Using Text Classification Methods to Detect Malware

Eoghan Cunningham[1], Oisín Boydell[1], Cormac Doherty[2], Benjamin Roques[2], and Quan Le[1]

[1] Centre for Applied Data Analytics Research (CeADAR), University College Dublin, Ireland
eoghan.cunningham@ucdconnect.ie, {oisin.boydell, quan.le}@ucd.ie
[2] Centre for Cybersecurity & Cybercrime Investigation (CCI), University College Dublin
{cormac.doherty, benjamin.roques}@ucd.ie

Abstract. In this paper we propose using text classification to detect malware. In our approach, we convert each binary executable to an assembly program, then use text analytics to classify whether the code is malicious or not. Using random forests as the classification model we achieve an F1 accuracy of 86%. Furthermore, to achieve this performance we only examined a limited portion of each assembly program. Our findings allow the development of malware detectors with fast responses as traditionally malware detectors need to parse the whole binary before making the decision. It also opens up the possibilities of using complex classification models like deep learning to detect malicious programs through analyzing code semantic.

Keywords: machine learning, text classification, cybersecurity, malware, reverse engineering

1 Introduction

Malware - software written with malicious intent - is a rapidly growing problem in cyber-security. In 2018 alone Symantec reported 246 million new malware variants [18]. The scale of this threat means that there is an urgent need to develop accurate as well as efficient classification tools for automatically detecting malware.

Most current methods for malware detection use signature based approaches to identify malware already described by human analysts or by behaviour analysis through generic heuristics detection [2]. For example, YARA rules [6] are a set of pattern matching rules, similar to regular expressions, designed to help malware researchers to identify and classify malware. They are constructed by malware analysts to match user defined patterns (binary or text) inside malware samples. When a new sample is discovered, if the sample matches an already existing YARA rule, then it can be classified to the corresponding malware family. However, these approaches are time consuming, require a high level of domain expertise, and can only be developed when the malware has already been identified and flagged; as such there is a need to develop tools which can

learn to detect new malware and malware variants automatically. This has stimulated research into applying machine learning to the malware detection task.

In this work we investigate the use of text classification methods to detect malware. We convert each input example, in the form of executable machine code in raw binary format, into an assembly program i.e. a representation of the underlying sequence of processor instructions. This static feature extraction is computationally efficient since it does not require running the executable, and the assembly code is a more amenable representation of an executable's behaviour as compared to other static analysis methods such as working with raw byte code sequences [14]. We aim to investigate the use of a portion of code instead of the whole assembly program to build malware detectors, as using only a portion of the program as input will allow the development of fast malware detectors, besides other benefits. Our contribution in this work is the analysis of how the performance of text classifiers vary with the number of assembly code instructions used to represent each binary. We find that the performance of classifiers trained in this way is relatively stable across a wide range of instruction sequence lengths, which shows that it is possible to develop very fast malware detection tools with high accuracy using only a small input data size.

2 Related Work

In this section we review literature on malware classification using disassembled executable code. For these works, a disassembler like IDAPro [3], Radare2 [4], or Ghidra [1] is used to convert binary executable code into a sequence of assembly instructions - each instruction includes an opcode (an assembly command) and the possible parameters that the command operates on. The next step is to convert the sequence of instructions into a sequence of tokens from a vocabulary - each instruction could be considered as a token, or the researchers may choose to keep only the opcode of each instruction as a token to reduce the token vocabulary size. Further feature extraction methods can then be applied on each sequence of tokens before being used as input to train a malware classifier.

The first group of works use n-gram (sequences of n consecutive tokens) statistics from the assembly code as input features for a classifier. In [20], the authors calculated the frequencies of bi-grams of opcodes, and then applied feature selection to keep only the 1,000 most important bi-gram opcodes sequences as input. They trained a number of popular classifiers on their dataset of 1,000 malware and 1,000 benign example binaries; and reported an accuracy of more than 90%. In [22] the authors studied different tokenizing approaches applied to the assembly code before extracting n-gram features to train a linear classifier. The author reported low accuracy of up to 77%.

The second group of works use generative models - e.g. Hidden Markov Models (HMMs) - to model the relationships over sequences of instructions. In [17], the authors trained a HMM for each of multiple malware families, then a test malware sample would be assigned to the family whose HMM gave it the highest score. In [15] the authors use HMMs to detect metamorphic viruses - malware which evades detection by inserting junk codes in its body.

Recently Deep Learning [10] has been emerging as a new classification model which can be applied across many diverse problem domains – including malware classification – due to its ability to learn efficient task based feature extraction given a dataset of input-label pairs. In [16] a convolutional neural network (CNN) on top of an opcode embedding layer was trained on opcode sequences to detect whether an android app

was malware or not, and it achieved a performance of 87% accuracy and 86% macro average F1 score on their large dataset configuration. In [13], a CNN was trained on one-hot encoded opcode sequences to classify malware into one of 9 families, with the neural network obtaining 91% F1 accuracy. [12] also train a CNN on opcode sequences through an opcode embedding layer to detect malware. Due to file size restriction the author split each assembly program into chunks of 10,000 opcode sequences and trained them independently; the performance of the neural network was not reported independently but as a component of an ensemble of classifiers.

In reviewing the literature, we notice that there is a lack of analysis of the effects of choosing different numbers of instructions from the assembly program on the performance of the classifiers. Another observation is that complex models like neural networks cannot easily process large assembly programs in whole due to memory and computing power constraints, which informs our work on using a subset of the first n instructions from each sample.

3 Methods

3.1 Dataset

There is a lack of standardised datasets used across the malware research community, not least in part due to the issue of distributing known malicious executables as well as the rapidly evolving nature of malware whereby older samples may not be representative of malware currently in the wild. Therefore we compiled our own datasets for our research from several sources. We used 5,000 unique malicious Microsoft Windows portable executable files (PE Win32) provided by UCD's Centre for Cybersecurity and Cybercrime Investigation (CCI). This collection was compiled over an eighteen month period from publicly available sources of threat intelligence, predominantly malicious URL feeds and phishing domains. These malware samples were complemented with over 5,000 benign (non-malicious) unique samples collected from online freeware sources, Windows ISO disk images and personal computers. In total, the dataset size is 17 Gbytes.

In order to evaluate the generality of our malware classification approach, we created an additional test dataset of 2,000 unique malware examples from an alternative source, VirusTotal [5], so as to account for any potential bias in the CCI sources. Again this was complemented with 2,000 additional unique benign samples collected from online freeware sources, Windows ISO disk images and personal computers.

3.2 Feature Extraction

Static Analysis is the term used for analysis of executable files, namely malware, that does not require their execution. Many statically extracted features have been shown to have predictive power in malware detection [11], [7], however the focus of our work is on using assembly instructions as produced from running a disassembler over each executable. We used the same disassembly software, Radare2, across all the excecutables in our datasets for consistency.

Function Resolution Given the predictive power of API calls [11] and motivated by the work in [22], we chose a feature representation for the assembly instructions that, where possible, resolved any memory addresses to the function name being called, according to the function import table. Any addresses that were not resolved to their function names are replaced with a fixed, unique string - 'adr' (Figure 1). After pre-processing, we tokenize the assembly code at the instruction level.

```
.
mov esi section.UPX1
lea edi [esi - adr]
push edi
mov ebp esp
lea ebx [esp - adr]
xor eax eax
push eax
cmp esp ebx
jne adr
inc esi
inc esi
push ebx
push adr
```

Fig. 1. An example of assembly code after pre-processing

Feature Representation We then considered the task of detecting malware executables as a binary text classification problem. We applied the *term frequency - inverse document frequency* (TF-IDF) metric from the field of information retrieval to represent the sequences of disassembled instructions as vectors, where the terms are represented by n-grams of assembly instructions. This prompted an analysis of the influence of n-gram length on vector dimensionality (vocabulary size) - a very high dimensional feature space is likely to lead to low generalization performance due to the curse of dimensionality [8]. Figure 2 shows the dimensionality of the feature vectors for various n-gram lengths. We chose to use n=2 (bi-grams) as this provides a balance between a compact vocabulary size and the ability to capture consecutive instructions. It is also the choice used in multiple research in malware classification [21][20].

3.3 Experiment Protocol

We carried out our experiments in two stages, using the dataset of 10,000 samples (5,000 maleware from CCI, 5,000 benign) as described above. In all experiments classifier performance was measured using macro averaged F1 score.

In the first stage we wanted to identify which classification algorithm performed best for a chosen number of instructions to be disassembled of 1000, using the feature extraction and representation approach described above. We considered six common

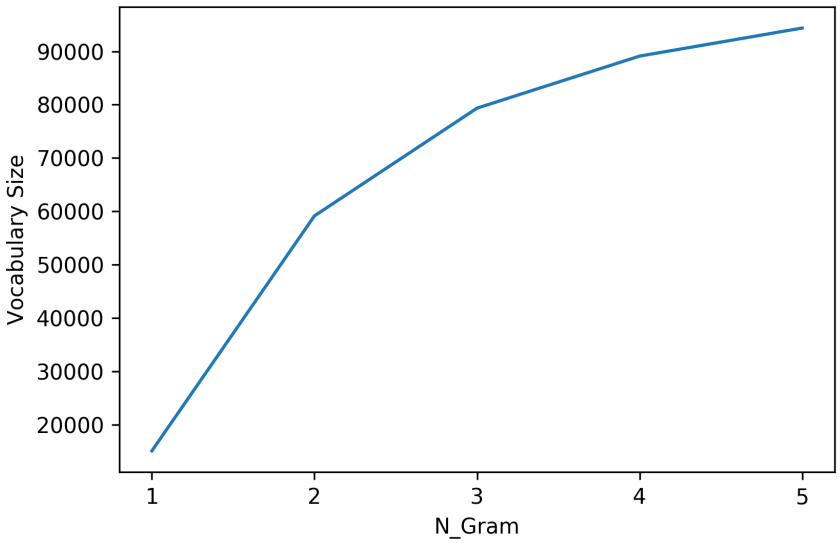

Fig. 2. Dimensionality of feature vectors by n-gram length

classifiers for our experiments: Logistic Regression, Naive Bayes, Random Forest, K Nearest Neighbors (KNN), Linear Support Vector Machines (Linear SVM), and XGBoost. We used the XGBoost classifier implemented in [9], and for other classifiers we used the library scikit-learn [19]; the hyper-parameters for all classifiers were set to default values (version 0.90 for XGBoost, and version 0.22 for scikit-learn). The dataset was randomly split into training and test sets with a ratio of 75%:25%, and we individually trained and evaluated all six classifiers. This was repeated five times and the F1 score of each repeat for each classifier was averaged.

In the second stage we analyzed how the performance of the best performing classifier from stage one varied when we used different numbers of instructions from the disassembled executables as input. For a chosen number of instructions n_inst, the classifier was only trained (and tested) with the first n_inst disassembled instructions from each executable. We tested various values of n_inst between 250 and 20,000, and used the same evaluation methodology as in stage one.

Finally, we wanted to understand how our classification approach generalises to a previously unseen malware situation. For this we trained the best performing classifier identified in stage one using the number of instructions, n_inst, which gave the best performance in the second stage on the full dataset of 5,000 malware examples from CCI and the 5,000 benign examples. We then tested the performance of our model on a test dataset comprising 2,000 malware examples from VirusTotal, along with 2,000 new benign examples.

4 Results

Table 1 shows the performance of the six classifiers when we used the first 1000 instructions of each assembly program to calculate its features, where random forests consistently achieved the best performance. As the result we chose random forests as the classifier for our next experiments.

	Run 1	Run 2	Run 3	Run 4	Run 5	Average F1
Logistic Regression	0.75	0.74	0.75	0.76	0.74	0.748
Naive Bayes	0.79	0.78	0.79	0.80	0.79	0.79
Random Forest	**0.86**	**0.84**	**0.84**	**0.85**	**0.84**	**0.846**
KNN	0.82	0.83	0.82	0.83	0.82	0.824
Linear SVM	0.80	0.78	0.79	0.80	0.79	0.792
XGBoost	0.81	0.80	0.82	0.81	0.80	0.808

Table 1. F1 score of the six classifiers, for each of the five random training/test splits, and the overall averaged F1 score for each (number of instructions = 1,000)

In the second experiment, we varied the number of instructions used in an assembly program in the series of values [250, 500, 1000, 2000, 3000, 4000, 5000, 7500, 10000, 15000, 20000]. The performance of the random forests for the chosen numbers of instructions is visualized in Figure 3. Random forests achieved the lowest average F1 score of 84.7% (84.5 % accuracy) with 1000 instructions, and it achieved the highest average F1 score of 86.4 % (86.7 % accuracy) with 7500 instructions.

As a further analysis on random forest classifier trained by assembly length of $7,500$ instructions, Figure 4 shows the time to train a random forest on a chosen number of instructions. We carried out our experiments on using 8 cores on an Intel Xeon E5 v3 processor with 53 GB memory.

For the final test, we trained a random forest on the whole dataset and tested it on the test set with malwares provided by VirusTotal. Our trained random forest achieved a performance of 81% F1 score, this showed our random forest model generalized to malware pieces from a different source.

Discussion

In the experiments our random forest classifier achieved 86.4 % macro-average F1 accuracy on a separate validation set, and 81% on a test set for the difficult scenario where malware executables come from a separate source (VirusTotal). This compares favourably with accuracy reported in other works, albeit on different datasets. However, our finding that good performance can be achieved by looking at only the first 7,500 instructions of the disassembled code is important. Reducing the quantity of data that needs to be disassembled and processed allows for the development of faster malware detectors than if the whole executable needs to be taken into account. We also believe that our analysis of model performance vs assembly length is useful in allowing developers to trade off model accuracy vs model speed for a specific use case:

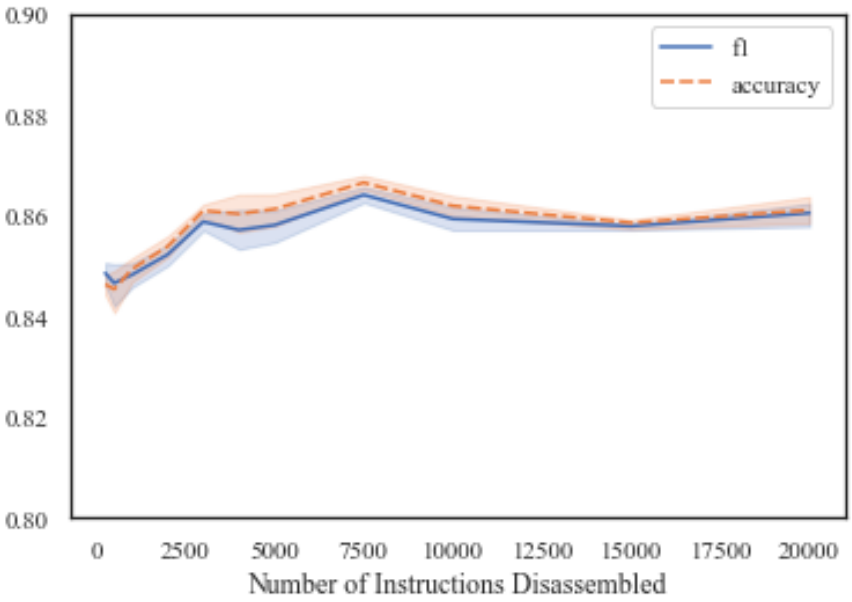

Fig. 3. Random Forest Classifier Performance by Assembly Length

for example, a malware detector scanning email might need to be very fast, whilst a detector protecting a high security network might trade speed for accuracy.

We believe that our approach of preprocessing assembly instructions is useful as it reduces the vocabulary size of tokens while preserving more information than using opcodes alone. However an important avenue of future work would be to explore preprocessing techniques further to reduce the vocabulary size of the tokens whilst keeping as much relevant information as possible, and whether our approach of selecting instructions from the start of the malware binary as opposed to other segments of the file is optimal.

5 Conclusion

In this paper we applied text classification to detect malicious programs, and we analyzed how the performance of a malware detector changed as a function of the number of instructions in the assembled code that it looked at. Our finding that a classification model only needs to look at a small portion of the disassembled code to detect malware opens up the possibilities to develop complex and powerful malware detection models on disassembled code. It also allows the development of fast malware detectors, which is necessary given the need to deal with a rapidly increasing number of malware variants today.

Acknowledgement. This research was supported by a research funding grant from Enterprise Ireland to CeADAR.

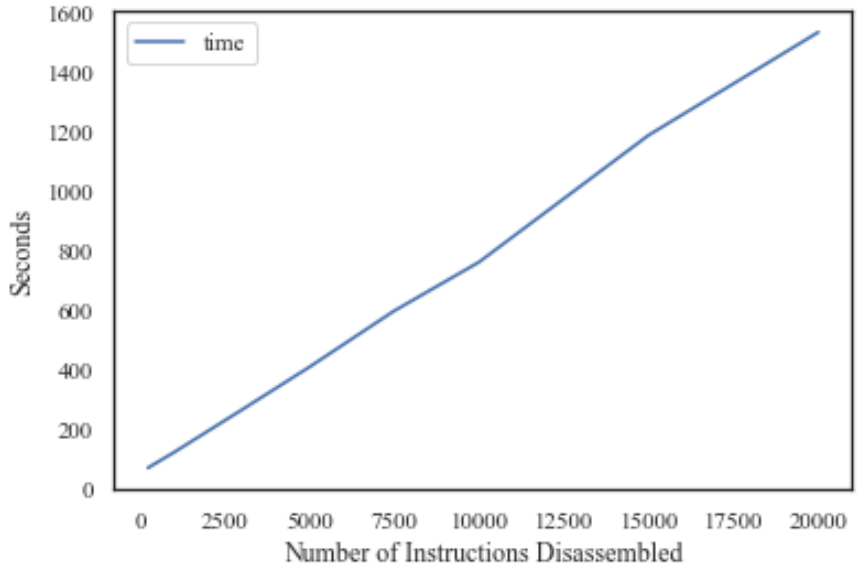

Fig. 4. Time to Train a Random Forest Classifier by Assembly Length

References

1. Ghidra - a software reverse engineering (sre) suite of tools developed by nsa's research directorate in support of the cybersecurity mission. `https://ghidra-sre.org/`, accessed: 2019-09-30
2. Heuristic and proactive detections. `https://encyclopedia.kaspersky.com/knowledge/heuristic-and-proactive-detections`, accessed: 2019-09-30
3. Ida pro combines an interactive, programmable, multi-processor disassembler coupled to a local and remote debugger and augmented by a complete plugin programming environment. `https://www.hex-rays.com/products/ida/`, accessed: 2019-09-30
4. Radare2 - a free toolchain for easing several low level tasks like forensics, software reverse engineering, exploiting, debugging. `https://rada.re/n/radare2.html`, accessed: 2019-09-30
5. Virustotal - free online virus, malware and url scanner. `https://www.virustotal.com/en`, accessed:2019-09-30
6. The yara project. `https://virustotal.github.io/yara/`, accessed: 2019-09-30
7. Biondi, F., Given-Wilson, T., Legay, A., Puodzius, C., Quilbeuf, J.: Tutorial: an overview of malware detection and evasion techniques. In: International Symposium on Leveraging Applications of Formal Methods. pp. 565–586. Springer (2018)
8. Bishop, C.M.: Pattern recognition and machine learning. Springer (2006)
9. Chen, T., Guestrin, C.: Xgboost: A scalable tree boosting system. In: Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining. pp. 785–794. ACM (2016)
10. Goodfellow, I., Bengio, Y., Courville, A.: Deep learning. MIT press (2016)

11. Ki, Y., Kim, E., Kim, H.K.: A novel approach to detect malware based on api call sequence analysis. International Journal of Distributed Sensor Networks 11(6), 659101 (2015)
12. Kim, D.: Improving Existing Static and Dynamic Malware Detection Techniques with Instruction-level Behavior. Ph.D. thesis, University of Maryland (2019)
13. Kolosnjaji, B., Eraisha, G., Webster, G., Zarras, A., Eckert, C.: Empowering convolutional networks for malware classification and analysis. In: 2017 International Joint Conference on Neural Networks (IJCNN). pp. 3838–3845. IEEE (2017)
14. Le, Q., Boydell, O., Mac Namee, B., Scanlon, M.: Deep learning at the shallow end: Malware classification for non-domain experts. Digital Investigation 26, S118–S126 (2018)
15. Lin, D., Stamp, M.: Hunting for undetectable metamorphic viruses. Journal in computer virology 7(3), 201–214 (2011)
16. McLaughlin, N., Martinez del Rincon, J., Kang, B., Yerima, S., Miller, P., Sezer, S., Safaei, Y., Trickel, E., Zhao, Z., Doupé, A., et al.: Deep android malware detection. In: Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy. pp. 301–308. ACM (2017)
17. Narra, U., Di Troia, F., Corrado, V.A., Austin, T.H., Stamp, M.: Clustering versus svm for malware detection. Journal of Computer Virology and Hacking Techniques 12(4), 213–224 (2016)
18. O'Gorman, e.a.: Symantec internet security threat report for 2019. Volume XXIV (2019)
19. Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., et al.: Scikit-learn: Machine learning in python. Journal of machine learning research 12(Oct), 2825–2830 (2011)
20. Santos, I., Brezo, F., Ugarte-Pedrero, X., Bringas, P.G.: Opcode sequences as representation of executables for data-mining-based unknown malware detection. Information Sciences 231, 64–82 (2013)
21. Shabtai, A., Moskovitch, R., Feher, C., Dolev, S., Elovici, Y.: Detecting unknown malicious code by applying classification techniques on opcode patterns. Security Informatics 1(1), 1 (2012)
22. Zak, R., Raff, E., Nicholas, C.: What can n-grams learn for malware detection? In: 2017 12th International Conference on Malicious and Unwanted Software (MALWARE). pp. 109–118. IEEE (2017)