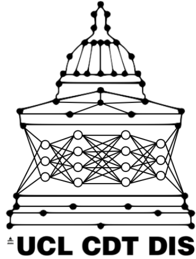


UCL CDT DIS Note

10th May 2021

Machine Learning for Static Malware Analysis

Emily Lewis, Toni Mlinarević, and Alex Wilkinson

University College London

Machine learning was used to classify executable files as either malicious or benign. Multiple models were trained on feature sets derived from several sample characteristics: PE headers, bytes n-grams, control flow graphs and API call graphs.A dataset of 32 967 benign and 74 924 malign samples was compiled and will be made available to the community where safely possible. An ensemble classifier was trained using the prediction outputs of the individual models as input features. All individual models performed well, and the ensemble markedly improved on these scores for a final classification accuracy of 98.9%. This demonstrates that multi-modal late fusion is a potent tool for malware detection. Classification of samples into malware families was also conducted using the call graph classifier. This displayed strong discriminative power for certain classes.

© 2021 University College London for the benefit of the Centre for Doctoral Training in Data Intensive Sciences. Reproduction of this article or parts of it is allowed as specified in the CC-BY-4.0 license.

Contents

1 Introduction **3**
- 1.1 Static Malware Analysis . . . 3
- 1.2 Portable Executable Format . . . 3

2 Dataset Construction **4**
- 2.1 Creating a Dataset . . . 4
- 2.2 Data Exploration . . . 4

3 Feature Selection **6**
- 3.1 PE Headers . . . 6
- 3.2 Bytes n-grams . . . 7
- 3.3 Graphs . . . 8
  - 3.3.1 Opcode Attributed Control Flow Graphs . . . 9
  - 3.3.2 API Call Attributed Call Graphs . . . 10
  - 3.3.3 Deep Graph Convolutional Neural Network Classifier . . . 11

4 Multimodal Ensemble **12**

5 Results and Discussion **13**
- 5.1 PE Headers . . . 13
- 5.2 Bytes n-grams . . . 15
- 5.3 Graphs . . . 16
  - 5.3.1 Binary Classification . . . 16
  - 5.3.2 Malware Family Classification . . . 18
- 5.4 Ensemble . . . 19

6 Conclusion **20**

7 Acknowledgements **21**

References **22**

1 Introduction

1.1 Static Malware Analysis

Efficient identification of malware is vitally important in an age when approximately 230 000 new malware samples are produced each day [1]. Ransomware attacks alone cost businesses millions in lost revenue per year and can disrupt crucial public services as demonstrated in the 2017 NHS WannaCry attack [2]. This threat will only grow with the rise of consumer cloud Infrastructure as a Service and Internet of Things networking.

The two main approaches to malware detection are static and dynamic analysis. Dynamic analysis involves executing samples in a secure sandbox for observation by a malware researcher. Suspicious behaviour might include unexpected changes to the file system or unusual network requests. Though effective this method is resource intensive and can be evaded by malicious programs which detect their run-time environment.

By contrast static analysis attempts to classify a sample without the need for execution. Signature based static analyses use hashing techniques. Programs are scanned for an identifying sequence of bytes and referenced against a database of known malign samples for classification. However this approach is vulnerable to previously unseen files and code obfuscation. Further static analysis extracts characteristics directly from a sample such as API calls, metadata strings and libraries to determine program behaviour. This analysis is capable of detecting zero-day attacks but requires intelligent interpretation of considerable amounts of data. Machine learning offers an opportunity to partially automate this process and enhance the ability of researchers to detect malware on massive scales.

1.2 Portable Executable Format

The majority of malware targets the Windows operating system (OS). The Portable Executable (PE) file format is the standard format used by executables, dynamic-linked libraries and others on 32-bit and 64-bit Windows systems. Consequently a significant proportion of malware samples are PE files. For instance 47% of all files submitted to VirusTotal, a free virus-scanning service, used the PE format [3, 4]. In addition to containing the binary code itself, a PE file contains structural information, describes how an OS should map a program into memory and which external functions are called via the import address table (IAT). From this header metadata and section information, characteristics can be extracted in the form of byte n-grams, opcode n-grams, converted strings and PE header field values. Examination of these features indicates program run-time behaviour and enables discrimination between malicious and benign Windows binaries.

2 Dataset Construction

2.1 Creating a Dataset

An important factor to the success of this project is the curation of a dataset that closely resembles the kind of benign and malicious binaries currently in circulation. Having such a dataset is essential in building machine learning models that can successfully generalise.

We obtain an initial dataset of benign samples from the binaries found on our Windows PCs. These contain some third-party software but are predominantly Windows 10 system files. These are supplemented with binaries taken from Windows Server 2000 and 2012, Windows XP, and Windows 7 installations. To add variety to the benign samples, an additional set of binaries are obtained by installing the 300 most popular packages from a Windows package manager. After combining the benign samples and removing duplicates using the Linux utility fdupes for checking MD5 hashes, we are left with 32 967 unique binaries.

Malicious samples are sourced from the malware repository VirusShare [5]. Samples uploaded to the site within the last two years are downloaded in bulk. The Windows binaries are then identified and queried for their VirusTotal report which gives the result of scans by 60–70 of the leading anti-virus (AV) software. We apply a threshold of 30 scans identifying the binary as malicious to reach a final set of 74 924 unique malware samples.

We understand that the compilation and curation of such a dataset is time consuming. To facilitate future research and encourage research reproducibility we are pursuing opportunities to safely share this data.

2.2 Data Exploration

We perform a limited exploration of the dataset using the metadata available. The timestamps of the binaries are extracted from the PE header and shown in Figure 1. The distributions are a reasonable reflection of the binaries typically found on Windows machines and of the malware that threatens them. The bimodal benign distribution is characteristic of the longer lifespan Windows system files and the frequently updated third-party software that makes up our dataset.

We are interested to know the types of malware in our dataset. To do this we take the AV scans from the VirusTotal report and have them vote on the malware family. Much of the malware is classified as generic and some of the families have very few malware samples associated with them. The malware families that can be identified are shown in Figure 2. The labelling is crude but sufficient to establish the diversity of our malware. There are examples of spyware (Zusy, Tepfer, Ursnif), ransomware (MSIL, Strictor, Lamer) and adware (Graftor, Razy, Adware).

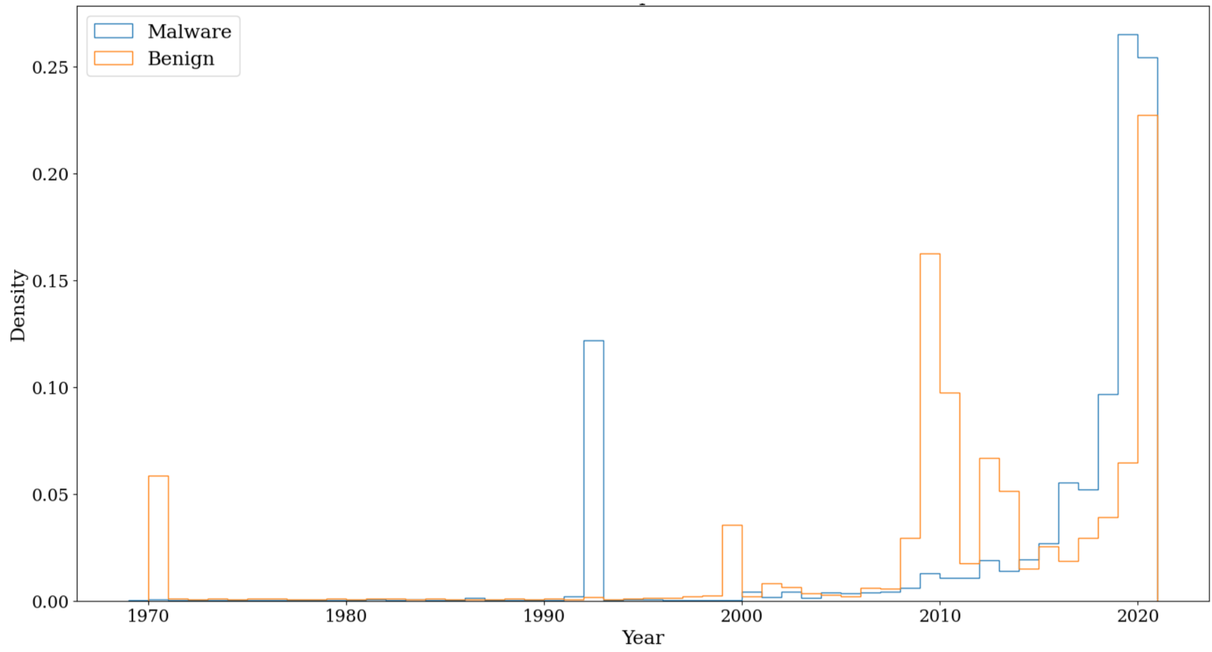


Figure 1: Density histogram of dataset compilation years. Note that binaries with a null timestamp default to 1970 and that a pre-2007 Delphi compiler bug gives binaries a 1992 timestamp.

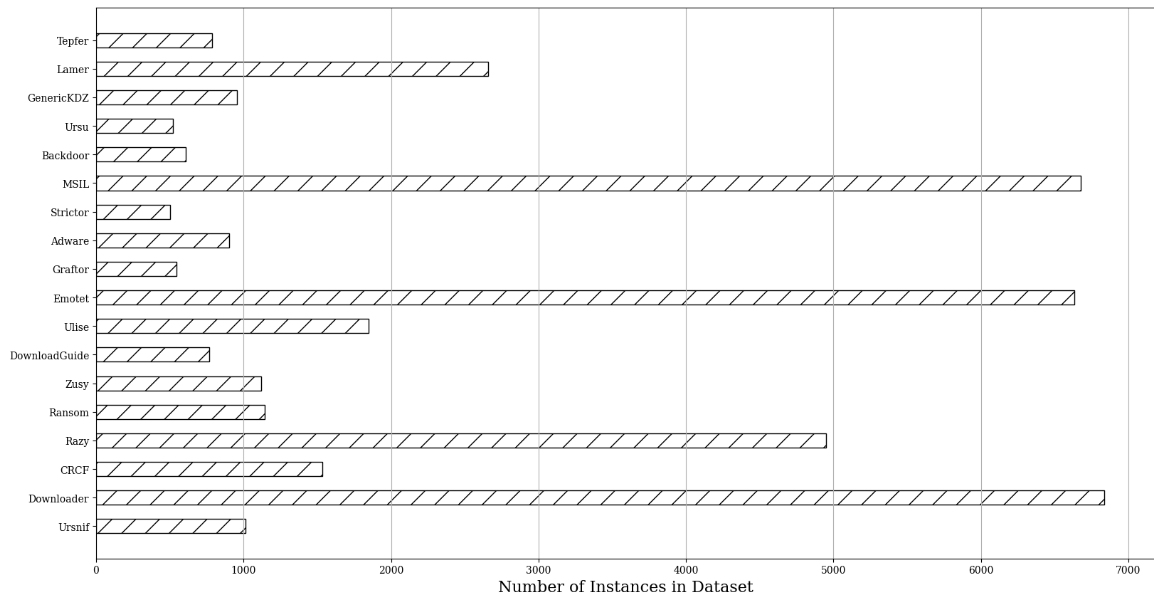


Figure 2: Malware families of malign dataset with more then 500 instances.

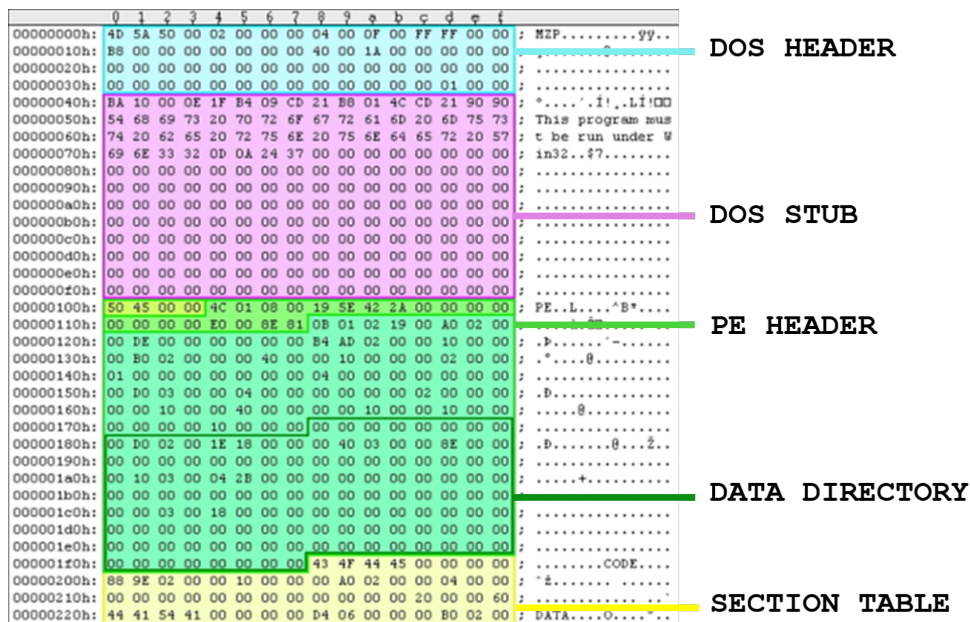


Figure 3: A hexdump from a sample binary. Memory addresses are in the left column, raw hexadecimal code in the middle and converted ASCII strings on the right. PE headers are highlighted.

3 Feature Selection

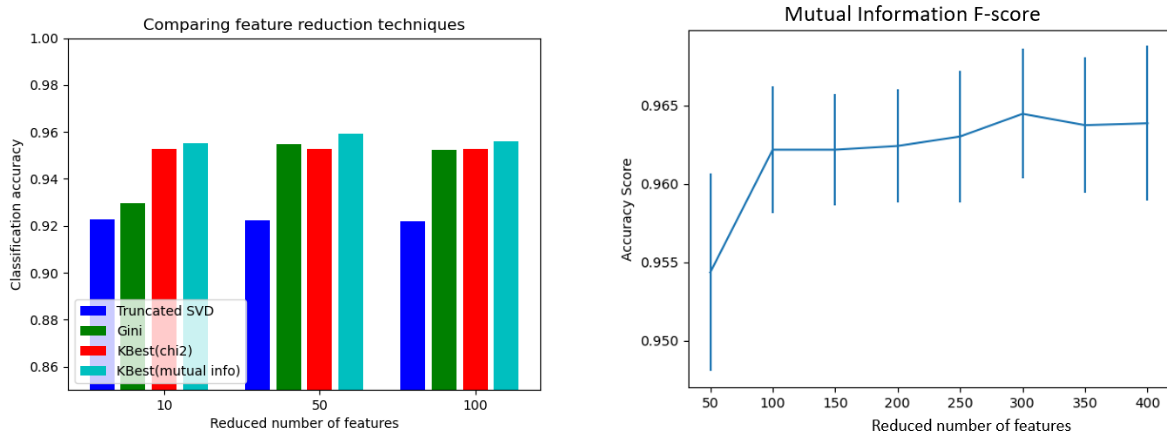
As previously discussed, PE executables present several features of interest to malware detection. Broadly we can consider a program either as a series of bytes, opcodes, function calls or described by a collection of metadata. Each requires a different approach to feature extraction, selection and pre-processing.

3.1 PE Headers

At the highest level of abstraction, we can exploit our knowledge of the PE format to extract metadata from the binary. A PE file consists of headers describing how the file should be executed and sections containing the body of the code. Headers are of particular use to malware research as many of their features are intrinsic to program structure and are difficult to manipulate without affecting functionality. In addition to describing the number and size of sections in a program, PE header fields contain characteristic metadata such as file type, subsystem version and preferred address when loaded into memory.

These values can be extracted as strings by a parser that employs an understanding of the PE format as demonstrated in Figure 3. In this work *pefile*, an open source python module, was used to gather PE header information and section names [6]. This produced a mixture of numerical and categorical values that were converted into machine learning interpretable features with one-hot encoding and scaled. For the largest samples this generated over 2000 features per file. Encoding multiplied this further to roughly 30 000 features. Dimensionality reduction via

feature selection was then necessary to reduce computational processing and improve classification performance. This costly step can potentially be avoided using an alternative regime such as Similarity Encoding [7]. Future work could explore this option.



(a) Effect of selection method on accuracy.

(b) Effect of feature number on accuracy.

Figure 4: A decision tree (DT) classifier was trained on feature sets of varying sizes selected via truncated singular value decomposition, Gini impurity calculated by another DT model, chi-squared and mutual information scoring.

Several feature selection methods and feature number maxima were compared for their effect on classification accuracy, these results are presented in Figures 4a and 4b. It was found that selecting the 300 features with the highest mutual information score produced the best accuracy for a reasonable training time. This is in line with the literature where information gain is popular for selecting features with optimal classification performance [8]. Further feature reduction to 113 features occurred during the training of the XGBoost model using feature importance scoring. More sophisticated feature selection or combinations of multiple methods were not explored and could provide further performance gains.

Features were extracted from 32 712 benign and 28 815 malware samples. The slight class imbalance is due to the parser better interpreting benign binaries as they are less likely to use unusual characters in section names. During the course of this work this limitation was removed so future research could expand the size of the dataset. The features were used to train a variety of models for binary classification implemented in sklearn using an 80:20 train-test split. Hyperparameter tuning was conducted using the hyperopt package to search the large parameter space. Models were optimised for recall over precision or accuracy to reduce the number of false negatives (malicious files classed as benign) as much as possible. These model configurations were validated on the training data using 5-fold cross-validation to avoid overfitting.

3.2 Bytes n-grams

Another set of features that we use to identify malware are sequences of bytes. Firstly, a hexadecimal representation of each executable file is generated using the xxd utility, where each byte is represented by a two-digit hexadecimal number. The n-grams in the file are then found by a

sliding window of n bytes. For example, the 2-grams in the sequence f37d339f would be f37d, 7d33 and 339f.

We use probably the simplest way to construct features for the machine learning model: we first create a list of distinct n-grams from the entire training dataset, and for each of those n-grams, check if it is present in each executable in the dataset. This was the method used by Kolter and Maloof [9]. The presence of an n-gram in a file was denoted by a value of 1, and absence with 0.

The number of distinct n-grams is very large, so they cannot all be used to train models. For example, the number of distinct 2-grams present in our training dataset was 65 536, which means all possible 2-grams were present in at least one executable.

A method is therefore needed to select the most useful n-grams. The method we use is to calculate the information gain for each n-gram, as also performed by Kolter and Maloof [9]:

$$I_i = \sum_{v_i \in \{0,1\}} \sum_{C \in \{C_j\}} P(v_i, C) \log \frac{P(v_i, C)}{P(v_i)P(C)}$$

where C is the class (malware or benign), v_i is the presence value for the ith n-gram, $P(v_j, C)$ is the fraction of samples in class C that have the value v_j for the ith n-gram, $P(v_j)$ is the fraction of all samples that have the value v_j for the ith n-gram, and $P(C)$ is the fraction of samples that are in class C.

We use 2-grams in our models, but in the future other values of n should be tried and the best one selected. Five hundred 2-grams with the highest information gain are selected, and four different models trained using these features: a decision tree, a boosted decision tree, a random forest, and a neural network. The models are trained on a dataset containing 17 338 benign and 50 704 malware samples.

3.3 Graphs

A natural representation of any program's execution is a directed graph. In this representation vertices are short blocks of execution that perform an isolated task whilst edges describe the relations between these blocks. For a control flow graph (CFG) the vertices are blocks of assembly instructions and the edges are control flow. For a call graph (CG) vertices are functions and the edges represent calling relationships between the functions. The potential of these graphs for static malware analysis has been recognised in the literature as a promising approach since it offers a unique level of generality in its representation. The use of PE header data to classify binaries can achieve excellent discriminative power on a dataset but may struggle to generalise to data of different formats or to attempts at obfuscation by malware authors. Conversely, n-gram byte sequences offer a representation that is highly generic but unlikely to match the discriminative power of the PE header data. Graph representations may provide a middle ground between generality and discriminative power. All binary formats have an associated graph structure and malicious behaviour is likely to produce similar patterns. A careful choice of feature vector for the graph's vertices can then help push for high classification accuracies.

There are many examples of graph-based representations being used for static malware classification. Often the graph structure is encoded into a fixed size representation in a pre-processing step [10, 11]. Other approaches have attempted to make use of graph matching networks [12]. The work using graph representations in this study most closely follows [13] where opcodes are used to create attributed CFGs that are passed through a classifier that uses graph convolutional layers. This is an appealing approach since the classifier learns to use the graph structure alongside the features.

In this work we consider CFGs with feature vectors derived from opcodes, as used in [13], and CGs with feature vectors derived from API calls. The use of API attributed CGs with the type of classifier we employ appears to be a new approach to static malware classification.

The *angr* binary analysis toolkit [14] is used to recover the CFG and CG of a binary. Using angr's Python framework, binaries are loaded into memory and a recursive disassembly is performed. Any graphs with vertices fewer than five or greater than 10 000 are discarded. The latter is done with GPU memory during the classifier training in mind and applies to a small fraction of the binaries. This is found to be unnecessarily cautious; a better approach would be to remove the limit and restrict the maximum total number of vertices in the mini-batches. We also apply the constraint that any disassembly taking longer than five minutes is aborted. This is required for processing a large dataset of binaries that can occasionally hang indefinitely during the disassembly. Successfully recovered graphs can then be examined with the tools provided by angr to construct a feature matrix. The resulting edge list and feature matrix is then saved in the numpy compressed array format.

3.3.1 Opcode Attributed Control Flow Graphs

To produce graphs that can be classified using machine learning algorithms we need to assign a feature vector to every vertex. For the CFG each vertex is a block of assembly instructions that contains no control flow until the final instruction where it exits. A small example CFG is shown in Figure 5.

To construct feature vectors we use the frequency of each opcode from the assembly instructions blocks. These define the operation being performed at each instruction, in Figure 5 they are listed in the second column of the block. Owing to the PE format's support of many different instruction sets, the number of possible opcodes is large. To reduce the number of entries in the feature vector only the most frequent 302 of the training set are used. This is the most basic method of feature selection and employing more advanced methods such as graph autoencoders or selection by information gain would almost certainly lead to a better representation of the data. Time constraints prevent us from making use of these techniques in this study. The feature space is then extended by an additional dimension to include the size in bytes of the block. This choice of a 303-dimensional feature space is made to match the CG feature space described in § 3.3.2.

A total of 40 269 malicious and 22 565 benign samples have their CFGs recovered. The average number of vertices is 1 857 and the average graph density is 0.0555.

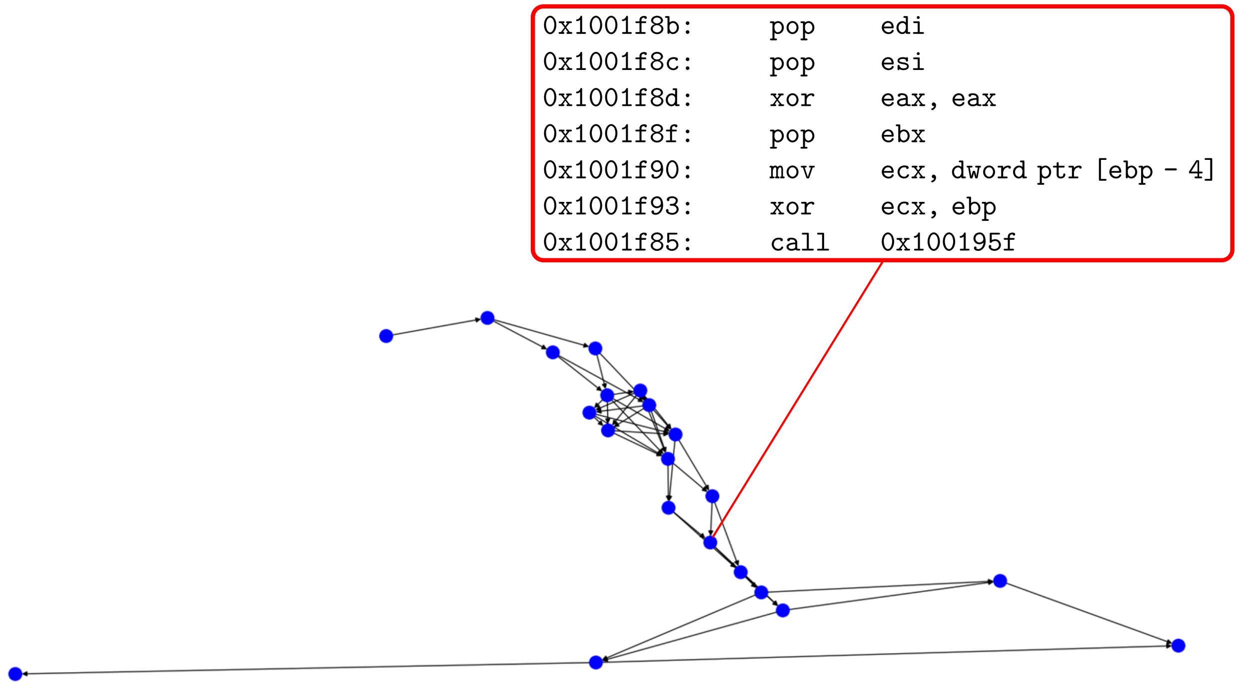


Figure 5: Example of the control flow graph recovered from a binary. The assembly instructions block contains memory addresses (first column), opcodes (second column), and operands (third column).

3.3.2 API Call Attributed Call Graphs

To construct feature vectors for a CG we use the Windows API calls made by the binaries. Once a CG is recovered, vertices that represent API calls are separated from vertices that represent internally defined subroutines. Due to the large number of possible unique API calls, the calls are categorised according to Microsoft documentations. The Microsoft Windows Software Development Kit[1] and Windows Driver Kit Device Driver Interface[2] documentations are scraped to find the header each API call is defined in and then the technologies that require the header for development. These technologies then define the categories, an example of this categorisation can be seen in Figure 6. The Microsoft C runtime library[3] is also scraped and the headers themselves are used for the categories. With these documentations we categorise a total of 73 698 unique API calls into 303 categories.

Figure 6 shows an example of a small CG. At an API call execution leaves the binary resulting in vertices of API calls having a null outdegree. A simple feature vector that marks the presence of API call categories would give all subroutine vertices zero vectors. For many graphs this would mean that a lot of the structure associated with long sequences of subroutines that lead to an API call would not be used in the classification. To prevent this an alternate feature vector is used. The length of the shortest path to each API call vertex is used to define the value of the feature vector in the feature space direction associated with each API call. If there is no path,

[1] https://github.com/MicrosoftDocs/sdk-api
[2] https://github.com/MicrosoftDocs/windows-driver-docs-ddi
[3] https://github.com/MicrosoftDocs/cpp-docs

a value of zero is assigned. This gives the feature space a notion of depth with respect to API calls.

A total of 38 703 malicious and 18 835 benign samples have their CGs recovered. The average number of vertices is 1 481 and the average graph density is 0.00949. The CGs of the malware with an associated malware family are also recovered for a total of 18 385 samples with 12 malware family labels, an average number of vertices of 1 760, and an average graph density of 0.00475.

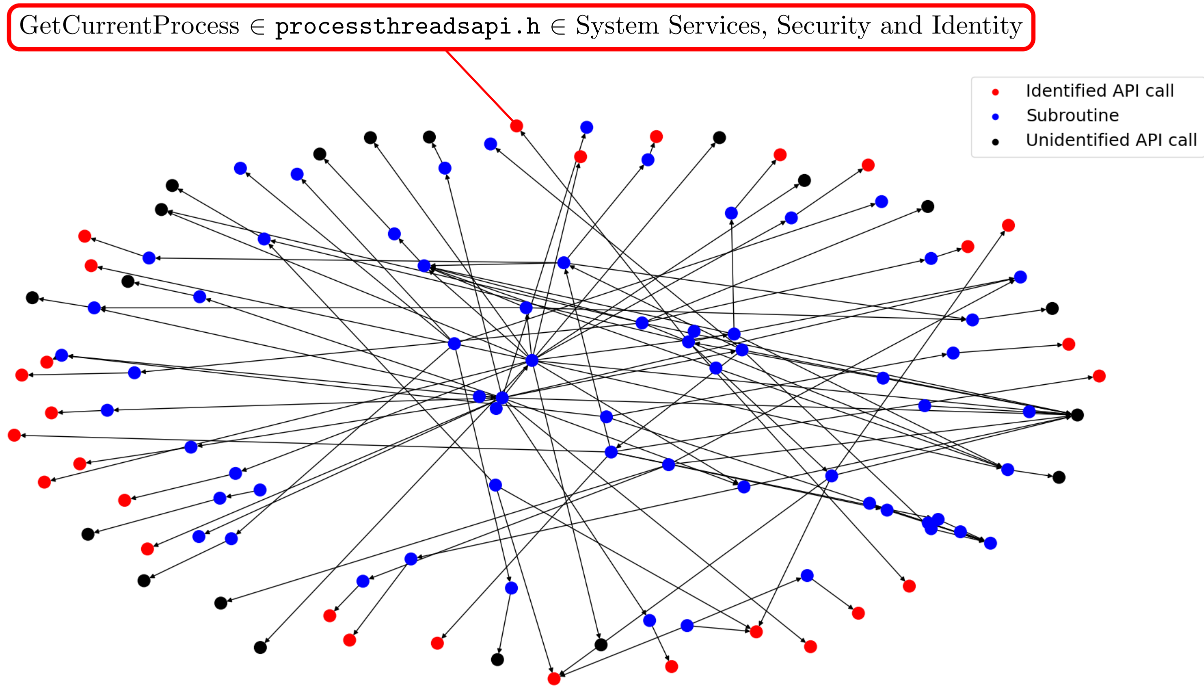


Figure 6: Example of the call graph recovered from a binary. An example of how the API calls are categorised is given in the red box. Unidentified API calls are due to calls not being categorised or being unresolvable.

3.3.3 Deep Graph Convolutional Neural Network Classifier

To classify both types of graph we implement in PyTorch a classifier that uses the Deep Graph Convolutional Neural Network (DGCNN) architecture introduced in [15]. This architecture is chosen since it has been proven in [13] to work in the domain of malware classification. An overview of the network can be seen in Figure 7 and a detailed description of its parts and motivation can be found in the original paper. The key components of the DGCNN are the graph convolution layers and the SortPooling layer. The former propagates vertex information according to the connectivity structure of the graph whilst the latter provides a consistent way to sort vertex features and unifies the sizes of the tensors going to the 1-dimensional convolutions.

The graph datasets are split into train : valid : test in the ratio 70 : 10 : 20. The validation sets are used to perform hyperparameter tuning by hand and to implement early stopping. A grid search is not performed due to the long training time. The tuning of model parameters such as the number of channels, activation functions, and dropout rates has limited effect. The most gain

in performance is obtained by tuning the optimiser used for training as ensuring stable training is the main challenge with the DGCNN. It is found that RMSprop with an initial learning rate of 0.00001 and a small learning rate decay every 2 epochs leads to the best performance. It seems unusual that an adaptive learning rate optimiser (where the learning rate parameter acts as the maximum learning rate) should require learning rate decay but it is found to be crucial in reaching higher performance on the validation set.

A DGCNN for binary malware classification is trained on both the CG and CFG datasets. For the CG dataset a DGCNN is also trained for multiclass classification of the malware family.

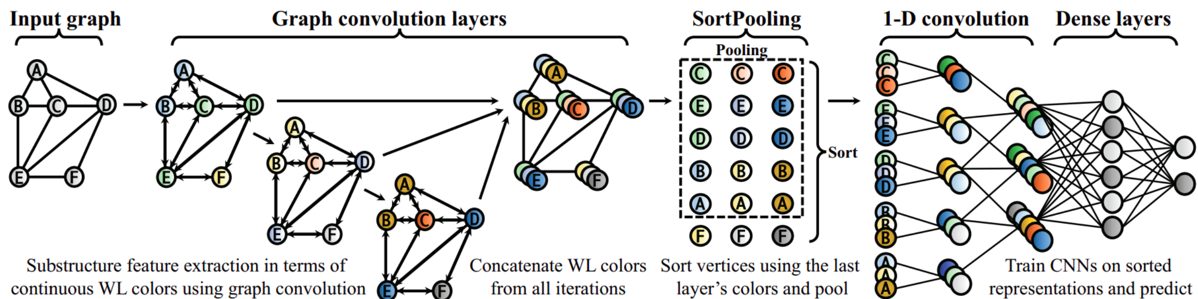


Figure 7: Architecture of the Deep Graph Convolutional Neural Network. Taken from [15].

4 Multimodal Ensemble

Finally we create a meta-ensemble which combines individual model predictions for ultimate classification. This approach is known as late fusion and refers to a process where one model is trained per feature type or modality and their decision values are fused via a mechanism such as averaging, voting or another learned model [16]. This is in contrast to early fusion which concatenates features extracted from each modality and then trains a single model on that joint representation. The benefit of late fusion is that multiple model types are permitted per modality, allowing for flexibility. Additionally as predictions are made separately it is easier to handle missing modalities. For instance, samples for which a CFG cannot be extracted can still be analysed by complementary alternative approaches. Due to this, the late fusion classifier is more robust and better able to predict on unseen samples. This approach is fairly novel and we are only aware of one other work using late fusion in this domain [17]. Even then that study used a different approach to feature modalities and focused on classification into malware families over malware detection.

The models combined for the ensemble are the XGBoost PE header model, the neural network bytes n-grams model, and the DGCNN trained on the CG dataset. Each of these models outputs the probability that an executable is malware. The ensemble model takes these predictions as input features and uses them to train a neural network as a meta-classifier. The predictions used to train the meta-model are obtained from a dataset separate to those used to train the individual models so as to ensure no information leakage. This dataset has 3 438 benign samples and 7 630 malware samples, and is used with an 80:20 train-test split.

5 Results and Discussion

5.1 PE Headers

The binary classification results of models trained on the PE header feature set are presented in Figure 9 and Table 1. The classification accuracy metric is slightly misleading in the case of an imbalanced dataset, so the additional values of precision, recall and area under ROC curves are also given. The PE-based classifiers display good discriminative power. When compared to other solely PE-metadata models in the literature the XGBoost classifier performs well and achieves a score of 0.978 accuracy compared to 0.976 found in Shalaginov et al [8]. This is most likely due to [8] using older, less performant models. All the classifiers favour recall over precision due to the dataset class imbalance and earlier choice to optimise for recall during hyperparameter tuning. If needed the precision-recall threshold of the models can be increased from 0.5 to completely remove the possibility of false negatives. When comparing like-for-like models in the literature, our random forest has an accuracy of 0.971, on par with Kumar et al.[3] who achieve 0.974 using a random forest on a raw feature set. Kumar et al.[3] then derive an integrated feature set using domain knowledge to improve accuracy to 0.988. This indicates that derived feature sets are worthy of exploration in the effort to improve performance.

The individual models were combined into ensembles using hard-voting and stacking. Logistic regression and a neural network were the stacking meta-classifiers tested. None of these approaches outperformed XGBoost, implying that the classifiers were negatively impacting each other. This is surprising as it was assumed that a collection of classifiers would balance out individual biases and produce better results.

A secondary benefit of XGBoost is that as a tree-based ensemble it offers good interpretabiltiy. This is important for a classifier that might act as a filter to flag suspicious samples for further inspection. The model can output features it considers important for making classifications, indicating which values a malware researcher may want to investigate further. XGBoost calculates feature importances by enumerating over the possible data splits proposed by each feature and selecting the highest. These values were extracted and are presented in Figure 8. It can be seen that aside from being suspicious of non-default section names, the classifier relies heavily on *ImageBase* and *VirtualAddress* for making classifications. This is promising as these variables refer to the preferred address of the program when loaded into memory. Non-standard values are often a good indication that a malware writer has tried to use an offset to avoid detection by AV [18]. From this we conclude that PE headers with XGBoost or other tree based ensembles and robust feature curation provide an excellent method for filtering malware. As a result, this was the model used to predict the decision values passed on to the final ensemble. A limitation to bear in mind for PE metadata models in general is that they rely on valid PE headers being available for each sample which is not always the case. With this limitation in mind we examine models based on other feature characteristics.

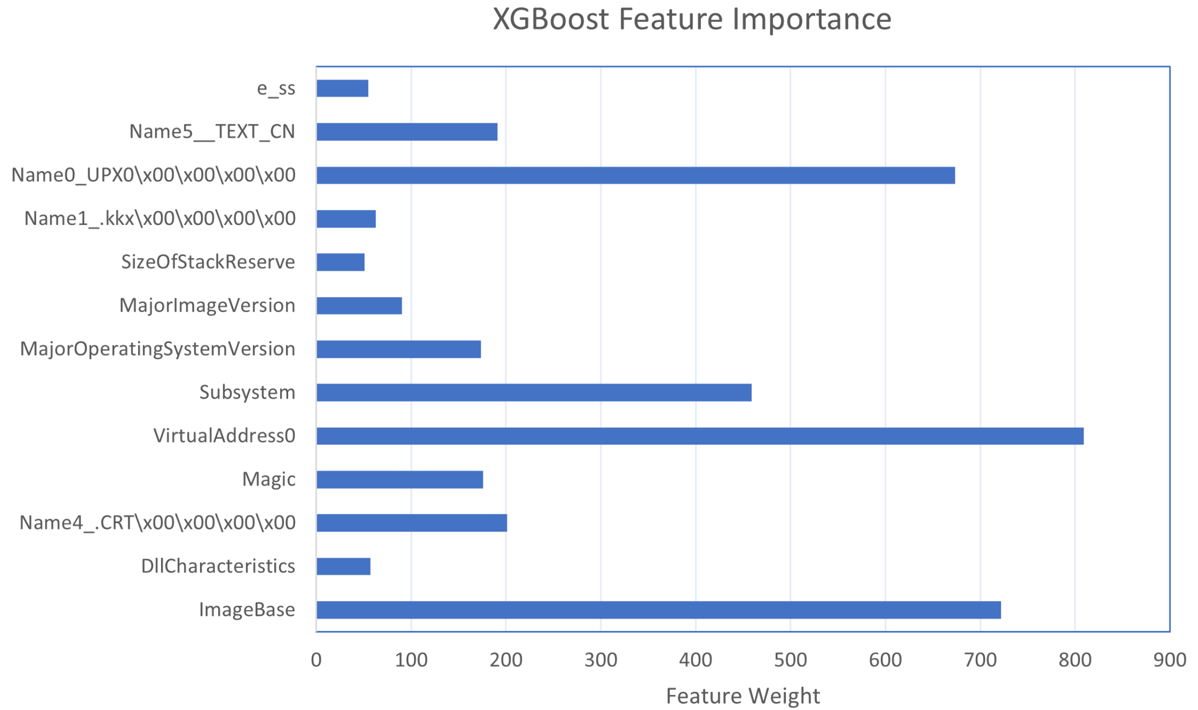


Figure 8: Feature weightings calculated by the XGBoost algorithm. *Name* features refer to PE section titles and ordering.

Table 1: Performance of individual classifiers and ensembles trained on the PE metadata. Recall and precision is with respect to the malware class.

Classifier	Accuracy	Precision	Recall
k-Nearest Neighbours	0.953	0.936	0.959
Stochastic Gradient SVM	0.913	0.891	0.917
Decision Tree	0.918	0.895	0.926
Random Forest	0.971	0.960	0.973
XGBoost	**0.978**	**0.970**	**0.980**
Neural network	0.958	0.945	0.960
AdaBoost	0.921	0.900	0.931
Hard Voting	0.967	0.957	0.968
Stacking Logistic Regression	0.975	0.966	0.978
Stacking Neural Network	0.975	0.965	0.977

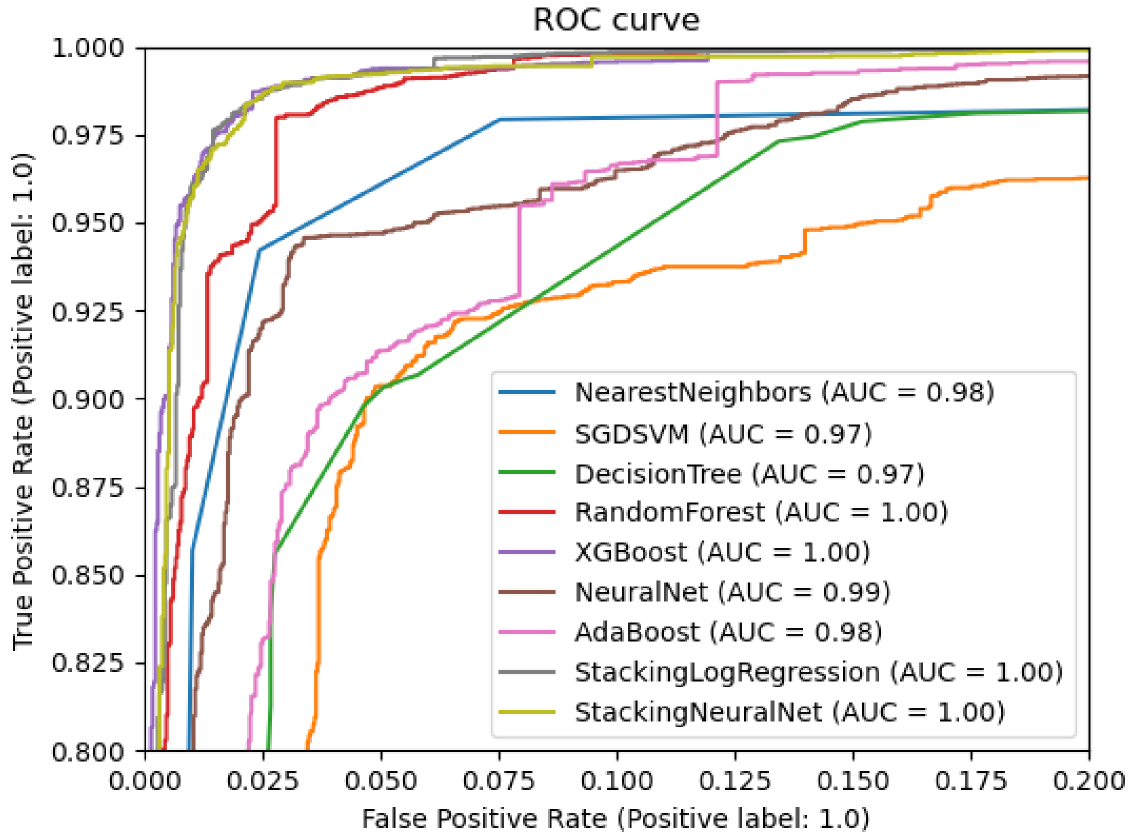


Figure 9: ROC and area under curve scores for PE header models. Positive label: 1.0 = Malware.

5.2 Bytes n-grams

The binary classification results for the models trained on the bytes 2-grams feature set are shown in Table 2, and the ROC curves are shown in Figure 10. These results are obtained for a test dataset of 2 567 benign samples and 6 797 malware samples.

The neural network has the best score for all considered metrics: accuracy, precision, recall, and the area under the ROC curve. This model was therefore chosen for the ensemble.

Table 2: Performance of the classifiers trained on the bytes 2-grams feature set, for a discrimination threshold of 0.5. Recall and precision is with respect to the malware class.

Classifier	Accuracy	Precision	Recall
Decision tree	0.902	0.915	0.953
Boosted decision tree	0.914	0.917	0.970
Random forest	0.944	0.943	0.983
Neural network	0.961	0.962	0.984

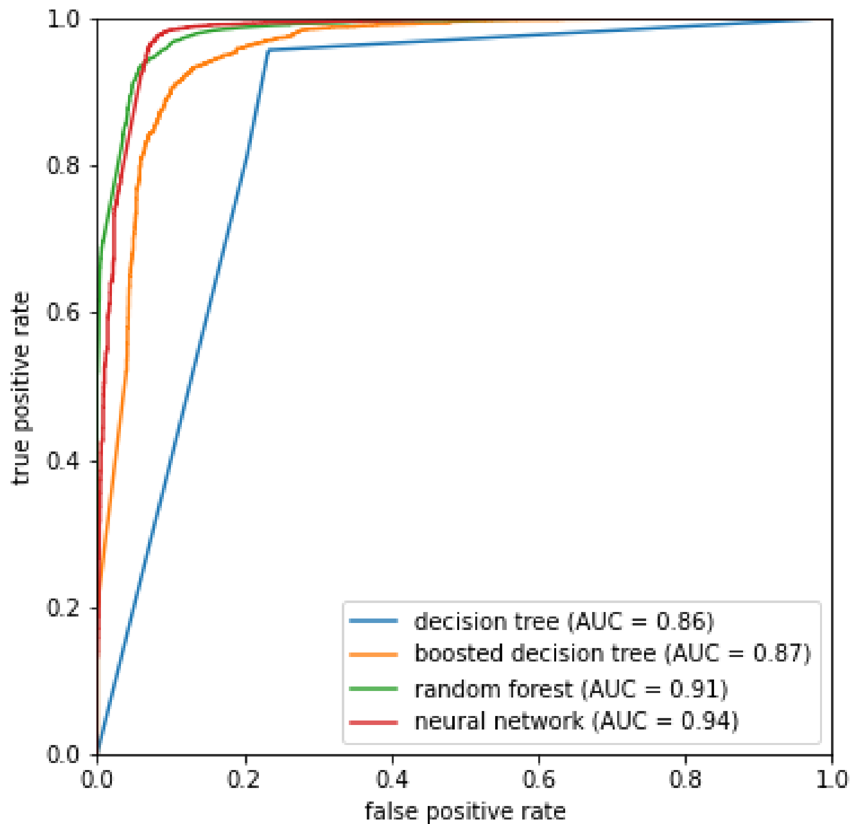


Figure 10: ROC curves and areas under ROC curves for the models trained on the bytes 2-grams feature set. The positive label corresponds to malware.

5.3 Graphs

5.3.1 Binary Classification

The binary classification results for the CG and CFG classifiers can be seen in Table 3. Both classifiers display strong discriminative power and favour recall over precision, likely because of there being more malware samples than benign in the training set. There are no direct comparisons of these results in the literature but the CG performance is inline with other works on static malware classification whilst the CFG slightly falls behind. It is hard to say in general if the CG's better performance here makes it a superior representation of the binaries than the CFGs. There is a lot of scope to improve the feature selection for the CFGs so more work is needed to compare the two types of graph. The overall performance demonstrates that there is potential in representing binaries as graphs for malware classification and that graph neural networks are capable of combining structural and feature information to learn malicious behaviour.

Table 3: Performance of the classifier trained on the call graphs and on the control flow graphs for a discrimination threshold of 0.5. Recall and precision is with respect to the malware class.

Classifier	Accuracy	Precision	Recall
Call Graph	0.954	0.958	0.975
Control Flow Graph	0.921	0.917	0.964

Table 4: Percentage of all API calls made up by some of the categories for the benign test set call graphs separated into correct and incorrect classifications.

Category	Correct Classification	Incorrect Classification
System Services	15.3	13.4
Security and Identity	6.40	6.67
Internationalisation for Win Apps	4.46	5.61
⋮	⋮	⋮
wchar header	2.01	0.31
Display Devices	1.85	3.23
Graphics Device Interface	1.84	3.23
Win Runtime C++ Reference	1.76	0.76
Kernel-Mode Driver	1.71	0.49
string header	1.61	0.26
⋮	⋮	⋮

A helpful property of the DGCNN classifier is that it outputs a probability which is thresholded to make a classification. The value of the probability can be used to inform on the classifier's confidence in its classification. Figure 11 shows the probabilities for correct and incorrect classifications for the CGs. The difference in confidence for the correct and incorrect classifications allows for considerable flexibility between precision and recall when choosing the discrimination threshold.

Interpreting deep learning models is a challenging task. One way to glean some understanding is to look at how different features of the test samples affect the classifier's performance. Table 4 shows some of the API calls of the benign CG test set samples for both correct and incorrect classifications. We see that wchar header, Win Runtime C++ Reference, and string header are all most prevalent in correct classifications. The API calls under these categories all facilitate the use of the C and C++ runtime libraries in the binaries. This seems to be indicative of benign behaviour and it looks like the classifier has learnt to leverage this. Also of note are Graphics Device Interface and Display Devices which are both most prevalent in incorrect classifications. These categories contain API calls associated with displaying pop-ups and opening windows on the user's display. It appears that the classifier has associated these API calls as potentially being malicious. From Figure 2 we know that the dataset contains examples of ransomware and it is also common for other malware families to be adapted for use as ransomware. Therefore, it is possible that the classifier has seen certain sequences of API calls in these categories that draw ransom notes and has learnt this as malicious behaviour. The data from Table 4 provides only some speculative ideas as to what is being leveraged by the CG classifier and ignores the importance of structural information from the graph. A more thorough examination with carefully selected test samples may produce some interesting insights.

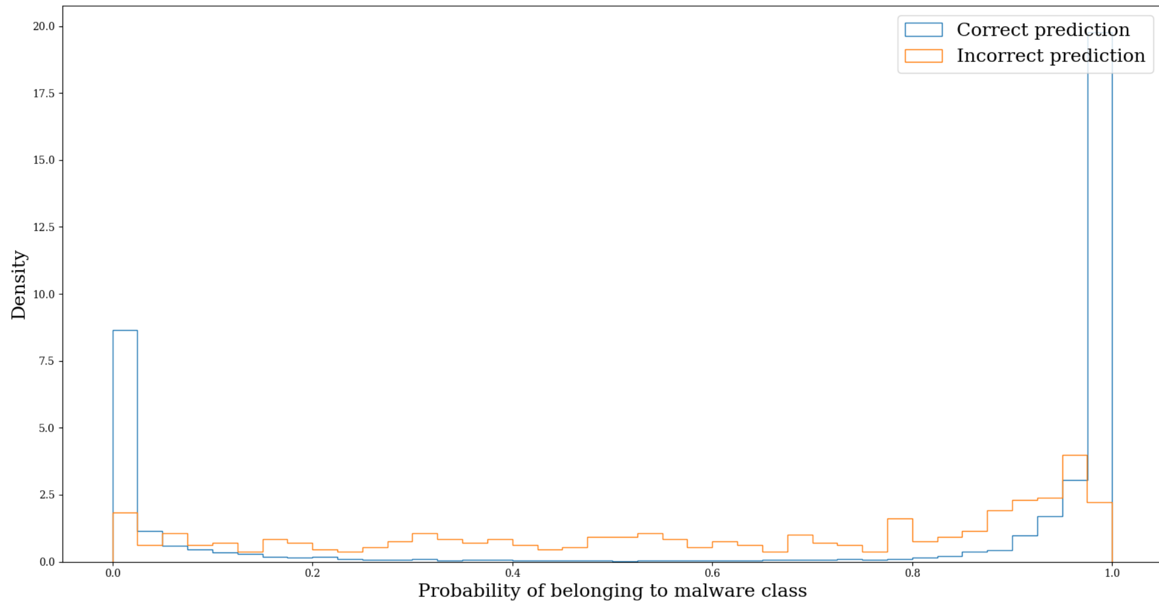


Figure 11: Density histogram of the call graph classifier's output probability that a test sample is malware for incorrect and correct classifications.

5.3.2 Malware Family Classification

The confusion matrix in Figure 12 summarises the performance of the CG classifier on the problem of malware family classification. This corresponds to a macro averaged F-score of 0.688. We see that the classifier achieves perfect recall and near perfect precision on the CRCF, DownloadGuide, and Lamer families. Performance on the other families spans from good to poor, notably the classifier fails to learn any discriminating characteristics of the Ulise family. Some of the incorrect predictions may be expected due to similarities in the families, Zusy and Ursnif are both banking trojans for example. The origins of other behaviours such as the tendency to label Ulise as Razy are mysterious. Excluding the poorly performing Ulise and Zusy families, the results are similar to what is found in [13] where CFGs are used to classify 13 malware families.

Overall, for a 12 class classification the performance is strong. The near perfect classification of some of the families demonstrate the high discriminative power that can be achieved by representing binaries with graphs. It should also be noted that the method of labelling the malware families is fairly crude compared to the hand labelled datasets sometimes used in the literature. This may be of some detriment to the classifier's performance.

Confusion Matrix

True labels \ Predicted labels	Ursnif	CRCF	Ransom	DownloadGuide	Ulise	Emotet	Adware	Razy	Backdoor	Zusy	Lamer	Tepfer
Ursnif	144 76.60%	0 0.00%	0 0.00%	0 0.00%	30 15.96%	3 1.60%	0 0.00%	2 1.06%	6 3.19%	3 1.60%	0 0.00%	0 0.00%
CRCF	0 0.00%	295 100.00%	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 0.00%
Ransom	30 14.22%	3 1.42%	119 56.40%	0 0.00%	3 1.42%	32 15.17%	6 2.84%	9 4.27%	0 0.00%	6 2.84%	2 0.95%	1 0.47%
DownloadGuide	0 0.00%	0 0.00%	0 0.00%	162 100.00%	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 0.00%
Ulise	34 11.41%	12 4.03%	4 1.34%	0 0.00%	30 10.07%	37 12.42%	14 4.70%	158 53.02%	4 1.34%	4 1.34%	1 0.34%	0 0.00%
Emotet	22 1.70%	0 0.00%	36 2.79%	0 0.00%	7 0.54%	1203 93.18%	3 0.23%	12 0.93%	4 0.31%	1 0.08%	1 0.08%	2 0.15%
Adware	13 10.83%	1 0.83%	1 0.83%	1 0.83%	6 5.00%	13 10.83%	63 52.50%	0 0.00%	18 15.00%	4 3.33%	0 0.00%	0 0.00%
Razy	10 2.84%	0 0.00%	20 5.68%	0 0.00%	15 4.26%	22 6.25%	7 1.99%	267 75.85%	0 0.00%	7 1.99%	0 0.00%	4 1.14%
Backdoor	13 10.57%	12 9.76%	5 4.07%	1 0.81%	3 2.44%	15 12.20%	5 4.07%	2 1.63%	54 43.90%	3 2.44%	8 6.50%	2 1.63%
Zusy	35 25.55%	18 13.14%	6 4.38%	0 0.00%	5 3.65%	2 1.46%	9 6.57%	0 0.00%	8 5.84%	46 33.58%	8 5.84%	0 0.00%
Lamer	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 0.00%	0 0.00%	337 100.00%	0 0.00%
Tepfer	0 0.00%	0 0.00%	0 0.00%	0 0.00%	3 1.83%	0 0.00%	1 0.61%	0 0.00%	1 0.61%	0 0.00%	0 0.00%	159 96.95%

Figure 12: Confusion matrix of call graph malware family classifier. Percentages are row-wise whilst the heat map is global.

5.4 Ensemble

The binary classification results for the final ensemble model and the individual models used to create it are shown in Table 5, and the ROC curves are shown in Figure 13. The results for the individual models differ from the ones in Tables 1, 2 and 3 because they were evaluated for a different, smaller dataset, the same one used to evaluate the ensemble, so that the performance can be compared.

The PE header model performs better than the n-grams and CG models in terms of accuracy and precision though not recall. This may be due to the differences in dataset balances and sizes or the smaller preprocessing overhead which allowed for more in depth hyper-parameter optimisation. Alternatively PE headers may simply provide a slightly richer source of features with discriminative power. The ensemble displays further improvement with better accuracy, recall and area under the ROC curve scores. This indicates that the different modes are complementary and reduce each other's weaknesses and biases to produce a more powerful overall classifier.

Table 5: Performance of the ensemble model and the models used to obtained the predictions for the ensemble model, for a discrimination threshold of 0.5. Recall and precision is with respect to the malware class.

Classifier	Accuracy	Precision	Recall
2-grams	0.964	0.964	0.987
call graphs	0.955	0.961	0.977
PE header	0.972	0.992	0.970
Ensemble neural network	0.989	0.991	0.995

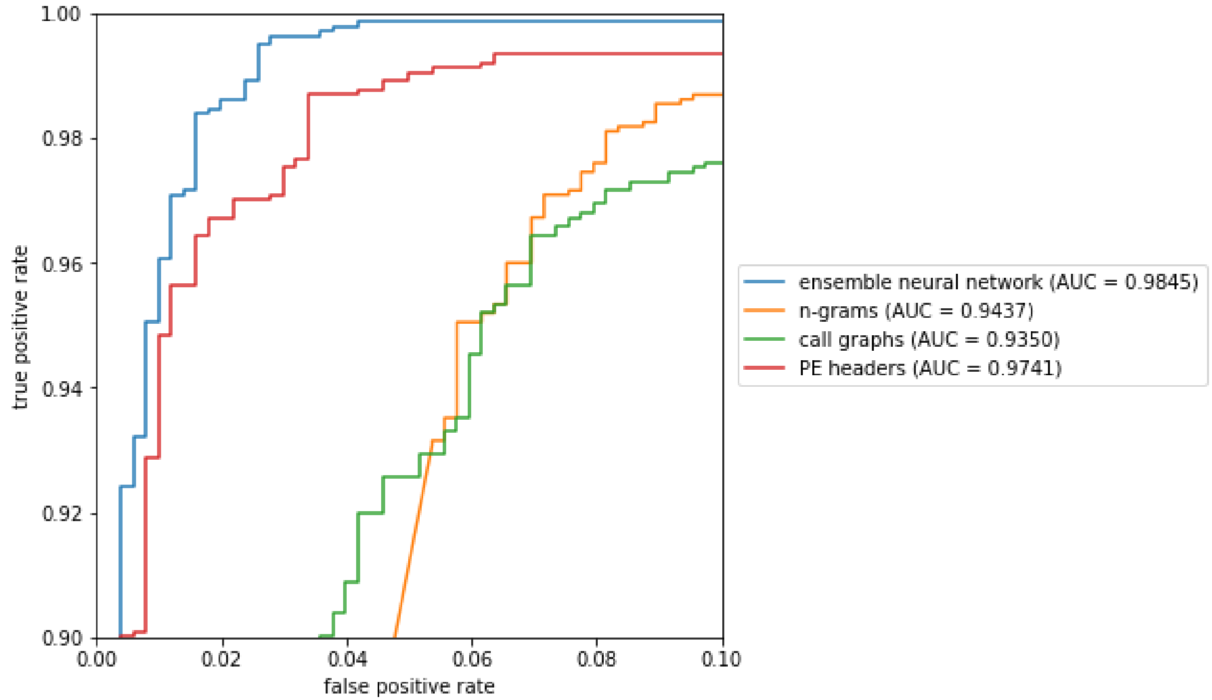


Figure 13: ROC curves for the ensemble model and the individual models used for the ensemble. The positive label corresponds to malware.

6 Conclusion

We trained several machine learning models to identify malicious executables using different features: PE headers, bytes n-grams, control-flow graphs and API call graphs. All models demonstrate very good classification performance, with the PE header model marginally outperforming the others. We created an ensemble model which uses the decision value predictions of the best of these models as input features. The ensemble displays an improved performance over individual models with a classification accuracy of 98.9%. This suggests that multi-modal late fusion is a valid tactic for combining classifiers for effective malware detection at scale.

7 Acknowledgements

We would like to thank NCC Group for the opportunity to perform this research and providing access to their resources. In particular we thank Matt Lewis, Jennifer Fernick, and Mostafa Hassan for their advice and support. We are also grateful to Tim Scanlon and Sebastien Rettie of UCL for their guidance and leadership during the project.

References

[1] PurpleSec. *2021 Cyber Security Statistics*. purplesec.us. URL: https://purplesec.us/resources/cyber-security-statistics/.

[2] S. Ghafur et al. "A retrospective impact analysis of the WannaCry cyberattack on the NHS". In: *npj Digital Medicine* 2.1 (2nd Oct. 2019), pp. 1–7. DOI: 10.1038/s41746-019-0161-6.

[3] Ajit Kumar, K. S. Kuppusamy and G. Aghila. "A learning model to detect maliciousness of portable executable using integrated feature set". In: *Journal of King Saud University - Computer and Information Sciences* 31.2 (1st Apr. 2019), pp. 252–265. DOI: 10.1016/j.jksuci.2017.01.003.

[4] Chronicle Security Ireland Limited. *VirusTotal*. URL: https://www.virustotal.com. Accessed: 01.02.2021.

[5] Corvus Forensics. *VirusShare*. URL: https://virusshare.com. Accessed: 01.02.2021.

[6] NCC-UCL Github. *extract.py*. URL: https://github.com/sebastien-rettie/cdt-nccgroup/blob/main/feature_extraction/pe/extract-pe.py.

[7] Patricio Cerda, Gaël Varoquaux and Balazs Kegl. "Similarity encoding for learning with dirty categorical variables". In: *Machine Learning* 107 (1st Sept. 2018). DOI: 10.1007/s10994-018-5724-2.

[8] Andrii Shalaginov et al. "Machine Learning Aided Static Malware Analysis: A Survey and Tutorial". In: *Advances in Information Security*. 3rd Aug. 2018, pp. 7–45. DOI: 10.1007/978-3-319-73951-9_2.

[9] Jeremy Z. Kolter and Marcus A. Maloof. "Learning to Detect Malicious Executables in the Wild". In: *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD '04. Seattle, WA, USA: Association for Computing Machinery, 2004, pp. 470–478. DOI: 10.1145/1014052.1014105.

[10] Na Huang et al. "Deep Android Malware Classification with API-Based Feature Graph". In: *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. 2019, pp. 296–303.

[11] Haodi Jiang, Turki Turki and Jason T. L. Wang. "DLGraph: Malware Detection Using Deep Learning and Graph Embedding". In: *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*. 2018, pp. 1029–1033.

[12] Shen Wang et al. "Heterogeneous Graph Matching Networks for Unknown Malware Detection". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, July 2019, pp. 3762–3770.

[13] Jiaqi Yan, Guanhua Yan and Dong Jin. "Classifying Malware Represented as Control Flow Graphs using Deep Graph Convolutional Neural Network". In: *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 2019, pp. 52–63. DOI: 10.1109/DSN.2019.00020.

[14] Yan Shoshitaishvili et al. "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis". In: *IEEE Symposium on Security and Privacy*. 2016.

[15] Muhan Zhang et al. *An End-to-End Deep Learning Architecture for Graph Classification.* 2018.

[16] Daniel Gibert, Carles Mateu and Jordi Planes. “The rise of machine learning for detection and classification of malware: Research developments, trends and challenges”. In: *Journal of Network and Computer Applications* (1st Mar. 2020). DOI: 10.1016/j.jnca.2019.102526.

[17] Mansour Ahmadi et al. “Novel Feature Extraction, Selection and Fusion for Effective Malware Family Classification”. In: 9th Mar. 2016. DOI: 10.1145/2857705.2857713.

[18] Sanjay Katkar. *Virus Bulletin :: Inside the PE file format.* URL: https://www.virusbulletin.com/virusbulletin/2006/06/inside-pe-file-format/ (visited on 10/05/2021).