Journal of Computer Virology and Hacking Techniques (2021) 17:279–297
https://doi.org/10.1007/s11416-021-00381-3

ORIGINAL PAPER

Data augmentation and transfer learning to classify malware images in a deep learning context

Niccolò Marastoni[1] · Roberto Giacobazzi[1] · Mila Dalla Preda[1]

Received: 11 December 2020 / Accepted: 19 March 2021 / Published online: 8 April 2021

Abstract
In the past few years, malware classification techniques have shifted from shallow traditional machine learning models to deeper neural network architectures. The main benefit of some of these is the ability to work with raw data, guaranteed by their automatic feature extraction capabilities. This results in less technical expertise needed while building the models, thus less initial pre-processing resources. Nevertheless, such advantage comes with its drawbacks, since deep learning models require huge quantities of data in order to generate a model that generalizes well. The amount of data required to train a deep network without overfitting is often unobtainable for malware analysts. We take inspiration from image-based data augmentation techniques and apply a sequence of semantics-preserving syntactic code transformations (obfuscations) to a small dataset of programs to generate a larger dataset. We then design two learning models, a convolutional neural network and a bi-directional long short-term memory, and we train them on images extracted from compiled binaries of the newly generated dataset. Through transfer learning we then take the features learned from the obfuscated binaries and train the models against two state of the art malware datasets, each containing around 10 000 samples. Our models easily achieve up to 98.5% accuracy on the test set, which is on par or better than the present state of the art approaches, thus validating the approach.

Keywords Deep learning · Binaries · Malware

1 Introduction

Malware (**mal**icious soft**ware**) comprises all programs that are written with the intent to perform some malicious activity. The proliferation of these types of programs has increased considerably in the last few years [28], as most households own one or more devices that can be attacked. It is thus imperative to find fast and reliable techniques to identify and fight new malware.

Modern-day malware samples are often heavily protected against reverse engineering and many types of program analysis. For example, malware is frequently modified through obfuscations [42]. These are syntactic code transformations that take a program in input and generate a different program that is more difficult to analyze while still maintaining its functionality [11]. The combination of obfuscations with the code optimization algorithms usually embedded in compilers makes the reverse engineering of malware harder, often slowing down or obstructing parts of the disassembly process [1]. For this reason it is imperative to find techniques that deal with the raw binary instead of relying on higher-level features stemming from reverse engineering attempts.

Obfuscations are widely used in malware [31,47] and they make it harder to classify emerging malware into their specific families. This task is called malware classification, and it is usually achieved with machine learning techniques. These can range from shallow models that require manual feature engineering before the training process, to deep learning models that can work directly on the raw data. The downside of shallow models is that they require specific domain expertise, which means that time and resources are needed to analyze the samples in the dataset before proceeding to the learning phase. On the upside, the input of human-engineered features usually renders the model and the results easier to

✉ Niccolò Marastoni
niccolo.marastoni@univr.it

Roberto Giacobazzi
roberto.giacobazzi@univr.it

Mila Dalla Preda
mila.dallapreda@univr.it

[1] University of Verona, Verona, Italy

interpret. With new malware spreading at an alarming pace, the cost of this manual work is simply unfeasible. Deep learning techniques can automatically extract the features from the dataset samples without the need for time consuming feature engineering or specific domain expertise. This advantage makes deep learning the go-to paradigm for malware classification.

One of the drawbacks of deep learning techniques, compared to shallower models, is their tendency to overfit when trained with small datasets [43]. This can be a problem in fields like program analysis, and especially in malware classification, as gathering enough samples with the proper ground-truth takes many resources and even more time.

This problem is also common in other fields, such as in image recognition and image classification [35]. The lack of enough training data is easily solvable in the vision context, as new data points can be generated from existing ones by applying some semantics-preserving transformations to the images such as rotations, translations in space or selective cropping. This process of generating new data from existing images is known as data augmentation and it is a staple of deep learning. This is a key factor in the approach that was first introduced in [26].

Another way to mitigate the problems that models can find with few data-points is to reuse a part of an already trained model, usually the part dedicated to feature extraction. These models can be trained with millions of data points and then they can be repurposed for a different problem setting by removing the head of the model (dense layer) and re-training a new head while "freezing" the rest of the network. This process is of course less time expensive, but it also does not incur the problem of needing more data to train, as the majority of the weights are "frozen" and thus do not appear as free variables. This technique is called transfer learning [34].

Since obfuscations generate syntactic variants of programs but maintain their semantics, they can be seen as data-augmentation transformations specific to code. This is the main intuition of [26], where 47 small programs with different semantics have been transformed iteratively by applying obfuscations and generating 200 variants each, resulting in a final dataset that contains 9 400 samples divided in 47 classes.

Generating the dataset this way allows for fine control on the size of the dataset itself and its class balance (that can often be a problem in real-world datasets [18]). Another upside of this technique is that it allows us to clearly see if there are some obfuscations that render the programs harder to classify than others, thus being more powerful or resilient to the particular learning architecture.

In this paper we generate a dataset of 18 800 obfuscated programs with the aforementioned technique, reaching a size that is roughly double than the one generated in [26] with the same technique. We then train two deep neural network models, a convolutional neural network (CNN) and a bi-directional long short-term memory (LSTM) and achieve an average accuracy around 93% on the generated dataset. We provide an analysis of the classification errors that highlight the strength of certain obfuscations against the classification effort and the weaknesses of the trained models.

To prove that the techniques and models used in this work are suited for real-life scenarios involving malware we train the CNN and the LSTM on two malware datasets heavily used in literature, the dataset from the The Microsoft Malware Classification Challenge hosted on Kaggle [20] (referred to as the MsM2015 dataset from now on) and the MalImg dataset [30]. These datasets will be thoroughly described in Sect. 4.

We then experiment with transfer learning, taking the features from the models trained on the custom dataset and using them to classify the two aforementioned malware datasets. We verify that it is indeed possible to use the features learnt from the classification of a custom-made dataset of binaries in order to classify a real-world dataset. This has the obvious potential of allowing big networks to be trained on huge datasets and then be reused for smaller and newer datasets at a very low cost. In Sect. 6 we show the results of our experiments with transfer learning by training a model on our custom generated dataset and then verifying that the learnt features can be used to classify the other two datasets. The positive results achieved make us believe that this technique has a lot of potential and could help mitigate the problems encountered when applying deep learning techniques to small malware datasets.

The main contributions of this work are:

- the design of neural network models that can classify obfuscated binaries from their images
- thorough comparison of the approaches
- validation of the models on two state of the art malware datasets
- successful transfer learning experiments between models trained with different datasets

2 Background

This section serves as a primer on a few of the key concepts that we use in our study.

2.1 Obfuscations

Obfuscations are program transformations that change the syntax of the program without altering its semantics. They are meant to confuse analyzers and reverse engineers, although

the amount of *confusion* added cannot yet be reliably measured [7,8].

Let $Prog$ be the set of all programs. An obfuscation is a program transformation $O : Prog \rightarrow Prog$ that given a program $P \in Prog$ produces a new program $O(P)$ with the same functionality as P but that *is unintelligible in some sense* [2].

In this work we use obfuscation as a data augmentation tool to generate the first dataset from which we design both our learning models. The programs in the datasets have been created by running the Tigress C obfuscator [10] on simple C programs.

2.2 Convolutional neural networks

CNNs are feed-forward neural network models that take inspiration from the human visual cortex and are widely used in image recognition and classification [23,38]. Their success in image-related learning tasks has been attributed to their translational invariance [3] which allows them to be used to solve problems such as face recognition [23] or handwritten character recognition [24]. In general, CNNs are very good at extracting spatial features from the data.

At their core, CNNs consist of at least one convolutional layer connected to a dense layer. The convolutional layer operates a convolution on the input in order to isolate the features that the network deems important during the training phase. This process greatly reduces the number of weights needed since the input images are condensed into a smaller feature set. The convolutional network usually is connected to a max pooling layer that contributes further to the reduced size of the features detected by combining the values of multiple neurons into a single value, usually the maximum value (thus max-pooling) or the average value. After a combination of the aforementioned layers the CNN architecture is completed by at least one fully connected (or dense) layer. This layer is responsible for the classification process itself and acts as a multi-layer perceptron that takes as input the features extracted from the previous layers. To prevent overfitting, regularization techniques are commonly used and can be applied anywhere in the network. This reduces the possibility of the network purely memorizing the training data, thus allowing for better generalization [22].

2.3 Long short-term memory

As stated in the previous subsection, CNNs excel at extracting spatial features from data. This is invaluable when classifying or generally working with natural images, but images extracted from code reveal different types of patterns altogether and therefore comprise a different learning problem [26].

The sequential nature of code, and thus of compiled programs, indicates that a learning model that works well with sequential data can be beneficial. Recurrent neural networks (RNNs), which are designed to process sequences of data of arbitrary length from beginning to end, is one example. Generally, the hidden state h_t of a RNN depends on the output of the previous state h_{t-1} and so on, which means that the state h_t contains a distributed representation of all the tokens observed in the sequence up to the time step t. This way, the network can generate probabilistic dependencies from previously seen data. One common pitfall of this structure is that the dependencies between tokens that are far apart from each other in the sequence are hard to manage. This stems from the nature of the gradient descent algorithm over time steps, which makes the components either decay or grow exponentially [4,17].

Long short-term memory was developed to mitigate precisely this problem [17]. By storing information at particular timesteps, LSTMs provide a mechanism to specify when to remember certain information, and more importantly, when to forget it.

The LSTM used in this work is bidirectional, meaning that the input stream is read in both directions at once. This is achieved with two LSTM models, one forward and one backward, which read the input in the two directions and then combine the resulting vectors to produce a unified result.

In Sect. 5 we outline the specific architecture of our neural networks.

2.4 Transfer learning

As mentioned in previous sections, a big problem of deep learning tasks is finding datasets that are big enough to allow learning without overfitting. Training deep networks is also very time consuming and requires adequate computing resources. Transfer learning can alleviate both these problems [36], by leveraging the information learnt from one task in order to solve a different task. Some common applications of transfer learning are of course in vision, where huge models trained from the ImageNet dataset [13] can be re-purposed for new image classification tasks.

To illustrate how transfer learning works we can imagine a typical CNN as described earlier, trained on a dataset of images for classification. The convolutional layers are tasked with extracting the features from the inputs while the dense layers towards the output provide the actual classification of the dataset. One way to properly perform transfer learning is to remove the dense layer from the CNN and "freeze" the convolutional layers (meaning, the gradient descent will not modify their weights). At this point a new dense layer can be applied with the proper output shape (the number of classes for the new learning problem) and the network can be trained on the new dataset. Since the convolutional

layers do not have to be trained again, the learning process is sped up considerably. The size of the dataset is also less important, since the number of free variables (weights) that can be modified is much smaller than those in the original network. The convolutional layers of the CNN will provide the necessary features for the classification task at virtually no cost as the feed-forward part of the network is relatively inexpensive.

In this work we use transfer learning between different datasets of images generated from compiled programs. This allows us to gauge if the features extracted from one dataset can be used to classify the others. The applications of this can lead to many interesting opportunities that will be discussed in future sections.

2.5 Bicubic interpolation

In order to resize the images in our custom dataset we use bicubic interpolation as implemented in OpenCV [6]. This algorithm infers the intensity of an unknown pixel by applying bicubic interpolation [21] over the neighboring 16 pixels. It is often used in place of its bilinear counterpart when the quality of the resulting image is more important than the computational resources used, which fits our scenario.

3 Related work

In this section we explore the most relevant works that approach the program classification problem using only compiled binaries. We do not include works that use features extracted from the source code of the programs or from the assembly, as our methodology is based on the premise of not possessing either.

It has to be noted that the focus of our study is not only on malware classification but on any obfuscated binary. This is why the two malware datasets have been used mainly for validation. Furthermore, none of the surveyed studies train an LSTM on the images extracted from binaries. This model provides new insights on the classification of images extracted from binaries.

Since the classification of binaries according to their images is done mainly on only two datasets, the MalImg and MsM2015 ones, we will group the related works accordingly.

MalImg papers To our knowledge, the first work that utilizes images extracted from binary files in order to classify them is by Nataraj et al. [30]. The general idea of this paper comes from the consideration that malware samples often appear similar in layout and texture when translated into images. For this reason this study approaches the feature selection with GIST [32], using wavelet decompositions of the images. The classifier used is a simple k-nearest neighbors with Euclidean distance as a measure of similarity. Other than the different models used and the different features used, our approach also does not assume that the samples from the same classes in our database look anything alike. In fact the opposite can be said, since the obfuscations applied generate visually different variants for each class. Nonetheless we use the dataset from this work to show that our approach can generalize to datasets that, unlike our custom one, have not been created ad hoc.

The study by Cui et al. [12] also considers malware classification as a means to its main goal. The paper argues (rightfully so) that the MalImg dataset is heavily imbalanced, which can potentially lead to less than accurate results. To quell the problem, they perform under-sampling of the dataset (removing samples from specific classes) and demonstrate that the process makes overfitting less of an issue.

Yakura et al. [46] designed a CNN with attention mechanism in order to highlight which parts of the malware image were being considered for the purpose of classifying them into families. Their approach allows the learning process to output specific regions of the image that are being used for classification, thus providing useful information for further manual analysis.

In [40], the authors take the deep residual network architecture with 50 layers from [16] and use it to classify malware from the MalImg dataset. The network is first trained on a typical object detection task, then the last layer is dropped from the model and a new dense model with 25 output nodes is added to classify the malware samples into their respective families. Other recent works such as [9] and [5] have proposed slight variations of the aforementioned approach, always pre-training their model on common image classification tasks.

The intuition of the previous papers is that the malware in the MalImg dataset present specific visual features that make them distinctive to the human eye [40], thus a neural network trained to classify real-life objects could carry enough features to also classify the malware samples.

Our approach starts with the assumption that we can already extract meaningful features from malware samples, so we need to verify that these features map easily to new datasets. For this reason we use a network that is pre-trained on a malware dataset in order classify a different malware dataset.

A CNN has been used for malware classification in [18], along with an extreme learning machine (ELM). The main goal of this work is to compare the efficiency of the two models when put to the task of classifying malware images. This study uses the images from the dataset of Nataraj et al. [30] to train their models and the results indicate that ELMs are more suited for the task at hand, being faster and more accurate than CNNs.

MsM2015 The MsM2015 dataset has been extensively used in literature for malware classification tasks. For example,

the work by Kang et al. [19] uses word-to-vec approach with an LSTM network to classify the samples in each family. As many other studies on this subject, they do not consider the binary as is but rather generate an assembly file for each sample and collect opcodes and API functions that will then constitute the bulk of the features utilized.

A very interesting work that uses both the MalImg dataset and images extracted from the binaries in the MsM2015 dataset is [45]. The goal of the paper is to evaluate cost-sensitive approaches to malware image classification and for this purpose the authors combine a CNN with various RNN models in order to gauge the effectiveness of the approaches.

4 Datasets

This section describes the datasets used in this work. We start from the methodology for generating of our custom OBF dataset and then summarize the peculiarities of MalImg and MsM2015.

4.1 OBF dataset

We start with 32 programs that are downloaded from a beginners programming website [37]. These are very simple programs that average 23 lines and they were selected so that the full images extracted from the programs would fit the models.

Small programs work well enough to illustrate the methodology but bigger programs can lend some validity. For this reason 15 more programs have been added to the original dataset, all taken from solutions of the Google Code Jam. These programs have been selected randomly and represent a more real-world scenario with source code that spans between 38 lines and 150.

Applying obfuscations We selected 8 obfuscations from Tigress and we applied them iteratively on the 47 files of the original dataset.

- *Flatten* changes the control flow of functions by inserting a switch statement that decides where each basic block flows next
- *Split* splits a function into two separated functions
- *RandomFuns* inserts a random function into the code
- *EncodeArithmetics* encodes any arithmetic operation into a semantically equivalent but syntactically harder to decipher operation
- *EncodeLiterals* initializes literal integers and strings with new functions
- *InitOpaque* adds specific data structures that can be used to insert opaque predicates
- *InitEntropy* adds new variables in order to collect entropy
- *InitImplicitFlow* initializes handlers for implicit flow analysis

These obfuscations have been selected because they represent different types of syntactic transformations. For example, the first 2 deal with structural transformations of the control flow graph while *EncodeArithmetics* and *EncodeLiterals* perform a more symbolic kind of transformation. *RandomFuns* has been added specifically because it adds new functions to the code, preserving the basic semantics of the original program but at the same time augmenting it. This is akin to what happens in some malware, where the payload is consistent between different samples of a family but the surrounding program can have different semantics [14].

All 8 obfuscations are applied to every sample in the initial dataset, generating 376 ($47 * 8$) new syntactic variants of the base programs.

Then the process is repeated, but this time the new programs are each obfuscated with a new transformation, thus generating 2632 ($376 * 7$) variants which are added to the initial dataset and the simple variants. The reason why only 7 obfuscations can be considered during the first repetition of the algorithm is that we do not allow for repeated obfuscations.

After all, most transformations do not result in a new interesting variant when applied twice. For example, *Flatten* applied to a function that has already been flattened would result in exactly the same program, as it is idempotent.

Repeating this algorithm until all possible combinations of obfuscations are depleted generates 17,055,360 samples. We limit the size of our dataset to 18,800 samples in order to have 400 variants for every initial program.

One of the advantages of generating such a dataset from scratch is that the resulting classes will be as balanced as needed. Some datasets in literature are heavily imbalanced, for example, both the MalImg dataset (used in this paper) and the GENOME [48] dataset for Android malware suffer from this. It is expected to find such imbalance in datasets found in the wild, but this leads to problems in the classification process, or more properly, in the accuracy measurement. For instance, let us imagine a dataset of malware that contains 25 classes distributed in a balanced way. Given a random sampling of the test set, a simple classifier could always guess one of the classes and get it right $1/25$ of the time, thus obtaining around 4% accuracy. This is evidently a bad result and would alert the researchers to the inherent incapability of the learning model. On the other hand, the MalImg dataset contains 9 458 samples divided into 25 classes and the largest class has 2 949 elements, with the second largest one closing in at 1 591. A simple classifier could learn to distinguish between these 2 big classes, then guess all samples to be contained in them and still reach almost 50% accuracy. Of course this is not a good result by itself, but it can trick anyone

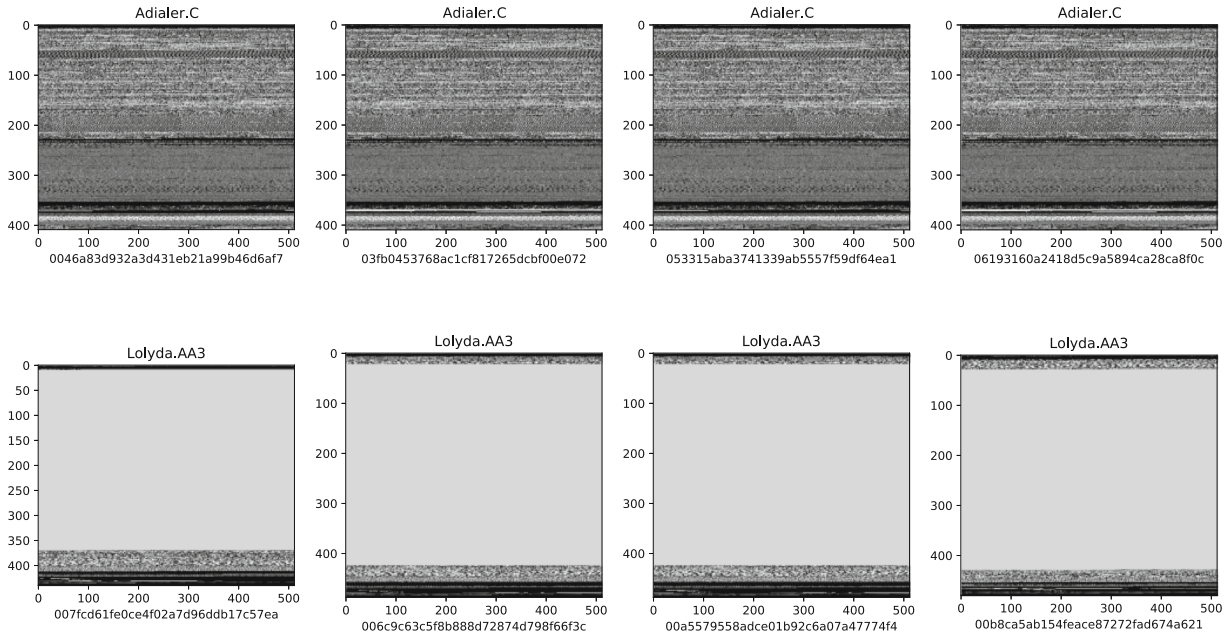

Fig. 1 8 samples from the classes '`adialer`' and '`Lolyda`' of MalImg

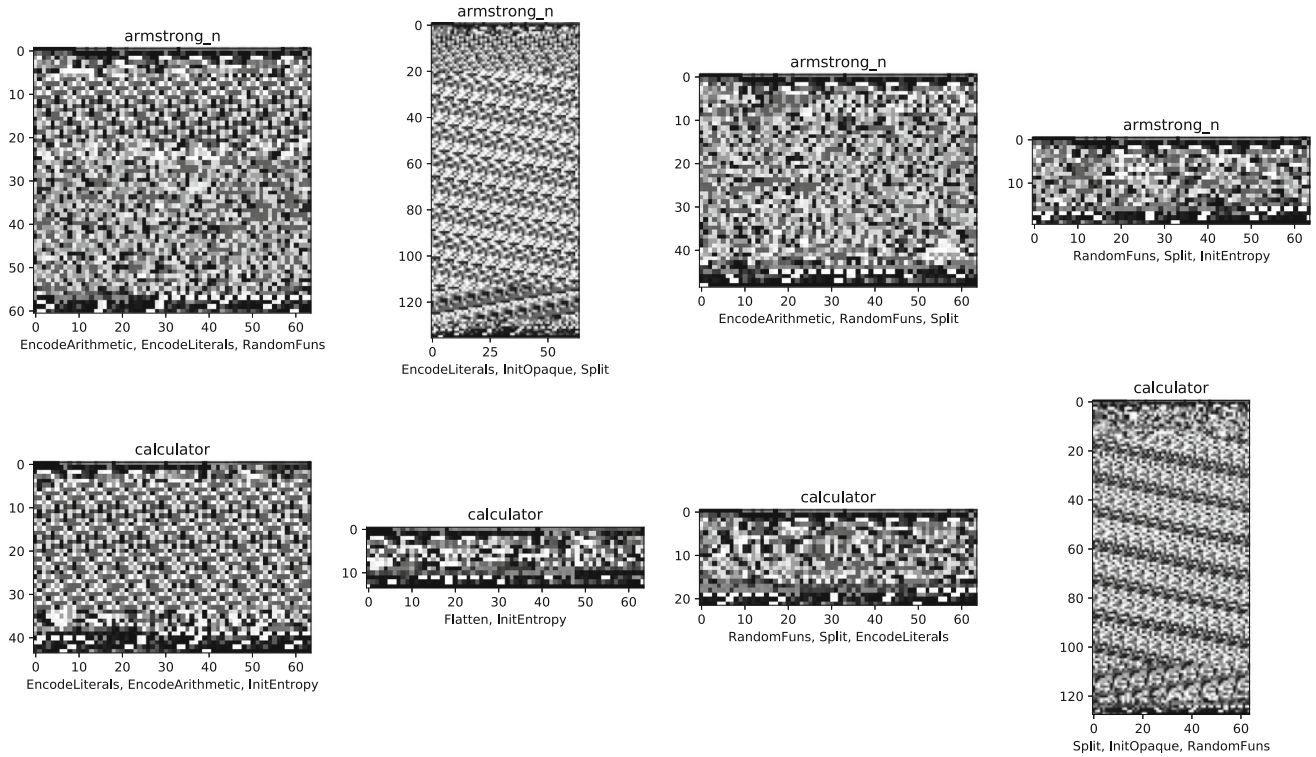

Fig. 2 8 samples from the classes armstrong_n and calculator of OBF

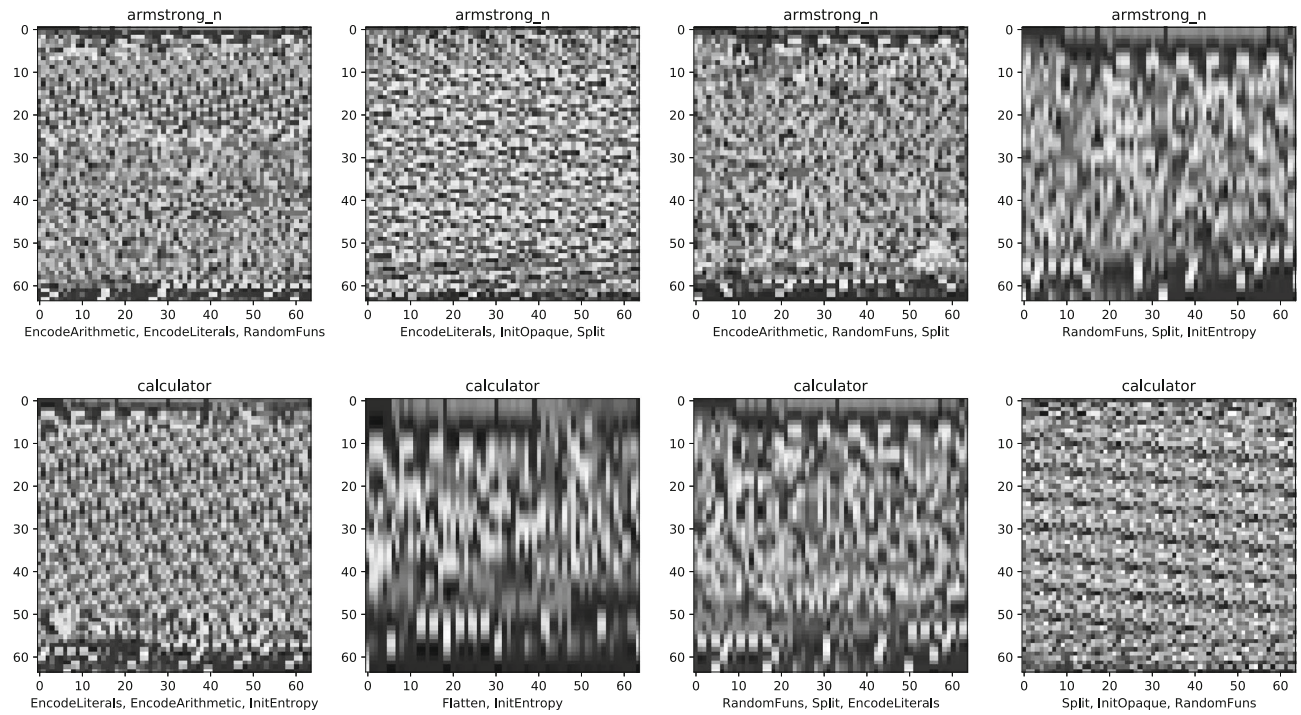

Fig. 3 8 samples from the classes '`armstrong_n`' and '`calculator`' of OBF, interpolated

into thinking that the model is actually learning something from the dataset when that is certainly not the case.

Another advantage of our dataset generation technique is that we can ensure that the obfuscations applied will cause pervasive structural changes in the binaries, generating visually distinctive images that belong in the same class. This is not always the case, especially for datasets that have been collected in the wild, since many code transformations do not act on the global structure of the executable file. This can be easily seen in Fig. 1, which shows samples from the same classes looking very similar. We will expand on this later.

The dataset generated from the simple programs iteratively obfuscated will be called OBF from now on. In Fig. 2 we show 8 programs taken randomly from two classes, '`armstrong_n`' and '`calculator`'. At the bottom of each figure there is a list of all the obfuscations applied to the specific sample. It should be evident that the syntactic transformations of the source code also result in visually distinctive binaries, and thus the images extracted from them also have distinctive features.

The images generated vary in size, from 24×64 for the smallest sample to 9800×64 for the largest one. Since image classification models in general require a uniform size for all the images, we use bicubic interpolation to bring them to one size. In Fig. 3 we show the same samples as in Fig. 2 but with interpolation applied to them. Even at first glance, the difference in the images caused by the obfuscations is still noticeable, if not more so.

4.2 Microsoft malware dataset [MsM2015]

The MsM2015 dataset consists of more than 20,000 malware samples for Windows, collected by the Microsoft corporation and distributed by Kaggle for a competition in 2015 [41]. Each sample in the dataset belongs to one of 9 known malware families. The dataset provides each datapoint as the hexadecimal representation of its executable and as a collection of metadata generated by IDA pro. The total size of the dataset is around 500 GB of data, making it less than nontrivial to work with.

For this study we utilized only the byte representation of the samples, loading it in python using the same technique applied in [26] and representing every sample as an image with a set width of 64. Each sample is then resized to either 64×64 or 256×64 using bicubic interpolation, generating a more manageable dataset. Since images generated from the MsM2015 dataset are sourced from real-life malware, they come in varying sizes that far surpass those of the OBF dataset. As a result, the interpolation process removes even more detail than it did in the first dataset.

4.3 MalImg

The MalImg dataset is a collection of 9458 malware samples divided into 25 families. The main characteristic of this dataset is that the malware samples are not provided directly, but rather as their images as they appear on disk. In a similar

Table 1 Accuracies of the bidirectional LSTM when trained with different timesteps and image shapes

Img size	Timesteps	Features	Accuracy (%)	Time (s)
32 × 128	32	128	92.4	279
128 × 32	128	32	92.6	743
64 × 64	64	64	**93.4**	392
64 × 128	64	128	92.8	402
128 × 64	128	64	93.2	712
128 × 128	128	128	93.1	641
32 × 256	32	256	92.9	364
256 × 32	256	32	92.4	1516
64 × 256	64	256	92.2	461
256 × 64	256	64	92.1	1251
128 × 256	128	256	91.9	673
256 × 128	256	128	92.9	1341

Highest accuracy value is indicated in bold

way to the work in [26], the bytes of the executable files are trivially mapped to floats, which will then be interpreted as pixel values of grayscale images.

As anticipated, the classes in the dataset are heavily imbalanced: the biggest one (‘`Allaple.A`’) contains 2949 samples, while the smallest one contains only 80 samples.

In Fig. 1 we show 8 random samples taken from two classes of the dataset. It should be readily evident that the images from each class have distinctive patterns that allow us to tell samples from one family apart from samples in the other family. This is again true for samples processed by bicubic interpolation, as shown in Fig. 4. The observation that different families of malware had a distinctive ‘`look`’ is part of what has driven the original work in [30]. This is not always the case with binaries. Being able to tell at a glance that two executable files belong to the same class is a luxury that we do not have in the OBF dataset. Fig. 2 shows this clearly, and in fact, the second sample in the first family appears most similar to the fourth sample in the second family.

4.4 Fixed size

Throughout the rest of this paper the size of the images considered for classification has been set to 64×64 and 256×64. This is done to optimize the training time and the classification accuracy for both models. The training accuracy for various image sizes (resulting in multiple time step values and features) can be seen in Table 1. As made evident by the experimental results, the two selected values are among the optimal ones according to classification accuracy. These experiments were done on the OBF dataset but empirical observations led us to keep the same fixed image size for the classification of all the datasets.

5 Experiment setup

In this section we describe the details of our experiments, from reproducibility and generalizability concerns to model architecture and considerations on how to best present our results.

5.1 Training, validation and testing sets

In order to guarantee a degree of generalization we split each dataset in training, validation and testing sets. During the learning stages we experimented with different ratios for the splits with the goal of maximizing the fairness of the generalization [39] but also to preserve as many samples as possible for the training stage. The ratio of the hold-out test set for all of the experiments in this paper is 0.2, meaning that 20% of the dataset is reserved for the testing phase that happens after the training and it is not used to tweak any of the parameters. Of the remaining samples in the dataset, 10% is kept for the validation set, which is used at every training epoch to calculate the validation loss, while the rest is the training set with which the weights of the network will be trained.

All the results from the subsequent sections have been achieved using the hold-out test set, averaged after running the experiments up to 20 times. This should ensure the generalizability of the approach, as any unfortunate division of the dataset should be prevented by the average over multiple random splits.

5.2 Classification scores

The main results in this paper are reported as measure of test set accuracy, that being the percentage of samples classified into the right class. Due to the unbalanced nature of the MalImg dataset we also report the precision, recall and f1 measure for each class.

Given the true positives as T_P and the false positives as F_P we can define precision as:

$$Precision = \frac{T_P}{T_P + F_P} \tag{1}$$

Intuitively, if a classifier has high precision for a specific class A it means that it guesses correctly when a sample belongs to A and will not classify foreign samples to this class. It is the percentage of correct guesses when the classifier guesses A. However this does not take into account the samples belonging to A that have been wrongly assigned to other classes. Recall, on the other hand, considers these mistakes which are called false negatives (F_N):

$$Recall = \frac{T_P}{T_P + F_N} \tag{2}$$

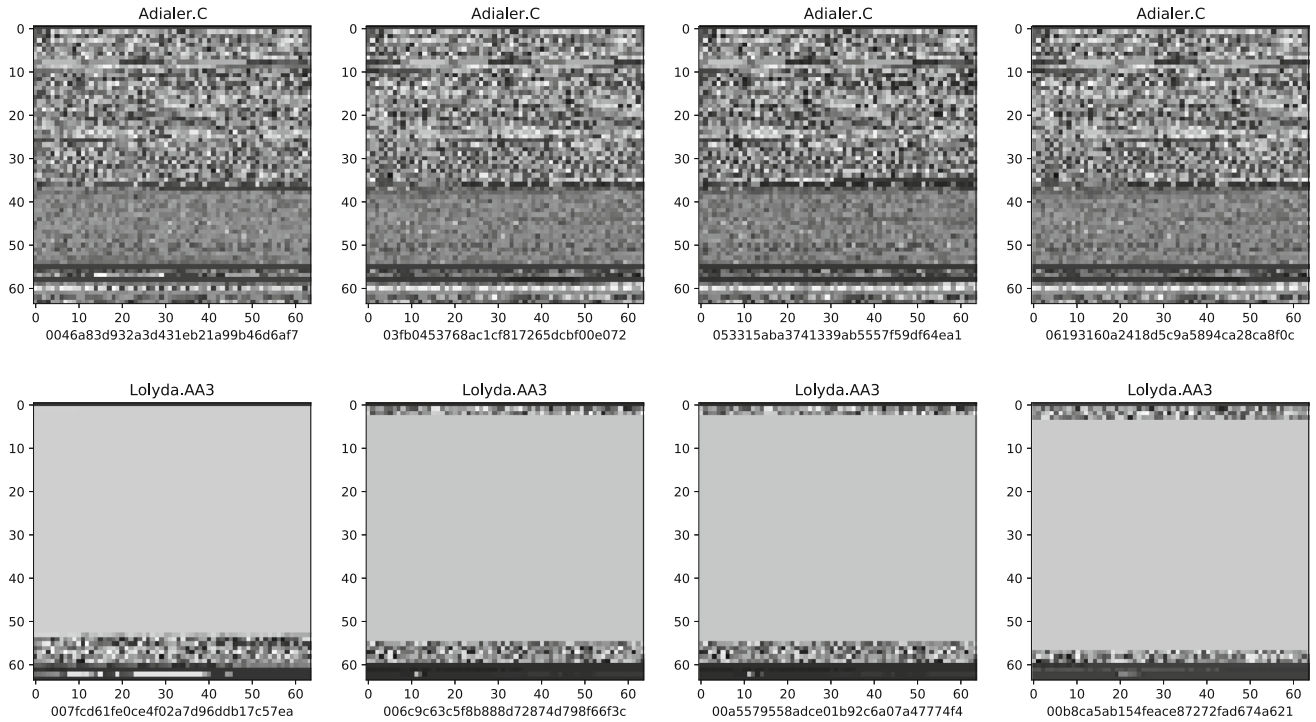

Fig. 4 8 samples from the classes '`adialer`' and '`Lolyda`' of MalImg, interpolated

Table 2 Average classification accuracy

	CNN		LSTM	
	64×64 (%)	256×64 (%)	64×64 (%)	256×64 (%)
OBF	90.9	92.3	**93.4**	92.6
MsM2015	90.8	92	**93.8**	90
MalImg	98.1	98.3	**98.5**	98.2

Highest accuracy value is indicated in bold

The recall of a classification task for the class A is the percentage of correct guesses made by the model when it should have guessed A.

To better visualize the difference between accuracy and these measures we can add the true negatives (T_N) and then define accuracy as:

$$Accuracy = \frac{T_P + T_N}{T_P + T_N + F_P + F_N} \tag{3}$$

Precision and recall are often combined in the $F1$ measure via their harmonic mean:

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{4}$$

5.3 Models

The two models considered have been fully coded in Python with Keras [33] and deployed on Colab notebooks freely available online to simplify the reproducibility of the study. The untrained models for these experiments can be found at [27] but we cannot provide a download link for the MsM2015 and the MalImg datasets as they are made available by each respective owner on their own terms. The OBF dataset is already available in the notebooks, along with methods that can adapt the other two datasets to the models.

In the following we provide a technical description of the models implemented.

CNN We implement a convolutional neural network in Keras, an almost direct translation of the one employed in [26] (which was built on barebones TensorFlow). The network has a base of two convolutional layers, each followed by a max pooling layer. Each convolutional layer has a kernel size of 5 and employs a rectified linear unit (relu) as the activation function, with 32 filters in the first layer and 64 in the second. Both max pooling layers have a size of 2×2 with padding set to '`same`' in order to avoid shrinking the input image. The head of the network is a dense layer followed by a dropout layer that flows in the last dense layer. The dropout layer is a form of regularization which, as anticipated in Sect. 2, prevents the network from memorizing the training set and

thus hopefully allows it to better generalize. With a value set to 0.2, the drop out layer randomly selects 20% of the neurons of the previous layer and sets their activation to zero. The network is then trained via gradient descent through the Adam optimizer with default learning rate and epsilon, while categorical cross-entropy is used as loss measure.

This network has been thoroughly tested on the MNIST dataset of hand-written digits [25], where it reaches accuracy values up to 99%. The purpose of testing it on a dataset with simple images is to show that, while the approach is meant to work on any image recognition task, it can generalize to arguably more difficult tasks. Further, according to the experiments in [26], it is the smallest network that can learn and generalize on the OBF dataset. It is important to contain the size of the network because bigger models tend to overfit and generally underperform unless they are trained with huge datasets [15].

We tested with many image sizes, taking the approach shown in [26] but using interpolation to resize the images instead of cropping and zero-padding. Among the various sizes we only show the results with square images of size 64 pixels by 64 pixels and 256 by 64 (64×64 and 256×64 in the tables) as they are the most significant.

LSTM The long short-term memory has also been implemented in Keras. It consists of two recurrent units, specifically bi-directional LSTM units with 141 and 94 units respectively. In order to quell overfitting the network has been equipped with a patience of 30 epochs, meaning that the model will stop learning 30 epochs after the validation loss has stopped decreasing.

The LSTM model clearly performs better than the CNN in all 3 datasets considered, given the initial input size of 64×64. The advantage of the LSTM model is not only in the improved accuracy but also in its considerably smaller size, although it results in a higher training time. An interesting aspect of the LSTM models is that they do not perform better with bigger images, in fact, the accuracy drops quite a bit. A way to slightly mitigate this effect is to encode the images as 64×256 (short and fat instead of tall and thin). This likely has something to do with the fact that the increase in the height of the image corresponds to an equal increase in the timesteps of the recurrent network. Keeping the number of timesteps to 64 maintains the performance of the network but the improvement on the classification results is not particularly noticeable.

We show the average accuracy for all models and datasets in Table 2.

6 Results and analysis

This section is devoted to presenting our results on both architectures presented in the previous section against the datasets

Table 3 Classification scores for the CNN on OBF

Classes	Precision	Recall	f1-score	Support
alien_lang	0.91	0.99	0.95	80.0
armstrong_n	0.89	0.84	0.86	80.0
bot_trust	0.99	0.95	0.97	80.0
calculator	0.93	0.80	0.86	80.0
candy_split	0.98	0.80	0.88	80.0
char_freq	0.96	0.92	0.94	80.0
count_digits	0.86	0.84	0.85	80.0
count_vowels	0.89	0.85	0.87	80.0
factorial	0.94	0.79	0.86	85.0
factorial_rec	0.97	0.88	0.92	75.0
factors	0.83	0.84	0.83	80.0
fair_warn	0.96	0.98	0.97	80.0
fib_1	0.81	0.79	0.80	80.0
fib_2	0.77	0.82	0.80	80.0
fly_swatter	0.81	0.90	0.85	80.0
gcd	0.92	0.69	0.79	85.0
gcd_rec	0.84	0.92	0.88	75.0
hello_world	0.95	1.00	0.98	80.0
lcm	0.80	0.74	0.77	80.0
leap_year	0.88	0.86	0.87	80.0
magicka	0.97	0.85	0.91	80.0
min_product	0.82	0.90	0.86	80.0
multibase_hap	1.00	0.95	0.97	80.0
n_palindrome	0.78	0.81	0.80	80.0
n_is_prime	0.84	0.86	0.85	80.0
n_sum_of_p	0.75	0.74	0.74	80.0
pos_or_neg	0.83	0.84	0.83	80.0
power_n	0.69	0.79	0.74	80.0
prime_n	0.83	0.81	0.82	80.0
pyramid	0.90	0.95	0.93	80.0
quot_remainder	0.74	0.92	0.82	80.0
remove_char	0.89	0.88	0.88	80.0
reverse_int	0.85	0.89	0.87	80.0
rotate	0.91	0.92	0.92	80.0
saving_univ	0.87	0.94	0.90	80.0
snapper_chain	0.88	0.89	0.88	80.0
store_struct	0.88	0.96	0.92	80.0
strcat	0.89	0.88	0.88	80.0
strcpy	0.87	0.90	0.88	80.0
stringsort	0.91	0.92	0.92	80.0
strlen	0.91	0.94	0.93	80.0
sum	0.89	0.84	0.86	80.0
theme_park	0.99	0.99	0.99	80.0
times_table	0.84	0.91	0.87	80.0
train_t_tab	0.94	0.94	0.94	80.0
watersheds	0.95	0.99	0.97	80.0
welcome_cjam	0.99	0.98	0.98	80.0

Table 4 Classification scores for the LSTM on OBF

Classes	Precision	Recall	f1-score	Support
alien_lang	0.99	0.99	0.99	80.0
armstrong_n	0.84	0.84	0.84	80.0
bot_trust	0.98	0.99	0.98	80.0
calculator	0.89	0.94	0.91	80.0
candy_split	0.93	0.82	0.87	80.0
char_freq	0.99	0.96	0.97	80.0
count_digits	0.82	0.88	0.85	80.0
count_vowels	0.90	0.88	0.89	80.0
factorial	0.85	0.90	0.88	78.0
factorial_rec	0.94	0.98	0.96	82.0
factors	0.96	0.88	0.92	80.0
fair_warn	1.00	0.99	0.99	80.0
fib_1	0.75	0.82	0.79	80.0
fib_2	0.79	0.85	0.82	80.0
fly_swatter	0.95	0.96	0.96	80.0
gcd	0.83	0.78	0.80	81.0
gcd_rec	0.87	0.82	0.84	79.0
hello_world	0.99	0.95	0.97	80.0
lcm	0.78	0.82	0.80	80.0
leap_year	0.87	0.84	0.85	80.0
magicka	0.99	1.00	0.99	80.0
min_product	0.97	0.90	0.94	80.0
multibase_hap	1.00	0.98	0.99	80.0
n_palindrome	0.89	0.82	0.86	80.0
n_is_prime	0.84	0.88	0.86	80.0
n_sum_of_p	0.92	0.88	0.90	80.0
pos_or_neg	0.91	0.91	0.91	80.0
power_n	0.70	0.84	0.76	80.0
prime_n	0.84	0.79	0.81	80.0
pyramid	0.87	0.91	0.89	80.0
quot_remainder	0.79	0.85	0.82	80.0
remove_char	0.95	1.00	0.98	80.0
reverse_int	0.91	0.78	0.84	80.0
rotate	0.93	0.95	0.94	80.0
saving_univ	1.00	0.99	0.99	80.0
snapper_chain	0.98	0.99	0.98	80.0
store_struct	0.94	0.96	0.95	80.0
strcat	0.81	0.80	0.81	80.0
strcpy	0.89	0.88	0.88	80.0
stringsort	0.94	0.99	0.96	80.0
strlen	0.91	0.92	0.92	80.0
sum	0.86	0.81	0.83	80.0
theme_park	0.98	1.00	0.99	80.0
times_table	0.86	0.80	0.83	80.0
train_t_tab	0.99	0.99	0.99	80.0
watersheds	1.00	1.00	1.00	80.0
welcome_cjam	0.99	1.00	0.99	80.0

Table 5 Classification scores for the CNN on MsM2015

Classes	Precision	Recall	f1-score	Support
Obfuscator.ACY	0.92	0.86	0.89	252.0
Simda	0.11	0.12	0.12	8.0
Ramnit	0.85	0.87	0.86	314.0
Vundo	0.88	0.82	0.85	89.0
Gatak	0.86	0.84	0.85	202.0
Lollipop	0.86	0.92	0.89	509.0
Kelihos_ver1	0.98	0.93	0.95	88.0
Tracur	0.69	0.65	0.67	145.0
Kelihos_ver3	1.00	0.99	1.00	566.0

Table 6 Classification scores for the CNN on MalImg

Classes	Precision	Recall	f1-score	Support
Adialer.C	1.00	1.00	1.00	26.0
Agent.FYI	1.00	1.00	1.00	19.0
Allaple.A	1.00	1.00	1.00	604.0
Allaple.L	1.00	1.00	1.00	314.0
Alueron.genJ	1.00	1.00	1.00	40.0
Autorun.K	1.00	1.00	1.00	25.0
C2LOP.P	0.82	0.85	0.84	27.0
C2LOP.geng	0.92	0.87	0.89	38.0
Dialplatform.B	1.00	1.00	1.00	43.0
Dontovo.A	1.00	1.00	1.00	34.0
Fakerean	0.97	1.00	0.99	75.0
Instantaccess	1.00	0.99	0.99	90.0
Lolyda.AA1	1.00	1.00	1.00	45.0
Lolyda.AA2	0.96	1.00	0.98	25.0
Lolyda.AA3	1.00	1.00	1.00	24.0
Lolyda.AT	1.00	1.00	1.00	29.0
Malex.genJ	1.00	0.97	0.99	35.0
Obfuscator.AD	0.96	1.00	0.98	25.0
Rbotgen	0.97	1.00	0.98	28.0
Skintrim.N	1.00	1.00	1.00	21.0
Swizzor.genE	0.70	0.59	0.64	27.0
Swizzor.genI	0.54	0.58	0.56	26.0
VB.AT	1.00	1.00	1.00	78.0
Wintrim.BX	1.00	1.00	1.00	13.0
Yuner.A	1.00	1.00	1.00	156.0

presented in Sect. 4. At the end of the section we briefly discuss how the results obtained can be analyzed.

6.1 OBF dataset

CNN On the OBF dataset the CNN model achieves an average accuracy of 92.3% on the hold-out test set with the input images resized to 256×64. This result is a definite improve-

ment from 88% which was the average accuracy on the same dataset with the CNN in [26], where the images had a bigger height (596 pixels) and were not compressed.

This tells us that the information needed by the CNN for the classification is not reduced by the interpolation process. In fact, interpolating the images results in better classification accuracy than the cropping and padding methods used in [26]. The better performance in the classification of 256×64 images reinforces the belief expressed in Marastoni et al. [26] that bigger size correlates with more information for the model and thus results in an easier classification process. In Table 3 we show the classification scores of the CNN.

LSTM The same consideration cannot be done by looking at the LSTM results, where the accuracy for the OBF dataset oscillates around 92.6% for images of size 256×64, compared to 93.4% for the smaller square images. This result has to be attributed to the difficulty encountered by the LSTM due to the increase of the timesteps, which also incurs a loss of speed. Whatever advantage there might be in having bigger images is then lost to the vanishing information in long recurrent networks. Interestingly enough, the classification accuracy does not improve when we switch the height with the width of the images and learn with the LSTM, while the training time does indeed decrease due to the reduced timesteps.

This result is definitely better than the CNN but incurs a processing time overhead. The training process for the LSTM takes about twice as long than the one for the CNN. This is expected, as CNNs are renowned for being very fast models, and furtherore the LSTM is bi-directional so the input must be scanned in 2 directions.

In Table 4 we show the classification scores of the LSTM for every family in OBF. It should be noted how the subdivision of the families in this dataset is more balanced than in the others. This is of course due to the fact that the OBF dataset is generated from scratch and thus does not suffer from class imbalance. We also noticed that the classification scores tend to have more outliers when the classes are distributed differently between the training set and the test set, thus this is avoided through a simple balancing algorithm that results in the almost uniform distribution clearly shown in Tables 4 and 3.

6.2 MsM2015 and MalImg datasets

The same architectures described above are used to classify malware samples from the MsM2015 and MalImg datasets.

CNN The accuracy for the CNN models trained on the two malware datasets is higher than the accuracy achieved on the OBF dataset. Classifying the hold-out test set samples of the MsM2015 dataset yields 92% accuracy on average, while on the MalImg we record results of up to 98.3% average accuracy. The classification scores for the individual classes of CNN on the MsM2015 and MalImg datasets are in Table 5 and Table 6 respectively.

LSTM The LSTM reaches 98.5% average accuracy on MalImg, which is a result comparable to recent works that use the same dataset [29,44]. The same architecture trained on the MsM2015 dataset achieves an average accuracy of 94.2%.

Since the MalImg dataset contains a noticeable class imbalance we provide various classification scores for the single classes with the LSTM in Table 7 and with the CNN in Table 6. Analyzing these scores it is easy to spot two classes in particular, '`Swizzor.genE`' and '`Swizzor.GenI`', that appear to be difficult to classify for both the LSTM and the CNN. The confusion matrix shown in Fig. 5 provides a clearer picture on the classification errors for these two classes, where it is clear that the models struggle with deciding whether some samples belong to the '`Swizzor.genE`' class or the '`Swizzor.GenI`' class. Since these malware samples are simple variants of the same family they appear very similar and cannot be reliably distinguished by our models. In contrast, the confusion matrix shows that the variants '`C2LOP.P`' and '`C2LOP.geng`' do not generate the same

Table 7 Classification scores for the LSTM on MalImg

Classes	Precision	Recall	f1-score	Support
Adialer.C	1.00	1.00	1.00	26.0
Agent.FYI	1.00	1.00	1.00	19.0
Allaple.A	1.00	1.00	1.00	604.0
Allaple.L	1.00	1.00	1.00	314.0
Alueron.genJ	1.00	1.00	1.00	40.0
Autorun.K	1.00	1.00	1.00	25.0
C2LOP.P	0.80	0.89	0.84	27.0
C2LOP.geng	0.88	0.92	0.90	38.0
Dialplatform.B	1.00	1.00	1.00	43.0
Dontovo.A	0.97	1.00	0.99	34.0
Fakerean	0.96	1.00	0.98	75.0
Instantaccess	1.00	1.00	1.00	90.0
Lolyda.AA1	1.00	1.00	1.00	45.0
Lolyda.AA2	1.00	1.00	1.00	25.0
Lolyda.AA3	1.00	1.00	1.00	24.0
Lolyda.AT	1.00	1.00	1.00	29.0
Malex.genJ	1.00	0.97	0.99	35.0
Obfuscator.AD	1.00	1.00	1.00	25.0
Rbotgen	1.00	1.00	1.00	28.0
Skintrim.N	1.00	1.00	1.00	21.0
Swizzor.genE	0.67	0.22	0.33	27.0
Swizzor.genI	0.46	0.62	0.52	26.0
VB.AT	1.00	0.99	0.99	78.0
Wintrim.BX	0.87	1.00	0.93	13.0
Yuner.A	1.00	1.00	1.00	156.0

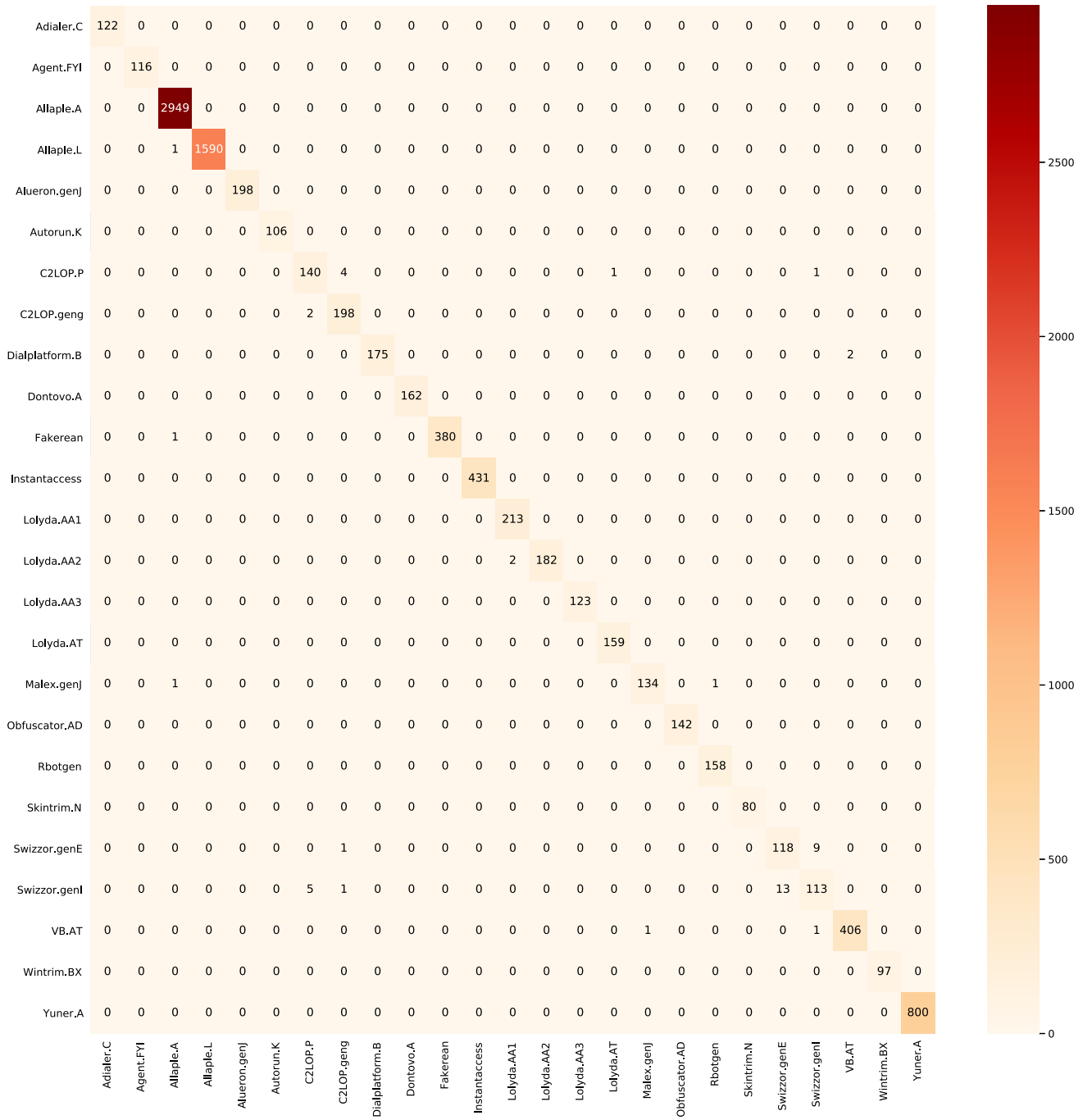

Fig. 5 Confusion matrix for the CNN on the MalImg dataset

amount of errors for our models as they appear to be different enough to be reliably classified.

In Tables 5 and 8 we show the scores for the classification of the MsM2015 dataset on the CNN and LSTM respectively. It is easy to notice that the class '`Simda`' is very hard to classify for both models, achieving an F1 score of 0.12 with the CNN and 0.31 with the LSTM. This is easily explained by the support shown as the last column, which is the number of samples against which the classifier has been tested (the samples in the test set for a specific class). The class '`Simda`' is very under-represented, in fact only 40 samples exist of this class in the whole dataset, while the biggest classes feature thousands of samples each. This of course leads to problems with the classification accuracy, as 40 samples is not nearly enough for the class to be relevant in the training process.

Both models generate better results for the two malware datasets compared to the OBF dataset. This is probably due to the nature of the obfuscated files of the OBF dataset, where

Table 8 Classification scores for the LSTM on MsM2015

Classes	Precision	Recall	f1-score	Support
Obfuscator.ACY	0.95	0.90	0.92	252.0
Simda	0.40	0.25	0.31	8.0
Ramnit	0.90	0.92	0.91	314.0
Vundo	0.92	0.93	0.93	89.0
Gatak	0.91	0.92	0.92	202.0
Lollipop	0.96	0.94	0.95	509.0
Kelihos_ver1	0.93	0.97	0.95	88.0
Tracur	0.84	0.88	0.86	145.0
Kelihos_ver3	1.00	1.00	1.00	566.0

the intra-class similarities and the inter-class differences are not as definite as the samples found in the wild for the other datasets. One advantage of the OBF dataset is the size of the original programs before obfuscation. Since the programs are very small and usually consist of a single function with few auxiliaries, it is easy to change the structure of the programs in a more effective way and this reflects on the binaries themselves (and thus on the images).

6.3 Transfer learning

Having trained a CNN on three different datasets we investigate whether the features learnt in one of them can generalize to the other two. In order to do this we employ transfer learning, a technique that is very popular nowadays as it allows to re-use an already trained architecture for a completely different problem.

As explained in 2, the convolution layers of the CNN are concerned with feature extraction, along with max pooling and activations. The head of the model, two dense layers and a drop out layer in our case, is tasked with using the features to classify the programs into their respective classes. This of course is also true for the bi-directional LSTM that we trained, removing the dense layer from a model and applying a new dense layer with proper outputs provides with a new model that can use pre-trained features for a new classification problem.

Thanks to this neat subdivision of tasks, it has become good practice to download pre-trained models from either big companies or research labs and re-use them for completely new purposes. This allows huge networks such as the ones trained on ImageNet [13] for days (often with very expensive equipment) to be used by people that would otherwise not have the possibility to access such architectures. The hope is that features extracted from a comprehensive image dataset such as ImageNet will hopefully generalize to other image-based learning problems.

In our case we train all the models with images extracted from binaries, thus there could be an interesting intersection in the features as they come from the same problem domain. What needs to be investigated is if the features extracted from one dataset of compiled programs can be used to classify another dataset in the same domain. This is akin to using the networks trained on ImageNet for different (and usually smaller) datasets, since it can save time and resources. At the same time it can open up future possibilities, where big networks are trained on huge datasets of images extracted from compiled programs which can then be repurposed in order to classify smaller and newer datasets.

In the rest of this section we illustrate our experiments with transfer learning with both models on all the datasets.

MsM2015 → MalImg [LSTM] A bi-directional LSTM model has been trained for 200 epochs on the MsM2015 dataset, achieving around 94% accuracy. After removing the dense layer we set the base as untrainable, thus preventing the optimizer from modifying the weights of the two LSTM units and preserving the features learned from the MsM2015 dataset. We then added a new dense layer with 25 outputs (as opposed to the 9 for the previous classification problem) and we trained on the MalImg dataset for a little over 321 epochs, achieving 98.4% accuracy on the hold-out dataset. This result is on par or even slightly better than most models trained using only the MalImg dataset. The performance of the LSTM on the MalImg dataset is already very good and adding the transfer learning actually increased the learning time from around 100 epochs to more than 400 for the same accuracy.

MsM2015 → MalImg [CNN] The CNN model performs very well on the MalImg dataset, taking between 45 to 50 epochs to achieve 98.4% accuracy. This is the best result for the CNN so far and the fastest training time.

After downloading a trained model for the MsM2015 dataset (which achieved around 92% accuracy on said dataset), we attach a new dense layer and retrain it for 80 epochs. The final accuracy is 98.2%.

These experiments led us to believe that there is some untapped potential in the process of transfer learning applied to our problem setting. The positive results could stem from the fact that the MsM2015 dataset, while consisting of less classes altogether, contains a comparable amount of samples. It also certainly helps that the programs from both datasets are for the Windows system, thus possibly sharing many visual features. This hypothesis is tested in the next attempt.

OBF → MalImg transfer [CNN and LSTM] The process described above has been tried with a CNN and an LSTM, both trained on the OBF dataset. The CNN model on the MalImg dataset, with the base of the network set as untrainable, achieved 97% accuracy on the hold out set. This confirms that the features learned by the CNN model to solve the classification problem for the OBF dataset are in fact applicable to the MalImg dataset. An analysis of the classification errors revealed that the same problems persist with the new

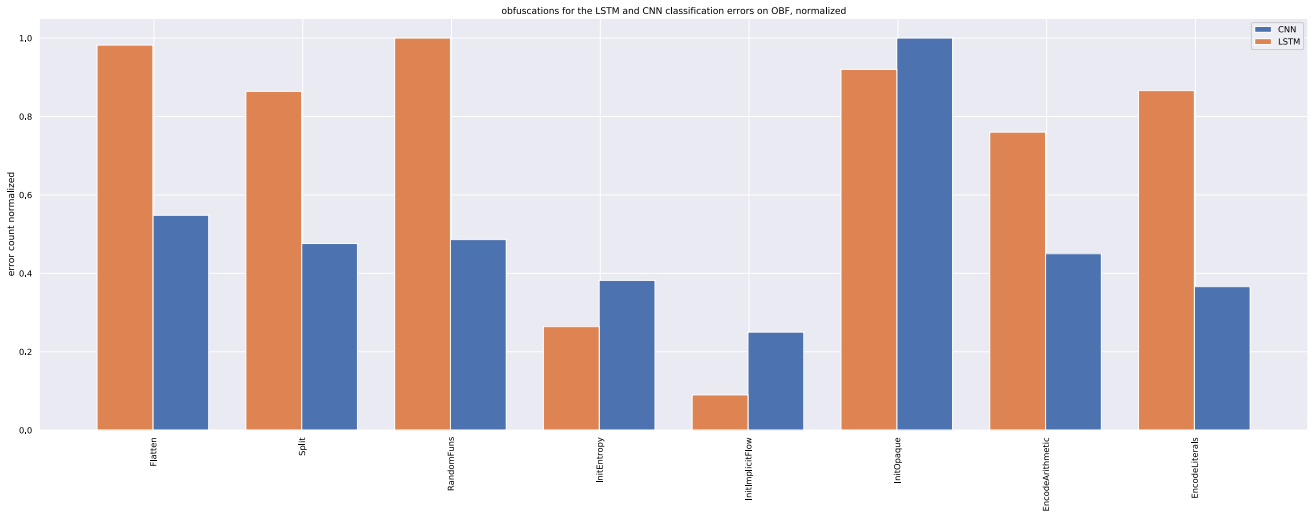

Fig. 6 Obfuscations frequency in the classification errors for the CNN and LSTM on the OBF dataset

model. In particular, the classes '`Swizzor.genE`' and '`Swizzor.genI`' are still very hard to tell apart. Since the new network allows updates only on the weights for the dense layers, the training time is also greatly reduced, going from an average 70 seconds to around 18 s with only a slight reduction in accuracy.

The LSTM model also improves on the training time but the accuracy decreases to around 87%, making it less viable. *OBF → MsM2015 [CNN and LSTM]* The models generated with transfer learning for the MsM2015 dataset perform slightly worse than the ones for the MalImg dataset. The average accuracy for the CNN model trained on OBF and then transferred to the classification of MsM2015 is around 78%, a full 14 points lower than its counterpart trained solely on the MsM2015 dataset. The LSTM does not fare better, its accuracy hovering around 70%.

These results come solely from training on images of size 64×64 so the lower accuracy could come from the original model not learning enough features from the OBF dataset. The MsM2015 dataset is also generally harder to learn from when compared to MalImg.

MalImg → MsM2015 [CNN and LSTM] For completeness we tried to apply the models trained on the MalImg dataset to classify samples in the MsM2015 dataset. The accuracy achieved by the CNN model obtained via transfer learning is around 76%, only slightly lower than the previous experiment with the model trained on OBF. Once again the LSTM model achieves a lower score than the CNN with around 70% accuracy.

This experiment suffers from the small size of the MalImg dataset and from the apparent difficulty inherent in classifying the MsM2015 dataset while only looking at the images extracted from its binaries.

6.4 Error analysis

As anticipated, the custom nature of the OBF dataset allows us to monitor the effect of the obfuscations on the classification results. In order to do this we collected all the mistakenly classified samples from every run of both models against the OBF dataset and counted the obfuscations that have been applied to such samples. The obfuscation count is then averaged through the runs (in our case 20 runs) and then finally normalized, so we can have a number between 0 and 1 that is easy to compare between the different models.

In Fig. 6 we show the result of this study in a bar graph. The CNN results (in the blue, wider bars) clearly show that the network has the most trouble classifying programs that have been transformed by the InitOpaque transformation, while the Flatten transformation, with around half the errors on average, comes in at second place. This big discrepancy between the errors makes it clear that the spatial features extracted by the CNN have trouble when programs are obfuscated with InitOpaque, possibly due to the huge amount of entropy added to the binary (easily seen in Fig. 2).

The LSTM (in orange, slimmer bars) has a more uniform distribution of the errors wrt the applied obfuscations. InitOpaque is still one of the obfuscations that are harder to classify, but is preceded by RandomFuns and Flatten, while being followed closely by Split and EncodeLiterals. What we can gauge from this analysis is that the LSTM does not have a particular weakness toward specific obfuscations but struggles relatively uniformly on the hardest obfuscations.

These model-specific observations make it clear that using one particular model to classify binaries by looking at their images has its drawbacks. The general lack of precision of the models trained on the OBF dataset further strengthen this idea.

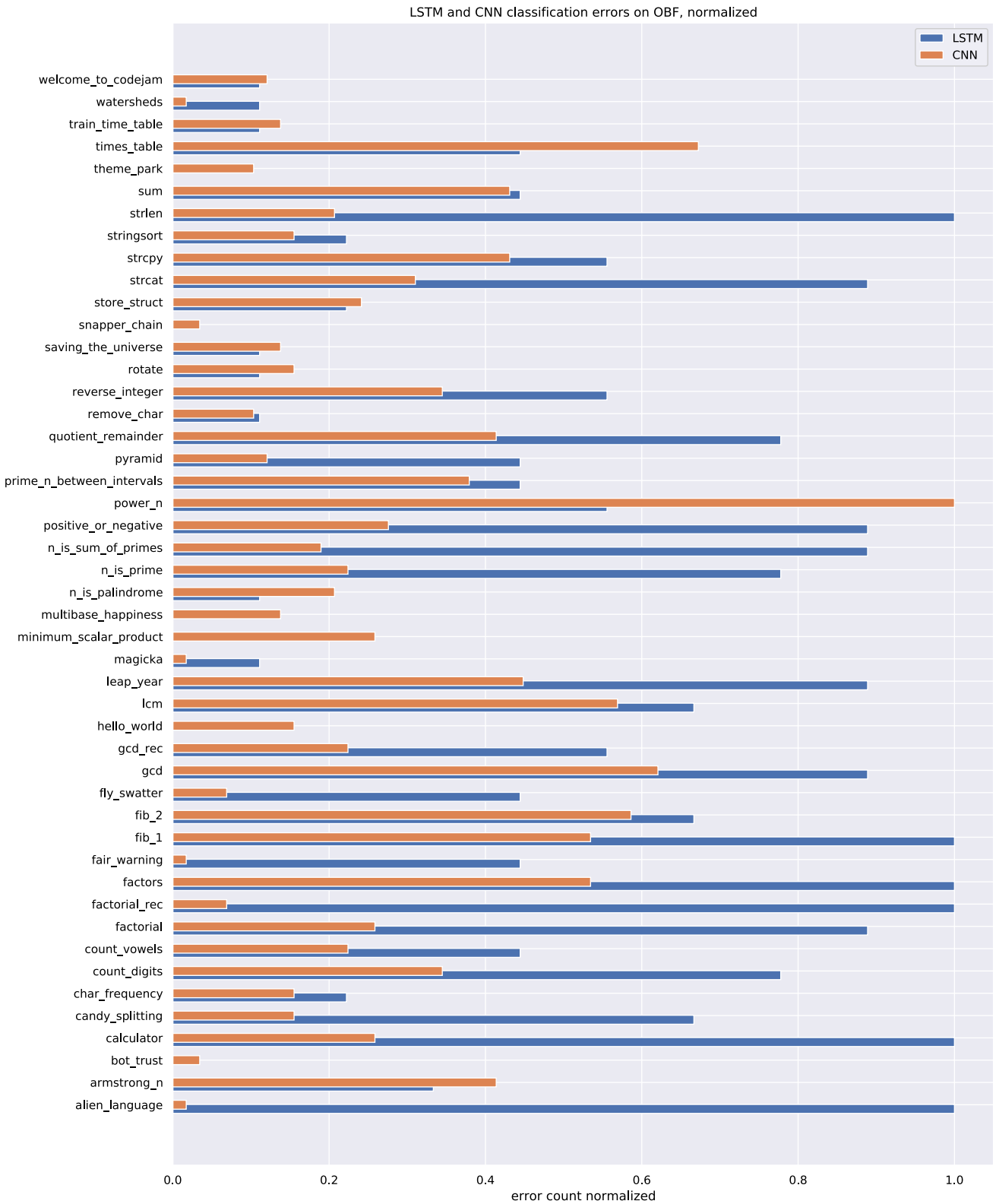

Fig. 7 Classification errors for the CNN and the LSTM on the OBF dataset, normalized

Another point of interest is that both models find it very easy to classify programs that are encoded with InitEntropy and InitImplicitFlow. This reinforces the observation that certain obfuscations are more effective than others against these classification techniques. As with images, the transformation used to fool the classifier can be more or less effective and part of this work is to show that this is the case.

In Fig. 7 we show the errors with a focus on the classes of the OBF dataset. The errors of the LSTM are again more uniformly distributed while the CNN presents few taller peaks. It is interesting to note that some programs that appear on the higher end of error count for the LSTM are among the easiest to classify for the CNN (i.e. '`factorial_rec`' and '`alien_language`'). This again highlights the difference in strength of the two models.

6.5 Comparison with existing works

We decided not to directly compare the accuracy values recorded with the two malware datasets in this paper with those achieved in previous works. The main reason for this is that the goal of this work is not to raise the ever-raising bar of classification accuracy in a specific domain, as that can usually be achieved by simply spending more time overtuning the parameters or throwing more expensive hardware at the problem. At the same time, the accuracy level that we report is always taken from a hold-out test set that has been extracted randomly from the main dataset. This is a different approach than the one taken in most works we surveyed

7 Conclusions and discussion

We have shown how two deep learning models (a CNN and an LSTM) fare in a classification task on a generated dataset and two real-life malware datasets. The models have been built with an image classification task in mind and have been fine-tuned with a custom generated dataset of obfuscated programs. The results clearly show that the LSTM model performs better in all the classification tasks, reaching 98.5% average accuracy on a hold-out set of the MalImg dataset which is on par or superior to other studies in the state of the art. The models trained with the MalImg and MsM2015 datasets have then been subjects of transfer learning experiments in order to generate new models that have been trained on one dataset, while classifying samples from the other. With the accuracy of both these classifiers generated through transfer learning we verified that the features learned from either dataset can be used to classify malware from the other. This is a great result because it means that, not only can we use the images extracted from executable malware samples in order to classify them into their respective families, we can also transfer the knowledge gathered during such process to classify a new malware dataset. Akin to the results in image recognition, this can potentially save a lot of computation time and resources when dealing with newly released datasets.

Discussion These promising results listed in this work led us to believe that there is a lot of untapped potential in transfer learning applied to malware classification. Further experiments are needed to have a more comprehensive view of the potentials and limitations of this technique. For example we envision future experiments with a different starting pool of programs to generate a new, bigger dataset to which we can apply the obfuscations. Having more classes in the OBF dataset would allow for more diversity and bigger programs could be included since the size limitations of [26] have been overcome with bicubic interpolation. The pool of obfuscations can also be considerably expanded.

Different visualization techniques could also be needed in the future. It is evident that merely changing the way the images are resized greatly impacts the learning process, this means that the way we extract images from the binaries is important. We wonder if designing new techniques that are not meant for generic images (such as bicubic interpolation) but catered specifically to programs could aid the learning process.

On a related note, it would be also interesting to see what features are extracted by the networks when encountering different obfuscations. An attention based network could point out which of these features is important to distinguish the programs in different classes and which ones are merely introduced by the obfuscations.

Funding Open access funding provided by Università degli Studi di Verona within the CRUI-CARE Agreement.

References

1. Andriesse, D., Chen, X., Van Der Veen, V., Slowinska, A., Bos, H.: An in-depth analysis of disassembly on full-scale x86/x64 binaries. In: 25th USENIX Security Symposium (USENIX Security 16), pp. 583–600 (2016)
2. Barak, B., Goldreich, O., Impagliazzo, R., Rudich, S., Sahai, A., Vadhan, S., Yang, K.: On the (im) possibility of obfuscating pro-

grams. In: Annual International Cryptology Conference, pp. 1–18. Springer (2001)
3. Bengio, Y., LeCun, Y., Henderson, D.: Globally trained handwritten word recognizer using spatial representation, convolutional neural networks, and hidden Markov models. In: Advances in Neural Information Processing Systems, pp. 937–944 (1994)
4. Bengio, Y., Simard, P., Frasconi, P.: Learning long-term dependencies with gradient descent is difficult. IEEE Trans. Neural Netw. **5**(2), 157–166 (1994)
5. Bhodia, N., Prajapati, P., Di Troia, F., Stamp, M.: Transfer learning for image-based malware classification. arXiv preprint arXiv:1903.11551 (2019)
6. Bradski, G.: The OpenCV Library. Dr. Dobb's Journal of Software Tools (2000)
7. Canavese, D., Regano, L., Basile, C., Viticchié, A.: Estimating software obfuscation potency with artificial neural networks. In: International Workshop on Security and Trust Management, pp. 193–202. Springer (2017)
8. Ceccato, M., Di Penta, M., Nagra, J., Falcarin, P., Ricca, F., Torchiano, M., Tonella, P.: The effectiveness of source code obfuscation: an experimental assessment. In: 2009 IEEE 17th International Conference on Program Comprehension, pp. 178–187. IEEE (2009)
9. Chen, L.: Deep transfer learning for static malware classification. arXiv preprint arXiv:1812.07606 (2018)
10. Collberg, C.: The tigress c diversifier/obfuscator. Retrieved August **14**, 2015 (2015)
11. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations (1997)
12. Cui, Z., Du, L., Wang, P., Cai, X., Zhang, W.: Malicious code detection based on cnns and multi-objective algorithm. J. Parallel Distrib. Comput. **129**, 50–58 (2019)
13. Deng, J., Dong, W., Socher, R., Li, L.J., Li, K., Fei-Fei, L.: Imagenet: A large-scale hierarchical image database. In: 2009 IEEE Conference on Computer Vision and Pattern Recognition, pp. 248–255. Ieee (2009)
14. Deshotels, L., Notani, V., Lakhotia, A.: Droidlegacy: Automated familial classification of android malware. Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop **2014**, 1–12 (2014)
15. Gibert, D., Mateu, C., Planes, J.: A hierarchical convolutional neural network for malware classification. In: 2019 International Joint Conference on Neural Networks (IJCNN), pp. 1–8. IEEE (2019)
16. 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)
17. Hochreiter, S., Schmidhuber, J.: Long short-term memory. Neural Comput. **9**(8), 1735–1780 (1997)
18. Jain, M., Andreopoulos, W., Stamp, M.: Convolutional neural networks and extreme learning machines for malware classification. J. Comput. Virol. Hacking Tech. **16**(3), 229–244 (2020)
19. Kang, J., Jang, S., Li, S., Jeong, Y.S., Sung, Y.: Long short-term memory-based malware classification method for information security. Comput. Electr. Eng. **77**, 366–375 (2019)
20. Kebede, T.M., Djaneye-Boundjou, O., Narayanan, B.N., Ralescu, A., Kapp, D.: Classification of malware programs using autoencoders based deep learning architecture and its application to the microsoft malware classification challenge (big 2015) dataset. In: 2017 IEEE National Aerospace and Electronics Conference (NAECON), pp. 70–75. IEEE (2017)
21. Keys, R.: Cubic convolution interpolation for digital image processing. IEEE Trans. Acoust. Speech Signal Process. **29**(6), 1153–1160 (1981)
22. Kukačka, J., Golkov, V., Cremers, D.: Regularization for deep learning: a taxonomy. arXiv preprint arXiv:1710.10686 (2017)
23. Lawrence, S., Giles, C.L., Tsoi, A.C., Back, A.D.: Face recognition: a convolutional neural-network approach. IEEE Trans. Neural Netw. **8**(1), 98–113 (1997)
24. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proc. IEEE **86**(11), 2278–2324 (1998)
25. LeCun, Y., Cortes, C., Burges, C.: Mnist handwritten digit database. AT&T Labs [Online]. http://yann.lecun.com/exdb/mnist**2** (2010)
26. Marastoni, N., Giacobazzi, R., Dalla Preda, M.: A deep learning approach to program similarity. In: Proceedings of the 1st International Workshop on Machine Learning and Software Engineering in Symbiosis, pp. 26–35 (2018)
27. Marastoni, N.: Niccolò Marastoni's personal website. https://niccolomarastoni.github.io/articles.html (2021)
28. McAfee: McAfee Labs Threats Report 2020. https://www.mcafee.com/enterprise/en-us/assets/reports/rp-quarterly-threats-nov-2020.pdf (2020)
29. Naeem, H., Ullah, F., Naeem, M.R., Khalid, S., Vasan, D., Jabbar, S., Saeed, S.: Malware detection in industrial internet of things based on hybrid image visualization and deep learning model. Ad Hoc Netw. **105**, 102154 (2020)
30. Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B.: Malware images: visualization and automatic classification. In: Proceedings of the 8th International Symposium on Visualization for Cyber Security, p. 4. ACM (2011)
31. OKane, P., Sezer, S., McLaughlin, K.: Obfuscation: the hidden malware. IEEE Secur. Priv. **9**(5), 41–47 (2011)
32. Oliva, A., Torralba, A.: Building the gist of a scene: the role of global image features in recognition. Prog. Brain Res. **155**, 23–36 (2006)
33. O'Malley, T., Bursztein, E., Long, J., Chollet, F., Jin, H., Invernizzi, L., et al.: Keras Tuner. https://github.com/keras-team/keras-tuner (2019)
34. Pan, S.J., Yang, Q.: A survey on transfer learning. IEEE Trans. Knowl. Data Eng. **22**(10), 1345–1359 (2009)
35. Perez, L., Wang, J.: The effectiveness of data augmentation in image classification using deep learning. arXiv preprint arXiv:1712.04621 (2017)
36. Pratt, L.Y., Mostow, J., Kamm, C.A., Kamm, A.A.: Direct transfer of learned information among neural networks. Aaai **91**, 584–589 (1991)
37. Programiz: C examples. https://www.programiz.com/c-programming/examples (2020)
38. Rawat, W., Wang, Z.: Deep convolutional neural networks for image classification: a comprehensive review. Neural Comput. **29**(9), 2352–2449 (2017)
39. Reitermanova, Z.: Data splitting. In: WDS **10**, 31–36 (2010)
40. Rezende, E., Ruppert, G., Carvalho, T., Ramos, F., De Geus, P.: Malicious software classification using transfer learning of resnet-50 deep neural network. In: 2017 16th IEEE International Conference on Machine Learning and Applications (ICMLA), pp. 1011–1014. IEEE (2017)
41. Ronen, R., Radu, M., Feuerstein, C., Yom-Tov, E., Ahmadi, M.: Microsoft malware classification challenge. arXiv preprint arXiv:1802.10135 (2018)
42. Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., Weippl, E.: Protecting software through obfuscation: Can it keep pace with progress in code analysis? ACM Comput. Surv. (CSUR) **49**(1), 1–37 (2016)
43. Shorten, C., Khoshgoftaar, T.M.: A survey on image data augmentation for deep learning. J. Big Data **6**(1), 60 (2019)
44. Vasan, D., Alazab, M., Wassan, S., Naeem, H., Safaei, B., Zheng, Q.: Imcfn: image-based malware classification using fine-tuned convolutional neural network architecture. Comput. Netw. **171**, 107138 (2020)

45. Venkatraman, S., Alazab, M., Vinayakumar, R.: A hybrid deep learning image-based analysis for effective malware detection. J. Inf. Secur. Appl. **47**, 377–389 (2019)
46. Yakura, H., Shinozaki, S., Nishimura, R., Oyama, Y., Sakuma, J.: Neural malware analysis with attention mechanism. Comput. Secur. **87**, 101592 (2019)
47. You, I., Yim, K.: Malware obfuscation techniques: a brief survey. In: 2010 International Conference on Broadband, Wireless Computing, Communication and Applications, pp. 297–300. IEEE (2010)
48. Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: 2012 IEEE Symposium on Security And Privacy, pp. 95–109. IEEE (2012)

Terms and Conditions

Springer Nature journal content, brought to you courtesy of Springer Nature Customer Service Center GmbH (“Springer Nature”).
Springer Nature supports a reasonable amount of sharing of research papers by authors, subscribers and authorised users (“Users”), for small-scale personal, non-commercial use provided that all copyright, trade and service marks and other proprietary notices are maintained. By accessing, sharing, receiving or otherwise using the Springer Nature journal content you agree to these terms of use (“Terms”). For these purposes, Springer Nature considers academic use (by researchers and students) to be non-commercial.
These Terms are supplementary and will apply in addition to any applicable website terms and conditions, a relevant site licence or a personal subscription. These Terms will prevail over any conflict or ambiguity with regards to the relevant terms, a site licence or a personal subscription (to the extent of the conflict or ambiguity only). For Creative Commons-licensed articles, the terms of the Creative Commons license used will apply.
We collect and use personal data to provide access to the Springer Nature journal content. We may also use these personal data internally within ResearchGate and Springer Nature and as agreed share it, in an anonymised way, for purposes of tracking, analysis and reporting. We will not otherwise disclose your personal data outside the ResearchGate or the Springer Nature group of companies unless we have your permission as detailed in the Privacy Policy.
While Users may use the Springer Nature journal content for small scale, personal non-commercial use, it is important to note that Users may not:

1. use such content for the purpose of providing other users with access on a regular or large scale basis or as a means to circumvent access control;
2. use such content where to do so would be considered a criminal or statutory offence in any jurisdiction, or gives rise to civil liability, or is otherwise unlawful;
3. falsely or misleadingly imply or suggest endorsement, approval , sponsorship, or association unless explicitly agreed to by Springer Nature in writing;
4. use bots or other automated methods to access the content or redirect messages
5. override any security feature or exclusionary protocol; or
6. share the content in order to create substitute for Springer Nature products or services or a systematic database of Springer Nature journal content.

In line with the restriction against commercial use, Springer Nature does not permit the creation of a product or service that creates revenue, royalties, rent or income from our content or its inclusion as part of a paid for service or for other commercial gain. Springer Nature journal content cannot be used for inter-library loans and librarians may not upload Springer Nature journal content on a large scale into their, or any other, institutional repository.
These terms of use are reviewed regularly and may be amended at any time. Springer Nature is not obligated to publish any information or content on this website and may remove it or features or functionality at our sole discretion, at any time with or without notice. Springer Nature may revoke this licence to you at any time and remove access to any copies of the Springer Nature journal content which have been saved.
To the fullest extent permitted by law, Springer Nature makes no warranties, representations or guarantees to Users, either express or implied with respect to the Springer nature journal content and all parties disclaim and waive any implied warranties or warranties imposed by law, including merchantability or fitness for any particular purpose.
Please note that these rights do not automatically extend to content, data or other material published by Springer Nature that may be licensed from third parties.
If you would like to use or distribute our Springer Nature journal content to a wider audience or on a regular basis or in any other manner not expressly permitted by these Terms, please contact Springer Nature at

onlineservice@springernature.com