Fast and Efficient Malware Detection with Joint Static and Dynamic Features Through Transfer Learning

Mao V. Ngo[1], Tram Truong-Huu[2(✉)], Dima Rabadi[3], Jia Yi Loo[4], and Sin G. Teo[4]

[1] Singapore University of Technology and Design, Singapore, Singapore
vanmao_ngo@sutd.edu.sg
[2] Singapore Institute of Technology, Singapore, Singapore
truonghuu.tram@singaporetech.edu.sg
[3] Penn State Shenango, Pennsylvania, USA
dqr5554@psu.edu
[4] Institute for Infocomm Research, A*STAR, Singapore, Singapore
{loojy,teosg}@i2r.a-star.edu.sg

Abstract. In malware detection, dynamic analysis extracts the runtime behavior of malware samples in a controlled environment and static analysis extracts features using reverse engineering tools. While the former faces the challenges of anti-virtualization and evasive behavior of malware samples, the latter faces the challenges of code obfuscation. To tackle these drawbacks, prior works proposed to develop detection models by aggregating dynamic and static features, thus leveraging the advantages of both approaches. However, simply concatenating dynamic and static features raises an issue of imbalanced contribution due to the heterogeneous dimensions of feature vectors to the performance of malware detection models. Yet, dynamic analysis is a time-consuming task and requires a secure environment, leading to detection delays and high costs for maintaining the analysis infrastructure. In this paper, we first introduce a method of constructing aggregated features via concatenating latent features learned through deep learning with *equally-contributed dimensions*. We then develop a *knowledge distillation* technique to transfer knowledge learned from aggregated features by a teacher model to a student model trained only on static features and use the trained student model for the detection of new malware samples. We carry out extensive experiments with a dataset of 86 709 samples including both benign and malware samples. The experimental results show that the teacher model trained on aggregated features constructed by our method outperforms the state-of-the-art models with an improvement of up to 2.38% in detection accuracy. The distilled student model not only achieves high performance (97.81% in terms of accuracy) as that of the teacher model but also significantly reduces the detection time (from 70 046.6 ms to 194.9 ms) without requiring dynamic analysis.

Keywords: Knowledge distillation · deep learning · 1D-CNN · machine learning static malware analysis dynamic malware analysis

1 Introduction

Malicious software, also known as *malware*, is targeted to steal information of computer users, spread the virus into the computer networks, encrypt data for ransom, or do other nefarious purposes. Detecting malware that could be legitimately downloaded by authorized users is a primitive feature of a reliable computer system. To this end, malware detection models using machine learning or deep learning have been proposed to classify samples as malicious or benign. The suspicious samples are analyzed by malware analysis techniques to extract the features used by the detection model.

Malware analysis techniques are categorized into two types: *static* analysis and *dynamic* analysis. Static analysis extracts features of examined samples by dissembling the samples using reverse engineering tools and looking for specific strings or patterns in the binary code; whereas dynamic analysis detonates the samples in an isolated (and secure) environment (i.e., sandboxes) to obtain runtime behavior. While static analysis is fast to collect features, it cannot deal with code obfuscation (e.g., polymorphism, metamorphism, or compression) techniques [19,21]. On the other hand, dynamic analysis is more robust against code obfuscation but it requires executing the samples in a sandbox, which is a time-consuming task [27]. In addition, dynamic analysis cannot detect malware equipped with evasive techniques such as anti-virtualization or time delay.

Existing works [5,10,26,29] use either dynamic features[1] [14,26,27], static features [2,17,29], or aggregation of static and dynamic features [10] to train a detection model. As presented in [10], the model trained with aggregated features can generally boost detection performance as it overcomes the drawbacks of static analysis and dynamic analysis. However, the selection of static and dynamic feature sets for aggregation is a crucial task as it can positively or negatively affect the model's performance. Furthermore, our preliminary experiments also show that naively concatenating static and dynamic feature vectors with heterogeneous dimensions faces an issue of the imbalanced contribution of the feature vectors to the model's performance. The longer feature vector may dominantly contribute to the model's performance while the shorter feature vector has less impact [23]. This also increases the dimension of the aggregated feature vector, thus subsequently increasing the model size and complexity. The work in [9] proposes to prune near-zero weights from large neural networks or quantize the model by using fewer bits for weights and biases but sacrificing the model performance. In this paper, we continue advocating the aggregation of static and dynamic features. We propose a novel aggregation approach that combines the latent representation of dynamic and static feature vectors with an equal size to train a detection model.

While aggregating dynamic and static features improves the performance of the detection model as both feature vectors complement to overcome the drawbacks of each other, the inference process for a new sample is still time-consuming

[1] Dynamic/static features are features extracted from dynamic/static analysis method.

due to the extraction of runtime behavior through dynamic analysis. To address this challenge, we propose a transfer learning approach using knowledge distillation (KD) [13] to transfer knowledge from a large *teacher* model trained with aggregated features to a small *student* model trained with only static features. The training of the student model aims to minimize a distillation loss function that is a weighted combination of cross-entropy loss and Kullback-Leibler (KL) divergence loss. Consequently, the trained student model enables a fast and efficient inference as it can leverage both rich features from aggregated features (i.e., dynamic and static features) while it does not require malware samples to be analyzed in sandboxes. This significantly reduces detection time and addresses the privacy issues of the sandboxes deployed in clouds. The trained student model can be deployed on commodity computers to perform malware detection and protect them from attacks.

In summary, we make the following contributions to this work:

- We propose a method of feature aggregation by using a deep learning model to learn the latent representations of static and dynamic feature vectors. The latent representations will have the same size, thus equally contributing to the malware detection model.
- We develop novel one-dimensional convolutional neural network (1D-CNN) architectures, each being used to train a detection model for individual static and dynamic feature vectors and the aggregated feature vector.
- We develop a knowledge distillation technique to transfer rich knowledge from a large teacher model trained with the aggregated feature vector to a small student model trained only with static features.
- We carry out extensive experiments to evaluate the proposed feature aggregation approach and the performance of the student model obtained from the knowledge distillation.

The rest of the paper is organized as follows. Section 2 discusses the related work. Section 3 presents the background of malware analysis and feature extraction. Section 4 presents deep learning-based malware detection models. Section 5 presents our proposed transfer learning approach for fast and efficient malware detection. Section 6 presents extensive experiments to evaluate the performance of the proposed approach. Section 7 concludes the paper.

2 Related Work

2.1 Static Malware Analysis

Static analysis is used to extract syntactical information (referred to as static features) from the binary Windows portable executable (PE) files such as imported and exported functions, base addresses of the section headers, debug information, or even operational codes (via disassembly tools such as Ghidra [6] or IDA Pro [12]). Operational code (opcode)-based static features have been widely used in literature [7,15,29,35]. In [7], the authors proposed to use the frequency of

top k sequences of the n-gram opcodes to reduce the useless information caused by undiscriminating instructions. Similarly, in [15], the authors proposed two opcode-based feature vectors used for malware detection: the frequency (i.e., the number of occurrences) and binary (i.e., whether the n-gram opcodes exist or not). In [35], the authors implemented a CNN model for malware detection and type classification (e.g., Backdoor, Trojan, Worm, etc.) using static features. The authors disassembled Windows binaries using the W32dasm [32] tool and integrated two extracted static features: API call frequency and opcode bigram. Such works that rely on only static features are vulnerable to obfuscation techniques and packing, which constructs samples to be hard to disassemble or extract their opcodes (anti-reverse engineered).

2.2 Dynamic Malware Analysis

The dynamic analysis uses a virtual environment (e.g., virtual machines and sandboxes) to collect malware runtime behavior. Based on dynamic analysis reports, specific features are extracted to build detection models to flag suspicious behavior such as an unusual pattern of API/system calls, or calls to the attacker's command-and-control (CnC) servers. Recently, Zhang *et al.* [34] proposed a deceptive engine called Scarecrow to exploit evasion techniques by transforming or camouflaging the physical end-user environment into an analysis-like environment from the view of evasive malware. But the battle between malware authors and defenders is never-ending. Novel evasive techniques exploit an observation that files are detonated in sandboxes emulating physical machines but without human interaction. So attackers develop malware that remains passive until it detects signs of human users, e.g., a mouse click, intelligent response to dialog boxes, or the user scrolls to the second page of a Rich Text Format (RTF) document [1]. Another tactic used by attackers is simply to delay the execution of the malicious code by adding extended sleep calls as malware samples are detonated in automatic sandbox environments within a short period (e.g., 2 minutes by default in the Cuckoo sandbox). Many reported malware (e.g., Trojan Nap–February 2013, Kelihos Botnet–2011) refrain from their suspicious behavior through a monitoring process (e.g., 10 minutes in Nap Trojan code).

2.3 AI-based Malware Detection

Static and dynamic features have their advantages and drawbacks. Thus, using heterogeneous features would help achieve harder-to-be-bypassed detection models. In [5], the authors compared the performance of HMM model on dynamic, static, and hybrid (train on dynamic and test on static) feature sets. Their results show the best performance obtained with the dynamic feature set. In [20], the authors developed a hybrid framework to classify Windows samples into benign and malicious using LCS and cosine similarity-based machine learning algorithms. The algorithms are trained on combined static and dynamic API sequences to learn malicious and benign behavior. In [10], the authors developed MalDAE that analyzes the correlation and semantic mapping between static

and dynamic API sequences. The hybrid API sequences of samples are used to train machine learning models to detect malicious samples. Compared to our proposed model, the above works rely only on the sequence of the API calls and ignore the API arguments. In [31], the authors proposed to convert binary files to images with pre-defined sizes and used a convolution neural network for malware detection. This approach is similar to static analysis as malware signatures create byte sequences that differentiate each other.

2.4 Transfer Learning for Malware Detection

Recently, Zhi *et al.* [36] used knowledge distillation (KD) to transfer knowledge from a large teacher model to a lightweight student model, both using dynamic features for malicious Android app detection. The student model still depends on dynamic analysis, which incurs long delays due to the execution of samples in an Android emulator environment to extract runtime behavior during the detection phase. In contrast, our lightweight student model uses only static features, which are fast to obtain and can be easily deployed on commodity computers.

3 Malware Analysis and Feature Extraction

In this section, we briefly introduce our static and dynamic malware analysis for feature extraction.

3.1 Static Analysis and Feature Extraction

We use two feature extraction methods to extract two feature vectors, namely EMBER and OPCODE (OPerational CODE).

EMBER Features. We use the Library to Instrument Executable Formats (LIEF) [25] to parse PE files, then extract EMBER (Endgame Malware BEnchmark for Research) features [2]. Specifically, the EMBER feature vector consists of eight groups: five groups of parsed features (e.g., general file information, header information, imported and exported functions, and section information), and three groups of format-agnostic features (e.g., byte histogram, byte entropy histogram, and string extraction). Eventually, we obtain a feature vector whose size is 2381 for each malware sample.

OPCODE Features. We use a free reverse engineering tool, Ghidra [6], to perform static analysis on binary PE files, resulting in an output report containing the OPCODE sequence. We construct 3-gram OPCODEs, each being a sequence of 3 consecutive OPCODEs and considered as a feature in the OPCODE feature vector. As the total number of 3-gram OPCODE is large (more than a million in our experiments), we first conduct feature reduction by excluding one near-identity dummy features that have Pearson correlations close to 1 (e.g., above a threshold of 0.95). And then, we do feature selection based on Mutual Information (MI), which is also used for feature selection of n-gram OPCODE

in Android malware detection [15]. Specifically, we calculate MI of each 3-gram OPCODE feature to a classification class (i.e., benign or malicious) to select the top 3-gram OPCODEs (e.g., above 98^{th} percentile). We empirically obtain an OPCODE feature vector whose size is 33 338 for training malware detection models and prediction of new samples.

3.2 Dynamic Analysis and Feature Extraction

We adopt the API-ARG feature extraction method developed in [26][2] to construct the dynamic feature vector. Here, we briefly describe the method for the sake of completeness. We use a Cuckoo sandbox host deploying with a Windows-7 OS on a virtual machine to capture the runtime behavior of malware samples. Based on the behavioral analysis JSON reports from the Cuckoo sandbox, we extract sequences of both Application Programming Interfaces (API) and their arguments. Suffixes of API names (e.g., `Ex, A, W, ExA, ExW`) are removed and similar API calls will be merged, e.g., `FindFirstFileExW` and `FindFirstFileExA` are merged into `FindFirstFile`. After pre-processing API arguments (e.g., convert integer argument into logarithmic bins, categorize special directory/path/url such as `system32`, specify if special registry keys or special commands are available), the API name is concatenated with each of its arguments to form separate string features. As a result, the number of features is a product of the number of API names and the number of their arguments. Finally, a Hashing Vectorizer function [30] is used to encode these string features into a binary vector, whose length is $2^{20} = 1\,048\,576$.

3.3 Feature Aggregation

We combine the static feature (i.e., EMBER and OPCODE), and the dynamic feature (API-ARG) to construct an aggregated feature vector as another feature engineering technique. The intuition of combining both the static and dynamic features is to obtain complimentary advantages of each type of feature. Suppose we aggregate a static feature vector $\mathbf{X}_{\text{static}}$ with input dimension d_s and a dynamic feature vector $\mathbf{X}_{\text{dynamic}}$ with input dimension d_d, intuitively, we can either (i) combine them in a weighted sum with padding zeros for a smaller dimensional vector, or (ii) concatenate them to produce a larger vector with dimension $d_s + d_d$. However, the above approaches face an issue of imbalanced contribution from individual feature vectors, especially when one vector is dominantly larger than the other (e.g., the API-ARG vector with 1 048 576 dimensions is much larger than the EMBER vector with 2381 dimensions).

In order to tackle the imbalanced contribution issue, instead of directly using raw feature vectors, we propose to extract and compress each raw feature vector into a smaller dimensional space of the same size. This can be done using a deep neural network as a feature extractor. With equal-sized latent representations, we can eliminate the imbalanced contribution issue when aggregating static and

[2] In [26], API-ARG was reported with a different name `Method2`.

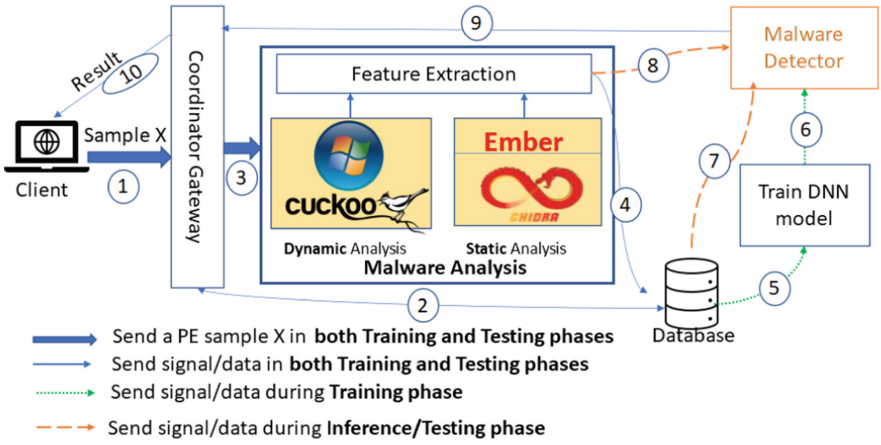


Fig. 1. Deep learning-based malware detection system.

dynamic feature vectors. Subsequently, we can either (i) concatenate the latent representation vectors forming a longer vector or (ii) combine these vectors via a weighted sum manner to the same dimensional vector. In this work, we choose the first option and then feed the aggregated vector into a fully connected (FC) layer for classification. The rationale behind this is that weighted parameters of the FC layer can be interpreted as tunable weights of the weighted sum of not only two representation vectors but also within different features in the projected vector. So concatenating two (or more) latent representation vectors and then feeding them into an FC layer is considered a comprehensive way to aggregate the latent representation vectors. We provide further details on the architectures of deep neural networks in Sect. 4.2.

4 Deep Learning-Based Malware Detection Models

Figure 1 shows workflows of training and testing/deploying stages in our deep learning (DL)-based malware detection system. First, a *Client* sends a portable executable (PE) sample X to a *Coordinator Gateway* for virus-scanning, as shown in step ①. The Coordinator Gateway calculates the hash value of the PE file, and checks the availability of its dynamic and static features in a *Database*, as shown in step ②. If they are available, the extracted features of the sample are retrieved, and proceed in step ⑤ to the model training phase or step ⑦ in the testing phase. If the PE file is new to the system, will be analyzed in the *Malware Analysis* module (step ③) (including *Dynamic Analysis* and *Static Analysis*), and output reports are passed through the *Feature Extraction* module to obtain dynamic features and static features (which have been presented in Sect. 3). These features are stored in the Database as shown in step ④, and sent to either: the *Malware Detector* module via step ⑧ if the system is in the inference stage, or the *Train DNN model* module via step ⑦ if the system is in

the training stage. The trained model is deployed for inference in step ⑥. In the inference stage, the detection result is sent back to the client via the Coordinator Gateway, as shown in lines ⑨ and ⑩.

The *Train DNN model* module trains a DL model for a binary classification task (i.e., benign or malicious) in a supervised learning manner. Instead of using a single deep neural network trained on a particular feature vector, we develop multiple neural network architectures, each being used for a feature vector either static, dynamic, or aggregated feature vectors. Below, we present DL-based network architectures for various feature vectors.

4.1 Neural Network Architectures for Individual Feature Vectors

Inspired by the success of residual deep neural networks [11] in computer vision, we design three one-dimensional residual deep neural networks, namely ResNet1D for learning the latent representation of individual feature vectors (i.e., EMBER, OPCODE, and API-ARG). Unlike images that have a two-dimensional input, malware features are represented as one-dimensional vectors. Therefore, we design our ResNet1D with a high stride step (up to 8 steps) within each ResNet1D-block. In Fig. 2, the Residual Conv1D block-1 shows the details of the ResNet1D-block that consists of two paths: (i) the bottom path is the main learning feature path, and (ii) the top path is a residual shortcut path. This design is similar to the *bottleneck* block [24] of ResNet. Due to the size of the model and compressing ratio design for each layer, we implement a single ResNet1D block for each layer to keep the model small while obtaining good performance. Detailed neural network architectures for the EMBER feature vector, OPCODE feature vector, and API-ARG feature vector are shown in Table 9, Table 10, and Table 11 (in Appendix B), respectively. For all three networks, we consider an expected number of layers for the whole 1D-CNN model and the size of each individual feature vector. We design stride steps and kernel sizes of these layers to incrementally compress input dimensions and end up with 1-dimensional output; while the padding (either 1 or 0) is added in order to make the last sliding window well fit the kernel size. Together with compression of the input dimensions, we expand the model in width by increasing the number of output channels, i.e., mostly double after each layer. These architectures were obtained after extensive experiments for parameter optimization.

Besides the ResNet1D block, we also consider other recent advanced variants of ResNet in the computer vision domain such as ResNeXt [33], inverted ResNeXt [33], and ConvNeXt [18]. We adopt key techniques of these variants used with conv-2D, implement them to conv-1D and adjust the ResNet1D block to construct ResNeXt-1D, Inverted RestNeXt-1D, and ConvNeXt-1D blocks, which are presented in Appendix C. Since we only replace the basic block architecture, we reuse the high-level network architecture as presented in Table 9, Table 10, and Table 11 for each individual feature vector. We changed the first layer of Conv1D with `kernel` $= 3$ and `padding` $= 1$ instead of `kernel` $= 1$ and `padding` $= 0$ after extensive ablation studies (Sect. A.2).

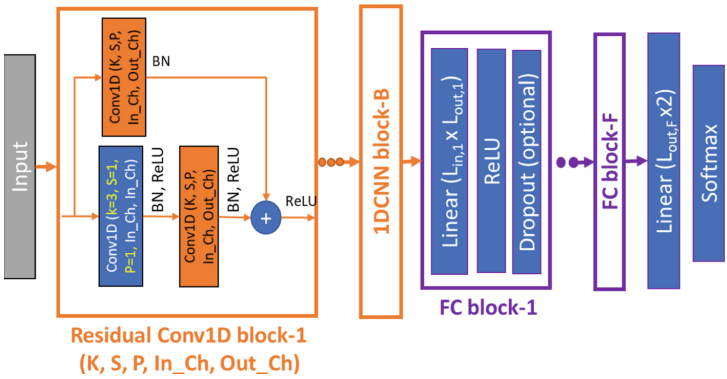


Fig. 2. Architecture of 1D-CNN with Residual Blocks.

4.2 Neural Network Architecture for Aggregated Feature Vectors

To construct the aggregated feature vector, besides the naive approach of concatenating original feature vectors into a long vector, we introduce a novel approach of using latent presentation learned by a 1D-CNN extractor.

Concatenate Original Feature Vectors. We can concatenate two static original feature vectors (EMBER with 2381 dimensions) and OPCODE with 33 338 dimensions) and dynamic original feature vector (API-ARG with 1 048 576 dimensions). However, this naive aggregation method yields an imbalanced contribution from three individual feature vectors. The largest feature vector—API-ARG—dominantly affects the aggregated feature vector. In Appendix C, we present 1D-CNN architectures for concatenating 2 original feature vectors (EMBER + API-ARG) and 3 original feature vectors (EMBER + OPCODE + API-ARG) in Table 12 and Table 13, respectively.

Concatenate Latent Representation Vectors. Each individual feature vector is passed through a respective 1D-CNN extractor for learning a latent representation vector, which ends up with 384 dimensions. Hence, concatenating them will eliminate the imbalanced contribution issue. The aggregated feature vector is then fed into a few layers of a feed-forward network to classify samples as either benign or malicious. Figure 3 shows the architecture of the entire deep learning model including 1D-CNN models for learning latent representation of original feature vectors and a feed-forward network for classification.

While the model trained with the aggregated feature vector leverages the advantages of static and dynamic features, the inference stage is still a time-consuming task for new samples that have not been analyzed in a sandbox to extract dynamic features. To address this challenge, we propose to use knowledge distillation to transfer knowledge from a large *teacher model trained on the aggregated feature vector* to a small *student model trained on a single static feature vector*. In the next section, we present such a knowledge distillation technique.

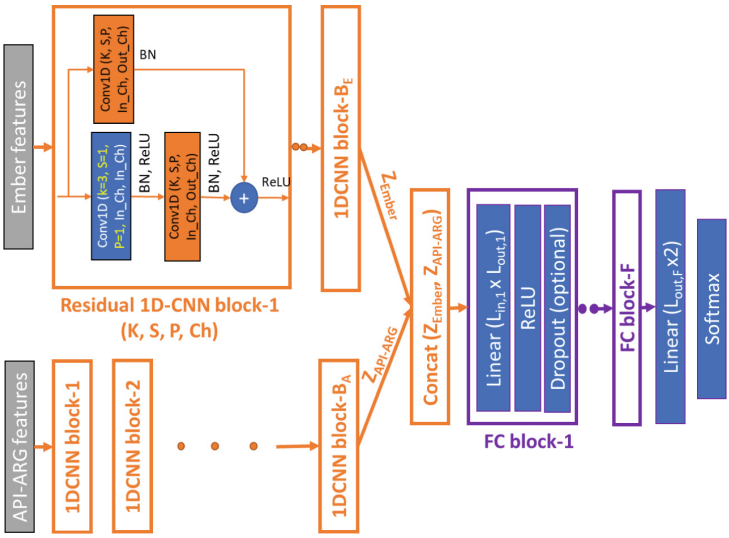


Fig. 3. Architecture of 1D-CNN for Learning Latent Representation and a Feed Forward Network for Classification.

5 Transfer Learning with Knowledge Distillation

Transfer learning is a technique of training a new model or fine-tuning a pre-trained model to adapt to a new task or a new domain. While transfer learning keeps the network architecture nearly the same or at least the first few feature extraction layers as the original model, the knowledge distillation (KD) approach does not have this constraint. Indeed, knowledge distillation tends to transfer the knowledge from a large *teacher* network to a smaller student network with an aim of approximating the performance of the *student* network as well as that of the teacher network. KD was first introduced by Bucila *et al.* [3] for model compression and became popular after Hinton *et al.* [13] generalized it. With a smaller architecture, the student model typically requires fewer computing resources and enables a fast inference. In this work, we adopt the KD idea to the cybersecurity domain with multiple objectives. We not only transfer knowledge from a large-complex network to a smaller one to achieve the original purpose of KD but we also leverage static and dynamic features of malware analysis to transfer the knowledge of the model trained on aggregated features to a model trained on static features only.

In order to perform KD, we define a joint loss function of both ground-truth labels (i.e., "hard" labels) and "soft" labels obtained from the distribution of class probabilities predicted by the teacher model. Specifically, the soft labels are the output of a `softmax` function of the teacher model's logits. However, in many cases, the probability distribution outputs the correct class with a very high probability and very close to zero for other classes. So, it does not provide much information beyond the ground-truth labels (i.e., hard labels). To address

this issue, Hinton *et al.* [13] proposed the concept of "softmax temperature". The softened `softmax` with a scaling temperature τ of a network f, given input $\boldsymbol{x}$ and the last output logit $\boldsymbol{z}$ has a probability denoted by $p_i^f(\tau)$ of class i calculated from the logit $\boldsymbol{z}^f$, and it is defined as follows:

$$p_i^f(\tau) = \frac{\exp\left(z_i^f/\tau\right)}{\sum_j \exp\left(z_j^f/\tau\right)}. \tag{1}$$

When $\tau = 1$, we get the standard `softmax` function. When τ increases, the probability distribution generated by the `softmax` function becomes softer, providing more fine-grained information about the similarity between the predictions of the teacher model and the ground-truth labels.

During the training of deep neural networks, we typically minimize cross-entropy loss between the output probabilities (i.e., after `softmax` function) and the target labels. In knowledge distillation, the objective of training a smaller student model is to minimize a linear combination of two losses: the cross-entropy (CE) loss with hard labels (as usual) and the Kullback-Leibler (KL) divergence loss between softened probability predictions of the student model and the teacher model. The combined loss is computed as follows:

$$\mathcal{L}_{KD-KL}(\boldsymbol{x}; \boldsymbol{W}) = \alpha \mathcal{L}_{CE}(\boldsymbol{p}^S(1), \boldsymbol{y}) + (1-\alpha)\mathcal{L}_{KL}(\boldsymbol{p}^T(\tau), \boldsymbol{p}^S(\tau)), \tag{2}$$

where $\boldsymbol{x}$ is the input feature vector, $\boldsymbol{W}$ is the parameter set of the student model, S is the student model, T is the teacher model, $\boldsymbol{y}$ is the ground-truth label, and α is a hyper-parameter of the linear combination. $\boldsymbol{p}^S(\tau)$ and $\boldsymbol{p}^T(\tau)$ are the softened probabilities with temperature τ of the student and teacher models, respectively. The cross-entropy loss is defined as in Eq. (3), and the KL-divergence loss is defined as in Eq. (4):

$$\mathcal{L}_{CE}(\boldsymbol{p}^S(1), \boldsymbol{y}) = \sum_i -y_i \log p_i^S(1), \tag{3}$$

$$\mathcal{L}_{KL}(\boldsymbol{p}^T(\tau), \boldsymbol{p}^S(\tau)) = \tau^2 \sum_i p_i^T(\tau) \log \frac{p_i^T(\tau)}{p_i^S(\tau)}. \tag{4}$$

In [13], the standard choice of α is 0.1 but it was also examined with a value of 0.5. The scaling temperature τ was recommended to take a value in the set of $\{3, 4, 5\}$ [13,16].

Figure 4 shows the process of knowledge distillation from a teacher model trained on the aggregated feature vector (e.g., aggregation of $\mathbf{X}_{\text{static}}$ and $\mathbf{X}_{\text{dynamic}}$) to a student model trained with a static feature vector only (i.e., $\mathbf{X}_{\text{static}}$). During the training process, a sample X is passed through the Malware Analysis module (see Fig. 1) including dynamic analysis and static analysis. The analysis reports are processed to extract the dynamic and static feature vectors, namely $\mathbf{X}_{\text{dynamic}}$ and $\mathbf{X}_{\text{static}}$, respectively. We aggregate dynamic and static feature vectors to train the teacher model (ref. Section 4.2). Subsequently, we use the teacher model to obtain the penultimate output layer (a vector of 384 dimensions), which is the `logit` vector Z^T in Fig. 4. Finally, we train the student model

on the static feature vector $\mathbf{X}_{\text{static}}$ with a loss function defined as a weighted combination of the CE loss and a KL-divergence loss.

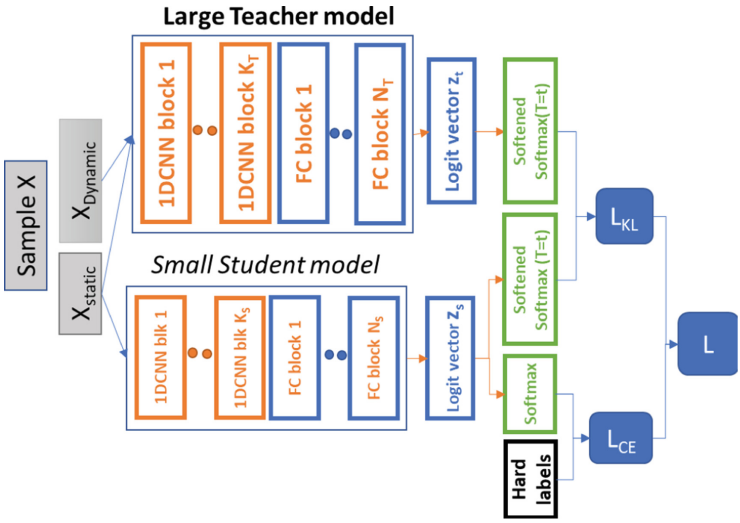


Fig. 4. Knowledge Transfer from a Teacher model to a Student model with knowledge distillation (KD).

As Kim *et al.* [16] showed that a large value of τ, strong softening, leads to *logit matching*, whereas a small τ results in *label matching*. They suggested that "`logit` matching has a better generalization capacity than label matching" when the data has correct labels. As a result, they proposed to use mean-squared error (MSE) loss to directly match the output `logit` of the teacher model and those of the student model, yielding better performance in most of the cases with correct labels, and remove an unreasonable assumption of zero-mean of the `logit` generated by the student model [13,16]. Furthermore, MSE loss does not require fine-tuning a suitable temperature to smooth the output probabilities. The KD loss function for enhancing the `logit` matching is then defined as follows:

$$\mathcal{L}_{KD-MSE}(\boldsymbol{x};\boldsymbol{W}) = \alpha\mathcal{L}_{CE}(\boldsymbol{p}^S(1),\boldsymbol{y}) + (1-\alpha)\mathcal{L}_{MSE}(\boldsymbol{z}^T,\boldsymbol{z}^S), \tag{5}$$

where $\mathcal{L}_{MSE}(\boldsymbol{z}^T,\boldsymbol{z}^S) = ||\boldsymbol{z}_S - \boldsymbol{z}^T||_2^2$. In this paper, we examine both loss functions, namely KD-KL loss in Eq. (2), and KD-MSE loss in Eq. (5) during the knowledge distillation from the teacher model to the student model.

Maximizing Transferred Knowledge with Ensemble of Multiple Teacher Models. Due to the randomness during the training process of the teacher model (e.g., using various training datasets, sample shuffle with batches), multiple teacher model instances could have different performances. To maximize

the amount of knowledge transferred to the student model, we use an ensemble approach to combine predictions of multiple teacher model instances in an average manner before computing the loss during the training of the student model, thus reducing the variance of the predictions of the teacher models. It is to be noted that the ensemble model increases the inference delay, but it only applies during knowledge distillation (i.e., training of the student model) from the cumbersome ensemble teacher model to the student model.

6 Performance Evaluation

6.1 Dataset

We use a private dataset that is bought from a security vendor in 2020. The dataset consists of 86 709 Windows portable executable (PE) samples (including 42 742 benign and 43 967 malicious samples)[3]. We split the dataset with an 80:20 ratio into a training set with 69 367 samples and a test set with 17 342 samples. We then split the training set with an 80:20 ratio into a development set and a validation set to fine-tune the best hyper-parameters (in Sect. 6.2). Subsequently, we use the best hyper-parameters to train the final model on the whole training set, and test on the test set for performance evaluation. We perform static and dynamic analysis as described in Sect. 3 to extract EMBER features, OPCODE features, and dynamic API-ARG features for each sample. To speed up the dynamic analysis process, we develop a distributed Cuckoo infrastructure that allows concurrent analysis of multiple samples in multiple sandboxes. A detailed description of the infrastructure is given in Appendix D.

Diversity of Malware Types in Dataset. As the security vendor does not provide the type of each sample, we use Kaspersky to determine the type of malicious samples and Detect it Easy (DIE) tool [8] to determine the type of benign samples. In Table 1, we present the results of our analysis. The results show that our dataset includes diverse malware and benign samples, making the detection model generalized to existing types of malware and benign samples. It is worth mentioning that there are 3 160 samples that we were not able to obtain their type (indicated as "Other" in Table 1). These include 3 145 samples that are classified as benign and 15 samples whose analysis report is not available.

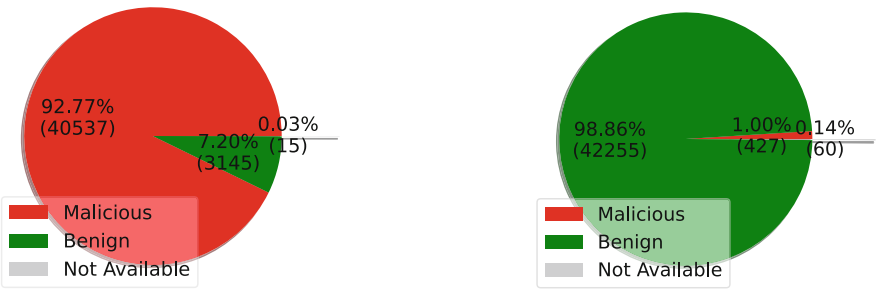
Ground-Truth Label Verification. We use the labels provided by the security vendor to train the detection model in a supervised learning approach. To avoid the scenario that the detection model is biased toward the ground-truth labels of the security vendor, we use Kaspersky to verify the ground-truth labels provided by the security vendor. In Fig. 5, we present the matching between the ground-truth labels provided by the security vendor and the labels obtained from Kaspersky. We observe that the labels provided by the security vendor

[3] The hash value of samples will be provided on request for experiment reproducibility.

Table 1. Dataset Description

	Types	**#Samples**	**Types**	**#Samples**
Malicious Samples	Trojan	16 154	Packed	50
	Virus	12 144	HackTool	44
	Worm	4 242	Exploit	30
	Adware	3 179	Rootkit	14
	Backdoor	2 661	Porn	10
	Risktool	1 290	RemoteAdmin	7
	Downloader	549	PswTool	5
	Hoax	153	Monitor	3
	DangerousObject	150	NetTool	2
	WebToolBar	119	Eicar	1
	Other	3 160		
Benign Samples	DLL	37 527	GUI	4 432
	Console	669	Driver	114

and Kaspersky agree for more than 92.77% of malicious samples and 98.86% of benign samples. There is 7.20% of samples detected as malicious by the anti-virus engine of the security vendor but undetected by Kaspersky. There are also 1% of samples considered benign by the anti-virus engine of the security vendor but detected as malicious by Kaspersky. Without loss of generality, we use the labels provided by the security vendor to train our detection models.



(a) Malicious Samples. (b) Benign Samples.

Fig. 5. Matching of sample labels between Kaspersky and the anti-virus engine of the security vendor. Percentage of matching and the number of samples.

6.2 Hyper-parameters Setting

In this paper, we use stochastic gradient descent (SGD)[4] optimizer with a momentum of 0.9 to train all the deep learning models for 100 training epochs. The initial learning rate is set as 0.02 and reduced by 10 times at the epoch 50^{th} for EMBER, API-ARG, and aggregated feature vectors. For OPCODE features, we also use the initial learning rate of 0.02 but reduce it by 10 times at the epoch 30^{th} and 80^{th}. We also use a weight-decay of 0.001 as a regularizer during the training of the models. Besides 1D-CNN models, we also train the XGBoost model, which is considered the state-of-the-art results in prior work [26]. We use the default parameters of the open-source XGBoost Python-based package [4].

Table 2. Performance (in percentage) on the test set of 1D-CNN (average of 5 runs, and an ensemble model) and XGBoost

Feature	Models	Accuracy	F1-score	FPR	FNR
EMBER	1D-CNN	97.62 ± 0.08	97.66 ± 0.08	2.46 ± 0.15	2.29 ± 0.12
	XGBoost	97.66	97.70	2.55	2.13
OPCODE	1D-CNN	95.91 ± 0.08	95.95 ± 0.08	3.78 ± 0.11	4.39 ± 0.11
	XGBoost	95.54	95.59	4.16	4.74
API-ARG	1D-CNN	96.64 ± 0.13	96.69 ±0.13	3.29 ± 0.20	3.42 ± 0.23
	XGBoost	96.48	96.55	4.01	3.03

6.3 Performance Metrics

We use accuracy, F1-score (F1-score=$\frac{2\text{TP}}{2\text{TP+FP+FN}}$) to measure the performance of different DL-based models. Moreover, for malware detection tasks, the false negative rate (FNR = $\frac{\text{FN}}{\text{FN+TP}}$) and false positive rate (FPR = $\frac{\text{FP}}{\text{FP+TN}}$) are also imperative metrics to compare the performance of detection models, where FN, FP, TN, TP are false negative, false positive, true negative, and true positive predicted samples, respectively. The smaller the values of FPR and FNR, the better the detection model.

6.4 Performance of Teacher Model

Experimental Results with Individual Feature Vectors. Table 2 shows detection performance on the test set of 1D-CNN models and XGBoost models[5] trained on individual feature vectors. We observe that for OPCODE and API-

[4] After experimenting with several optimizers, e.g., Adam, AdamW, RMSprop, SGD, and SGD with momentum, we observed that SGD with momentum performs best.

[5] There is no variance of XGBoost's result because it does not include any stochastic in models; therefore yielding the same results regardless of running many times.

ARG features, the average performance of 1D-CNN models (over 5 different trained models) is much higher than those of XGBoost models, which are state-of-the-art models used in [2,26]. For EMBER features, the average accuracy of 1D-CNN models is slightly lower than that of XGBoost models, but our best model (among the 5 trained models) outperforms the XGBoost models.

Experimental Results with Aggregated Feature Vectors. Among three individual feature vectors shown in Table 2, the models trained on the EMBER feature vector outperform the models trained on OPCODE and API-ARG feature vectors by a significant margin. Therefore, we always use EMBER features in aggregated features. We conduct two ways of feature aggregation:

- `Agg2` concatenates EMBER and API-ARG;
- `Agg3` concatenates EMBER, OPCODE, and API-ARG.

With two feature aggregation methods described in Sect. 4.2, we evaluate 4 possible aggregated features as follows:

- `Agg2-Org` concatenates original feature vectors of EMBER and API-ARG,
- `Agg2-Lat` combines latent representation vectors of EMBER and API-ARG,
- `Agg3-Org` combines original vectors of EMBER, OPCODE and API-ARG,
- `Agg3-Lat` concatenates latent representation vectors of EMBER, OPCODE and API-ARG.

Table 3 shows experimental results of 1D-CNN models and the XGBoost model. We can see that 1D-CNN models of `Agg2-Lat` and `Agg3-Lat` obtain a detection accuracy (both average and ensemble) higher than those of `Agg2-Org` and `Agg3-Org`, respectively. It is because `Agg2-Lat` and `Agg3-Lat` combine equally-dimensional latent representation vectors that learn important latent features from the original feature vectors. Specifically, in `Agg2-Org` and `Agg3-Org`, the original feature vector of API-ARG with high dimension significantly dominates other static original feature vectors. In contrast, `Agg2-Lat` and `Agg3-Lat` concatenate the latent representation vectors with the same dimension. The 1D-CNN model trained on `Agg2-Lat` feature vector obtains an average test accuracy of 97.95% that is higher than the performance of the 1D-CNN model trained with `Agg2-Org`; and its ensemble model achieves the highest accuracy, F1-score, and the lowest FNR among all the evaluated models (i.e., 98.11% for accuracy, 98.14% for F1-score, 1.88% for FPR and 1.89% for FNR).

We also observe that including more static features (i.e., concatenating both EMBER and OPCODE feature vectors) into the aggregated feature vector does not always help improve the performance of the models. Specifically, compared to `Agg2-Org`, `Agg3-Org` only improves accuracy and F1-score for 1D-CNN models but reduces performance for XGBoost models as shown in Table 3. Similarly, compared to `Agg2-Lat`, `Agg3-Lat` deteriorates in all performance metrics for both 1D-CNN and XGBoost models. This experiment shows that OPCODE features have negative effects on the aggregated features, especially when aggregating latent representation vectors with the same dimension. Thus, we need to carefully select individual feature vectors that help improve the performance of

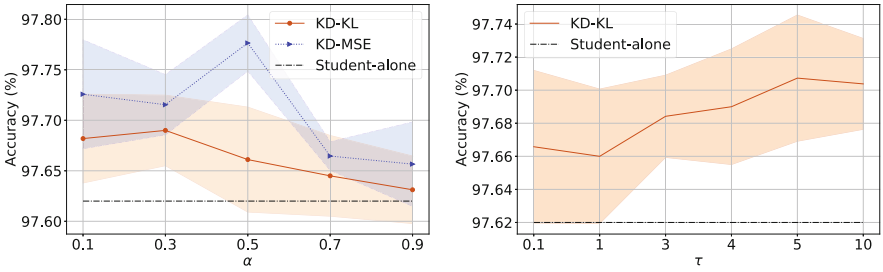
the models trained on the aggregated feature vector. Among all the methods, the ensemble of the models trained with `Agg2-Lat` obtains the best accuracy of 98.11%. We use this ensemble as the teacher model for transfer learning.

6.5 Transfer Learning for EMBER Features

In this section, we present experimental results of transfer learning from the teacher model trained with `Agg2-Lat` to a distilled student model trained with EMBER features. First, we fine-tune the scaling temperature hyper-parameter τ and the weighted scaling factor α for the loss functions in Eq. (2) and Eq. (5), then we evaluate the performance of the distilled student model.

Table 3. Performance (in percentage) of Teacher models trained with different aggregated feature vectors

Feature	Models	Accuracy	F1-score	FPR	FNR
Agg2-Org	1D-CNN	97.68 ± 0.05	97.72 ± 0.05	2.56 ± 0.15	2.08 ± 0.11
	Ensemble	97.89	97.76	2.11	2.42
	XGBoost	97.99	98.02	2.15	1.87
Agg2-Lat	1D-CNN	97.95 ± 0.04	97.98 ± 0.04	2.18 ± 0.11	1.92 ±0.12
	Ensemble	**98.11**	**98.14**	1.88	**1.89**
Agg3-Org	1D-CNN	97.85 ± 0.03	97.88 ± 0.03	2.16 ± 0.08	2.15 ± 0.08
	Ensemble	97.93	97.96	2.00	2.14
	XGBoost	97.96	97.99	2.21	1.87
Agg3-Lat	1D-CNN	97.90 ± 0.06	97.93 ± 0.06	2.20 ± 0.08	2.00 ± 0.09
	Ensemble	98.05	98.08	**1.82**	2.07



(a) With different weighted parameter α in KD-KL loss and KD-MSE loss functions. (b) With different values of τ in KD-KL loss (with the best $\alpha = 0.3$).

Fig. 6. Performance of Transfer Learning with EMBER Features.

Tuning α for KD-KL Loss and KD-MSE Loss Functions. As suggested in [13], the weighted parameter α should be 0.1, and the temperature $\tau \in \{3, 4, 5\}$. Nevertheless, we conduct a grid search to find the best α for our dataset. We transfer the knowledge from the teacher model to the distilled student models with EMBER features, with $\alpha \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$ and a fixed temperature $\tau = 4$. Figure 6a shows the accuracy of these distilled students. We observe that the distilled models using the KD-MSE loss perform better than the distilled models using the KD-KL loss. This result aligns well with the results presented in [16]. We also observe that the distilled student models with a small value of α perform better than those with a large value of α for both loss functions; it is because the distilled student models can learn more transferred knowledge from the "soft" labels predicted by the teacher models. Thus, we use the best value of α for KD-KL ($\alpha = 0.3$) and KD-MSE ($\alpha = 0.5$) for further evaluation. This result also aligns with the results presented in [13].

Table 4. Performance (in percentage) of student models transferred with different loss functions for EMBER features

Methods	Accuracy	F1-score	FPR	FNR
Student-alone	97.62 ± 0.08	97.66 ±0.12	2.45 ± 0.15	2.29 ±0.12
KD-KL	97.71 ± 0.04	97.74± 0.04	2.38 ±0.09	2.21 ± 0.14
KD-MSE	**97.78 ± 0.03**	**97.81 ± 0.03**	**2.26 ± 0.07**	**2.19 ± 0.08**

Tuning Scaling Temperature Hyper-parameter τ for KD-KL Loss. With the selected value of $\alpha = 0.3$ for KD-KL, we run experiments with different scaling temperatures $\tau \in \{0.1, 1, 3, 5, 7, 10\}$. The mean and standard deviation of the accuracy of the distilled KD-KL student models for EMBER features (average over 5 runs) are shown in Fig. 6b. We can see that when increasing τ's value, the accuracy of the distilled student model slightly improves. This performance trend aligns with results from [16]. The best accuracy is achieved with $\tau = 5$ that aligns with the recommendation in [13].

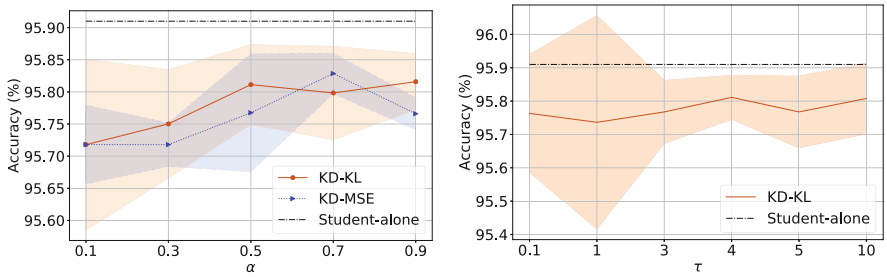
In Table 4, we present a performance comparison between the student-alone model (i.e., the model trained with EMBER features without KD), and transferred KD models with KD-KL loss and KD-MSE loss (with their best hyper-parameters). We observe that the distilled student models outperform the student-alone model in all the metrics. The best-distilled student model (KD-MSE) obtains an accuracy of 97.81%, which is 0.11% higher than that of the best student-alone model. Compared to the XGBoost model trained with OPCODE features (shown in Table 2), this distilled student model improves the detection accuracy by 2.38%. The results also show that the performance of the distilled student models approximates that of the teacher models (i.e., 97.81% vs. 98.11%) and is even better than the model trained with the aggregated feature vector (i.e., 1D-CNN with `Agg2-Org` achieves an accuracy of 97.68% shown in Table 3).

When aggregating latent representations of EMBER features and dynamic API-ARG features, `Agg2-Lat` can improve 0.33% of accuracy compared to the EMBER student-alone (98.11% in Table 3 vs. 97.62% in Table 4). But this incurs a significant delay in inference due to dynamic analysis and results in a large model (discussed further in Sect. 6.7). When using knowledge distillation to train a distilled KD-MSE student model, we obtain an improvement of 0.16% of accuracy, which is about 50% of the improvement of the model trained with `Agg2-Lat` but *without any additional cost or delay due to dynamic analysis*. The model also keeps unchanged in terms of size and inference time. It is to be noted that even though the absolute number of 0.16% of improvement is quite small, the standard deviation of the result is very small (0.03%), which shows reliable and consistent improvement.

6.6 Transfer Learning with OPCODE Features

We also study KD from the teacher model to a 1D-CNN student model trained with OPCODE features.

Tuning α for KD-KL Loss and KD-MSE Loss Functions. We conduct a similar fine-tuning of the α hyper-parameter for KD-KL loss and KD-MSE loss by using the same teacher as used for EMBER features (i.e., `Agg2-Lat`). The results are shown in Fig. 7a. We observe an interesting result that regardless of α for both loss functions, all the distilled student models trained with OPCODE features obtain a lower accuracy than the OPCODE student-alone model even including the upper bound of the shaded areas.



(a) With different values of hyper-parameter α in KL-loss (with $\tau = 4.0$) and MSE-loss. (b) With different hyper-parameter τ in KL-loss (with $\alpha = 0.5$).

Fig. 7. Performance of Transfer Learning with OPCODE Features.

Tuning scaling temperature hyper-parameter τ for KD-KL Loss. We fine-tune temperature hyper-parameter τ with the best $\alpha = 0.5$ obtained in Fig. 7a. As shown in Fig. 7b, the average accuracy of the distilled student models trained with OPCODE features and all the evaluated values of τ is lower than

Table 5. Comparison between student-alone model and distilled models trained with OPCODE features

Methods	Accuracy	F1-score	FPR	FNR
Student-alone	**95.91 ± 0.08**	**95.95± 0.08**	**3.77 ± 0.11**	4.40 ± 0.11
KD-KL-2F	95.81 ± 0.11	95.86±0.11	4.06±0.15	**4.32±0.14**
KD-MSE-2F	95.83 ± 0.03	95.87±0.03	3.93± 0.09	4.40±0.07
KD-KL-3F	95.75 ± 0.04	95.80±0.04	4.07± 0.17	4.43±0.16
KD-MSE-3F	95.78 ± 0.09	95.83 ±0.09	3.96±0.14	4.48±0.11

that of the student-alone model. We conjecture that the KD technique can only transfer knowledge from a teacher to a student model trained with a feature vector that has a positive contribution to the teacher model (e.g., EMBER); while it fails to do so if the feature vector used to train the student model has a negative contribution to the teacher model (e.g., OPCODE) when aggregation. We remind the reader about the results presented in Sect. 6.4, `Agg3-Lat` (with OPCODE) performs worse than `Agg2-Lat` (without OPCODE). It is also to be noted that the student-alone model trained with OPCODE features without KD has the lowest performance compared to the models trained with EMBER features or API-ARG features. This aligns well with the analogy that a student who is not strong at learning will not be able to acquire knowledge from the teacher even though the teacher is strong.

Since the results from Fig. 7a and Fig. 7b were obtained by transferring the knowledge from the teacher trained without OPCODE features to a student using OPCODE features, we conduct further experiments with transfer learning from a teacher with OPCODE features included in the aggregated feature vector (i.e., `Agg3-Lat`) to a student model trained with OPCODE features. We use the best hyper-parameter obtained above ($\alpha = 0.5, \tau = 4$ for KD-KL, and $\alpha = 0.7$ for KD-MSE) to conduct these experiments. The last two rows in Table 5 show the performance (in percentage) of the distilled student models trained with OPCODE features (i.e., KD-KL-3F and KD-MSE-3F) with knowledge transferred from softened labels obtained from the teacher model trained with `Agg3-Lat` feature vector. We can see that the performance of KD-KL-3F and KD-MSE-3F is even worse than that of KD-KL-2F and KD-MSE-2F. This is because the teacher model trained with `Agg3-Lat` feature vector performs poorer than the teacher model trained with `Agg2-Lat` feature vector. This aligns with another analogy that a poor teacher cannot transfer his/her knowledge to students and could deteriorate the performance of the students if the knowledge is not effective.

6.7 End-to-End Detection Delay

In Table 6, we report a breakdown of end-to-end (E2E) detection delay that consists of three components: (i) malware analysis delay, (ii) feature extraction

(*Feat-Extr.*) delay, and (iii) inference delay for different detection methods. We can see that (1) our proposed method—KD-MSE and EMBER method have the least E2E delay for detecting malware (i.e., 194.9 ms); and (2) detection methods involving dynamic analysis (i.e., API-ARG, and `Agg2-Lat`) have significantly long E2E delay compared to that of other methods (OPCODE, EMBER, and KD-MSE), at least 7 × longer.

Table 6. Breakdown of end-to-end detection delay (in ms)

Methods	Analysis	Feat-Extr.	Inference	E2E Delay
OPCODE	12 243.7	62.0	0.4	12 306.1
API-ARG	69 120.0[(a)]	720.0	1.2	69 841.2
Agg2-Lat	69 120.0	914.7	11.9[(b)]	70 046.6
EMBER	194.7[(c)]		0.2	**194.9**
KD-MSE	194.7[(d)]		0.2	**194.9**

[(a)] The delay of dynamic analysis of malicious samples is usually much higher than those of benign samples, e.g., the average delay to complete analysis of a malicious sample is 120 960 ms, while this figure for a benign sample is only 60 480 ms.
[(b)] Inference delay of the ensemble model from 11 models.
[(c)] For EMBER features, we measure analysis delay and feature-extraction delay jointly in a single program.
[(d)] Analysis delay for KD-MSE is similar to EMBER as it uses EMBER feature for detection.

Looking into each delay component of API-ARG and `Agg2-Lat` methods, we see the *Feat-Extr.* and *Inference* delays are minuscule compared to *Analysis* delay[6] (which consumes above 98% of E2E delay). Thus, removing dynamic *Analysis* delay will significantly speed up malware detection time in a real-world deployment. This explains why static analysis is widely used in many commercial-off-the-shelf antivirus products. The experimental results also show that among the static analysis-based methods, OPCODE needs a much longer time for inference compared to EMBER and KD-MSE in both feature extraction and inference. In summary, our KD-MSE approach achieves a detection as fast as the fastest static method (i.e., EMBER), while approximating the performance of the `Agg2-Lat` method that uses both dynamic and static features.

It is worth mentioning that reducing the inference/detection overhead is crucial for a detection model as delaying the detection affects the usability of the system and raises security issues: malware samples may compromise the host before being detected. Nevertheless, we note that the training time of the models (e.g., teacher model and student model) is a one-time cost and it could be

[6] Analysis delays are calculated as the average time of executing samples in the test set (i.e., 17 342 samples) using dynamic or static analysis.

done offline with sufficient computing resources. This reflects practical scenarios where security companies nowadays deploy large-scale computing systems in their premise for pre-training and updating detection models, which are in turn deployed on commodity computers of their end-users.

7 Conclusion

In this paper, we developed various 1D-CNN models for malware detection using static and dynamic features. We addressed a limitation of the feature aggregation that naively concatenates original static and dynamic feature vectors by using a 1D-CNN model to learn the latent representation of each original feature vector with a fixed size before aggregation. We tackled the dilemma between the benefits of dynamic analysis and its massive delay in the deployment phase by developing a *knowledge distillation* technique to *transfer* rich knowledge from a big teacher model (trained with aggregated features) to a small student model (trained with static features only). We carried out extensive experiments with a dataset of 86 709 samples to evaluate the performance and efficiency of our proposed technique. The experimental results show that the proposed technique outperforms existing methods with an improvement of up to 2.38% in terms of detection accuracy. The proposed technique also significantly reduces the end-to-end delay for sample prediction from 70 046.6 ms to 194 ms. In other words, the proposed technique enables fast and efficient malware detection. Our work also found that transfer learning is not always successful if the student feature vector has negative effects on the performance of the teacher when aggregated with the teacher feature vector.

A Ablation Studies

Table 7. Performance (in percentage) of API-ARG student models transferred with different loss functions

Methods	Accuracy	F1-score	FPR	FNR
Student-alone	96.64 ± 0.13	96.69 ±0.13	3.29 ± 0.20	3.42 ±0.23
KD-KL	**96.77 ± 0.02**	**96.81± 0.02**	**3.19 ± 0.07**	3.26 ± 0.06
KD-MSE	96.73 ± 0.08	96.77 ± 0.08	3.29 ± 0.13	**3.24 ± 0.08**

A.1 Transfer Learning for API-ARG Features

Even though transferring knowledge to a student model trained with dynamic features is not a focus of our work, we evaluate the performance of the distilled student models trained with dynamic API-ARG features in this section

for completeness. Since the training time for a student model trained with API-ARG features is much longer than that for the models trained with EMBER features or OPCODE features, we only evaluate one set of the recommended hyper-parameters from [13]: $\alpha = 0.1, \tau = 5$ for KD-KL, and $\alpha = 0.1$ for KD-MSE. As shown in Table 7, the distilled student models for API-ARG features perform better than the student-alone model. The performance trends also follow our conjecture that knowledge distillation helps transfer knowledge to a distilled student model with a dynamic API-ARG feature vector that *positively contributes* to the teacher model (i.e., `Agg2-Lat`). We also observe that the KD-KL student model obtains better performance compared to the KD-MSE student model, which is different from the results of the case where the student model is trained with EMBER features. This explains why we explored two loss functions during transfer learning for completeness, even though in [16] the authors stated that the KL-loss function is more suitable for a noisy label case, which is not our case since the labels of the malware samples are stabilized and provided by the security vendor.

A.2 Experiments with Different Neural Network Architectures

Besides ResNet1D used for all the experiments presented above, we adopt recent advanced neural network architectures of ResNet in computer vision such as ResNeXt [33], inverted ResNeXt [33], and recently ConvNeXt [18]. We adjust our ResNet1D basic block and implement its variants, namely ResNeXt1D, Inverted ResNeXt1D, and ConvNeXt1D basic blocks accordingly, presented in Fig. 8. We reuse the high-level neural network architectures as defined for ResNet1D in Table 9, Table 10, and Table 11 for each individual feature vector (i.e., EMBER, OPCODE, and API-ARG).

Table 8. Accuracy (in percentage) and size (in MB) of models with different types of basic blocks (see Fig. 8) for EMBER, OPCODE, and aggregated feature vectors

Models		EMBER	OPCODE	Agg2-Lat
ResNet1D ($K = 3$)	Acc.	**97.62 ± 0.08**	**95.91 ± 0.08**	**97.95 ± 0.04**
	Size	7.51	16.67	271.27
ResNet1D ($K = 1$)	Acc.	97.61 ± 0.06	95.79 ± 0.08	97.91 ± 0.03
	Size	7.11	**16.27**	**269.98**
ResNeXt1D	Acc.	97.57 ± 0.04	95.79 ± 0.11	97.88 ± 0.07
	Size	**5.44**	28.84	405.03
Inverted ResNeXt1D	Acc.	97.56 ± 0.06	92.68 ± 6.9	97.80±0.06
	Size	6.40	37.47	611.37
ConvNeXt1D	Acc.	97.08 ± 0.02	95.23 ± 0.05	97.40 ± 0.06
	Size	6.40	21.66	230.55

Table 9. Detailed network architecture for EMBER feature vector with 2 381 dimensions

Block	Kernel size	Stride	Padding	Out channels
conv1	7	4	1	24
conv2	7	5	1	48
conv3	7	4	0	96
conv4	7	4	1	192
conv5	7	1	0	384
FC1	384×128			
FC2	128×2			

The average performance over 5 runs of ResNet1D (with kernel size $K = 3$ and $K = 1$) and other variants of the student-alone models trained with EMBER, OPCODE features, and the teacher model trained with `Agg2-Lat` feature vector are presented in Table 8. We observe that ResNet1D with kernel size $K = 3$ obtains the highest accuracy and small size for all the models trained with EMBER, OPCODE, and `Agg2-Lat` feature vectors. This explains why we chose ResNet1D with kernel size $K = 3$ as our basic block for all the experiments presented above.

B Implementation of Neural Network Architecture for Individual Feature Vectors

Detailed implementation of neural network architectures for the EMBER feature vector, OPCODE feature vector and API-ARG feature vector are shown in Table 9, Table 10, and Table 11, respectively.

Table 10. Detailed network architecture for OPCODE feature vector with 33 338 dimensions

Block	Kernel size	Stride	Padding	Out channels
conv1	10	5	1	16
conv2	12	5	0	24
conv3	12	5	0	48
conv4	10	5	0	96
conv5	12	5	0	192
conv6	9	1	0	384
FC1	384×128			
FC2	128×2			

Table 11. Detailed network architecture for API-ARG feature vector with 1 048 576 dimensions

Block	Kernel size	Stride	Padding	Out channels
conv1	11	5	0	16
conv2	11	5	1	24
conv3	12	7	0	48
conv4	11	5	0	72
conv5	11	7	2	96
conv6	11	8	0	192
conv7	11	5	0	240
conv8	3	1	0	384
FC1	384×128			
FC2	128×2			

C Neural Network Architecture of Aggregated Original Features

In Table 12 and Table 13, we present the architectures of the 1D-CNN models for aggregated feature vectors from 2 original feature vectors (`Agg2-Org`—EMBER + API-ARG) and 3 original feature vectors (`Agg3-Org`—EMBER + OPCODE + API-ARG), which are developed and evaluated in our work.

D Speeding up Dynamic Analysis with Distributed Cuckoo Infrastructure

As we discussed earlier, it is a time-consuming task of dynamic analysis to produce analysis reports for malware samples. It depends on the user-defined parameter of the longest time that a sample is analyzed in the Cuckoo sandbox but the default setting is two minutes. Based on our daily experiments, using a single Cuckoo sandbox, we were able to analyze 8000 samples per week. We could use multiple sandboxes and manually submit malware samples to speed up the analysis. However, this is quite laborious as the virtual machines hosting Cuckoo sandboxes frequently crash, thus requiring close monitoring for fixing occurring issues.

In this work, we developed a parallel dynamic analysis infrastructure using the preliminary version of distributed Cuckoo [28]. We enriched the infrastructure by adding more automation for fault tolerance, which includes

– A re-submission mechanism to resubmit the samples that have not been successfully analyzed. The number of resubmissions is a user-predefined parameter (e.g., three times).

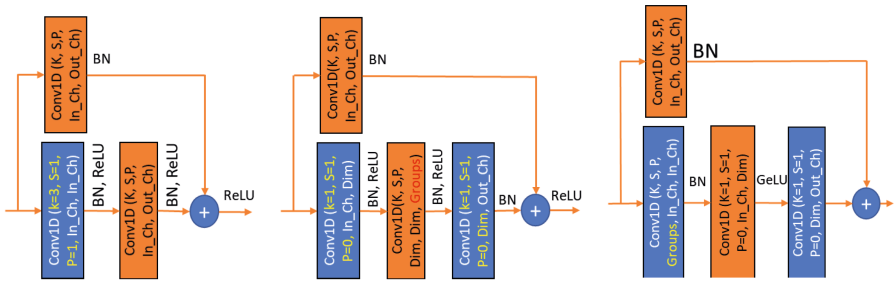
Table 12. Detailed network architectures for `Agg2-Org` feature vector (2 381 + 1 048 576 = 1 050 957 dimensions)

Block	Kernel size	Stride	Padding	Out channels
conv1	11	6	1	16
conv2	16	8	1	24
conv3	16	8	1	48
conv4	12	4	0	96
conv5	12	5	0	192
conv6	11	4	0	192
conv7	8	4	0	384
conv8	7	1	0	384
FC1	384×128			
FC2	128×2			

Table 13. Detailed network architectures for `Agg3-Org` feature vector (2 381 + 33 338 + 1 048 576 = 1 084 295 dimensions)

Block	Kernel size	Stride	Padding	Out channels
conv1	11	6	0	16
conv2	15	8	2	24
conv3	15	8	1	48
conv4	11	4	0	96
conv5	11	5	1	192
conv6	7	5	1	192
conv7	8	4	0	384
conv8	6	1	0	384
FC1	384×128			
FC2	128×2			

- A monitoring mechanism to check the status of virtual machines whether they are in normal working conditions or crashed. We implement an active monitoring technique that periodically sends monitoring requests to Oracle VirtualBox [22] to check virtual machine status (Cuckoo uses Oracle VirtualBox to host virtual machines and sandboxes).
- A virtual machine (VM) instantiation mechanism to instantiate new VMs to replace the crashed ones. This process is automatically invoked when the monitoring system detects a crashed VM. This allows us to maximize the utilization of computing resources (i.e., the available VMs hosted on physical servers).



(a) ResNet1D block with kernel size $K = 3$ (b) ResNeXt1D & Inverted ResNeXt1D blocks [33] (c) ConvNeXt block [18]

Fig. 8. Different block architectures for 1D-CNN models.

With the developed infrastructure and the available resources in our lab, we could run up to 12 Cuckoo sandboxes at the same time. The implemented fault-tolerance mechanisms also relieve us from the burden of manual monitoring and submission. With more than 86 000 samples used in our experiments (discussed further in Sect. 6), we could complete the analysis in just 10 days.

E Discussion on Model Updating

As new malware samples are introduced and evolve daily, model updating is needed to keep the model up to date, thus being able to handle such new samples. However, we believe that this is worth for separate work to develop new methods for model updating such as online learning and dealing with data drift problems. A naive solution is to retrain the teacher model and re-transfer to the student model. The old and new student models are tested on a new test set, and we only deploy the new student if it outperforms the old one by a certain threshold. We will keep this for our future work.

References

1. Abhishek, S., Zheng, B.: Hot knives through butter: Evading file-based sandboxes. Technical report, Fire Eye (2013)
2. Anderson, H.S., Roth, P.: EMBER: an open dataset for training static PE malware machine learning models. arXiv preprint arXiv:1804.04637 (2018)
3. Bucila, C., Caruana, R., Niculescu-Mizil, A.: Model compression. In: Proceedings of ACM SIGKDD, pp. 535–541. KDD 2006, ACM, New York, NY, USA (2006)
4. contributors, T.X.: Extreme gradient boosting open-source software library. https://xgboost.readthedocs.io/en/latest/parameter.html. Accessed 12 Mar 2022
5. Damodaran, A., Di Troia, F., Visaggio, C.A., Austin, T.H., Stamp, M.: A comparison of static, dynamic, and hybrid analysis for malware detection. J. Comput. Virol. Hack. Techn. **13**(1), 1–12 (2017)

6. Directorate, N.R.: Ghidra: a software reverse engineering (SRE) suite of tools in support of the cybersecurity mission. https://ghidra-sre.org/. Accessed 12 June 2022
7. Fan, Y., Ye, Y., Chen, L.: Malicious sequential pattern mining for automatic malware detection. Expert Syst. Appl. **52**, 16–25 (2016)
8. horsicq GitHub: Detect it easy, or abbreviated "die" is a program for determining types of files, 13 June 2022. https://github.com/horsicq/Detect-It-Easy#
9. Han, S., Mao, H., Dally, W.J.: Deep compression: compressing deep neural networks with pruning, trained quantization and huffman coding. arXiv preprint arXiv:1510.00149 (2015)
10. Han, W., Xue, J., Wang, Y., Huang, L., Kong, Z., Mao, L.: MalDAE: detecting and explaining malware based on correlation and fusion of static and dynamic characteristics. Comput. Secur. **83**, 208–233 (2019)
11. He, K., Zhang, X., Ren, S., Sun, J.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
12. Hex-Rays: Ida pro: A binary code analysis tool–a powerful disassembler and a versatile debugger. https://hex-rays.com/IDA-pro/. Accessed 12 June 2022
13. Hinton, G., Vinyals, O., Dean, J.: Distilling the knowledge in a neural network (2015)
14. Kadiyala, S.P., Kartheek, A., Truong-Huu, T.: Program Behavior Analysis and Clustering using Performance Counters. In: Proceedings of 2020 DYnamic and Novel Advances in Machine Learning and Intelligent Cyber Security (DYNAMICS) Workshop. Virtual Event, December 2020
15. Kang, B., Yerima, S.Y., Mclaughlin, K., Sezer, S.: N-opcode analysis for android malware classification and categorization. In: 2016 International Conference On Cyber Security And Protection Of Digital Services (Cyber Security) (2016)
16. Kim, T., Oh, J., Kim, N., Cho, S., Yun, S.: Comparing kullback-leibler divergence and mean squared error loss in knowledge distillation. CoRR abs/2105.08919 (2021)
17. Kundu, P.P., Anatharaman, L., Truong-Huu, T.: An empirical evaluation of automated machine learning techniques for malware detection. In: Proceedings of the 2021 ACM Workshop on Security and Privacy Analytics, pp. 75–81. Virtual Event, USA (2021)
18. Liu, Z., Mao, H., Wu, C.Y., Feichtenhofer, C., Darrell, T., Xie, S.: A convnet for the 2020s. arXiv preprint arXiv:2201.03545 (2022)
19. Moser, A., Kruegel, C., Kirda, E.: Limits of static analysis for malware detection. In: ACSAC 2007, pp. 421–430 (2007)
20. Ndibanje, B., Kim, K.H., Kang, Y.J., Kim, H.H., Kim, T.Y., Lee, H.J.: Cross-method-based analysis and classification of malicious behavior by API calls extraction. Appl. Sci. **9**(2), 239 (2019)
21. Or-Meir, O., Nissim, N., Elovici, Y., Rokach, L.: Dynamic malware analysis in the modern era-a state of the art survey. ACM Comput. Surv. **52**(5) (2019)
22. Oracle: Oracle virtualbox. https://www.virtualbox.org/. Accessed 12 June 2022
23. Oramas, S., Nieto, O., Sordo, M., Serra, X.: A deep multimodal approach for cold-start music recommendation. In: Proceedings of 2nd Workshop on Deep Learning for Recommender Systems, pp. 32–37. Como, Italy, August 2017
24. Pytorch: Resnet implementation in pytorch. https://github.com/pytorch/vision/blob/main/torchvision/models/resnet.py. Accessed 12 June 2022
25. Quarkslab: Lief: Library to instrument executable formats. https://lief-project.github.io/. Accessed 18 Feb 2022

26. Rabadi, D., Teo, S.G.: Advanced windows methods on malware detection and classification. In: Annual Computer Security Applications Conference, pp. 54–68. ACSAC 2020, Association for Computing Machinery, New York, NY, USA (2020)
27. Rhode, M., Burnap, P., Jones, K.: Early-stage malware prediction using recurrent neural networks. Comput. Secur. **77**, 578–594 (2018)
28. Sandbox, C.: Distributed cuckoo. https://cuckoo.readthedocs.io/en/latest/usage/dist/. Accessed 12 June 2022
29. Santos, I., Brezo, F., Ugarte-Pedrero, X., Bringas, P.G.: Opcode sequences as representation of executables for data-mining-based unknown malware detection. Inf. Sci. **231**, 64–82 (2013)
30. Scikit-Learn: Hashing vectorizer function. https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.HashingVectorizer.html. Accessed 18 Feb 2022
31. Tan, W.L., Truong-Huu, T.: Enhancing robustness of malware detection using synthetically-adversarial samples. In: GLOBECOM 2020–2020 IEEE Global Communications Conference, Taipei, Taiwan, December 2020
32. URsoftware-W32DASM: W32dasm: a disassembler tool made to translate machine language back into assembly language. https://www.softpedia.com/get/Programming/Debuggers-Decompilers-Dissasemblers/WDASM.shtml. Accessed 12 June 2022
33. Xie, S., Girshick, R., Dollár, P., Tu, Z., He, K.: Aggregated residual transformations for deep neural networks. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 1492–1500 (2017)
34. Zhang, J., et al.: Scarecrow: deactivating evasive malware via its own evasive logic. In: 2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pp. 76–87 (2020)
35. Zhang, J., Qin, Z., Yin, H., Ou, L., Zhang, K.: A feature-hybrid malware variants detection using CNN based opcode embedding and BPNN based API embedding. Comput. Secur. **84**, 376–392 (2019)
36. Zhi, Y., Xi, N., Liu, Y., Hui, H.: A lightweight android malware detection framework based on knowledge distillation. In: Yang, M., Chen, C., Liu, Y. (eds.) NSS 2021. LNCS, vol. 13041, pp. 116–130. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-92708-0_7