the source and target domains contain identical malware families. Conversely, the open-set DA scenario allows for new malware families in the target domain that are not present in the source domain. We have conducted experiments under both closed and open-set conditions. Our findings demonstrate that our approach surpasses the current state-of-the-art methods in closed-set family-level classification and can be seamlessly adapted for open-set classification. Furthermore, we provide evidence supporting the superior performance of our approach, which effectively aligns samples from the same malware family across pre-drift and post-drift domains within the latent space. Additional details are provided in Section VII-C.

[5]https://nullsecurity.net/tools/binary.html

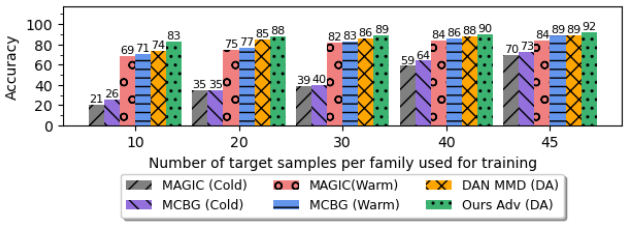

Fig. 9. Accuracy on post-drift data using $10-45$ labeled data from each post-drift family. Table XI in the Appendix shows the F1-score.

A. Closed-set DA

1) Dataset: In this experiment, we use the MalwareDrift dataset [28] [45]. The dataset encompasses samples from 7 distinct malware families, namely Bifrose, Creeinject, Obfuscator, Vbinject, Vobfus, Winwebsec, and Zegost, each representing various types of malware such as Trojans, worms, adware, and backdoors. Samples within each family are categorized into pre-drift and post-drift segments, with the partition determined at points of significant evolution. The evolution was determined using an ML-based tracking method that trains an SVM on malware families over time windows. The χ^2 statistic is employed to quantify variations in SVM weights over these sliding time windows. Significant spikes in the χ^2 timeline indicate alterations in the malware's characteristics. Previous work by Ma et al. [28] has shown that models trained on the pre-drift data perform poorly on the post-drift data. Figure 13 in the Appendix shows performance data from [28] on pre-drift and post-drift testing data for five models trained solely on pre-drift training data. The reduction in accuracy when tested on the post-drift data ranges from 23.0% to 38.7%.

2) Source and target datasets setup: The source domain includes pre-drift data from seven malware families and benign Windows samples, while the target domain comprises post-drift data from the same seven families and different benign Windows samples. Our objective is to train a classifier to identify whether a sample from the target domain belongs to one of the eight classes (benign or one of the seven malware families) using a limited number of target samples. The post-drift data is ordered based on time and grouped by family. For the target training set, the first $10, 20, 30, 40, 45$ labeled samples from each class in the post-drift dataset are selected. The remaining samples form the target testing set. This ensures that the target testing set maintains the same class ratio as the original post-drift dataset, preventing spatial bias [36]. In all experiments, the fraction of selected target training samples is at a maximum of 19% of the total post-drift dataset. All pre-drift data is used as the source training set. The source and target datasets are selected without temporal bias [36], as all data in the training set precedes the testing data temporally.

3) Baselines: We use the same baselines approach listed in Section IV-A3. We include the implementation details of each baseline in Appendix A-F.

4) Results: The results, presented in Figure 9, show that:

- Our method significantly outperforms other baseline methods in this closed-set family family classification task. It achieves an 83% accuracy in classifying family labels for the post-drift set with just 10 labeled samples per class.
- In comparison to other baselines, DAN exhibits superior performance. MAGIC (Cold) and MCBG (Cold) fail to converge due to insufficient data, especially for complex tasks like multi-class classification. MAGIC (Warm) and MCBG (Warm), while not as effective as ours, have a considerable performance boost compared to when the models are trained in a cold-start setting.

B. Open-set DA

In open-set classification, the target dataset includes unknown malware families that are not present in the source. There are state-of-the-art approaches for this open-set DA problem in the vision domain [35], [39]. Such approaches are able to (1) accurately classify target samples belonging to known classes and (2) detect and label unknown classes as "unknown". We have demonstrated the first capability with our closed-set experiment and now extend our approach to achieve the second capability in open-set malware family classification.

To handle unknown classes, we first ensure that these samples are not misclassified as benign, which is essential to prevent model evasion attacks. We then determine if the test sample is an outlier relative to the existing classes in the target training data, as outliers may signify new malware families. These outlier samples are set aside, and once a sufficient number is collected, they can be clustered using our graph-based approach. Human analysts would then typically review and assign new family labels to samples from each cluster, as commonly done in various existing approaches [25]. These new classes can be used to update our model or to train additional classifiers as needed.

To detect new families as outliers or unknowns, we introduce an outlier detection module based on one-class SVM that seamlessly integrates with our trained model. We train a one-class SVM on the learned latent graph representations of the post-drift training data and use the trained SVM to classify the latent representations from the test samples (we pass them to the generator to obtain such representations). The SVM assigns a label of (-1) to test samples identified as outliers. We then reassign an "unknown" label to these outlier samples.

We utilize the same source and target datasets setup as in the closed-set domain DA and train our model using the source training data along with 45 labeled samples from each class in the target training set. During testing, we provide three datasets, each containing 1, 5, and 9 new families from the Big-15 dataset, respectively. Our approach is evaluated on these testing sets using the following metrics: **Evasion success rate:** The percentage of samples from unseen malware families misclassified as benign. **Outlier detection rate:** The percentage of samples from unseen malware families correctly identified as the "unknown" class.

The results of both metrics are presented in Table IV. When our model is tested on new malware families, the evasion rate remains consistently low (below 5%), demonstrating robustness against evasion attempts. Additionally, the outlier detection module, trained on post-drift data features extracted by the generator, effectively identifies unknown samples. Figure 10 visualizes the decision boundary of the outlier detection module, with white dots representing latent representation from existing family observations and yellow dots representing latent representation from new family observations.

TABLE IV. EVASION RATE AND DETECTED UNKNOWN SAMPLES RATIO ON THREE TESTING DATASETS WITH 1, 5 AND 9 NEW FAMILIES.

Metrics	1 new family	5 new families	9 new families
Evasion success rate (%)	1.05	4.04	4.46
Detected/Total unknown samples	2396/2476	7662/9122	8921/10711

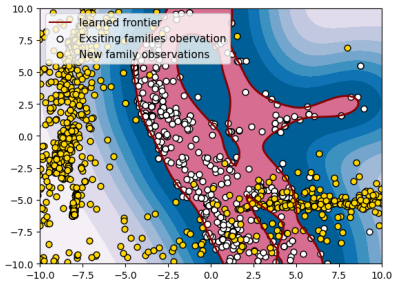

Fig. 10. Visualization of the decision function learned by the outlier detection module: the new family observations are outside the learned frontier.

C. Visualization of extracted features

We show evidence that the features extracted by our generator are drift-invariant. Figure 11 shows the effect of DA on the distribution of latent-space features from two malware families in the MalwareDrift dataset. The t-SNE visualizations show the latent graph representation (learned features) from our adaptation model (right) and the features learned by the MCBG model trained only on pre-drift data (left). DA effectively reduces distribution divergence between pre-drift and post-drift data, as samples from the same malware family in both domains are well clustered together. The clear boundary between these clusters indicates that the learned representations are class-discriminative. Conversely, the MCBG model suffers from a distribution shift, with post-drift data points diverging from pre-drift data points within the same family.

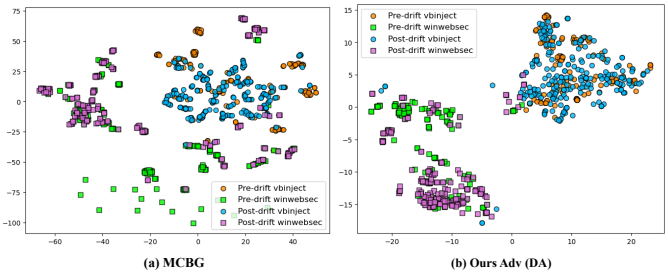

Fig. 11. The effect of adaptation on the distribution of the extracted features from pre-drift and post-drift data (best viewed in color). Each point represents a sample. Yellow and blue dots: pre-drift and post-drift "vbinject". Purple and green boxes: pre-drift and post-drift "winwebsec".

VIII. DISCUSSION

Computational Runtime. We have recorded the average runtime for each step in our pipeline for the experiments on

MB-24 (Section VI), and the results are in Appendix A-G. Our training overhead is acceptable compared to the warm-start and cold-start training strategies. The majority of the pipeline time is dedicated to data extraction and transformation, which is typical for most ML-based methods in novel applications. Extracting content-based features is the most time-consuming because it involves processing assembly files, which can be several gigabytes in size, depending on the binary. Converting the binary to an image representation is the fastest among all the representation processes, but as we have shown in the experiments, the image representation performs the worst when the training data size is small. Our graph representation processing time is between the two, achieving the best or comparable results on the datasets we have used in our evaluation.

System Architecture. In this paper, we focus on adapting malware for one system architecture or one instruction set. We carried out our evaluations on x86, given the notable absence of work on Windows malware. However, our approaches also apply to other systems, such as Android malware.

Robustness of Graph Representation. Ling et al. [26] demonstrated that most model evasion attacks on malware CFGs are ineffective. Effective attacks require complex transformations, such as call-based redividing transformation [26]. Adversarial training is the most effective defense against these attacks. To implement adversarial training, one can generate adversarial samples from various attacks and incorporate them into both domains. The training process and model remain unchanged, with the addition of new adversarial samples for each domain. Future work will further explore the integration of this approach with our DA method.

IX. Related work

Supervised Malware Classification. The learning-based approaches for analyzing executable files fall into three main categories, namely learning from static features [1], [2], [7], [33], [41], [43], [44], [51], dynamic behaviors [16], [54], or a combination of both [11]. Control Flow Graph (CFG), derived from assembly code, has shown exceptional efficacy in malware classification [47], [51]. Among them, MCBG is the closest to our approach in that it uses CFGs and GIN. However, MCBG is designed for supervised learning with the assumption of sufficient labeled samples. It lacks the capability for DA and has not been tested on drifted malware. In contrast, we address concept drift in malware classification, where drifted samples are typically scarce, by using adversarial DA on CFGs.

Malware Classification with Concept Drift. The state-of-the-art solutions for mitigating concept drift employ a similar two-step approach: (1) Sample Selection: Identifying and manually labeling high-impact labels that are critical for learning the distribution of new malware. (2) Model Retraining: Updating the models with these newly labeled data. This procedure is repeated to ensure that the classifier keeps up to date with the latest malware. The primary difference between these solutions is the sample selection technique used. Jordaney et al. [17] and Barbero et al. [5] proposed conformal evaluators that identify and reject examples that differ from the training distribution. Yan et al. [51] introduced a method called CADE to detect drifting samples based on contrastive representation learning. Chen et al. [9] proposed a novel uncertainty score and a pseudo loss for sample selection. For the second step, the models are updated using either cold-start or warm-start learning [9]. We focus on the second step, introducing a novel approach based on DA. Our experiments demonstrate that this approach outperforms existing retraining methods. We believe our work is orthogonal to the sample selection techniques and they could be incorporated into ours. Drift detection techniques could be used to select a minimal subset of data that diverges from the previous samples. Subsequently, these identified samples could serve as the post-drift data for training our model.

Domain Adaptation. The principle behind DA is to leverage a labeled dataset from a source domain to assist in classifying data from a related yet distinct target domain that lacks labeled samples. Here, we briefly discuss two previous DA approaches closely related to ours. The first approach focuses on aligning the statistical distribution shift between the source and target domains through a Domain Adversarial Network (DAN) [27]. A DAN utilizes the MMD loss to compare and reduce distribution shifts. The second approach focuses on learning a hidden representation using two rival networks: a generator and a discriminator. The Domain Adversarial Neural Network (DANN) [14] is one of the prominent methods of this kind. However, DAN and DANN have only been used and evaluated for image classification tasks.

Some approaches were proposed for DA on graph-structured data. AdaGCN [10] is designed for node-level classification tasks and was evaluated on predicting paper topics in citation networks. In our work, we address a graph-level classification problem for classifying malware with concept drift. This requires redesigning the losses and the model, leading to significant differences from prior works. Also, our solution includes a carefully designed pipeline with two other components to produce CFGs with high-quality embeddings. As shown in Section V, the absence of integration of those CFGs results in worse performance with the same DA technique. Recently, DA has been used to address data scarcity in security functions like network intrusion [42] and vulnerability detection [23], [56]. VulGDA [23] and CPVD [56] use DA for cross-project vulnerability detection at the source code level, transforming Code Property Graphs into vector features and learning transferable features with MMD loss and adversarial training, respectively. Our approach differs from VulGDA and CPVD in terms of purpose, operational level, and model input. VulGDA and CPVD work at the source code level for a different security function and cannot be adapted for malware detection, as most malware is in binary form. Also, both approaches utilize intermediate vector representations as the input, derived from training a GNN with source labels, which may introduce a source-only bias in the target vectors. In contrast, our approach employs graphs directly as inputs to the generator, thereby mitigating potential biases originating from the source domain.

X. Conclusion

This paper addresses the classification of drifted malware and the challenge of adapting models with limited labels. Our approach is based on adversarial domain adaptation. It operates on CFGs, exploiting the consistent characteristics of malware in terms of assembly code semantics and control

flow execution. Our process comprises three main elements: constructing CFGs, extracting vertex features, and adapting to drifted samples. We have extensively compared our training approach with approaches from previously published work or adapted to support malware classification. The experimental results show that our approach outperforms others in three distinct adaptation tasks with increasing adaptation complexity: evaluation on research datasets, evaluation on the latest real-world malware samples, and classification of multiple malware families that have evolved. We also demonstrate that our adaptation component can be used with other malware representations to improve performance while our graph representation achieves the best results. We conclude that our approach can effectively improve the model performance when trained with scarce new labels.

ACKNOWLEDGMENT

We thank Md Ajwad Akil from cyber2slab of Purdue University for their valuable time and effort in collecting the MB-24 dataset. The work reported in this paper has been supported by the National Science Foundation (NSF) under Grants 2229876 and 2112471.

REFERENCES

[1] H. Aghakhani, F. Gritti, F. Mecca, M. Lindorfer, S. Ortolani, D. Balzarotti, G. Vigna, and C. Kruegel, "When malware is packin'heat; limits of machine learning classifiers based on static analysis features," in *Network and Distributed Systems Security (NDSS) Symposium 2020*, 2020.

[2] M. Ahmadi, D. Ulyanov, S. Semenov, M. Trofimov, and G. Giacinto, "Novel feature extraction, selection and fusion for effective malware family classification," in *Proceedings of the sixth ACM conference on data and application security and privacy*, 2016, pp. 183–194.

[3] K. Allix, T. F. Bissyandé, J. Klein, and Y. Le Traon, "Androzoo: Collecting millions of android apps for the research community," in *Proceedings of the 13th International Conference on Mining Software Repositories*, ser. MSR '16. New York, NY, USA: ACM, 2016, pp. 468–471. [Online]. Available: http://doi.acm.org/10.1145/2901739.2903508

[4] H. S. Anderson and P. Roth, "EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models," *ArXiv e-prints*, Apr. 2018.

[5] F. Barbero, F. Pendlebury, F. Pierazzi, and L. Cavallaro, "Transcending transcend: Revisiting malware classification in the presence of concept drift," in *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2022, pp. 805–823.

[6] S. Bhardwaj, A. S. Li, M. Dave, and E. Bertino, "Overcoming the lack of labeled data: Training malware detection models using adversarial domain adaptation," *Computers & Security*, p. 103769, 2024.

[7] N. Bhodia, P. Prajapati, F. Di Troia, and M. Stamp, "Transfer learning for image-based malware classification," *arXiv preprint arXiv:1903.11551*, 2019.

[8] T. Caliński and J. Harabasz, "A dendrite method for cluster analysis," *Communications in Statistics-theory and Methods*, vol. 3, no. 1, pp. 1–27, 1974.

[9] Y. Chen, Z. Ding, and D. Wagner, "Continuous learning for android malware detection," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 1127–1144.

[10] Q. Dai, X.-M. Wu, J. Xiao, X. Shen, and D. Wang, "Graph transfer learning via adversarial domain adaptation with graph convolution," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 5, pp. 4908–4922, 2022.

[11] S. Dambra, Y. Han, S. Aonzo, P. Kotzias, A. Vitale, J. Caballero, D. Balzarotti, and L. Bilge, "Decoding the secrets of machine learning in malware classification: A deep dive into datasets, feature extraction, and model performance," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, 2023, pp. 60–74.

[12] D. L. Davies and D. W. Bouldin, "A cluster separation measure," *IEEE transactions on pattern analysis and machine intelligence*, no. 2, pp. 224–227, 1979.

[13] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database," in *2009 IEEE conference on computer vision and pattern recognition*. Ieee, 2009, pp. 248–255.

[14] Y. Ganin, E. Ustinova, H. Ajakan, P. Germain, H. Larochelle, F. Laviolette, M. Marchand, and V. Lempitsky, "Domain-adversarial training of neural networks," *The journal of machine learning research*, vol. 17, no. 1, pp. 2096–2030, 2016.

[15] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *Communications of the ACM*, vol. 63, no. 11, pp. 139–144, 2020.

[16] Y. Hou, T. Truong-Huu, Z. Chen, C.-K. Kwoh, and S. G. Teo, "Proteus: Domain adaptation for dynamic features in ai-assisted windows malware detection," in *2023 IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE, 2023, pp. 1322–1331.

[17] 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 *26th USENIX security symposium (USENIX security 17)*, 2017, pp. 625–642.

[18] Z. Kan, F. Pendlebury, F. Pierazzi, and L. Cavallaro, "Investigating labelless drift adaptation for malware detection," in *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*, 2021, pp. 123–134.

[19] T. N. Kipf and M. Welling, "Variational graph auto-encoders," *arXiv preprint arXiv:1611.07308*, 2016.

[20] S. Kumar *et al.*, "Mcft-cnn: Malware classification with fine-tune convolution neural networks using traditional and transfer learning in internet of things," *Future Generation Computer Systems*, vol. 125, pp. 334–351, 2021.

[21] A. S. Li, E. Bertino, X.-H. Dang, A. Singla, Y. Tu, and M. N. Wegman, "Maximal domain independent representations improve transfer learning," *arXiv preprint arXiv:2306.00262*, 2023.

[22] A. S. Li, A. Iyengar, A. Kundu, and E. Bertino, "Transfer learning for security: Challenges and future directions," *arXiv preprint arXiv:2403.00935*, 2024.

[23] X. Li, Y. Xin, H. Zhu, Y. Yang, and Y. Chen, "Cross-domain vulnerability detection using graph embedding and domain adaptation," *Computers & Security*, vol. 125, p. 103017, 2023.

[24] X. Li, Y. Qu, and H. Yin, "Palmtree: Learning an assembly language model for instruction embedding," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3236–3251.

[25] J. Liang, W. Guo, T. Luo, H. Vasant, G. Wang, and X. Xing, "Fare: enabling fine-grained attack categorization under low-quality labeled data," in *Proceedings of The Network and Distributed System Security Symposium (NDSS)*, 2021.

[26] X. Ling, Z. Wu, B. Wang, W. Deng, J. Wu, S. Ji, T. Luo, and Y. Wu, "A wolf in sheep's clothing: Practical black-box adversarial attacks for evading learning-based windows malware detection in the wild," in *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, 2024, pp. 7393–7410.

[27] M. Long, Y. Cao, J. Wang, and M. Jordan, "Learning transferable features with deep adaptation networks," in *International conference on machine learning*. PMLR, 2015, pp. 97–105.

[28] Y. Ma, S. Liu, J. Jiang, G. Chen, and K. Li, "A comprehensive study on learning-based pe malware family classification methods," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1314–1325.

[29] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "Mamadroid: Detecting android malware by building

markov chains of behavioral models," *arXiv preprint arXiv:1612.04433*, 2016.

[30] L. Massarelli, G. A. Di Luna, F. Petroni, L. Querzoni, R. Baldoni *et al.*, "Investigating graph embedding neural networks with unsupervised features extraction for binary analysis," in *Proceedings of the 2nd Workshop on Binary Analysis Research (BAR)*, 2019, pp. 1–11.

[31] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu, "Context-aware, adaptive, and scalable android malware detection through online learning," *IEEE Transactions on Emerging Topics in Computational Intelligence*, vol. 1, no. 3, pp. 157–175, 2017.

[32] A. Narayanan, L. Yang, L. Chen, and L. Jinliang, "Adaptive and scalable android malware detection through online learning," in *2016 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2016, pp. 2484–2491.

[33] L. Nataraj, S. Karthikeyan, G. Jacob, and B. S. Manjunath, "Malware images: visualization and automatic classification," in *Proceedings of the 8th international symposium on visualization for cyber security*, 2011, pp. 1–7.

[34] Y. M. Pa Pa, S. Tanizaki, T. Kou, M. Van Eeten, K. Yoshioka, and T. Matsumoto, "An attacker's dream? exploring the capabilities of chatgpt for developing malware," in *Proceedings of the 16th Cyber Security Experimentation and Test Workshop*, 2023, pp. 10–18.

[35] P. Panareda Busto and J. Gall, "Open set domain adaptation," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 754–763.

[36] F. Pendlebury, F. Pierazzi, R. Jordaney, J. Kinder, and L. Cavallaro, "{TESSERACT}: Eliminating experimental bias in malware classification across space and time," in *28th USENIX security symposium (USENIX Security 19)*, 2019, pp. 729–746.

[37] R. Ronen, M. Radu, C. Feuerstein, E. Yom-Tov, and M. Ahmadi, "Microsoft malware classification challenge," *arXiv preprint arXiv:1802.10135*, 2018.

[38] P. J. Rousseeuw, "Silhouettes: a graphical aid to the interpretation and validation of cluster analysis," *Journal of computational and applied mathematics*, vol. 20, pp. 53–65, 1987.

[39] K. Saito, S. Yamamoto, Y. Ushiku, and T. Harada, "Open set domain adaptation by backpropagation," in *Proceedings of the European conference on computer vision (ECCV)*, 2018, pp. 153–168.

[40] G. Salha, R. Hennequin, and M. Vazirgiannis, "Keep it simple: Graph autoencoders without graph convolutional networks," *arXiv preprint arXiv:1910.00942*, 2019.

[41] A. Singh, A. Handa, N. Kumar, and S. K. Shukla, "Malware classification using image representation," in *Cyber Security Cryptography and Machine Learning: Third International Symposium, CSCML 2019, Beer-Sheva, Israel, June 27–28, 2019, Proceedings 3*. Springer, 2019, pp. 75–92.

[42] A. Singla, E. Bertino, and D. Verma, "Preparing network intrusion detection deep learning models with minimal data using adversarial domain adaptation," in *Proceedings of the 15th ACM Asia conference on computer and communications security*, 2020, pp. 127–140.

[43] M. Someya, Y. Otsubo, and A. Otsuka, "Fcgat: Interpretable malware classification method using function call graph and attention mechanism," in *Proceedings of Network and Distributed Systems Security (NDSS) Symposium*, vol. 1, 2023.

[44] D. Vasan, M. Alazab, S. Wassan, H. Naeem, B. Safaei, and Q. Zheng, "Imcfn: Image-based malware classification using fine-tuned convolutional neural network architecture," *Computer Networks*, vol. 171, p. 107138, 2020.

[45] M. Wadkar, F. Di Troia, and M. Stamp, "Detecting malware evolution using support vector machines," *Expert Systems with Applications*, vol. 143, p. 113022, 2020.

[46] F. Wang, G. Chai, Q. Li, and C. Wang, "An efficient deep unsupervised domain adaptation for unknown malware detection," *Symmetry*, vol. 14, no. 2, p. 296, 2022.

[47] B. Wu, Y. Xu, and F. Zou, "Malware classification by learning semantic and structural features of control flow graphs," in *2021 IEEE 20th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, 2021, pp. 540–547.

[48] Y. Xie, J. Yi, J. Shao, J. Curl, L. Lyu, Q. Chen, X. Xie, and F. Wu, "Defending chatgpt against jailbreak attack via self-reminders," *Nature Machine Intelligence*, vol. 5, no. 12, pp. 1486–1496, 2023.

[49] K. Xu, Y. Li, R. Deng, K. Chen, and J. Xu, "Droidevolver: Self-evolving android malware detection system," in *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2019, pp. 47–62.

[50] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, "How powerful are graph neural networks?" *arXiv preprint arXiv:1810.00826*, 2018.

[51] J. Yan, G. Yan, and D. Jin, "Classifying malware represented as control flow graphs using deep graph convolutional neural network," in *2019 49th annual IEEE/IFIP international conference on dependable systems and networks (DSN)*. IEEE, 2019, pp. 52–63.

[52] L. Yang, A. Ciptadi, I. Laziuk, A. Ahmadzadeh, and G. Wang, "Bodmas: An open dataset for learning based temporal analysis of pe malware," in *4th Deep Learning and Security Workshop*, 2021.

[53] 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, pp. 2327–2344.

[54] H. Yin, B. Lou, and P. Reiher, "A method for summarizing and classifying evasive malware," in *Proceedings of the 26th International Symposium on Research in Attacks, Intrusions and Defenses*, 2023, pp. 455–470.

[55] A. Zewe, "Avoiding shortcut solutions in artificial intelligence," https://news.mit.edu/2021/shortcut-artificial-intelligence-1102, 2021, [Online; accessed April 2024].

[56] C. Zhang, B. Liu, Y. Xin, and L. Yao, "Cpvd: Cross project vulnerability detection based on graph attention network and domain adaptation," *IEEE Transactions on Software Engineering*, 2023.

APPENDIX A

A. Implementation details and additional results on Big-15

1) Our model implementation details: We implemented our model using TensorFlow and Spektral, a library for GNN. The generator has 3 `GIN` layers and ends with a global average pooling, and a dense layer with 256 neurons. The architecture of the classifier consists of 2 fully connected layers (`FC_1`, `FC_OUT`). The number of neurons in `FC1` is 256. `FC_OUT` is the output layer for label prediction. The discriminator has two layers with 256 hidden units and is followed by the softmax layer for domain prediction. Batch normalization is applied in each hidden layer. For training the model, we use the Adam optimizer with a learning rate of $1e-3$ for 60 epochs. The batch size is 16. The coefficient of the loss $\mathcal{L}_g$ is set to 0.1, which allows the discriminator to be less sensitive to noisy signals during training. We set $\lambda = 0.1$ in $\mathcal{L}_c$ since we have more labeled data from the source.

2) Baseline implementation details: MAGIC (Cold): MAGIC utilizes handcrafted features to represent each basic block, which are listed in Table V. The classification model is DGCNN, which uses a SortPooling layer. Our DGCNN topology is based on the ones used in [51] to be comparable to previous work. We train the model using the Adam optimizer with a learning rate of $1e-3$ for 60 epochs.

MCBG (Cold): MCBG uses the GIN-JK model as the classifier. We use their default GIN-JK model with the same hyperparameters used in [47]. MAGIC is trained using the Adam optimizer with a learning rate of $1e-3$ for 60 epochs.

MAGIC (Warm) and MCBG (Warm): In [9], warm-start learning takes an older model and continues training the entire model with new samples. However, we found that freezing the weights of the initial layer of the older model performs better on Big-15, so we opted for this approach. In fact, this

TABLE V. BASIC BLOCK ATTRIBUTES DEFINED IN MAGIC

Attribute Type	Attribute Description
token-level	# Numeric Constants # Transfer Instructions # Call Instructions # Arithmetic Instruction # Compare Instructions # Mov Instructions # Termination Instructions # Data Declaration Instructions # Total Instructions
block-level	# Offspring, i.e., Degree # Instructions in the Block

approach aligns with a commonly used fine-tuning paradigm in transfer learning, where certain layers are frozen to retain the knowledge gained from the source data.

DAN: The network architecture of the feature extractor and classifier is the same as the generator and classifier mentioned in Appendix A-A1. The feature extractor has 3 `GIN` layers and ends with a global average pooling, and a dense layer with 256 neurons. The architecture of the classifier consists of 2 fully connected layers (`FC_1`, `FC_OUT`). The number of neurons in `FC1` is 256. `FC_OUT` is the output layer for label prediction. Batch normalization is applied on each hidden layer. For training the model, we use the Adam optimizer with a learning rate of $1e-3$ for 60 epochs. The batch size is 16. The coefficient of the MMD loss is set to 1, followed by [27]. We set $\lambda = 0.1$ in $\mathcal{L}_c$ as well.

3) Graph-based clustering implementation details: First, we train a graph autoencoder to derive a 256-dimensional feature vector for each malware graph. In the consensus clustering algorithm, we apply three different clustering predictors: Gaussian Mixture Model (GMM), HDBSCAN, and K-means. For each predictor, we follow the best practice for selecting the parameters. For K-means clustering, we use a heuristic approach based on inertia values to determine the appropriate cluster number. In the case of GMM, all combinations of six components and four covariance types are explored to identify the model with the lowest Bayes Information Criterion. Regarding HDBSCAN, in order to reduce the number of outliers, we set the parameters as follows: $min\_cluster\_size = 2$ and $min\_samples = 2$. All three clustering solutions and the original labels are considered during the consensus matrix update. Finally, the GMM model is utilized to derive the final clusters for our analysis.

4) Additional results: The F1 scores of all the baselines and our approach evaluated on the original label set of Big-15 are listed in Table VII, while the F1 scores for the evaluation on the clustering label set are listed in Table VIII. The accuracy of all the methods on each target family based on the original label set and cluster label set of Big-15 is presented in Figure 12. The difference between the averaged accuracy of each approach based on the original label and cluster label is shown in Table VI.

B. Additional details on the content-based features of Big-15

The categories of features based on the content of the assembly files are summarized below.

- Symbol. The frequencies of symbols -, +, *,], [, ?, and @ are counted due to their common occurrence in code designed to evade detection.

TABLE VI. IMPACT OF MORE ACCURATE LABELS ON VARIOUS TRAINING STRATEGIES. THE TABLE SHOWS THE DIFFERENCE IN AVERAGE ACCURACY, DETERMINED BY THE ORIGINAL AND CLUSTER LABEL ASSIGNMENT. A POSITIVE VALUE INDICATES AN INCREASE IN PERFORMANCE RELATIVE TO THE ORIGINAL LABEL, WHILE A NEGATIVE VALUE INDICATES A DECREASE.

Strategy	Method	Target samples					
		20	50	100	200	300	500
Cold	MCBG	-1.2	-0.4	-1.2	-0.5	0.5	1.2
	MAGIC	1.1	1.4	-0.8	-1.6	-1.3	-1.1
Warm	MCBG	-4.2	-5.8	-3.3	-1.9	-0.6	0
	MAGIC	-1	-0.4	-0.3	-1.8	-1.7	-0.1
DA	DAN (MMD)	-1.5	-2.4	-0.2	0.1	-0.4	-0.2
	Ours (Adv)	-1.5	-1.2	0.1	-0.2	-0.2	-0.2

- Opcode. We have chosen a subset of 93 opcodes from the x86 instruction set, selected for their common usage and their frequent appearance in malicious code. We then measure the frequency of these opcodes in each malware sample.
- Register. The utilization frequency of registers has proven valuable in classifying malware families. Consequently, we have integrated 26 register features into the feature set for this purpose.
- Windows API. We also measure the frequency of use of Windows API in the assembly code. We have selected the top 794 frequent APIs used by malicious binaries based on the study in [2].
- Section. A PE file typically comprises predefined sections such as .text, .data, .bss, .rdata, .edata, .idata, .rsrc, .tls, and .reloc. Due to packing, these default sections can be altered, rearranged, and new sections may be introduced. Consequently, 26 section features are statistical attributes that capture the proportion of each section within the entire assembly file.
- Data Define. Certain malware samples may lack any API calls and primarily consist of a few operation codes, often due to packing. Specifically, they frequently feature data definition instructions like db, dw, and dd. As a result, we have incorporated 18 statistical features that summarize the distinct characteristics of data definition instructions into our analysis.
- Others. We have calculated both the file size and the number of lines within the file.

C. Implementation details on content-based baselines and our approach

1) SVM and MLP: For SVM, we set $C = 0.1$ to be consistent with [9]. The MLP consists of 8 fully connected layers (`FC_1`,`FC_1`,..., `FC_OUT`), with 100 neurons in `FC1-5` and 400 neurons in `FC5-6`, while `FC_OUT` serves as the output layer for label prediction. MLP is trained using the Adam optimizer with a learning rate of $1e-3$ for 60 epochs. In the warm-start of MLP, we continue training all the layers of the older model.

2) Our approach: Our model is implemented to be comparable with the MLP. The generator comprises 5 dense layers, each with 100 neurons. The classifier architecture consists of 3 fully connected layers, with the first two layers having 400 neurons each and the final layer serving as the output layer. Essentially, the combined generator and classifier components

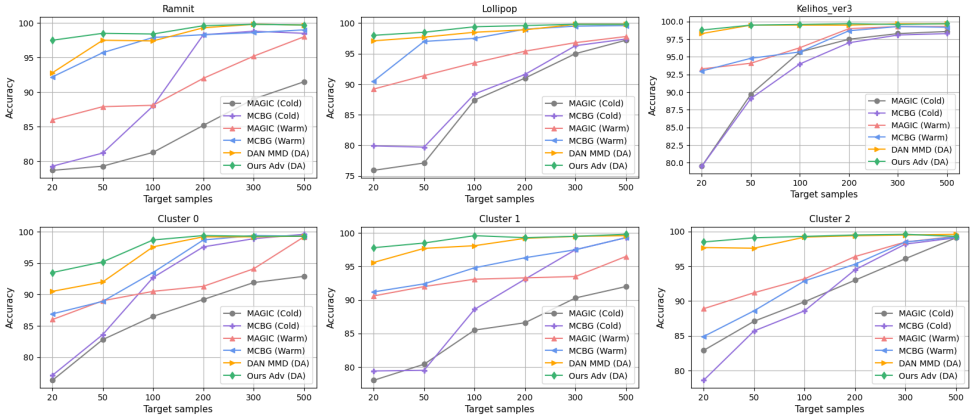

Fig. 12. Given fixed target training labels, we compute the average accuracy on the target testing data for different baseline techniques and our method. The top three figures are evaluated on the target family based on the original labels of Big-15, and the bottom three figures are for the clustering labels.

TABLE VII. EVALUATION RESULTS (F1 SCORE) BASED ON THE ORIGINAL LABEL SET OF BIG-15.

Strategy	Method	Ramnit: target samples						Lollipop: target samples						Kelihos_ver3: target samples					
		20	50	100	200	300	500	20	50	100	200	300	500	20	50	100	200	300	500
Cold	MCBG	79.3	81.3	88.1	98.3	98.2	98.5	79.9	79.7	88.3	91.6	96.2	97.4	80.0	89.1	94.0	97.0	98.0	98.3
	MAGIC	78.6	79.3	81.2	85.3	89.0	91.5	75.9	77.1	87.4	91.0	95.0	97.2	79.5	89.7	95.7	97.5	98.3	98.6
Warm	MCBG	92.2	95.7	**97.9**	98.3	98.5	99.0	90.5	97.0	97.5	99.0	99.5	99.6	93.0	94.8	95.7	98.7	99.3	99.2
	MAGIC	86.0	87.8	88.2	92.0	95.2	98.0	89.1	91.4	93.4	95.4	96.8	97.8	93.3	94.0	96.3	99.0	99.3	99.3
DA	DAN (MMD)	**92.8**	**97.5**	97.5	**99.3**	**99.8**	**99.7**	**97.1**	**97.6**	**98.5**	**98.9**	**99.8**	**99.8**	**98.3**	**99.5**	**99.5**	**99.4**	**99.6**	**99.7**
	Ours (Adv)	**97.5**	**98.5**	**98.4**	**99.6**	**99.8**	**99.7**	**98.0**	**98.5**	**99.6**	**99.6**	**99.8**	**99.8**	**98.5**	**99.5**	**99.6**	**99.7**	**99.6**	**99.7**
		↑4.7	↑1	↑0.5	↑0.3	↑0	↑0	↑0.9	↑0.9	↑1.1	↑0.7	↑0	↑0	↑0.2	↑0	↑0.1	↑0.3	↑0	↑0

TABLE VIII. EVALUATION RESULTS (F1 SCORE) BASED ON THE CLUSTER LABEL OF BIG-15.

Strategy	Method	Cluster 0: target samples						Cluster 1: target samples						Cluster 2: target samples					
		20	50	100	200	300	500	20	50	100	200	300	500	20	50	100	200	300	500
Cold	MCBG	77.1	83.6	92.7	97.6	98.9	99.6	79.6	78.9	88.5	93.1	97.5	99.3	78.6	85.7	88.6	94.5	98.1	99.2
	MAGIC	76.3	82.8	86.5	89.2	91.9	92.9	78.0	80.4	85.1	86.6	90.3	92.0	82.9	87.0	89.9	93.0	96.1	99.1
Warm	MCBG	86.9	88.9	93.5	98.6	**99.4**	**99.4**	91.1	92.4	94.8	96.3	97.5	99.3	85.0	88.6	93.0	95.5	98.5	99.2
	MAGIC	86.0	89.0	90.5	91.3	94.1	99.2	90.6	92.0	93.1	93.3	93.5	96.5	88.0	91.2	93.2	96.5	98.5	99.3
DA	DAN (MMD)	**90.7**	**92.1**	**97.6**	**99.2**	99.2	99.3	**95.7**	**97.7**	**98.1**	**99.2**	**99.5**	**99.6**	**97.7**	**97.6**	**99.2**	**99.4**	**99.5**	**99.6**
	Ours (Adv)	**93.4**	**95.2**	**98.7**	**99.4**	**99.2**	**99.3**	**97.8**	**96.5**	**99.5**	**99.3**	**99.5**	**99.8**	**98.5**	**99.1**	**99.3**	**99.5**	**99.6**	**99.3**
		↑2.7	↑3.1	↑1.1	↑0.2	↓0.2	↓0.1	↑2.1	↓1.2	↑1.4	↑0.1	↑0	↑0.2	↑0.8	↑1.5	↑0.1	↑0.1	↑0.1	↓0.3

form the classifier model described above. The discriminator has four layers with 400 hidden units, followed by the softmax layer for domain prediction. The hyperparameters remain consistent with those outlined in Appendix A-A1.

3) DAN: The feature extractor comprises 5 dense layers, each with 100 neurons. The classifier architecture consists of 3 fully connected layers, with the first two layers having 400 neurons each and the final layer serving as the output layer. The hyperparameters remain consistent with those outlined in Appendix A-A2.

4) Additional results on representation evaluation: The F1-score can be found in Table IX.

D. Implementation details on image-based baselines

1) Our approach: The generator has the architecture of (`Conv2D`, `MaxPooling`, `Conv2D`, `MaxPooling`, `Flatten`). The `Conv2D` layers has $\{32, 64\}$ 3×3 filters respectively. The classifier consists of 2 fully connected layers (`FC_1`, `FC_OUT`). The number of neurons in `FC1` is 256. `FC_OUT` is the output layer for label prediction. The discriminator has two layers with 1024 hidden units and is followed by the softmax layer for domain prediction. Batch normalization is applied on each hidden layer. For training the model, we use the Adam optimizer with a learning rate of $1e-3$ for 60 epochs. The batch size is 32. The coefficient of the loss $\mathcal{L}_g$ is

TABLE IX. F1 OF BASELINES ACROSS VARIOUS MALWARE REPRESENTATIONS. IN TERMS OF CONTENT AND IMAGE REPRESENTATION, WE ILLUSTRATE THE PERCENTAGE CHANGE IN ACCURACY FROM OUR ADVERSARIAL (ADV) DA METHOD TO THE PEAK PERFORMANCE AMONG ALL BASELINES WITHIN THE SAME REPRESENTATION. ULTIMATELY, WE DEMONSTRATE THE ENHANCEMENTS OF OUR FULL PIPELINE (GRAPH + ADV DA) COMPARED TO BOTH CONTENT + ADV DA AND IMAGE + ADV DA.

Malware Representation	Strategy	Method	Target samples					
			20	50	100	200	300	500
Content-based (CB)	Cold	SVM	67.1	71.4	75.1	78.9	82.2	85
		MLP	77	79.4	85.2	91.5	92.9	94.8
	Warm	SVM	70.2	72.7	78.6	82.9	86.7	89.7
		MLP	**90.9**	93.4	95.3	96.2	**97.5**	**97.9**
	DA	DAN (MMD) + MLP	90.8	**93.9**	**95.4**	**96.9**	97.1	97.5
		Ours (Adv) + MLP	**94.1**	**95.2**	**96.2**	**97.1**	**97.7**	**97.8**
			↑3.2	↑1.3	↑0.8	↑0.2	↑0.2	↓0.1
Image-based (IB)	Cold	ResNet-50	60.6	60.6	70.6	74.3	77	84.4
	Warm	ResNet-50	**86.2**	**87.7**	89.5	**91.9**	**93.5**	**94.4**
	DA	DAN (MMD) + CNN	85.4	87.5	**89.9**	91.6	93.1	94
		Ours (Adv) + CNN	**88.6**	**90**	**92.2**	**93.1**	**94**	**94.6**
			↑2.4	↑2.3	↑2.3	↑1.2	↑0.5	↑0.2
Graph-based (GB)	DA	Ours (Adv) + GIN	**96.6**	**96.9**	**99.2**	**99.4**	**99.4**	**99.5**
Improvement over CB: Ours (Adv)			↑2.5	↑1.7	↑3	↑2.3	↑1.7	↑1.7
Improvement over IB: Ours (Adv)			↑8	↑6.9	↑7	↑6.3	↑5.4	↑4.9

set to 0.1. We set $\lambda = 0.1$ in $\mathcal{L}_c$.

2) ResNet (Cold) and (Warm): The original ResNet-50's output layer contains 1000 neurons. As our task focuses on predicting whether a sample is malware or benign software, we modified the model by removing its last layer, adding a global average pooling layer, incorporating a fully connected layer with 256 neurons, and appending an output layer with 2 neurons. For training the model, we use the Adam optimizer with a learning rate of $1e-3$ for 60 epochs.

3) DAN: The feature extractor also has the same architecture as the generator and so as the classifier. For training the model, we use the Adam optimizer with a learning rate of $1e-3$ for 60 epochs. The batch size is 32. The coefficient of the MMD loss is set to 1. We set $\lambda = 0.1$ in $\mathcal{L}_c$.

4) Additional results on representation evaluation: The F1-scores can be found in Table IX.

E. Implementation details and additional results on MB-24

1) Implementation: The implementation of the baseline models and our approach remain consistent with those outlined in Appendix A-A1 and A-A2. We set $\lambda = 0.5$ in $\mathcal{L}_c$ for DAN and our approach.

2) Additional results on MB-24 evaluation: The F1-scores can be found in Table X.

TABLE X. EVALUATION RESULTS (F1 SCORE) ON THE MB-24 DATASET

Strategy	Method	Target samples					
		20	50	100	200	300	500
Cold	MCBG	53.8	56.9	59.2	62.7	70.4	71.6
	MAGIC	57.4	61.5	67.4	69.2	70.2	71.7
Warm	MCBG	68.1	**71.7**	73.1	74.8	76.7	77.5
	MAGIC	68.7	71.5	73.4	75	76.1	77.3
DA	DAN (MMD)	**70.5**	71.5	**73.7**	**78.4**	**78.8**	**80.5**
	Ours (Adv)	**72.1**	**73.7**	**76.1**	**79.2**	**79.8**	**82.4**
		↑1.6	↑2	↑2.4	↑0.8	↑1	↑1.9

3) Additional results on the obfuscated malware: The first experiment was conducted using task (b) in Figure 6. The source malware data was collected from March to May, the target training malware was from August, and the target testing malware was from September. All target testing malware samples were obfuscated using Hyperion, while the source and target training malware samples remained unobfuscated. Neither the baselines nor our model were exposed to obfuscated samples during training. The goal was to compare the performance of each method, trained with unobfuscated malware when tested on obfuscated malware. The results of this experiment are reported in Figure 13. The experiment shows that our approach experiences the least accuracy degradation when tested on obfuscated malware samples that were not seen during training. For a comparison, please refer to Figure 7, which reports the results when the target testing set is not obfuscated.

F. Implementation details and additional results on MalwareDrift

The implementation of the baseline models and our approach remain consistent with those outlined in Appendix A-A1 and A-A2, except for the output layer, which is modified to accommodate 8 classes.

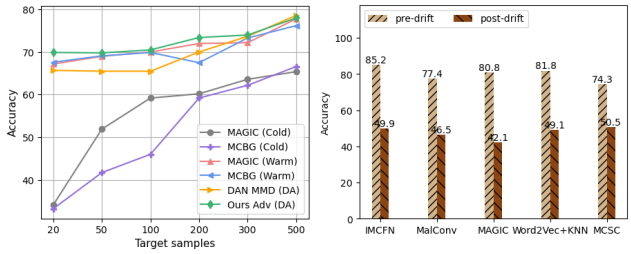

Fig. 13. **Left**: Accuracy on the obfuscated September testing data for different baseline techniques and our method. **Right**: Accuracy of source-only approach on the MalwareDrift dataset.

Previous work by Ma et al. [28] has shown that models trained on the pre-drift data perform poorly on the post-drift data. Figure 13 shows performance data from [28] on pre-drift and post-drift testing data for five models trained solely on pre-drift training data. As we can see from the figure, the reduction in accuracy ranges from 23.0% to 38.7%. It is thus clear that all considered models experience a substantial performance decrease when confronted with code evolution within the same malware families.

We report the accuracy and F1 score of all the methods evaluated on post-drift data using $10-45$ labeled data from each post-drift family in Table XI.

TABLE XI. ACCURACY AND F1 SCORE ON POST-DRIFT TESTING DATA USING A FEW LABELED DATA FROM EACH FAMILY IN POST-DRIFT TRAINING DATA.

Metric	Strategy	Method	Target samples per family				
			10	20	30	40	45
Accuracy	Cold	MCBG	25.5	35.3	40.0	63.5	73.2
		MAGIC	21.0	34.5	39.4	59.0	69.5
	Warm	MCBG	71	77.0	83.0	85.7	88.9
		MAGIC	69.0	74.5	82.0	83.5	83.5
	DA	DAN (MMD) + GIN	**74.0**	**84.5**	**86.2**	**88.0**	**89.0**
		Ours (Adv) + GIN	**83.0**	**87.5**	**88.7**	**90.0**	**91.6**
			↑9	↑3	↑2.5	↑2	↑2.6
F1	Cold	MCBG	25.5	35.2	40.0	63.5	73.2
		MAGIC	22.0	34.5	39.4	58.0	68.5
	Warm	MCBG	71.5	77.0	82.5	85.5	88.5
		MAGIC	69.0	74.5	81.0	83.5	83.5
	DA	DAN (MMD) + GIN	**75.0**	**84.5**	**86.2**	**88.0**	**89.0**
		Ours (Adv) + GIN	**82.0**	**87.5**	**88.7**	**89.0**	**90.7**
			↑7	↑3	↑2.5	↑1	↑1.7

G. Computational runtime

Both CFG extraction (Section II-C) and vertex feature extraction (Section II-D) are dependent on the node count within the CFG. Therefore, we present the extraction times for a malware CFG containing $17,564$ basic blocks/nodes. The CFG extraction process takes only 2.84 seconds to extract a complete CFG with $17,564$ basic blocks. In contrast, the vertex feature extraction is the most time-consuming step. For a CFG with $17,564$ nodes, the encoding process using PalmTree requires 3.4 minutes.

Training, on the other hand, is relatively faster. The training duration is affected by the size of the source and target datasets, the total number of epochs, and the GPU used. For example, training on the MB-24 dataset, which includes $7,907$ source training samples and 200 target training samples over 80 epochs, takes 13.5 minutes on an NVIDIA RTX 4090D. This is highly efficient for training a deep learning model.

APPENDIX B
ARTIFACT APPENDIX

A. Description & Requirements

1) How to access: The code for the artifact can be accessed at: https://github.com/gloryer/malware-detection-concept-drift/tree/main?tab=readme-ov-file. The required data to run this artifact can be accessed at: https://zenodo.org/records/14213306.

2) Hardware dependencies: We have successfully run the code with the following hardware

- CPU: Intel® Core™ i7-6700K CPU @ 4.00GHz × 8
- GPU: Nvidia RTX 3090 (24 GB)
- Memory: 64 GB

Additionally, we recommend 100 GB of available disk space to store the data.

3) Software dependencies: The code was tested on Ubuntu 20.04 LTS and using Python 3.8. It should also run on Ubuntu 22.04 LTS and later stable versions. Most of the code is built with TensorFlow, but PalmTree was developed using PyTorch, so some notebooks also require PyTorch installation. The versions used are TensorFlow 2.9.0 and PyTorch 2.4. Please follow the README in our GitHub repository to set up the environment and solve software dependencies.

B. Artifact Installation & Configuration

To set up the environment, we provided a `requirements.txt` (with the exact versions we used) in our repository. We recommend creating a virtual environment and installing the packages. All the required libraries can be installed with

```
pip install -r requirements.txt
```

The requirements were generated using `pip freeze` and modified manually to consider only the required packages. If a package is missing or you run into a problem, please feel free to contact us.

C. Experiment Workflow

Once the GitHub repository is cloned, please download the data and store them under the directory `malware-detection-concept-drift`. Run `tar -xzvf data.tar.gz` to extract the compressed file and do not change the name of the extracted folder (the name should be `/data`. Detailed instructions can be found in the GitHub README file. The experiments are designed to help the readers work through our pipeline presented in the paper (see the following evaluation section). We are seeking the artifact available badge and the artifact functional badge for this artifact. To make the review process convenient, we provided ipynb notebooks to run the experiments.

D. Major Claims

Following are the major claims we make for the artifact available and artifact functional badge

- (C1): Our artifact extracts CFGs from assembly files, and the raw CFGs are further processed to generate the node-level embeddings. This is proved by E1 and E2.
- (C2): We find that the original labels of Big-15 are not well separated. We can create denser clusters for the same dataset using the graph-based clustering algorithm presented in Section III. This is proved by E3 and E4.
- (C3): Our artifact can be successfully run to train our domain adaptation models and generate evaluation metrics (to produce our results in Figure 3 and Table II) in the paper. This is proved by E5.

E. Evaluation

Please run the following experiments to verify the claims.

1) Experiment (E1): [CFG Extraction] [10 human-minutes + 5 compute-minutes]: The following experiment is to extract CFGs from assembly files of malware/benign binaries.

- From the `CFG` directory, run the notebook `extract_cfg.ipynb`
- We only include five example ASM files from the Big-15 dataset to show the code is functional. However, the code is scalable to process a large number of files. You can also view the node(s) and edges printed in pretty json in the second cell.

2) Experiment (E2): [Vertex Feature Extraction] [15 human-minutes + 20 compute-minutes]: The following experiment is to generate the embeddings for the nodes of CFG using the pre-trained PalmTree model.

- From the `CFG` directory, run the notebook `generate_embeddings.ipynb`
- We only include five CFGs to show the code is functional. However, the code is scalable and can perform over a very large number of files.

3) Experiment (E3): [Graph-based Clustering Phase 1] [20 human-minutes + 20 compute-minutes]: The following experiment is to generate the graph embedding with a graph autoencoder. The graph embeddings will be used in Phase 2 of the graph-based clustering.

- From the `graph_based_clustering` directory, run the notebook `generate_graph_embeddings.ipynb`
- To simplify the amount of work for testing, we demonstrate with just five graphs obtained from E2, but the code is designed to process many graphs.

4) Experiment (E4): [Graph-based Clustering Phase 2] [1 human-hour + 1 compute-hour]: The following experiment is to generate the cluster labels with our weighted consensus clustering algorithm.

- From the `graph_based_clustering` directory, run the notebook `consensus_clustering.ipynb`
- Within the notebook, you'll find cells for t-SNE visualizations of the Big-15 dataset, showing both the original labels and the new cluster labels. Each cell will generate a figure in the output, which can be compared to Figure 4.

5) Experiment (E5): [Domain Adaptation] [2 human-hours + 4 compute-hours]: The following experiment is to generate our models' performance metrics on the Big-15 dataset. Follow these instructions to train our model:

- Run the following notebooks in any order
  `Train_origin_label_graph.ipynb`
  `Train_cluster_label_graph.ipynb`
  `Train_cluster_label_image.ipynb`
  `Train_cluster_label_content.ipynb`
- The first two notebooks will generate the metrics for our approach, as shown in Figure 12. The third and fourth notebooks will generate the metrics for our approach using image representations and content-based representations, respectively, corresponding to Table II.

F. Notes

Each notebook can be run without any modification to the cells. Since there are many cells, we recommend using Kernel → Run all cells (as found in the Jupyter interface). For quicker and more convenient execution, we set the number of epochs to a low value, minimizing runtime. However, the notebooks can be adjusted to use a larger number of epochs. To do this, follow the cells in the training notebooks, where we've indicated in the code comments how to modify the epoch setting.