

# Malware Classification Using Static Disassembly and Machine Learning

Zhenshuo Chen<sup>1</sup>[0000-0003-2091-4160], Eoin Brophy<sup>1</sup>[0000-0002-6486-5746], and Tomas Ward<sup>1,2</sup>[0000-0002-6173-6607]

<sup>1</sup> Dublin City University, Dublin, Ireland

<sup>2</sup> Insight Centre for Data Analytics, Dublin, Ireland

**Abstract.** Network and system security are incredibly critical issues now. Due to the rapid proliferation of malware, the traditional analysis methods struggle with enormous samples. In this paper, we propose four small-scale and easy-to-extract features, including sizes and permissions of PE sections, content complexity, and import libraries, to classify malware families, and use automatic machine learning to search for the best model and hyper-parameters for each feature and their combinations. Compared with detailed behavior-related features like API sequences, proposed features provide macroscopic information about malware. The analysis is based on static disassembly scripts and hexadecimal machine code. Unlike dynamic behavior analysis, static analysis is resource-efficient and offers complete code coverage, but is vulnerable to code obfuscation and encryption. The results demonstrate that features which work well in dynamic analysis are not necessarily effective when applied to static analysis. For instance, API 4-grams only achieve 57.96% accuracy and involve a relatively high dimensional feature set (5000 dimensions). In contrast, the novel proposed features together with a classical machine learning algorithm (Random Forest) presents very good accuracy at 99.40% and the feature vector is of much smaller dimension (40 dimensions). We demonstrate the effectiveness of this approach through integration in IDA Pro, which also facilitates the collection of new training samples and subsequent model retraining.

**Keywords:** Malware Classification · Reverse Engineering · Machine Learning · System Security.

## 1 INTRODUCTION

Network and system security are incredibly critical issues at the moment. According to [11], 142 million threats were being blocked every day in 2019. Furthermore, new types of malware are appearing all the time and are increasingly aggressive. For instance, the use of malicious PowerShell scripts increased by 1000% in the same year. To make matters worse, anti-anti-virus techniques used by attackers are also steadily improving. The use of polymorphic engines allows malware developers to mutate existing code while retaining the original functions unchanged. This is achieved, for example, through the use of obfuscation

and encryption. Code obfuscation uses needlessly roundabout expressions and data to make source or machine code challenging for humans to understand. A simple way is to use roundabout expressions. Attackers can also use jump instructions to make the real execution flow different from the disassembly script. Code encryption packs and encrypts executable files on the disk. They will decrypt themselves during execution. It means they are nearly impossible to analyze just by static disassembly relying instead on execution and reviewing of system logs. These anti-anti-virus techniques have now led to a rapid proliferation of malware which traditional analysis methods struggle to cope with as these rely on signature matching and heuristic rules. They have the same drawback: unseen samples must be manually analyzed before creating signatures or heuristic rules. However, analysts cannot review each unknown file in practice. Machine learning approaches in contrast do not rely on understanding code and malicious behaviors. After training with a wide range of known samples, such methods can more easily identify potential malware compared to human experts. Some automatic models have been applied in related fields, such as malware homology analysis by dynamic fingerprints in [12], and gray-scale image representation of malware in [7], which did not require disassembly or code execution.

We adopt a machine learning approach in this work. The primary exploration and experiments of this paper are as follows:

- The API  $n$ -gram, an efficient dynamic behavior feature usually generated by system event logging, is applied to static analysis. The result demonstrates that actual API sequences are hard to extract from disassembly scripts. Inaccurate  $n$ -grams have a substantial negative impact on classification.
- A simpler variant of the import library feature is proposed, described in Section. 3.4. It uses One-Hot Encoding to indicate whether a library is imported by malware. Compared with the number of APIs imported by each library, this variant is easier to extract and more reliable. Because hiding import libraries is more difficult than hiding API calls. However, this feature does not provide as much detailed information as API calls.
- Three new small-scale features are proposed: section sizes, section permissions, and content complexity. The feature descriptions are in Section. 3.5, Section. 3.6, and Section. 3.7 respectively. They can provide macroscopic information about malware.
- Automatic machine learning is used to find the best model and hyper-parameters for each feature and its combination.
- A method of using the classifier in practice is proposed. With the help of IDA Pro<sup>3</sup>, the most popular reverse analysis tool, new training data can be generated from the latest known malware. It also provides a Python development kit and using this the classifier proposed here is implemented as an IDA Pro plug-in. This allows an analyst using IDA Pro to process a malware sample and perform classification immediately within their workflow.

---

<sup>3</sup> <https://hex-rays.com/products/ida/support/idadoc/index.shtml>

## 2 RELATED WORK

In this section, we primarily discuss malware analysis combined with machine learning and hand-designed static features. Since their processes can be explained in terms of these underlying, interpretable features, analysts can undertake deeper exploration according to the model output. We also examine the small number of deep learning models which have been applied to the problem. These utilize image and byte representations to achieve their goals.

In [12], Zheng Rongfeng et al. used strings, registry changes and API sequences to distinguish whether a new sample was a variant of a known sample. One shortcoming of this method was that accurate registry changes and API sequences must be recorded at execution, which requires costly virtual machines. And due to the lack of enforcement conditions, not all malicious behaviors can be recorded. Another serious issue is that malware can hide signatures through polymorphic engines and packers with the purpose of being harder for anti-virus software to detect. In [6], Maleki Nahid et al. proposed a binary classification system for packed samples. They unpacked files and extracted PE Headers [5], then used the forward selection method to pick seven features. Unlike the previous, their model did not completely rely on detailed assembly instructions and system behaviors, but introduced macroscopic information, such as the number of executable sections and the debug information flag. However, it was unsuitable for files that could not be unpacked.

In the face of these problems which are difficult to solve by fingerprinting, Nataraj et al. proposed an innovative method in [7]. They converted a malware sample's byte content into a gray-scale image. The images belonging to the same family appear similar in layout and texture. They used  $K$ -Nearest Neighbors algorithm to determine whether the samples were derived from the same origin. Neither disassembly nor code execution was required, and simple transformations by polymorphic engines usually do not affect the general image layout. As a limitation, image representation has a problem that two malware images can be similar even if they belong to different families, because the same visual resources are used among these samples, like icons and user interface components. And in [2], Gibert et al. thought image representation might produce non-existing spatial correlations between pixels in different rows.

In recent years, some deep learning models have also been applied in the field of malware classification. In [3], Kalash et al. combined Convolutional Neural Networks with gray-scale image representation. Their model architecture was based on VGG-16 [10] and achieved 99.97% accuracy, which is the best result we have found to date. In [8], Raff et al. built a Convolutional Neural Network and used all bytes of a sample as raw data. But instead of training the network on raw bytes, they inserted an embedding layer to map each byte to a fixed-length feature vector. It could reduce incorrect correlations between two bytes. In other words, the certain bytes are closer to each other than other values, which is incorrect in terms of the assembly instruction context. And using Convolutional Neural Networks with a global max-pooling could increase the robustness when facing minor alterations in bytes. In contrast, traditional byte  $n$ -gram methods

are dependent on exact matches. For deep learning models with byte-based representation, Gibert et al. thought the main advantage of such an approach is that it can be applied to samples from different systems and hardware, because they are not affected by file formats [2]. However, the size of byte sequences is too large and the meaning of each byte is context-dependent. Byte-based representation does not contain this information. Another challenge is that adjacent bytes are not always correlated because of jumps and function calls.

Apart from these models, Gibert et al. mentioned several challenges in the face of malware analysis [2]. One of them is *Concept Drift*. In many other machine learning applications like digit classification, the mapping learned from historical data will be valid for new data in the future, and the relationship between input and output does not change over time. But for malware, due to function updates, code obfuscation and bug fixes, the similarity between previous and future versions will degrade slowly over time, decaying the detection accuracy. Furthermore, the interpretation of models and features should also be considered. When an incorrect classification happens, analysts need to understand why and know how to fix it. This is challenging without clear interpretability and explainability. Even in the absence of miss-classifications, analysts prefer to understand how a classification has been arrived at. This is the main reason why we did not choose a deep learning model.

### 3 FEATURE EXTRACTION

The dataset used in the paper is from the 2015 Microsoft Malware Classification Challenge [9]. It contains 10868 malware samples representing a mix of nine families. Each sample has two files of different forms: machine code and disassembly script generated by IDA Pro. In practice, malware is in the form of executable files. However for safety reasons, Microsoft does not provide raw files but only processed machine code and disassembly scripts. Without executable files, dynamic analysis cannot be conducted. All features can only be extracted from static text. The features described in Section. 3.4, Section. 3.5, Section. 3.6 and Section. 3.7 are proposed by us. API  $n$ -grams in Section. 3.2 is an effective feature in dynamic behavior analysis. We tested it to check if it is also applicable for static disassembly analysis.

#### 3.1 File Size

The file size is the simplest feature, containing the sizes of disassembly and machine code files, and their ratios. File sizes vary according to the functional complexity of different malware families. And size ratios may represent code encryption. If a sample is encrypted, disassembly may fail and the ratio of its machine code size to the disassembly size will be different from the other samples.

### 3.2 API 4-gram

The API sequence is almost the most commonly used feature. It directly uses malicious or suspicious API sequences to classify malware. Each malware family has distinct functions. For instance, `Lollipop` is Adware showing advertisements as users browse websites. `Ramnit` can disable the system firewall and steal sensitive information. These functions rely on different APIs.

API sequences should be extracted by dynamic execution since it can reduce the negative impact of code encryption. However, because of the dataset limitation, we can only use regular expressions to match `call` and `jmp` instructions whose target is an import API from static disassembly scripts. It has a huge negative impact on accuracy and we will discuss this in detail in a later section. Finally, 402972 API 4-grams were extracted and only the 5000 most frequent items were retained.

### 3.3 Opcode 4-gram

The opcode sequence is also commonly used. It focuses on disassembly instructions. Opcodes are defined by CPU architectures, not by systems as in the case of APIs. So they are compatible with different systems built on the same architecture. 1408515 opcode 4-grams were extracted and only the 5000 most frequent items were retained.

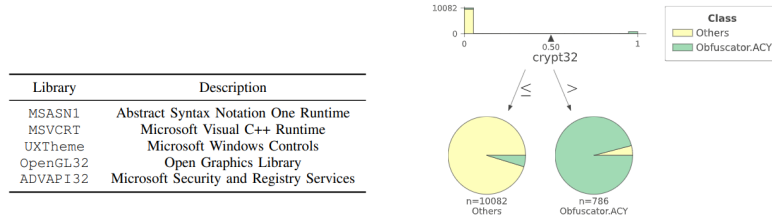
### 3.4 Import Library

As mentioned in Section. 3.2, each malware family has distinct functions. They must import system or third-party libraries to achieve. So a typical machine learning feature is the number of APIs per imported library used by malware. But API numbers would be inaccurate if malware calls an API with dynamic methods such as by `GetProcAddress`.

We proposed a simpler variant, using One-Hot Encoding to indicate whether a library is imported by malware. It is easier to extract and more reliable because in terms of system security, it is not as susceptible to anti-anti-virus techniques as the number of APIs. There are 570 different import libraries in the dataset. The 300 with the highest number of occurrences among them were retained. Fig. 2 demonstrates how this feature distinguishes `Obfuscator.ACY` from others and provides rough ideas about functionality. `Crypt32` is a cryptographic library. `Obfuscator.ACY` may rely it on encryption. Most samples from other classes are not encrypted so they do not import this library. We also extracted top libraries based on Gini Impurity as in Fig. 1.

### 3.5 PE Section Size

PE files consist of several sections. Each section stores different types of bytes and has attributes. The number of sections, their uses and attributes are defined by software development tools and programmers based on functionality.



**Fig. 1.** The most important libraries **Fig. 2.** A Decision Tree for Obfuscator.ACY

This feature focuses on section sizes. Each section has two types of sizes: a virtual size and a raw size. They are `VirtualSize` and `SizeOfRawData` fields of structure `IMAGE_SECTION_HEADER` in PE Headers, respectively. The raw size is exactly the size of a section on the disk, and the virtual size is the size of a section when it has been loaded into memory. For instance, a section may store only uninitialized data whose values are only available after startup. There is no need to allocate space for it on the disk, so the raw size is zero, but the virtual size is not. The ratio of the two types of sizes is also included in the feature. The dataset contains 282 sections with different names. Each section has three attributes, so the full feature has 846 dimensions. After feature selection using Random Forest based on Gini Impurity, only the 25 most essential dimensions were retained. Most of them are standard sections defined by software development tools as in Fig. 3.

### 3.6 PE Section Permission

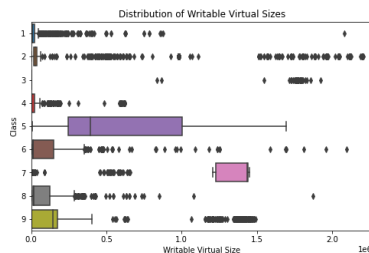
PE sections have access permissions, which are combinations of readable, writable and executable. We calculated the total size of readable data, writable data and executable code separately for each malware sample. Like the previous, each permission has three attributes: a virtual size, a raw size and a ratio of the two sizes. This feature can be regarded as a summary of PE section sizes and provides a more general view with only nine fixed dimensions. For example, Fig. 4 shows the distribution of writable virtual sizes. Four Backdoor classes (3, 5, 7, and 9) have relatively large writable space. This is possibly because they need to steal sensitive information or download other files from the Internet to run, which require enough memory space.

Additionally, we think these two PE section features (sizes and permissions) have compatibility with Linux systems. Linux uses the *Executable and Linkable Format (ELF)* for executable files. It has similar section structures to the PE format.

### 3.7 Content Complexity

Content complexity is a new feature type for malware classification. What we propose here has six fixed dimensions: the original sizes, compressed sizes and

Section	Stored Data
text	Executable code
data	Normal data
idata	Import libraries and functions
rdata	Read-only data
bss	Block Starting Symbols
tls	Thread Local Storage



**Fig. 3.** The most important sections **Fig. 4.** The distribution of writable virtual sizes

compression ratios of disassembly and machine code files. We used Python’s `zlib` library to compress samples and recorded size changes. This approximates function complexity, code encryption and obfuscation. Fig. 5 is from the sample with the largest disassembly compression ratio of 12.8. It might be obfuscated with repetitive, roundabout instructions. In contrast, Fig. 6 has the smallest disassembly compression ratio of 2.3. The disassembly failed and IDA Pro can only output its original machine code. This is because the sample is encrypted and packed by UPX, a famous open-source packer for executable files. In addition to this, the use of complex, rare instructions can also lead to low compression ratios.

```

mov    eax, esi    dd 66E5C7CBh, 0BABBBB02h
ror    eax, 8      dd 8ED64AF8h, 0F93780DFh
mov    ecx, esi    dd 3C99EBD7h, 0AF12FC33h
ror    ecx, 5      dd 6953D3A9h, 0C7012601h
    
```

**Fig. 5.** Snippet with the largest compression ratio **Fig. 6.** Snippet with the smallest compression ratio

In theory, this feature can be used directly in the malware classification of any other CPU architecture and system. It has better compatibility than others because it does not rely on any platform-related characteristics and structures. However, CPUs have different instruction sets. For instance, Intel x86 is based on Complex Instruction Set Computing, while ARM is based on Reduced Instruction Set Computing. It may affect classification accuracy.

## 4 EXPERIMENTS

For each feature and its combination, we used automatic machine learning library `auto-sklearn` to search for the best parameters, relying on Bayesian optimization, meta-learning and ensemble construction [1]. 80% of the dataset was used as a training set and `auto-sklearn` evaluated models on it using 5-fold

cross-validation. The models include  $K$ -Nearest Neighbors, Support Vector Machine and Random Forest. All experiments were conducted on 64-bit Ubuntu, Intel(R) Core(TM) i7-6700 CPU (3.40GHz) with 12GB RAM. Each model’s parameter search process lasted up to one hour. After `auto-sklearn` had determined a model’s optimal parameters, we used the remaining 20% as a test set to calculate classification accuracy. The results are shown in Table. 1, sorted in increasing order of accuracy. Random Forest provided the best performance in all experiments.

**Table 1.** The feature accuracy

Feature(s)	Dimension	Best Accuracy
All Features	1812921 $\rightarrow$ 10343	0.9948
Section Size, Section Permission, Content Complexity	861 $\rightarrow$ 40	0.9940
Section Size, Section Permission, Content Complexity, Import Library	1431 $\rightarrow$ 340	0.9922
Opcode 4-gram	1408515 $\rightarrow$ 5000	0.9908
File Size, API 4-gram, Opcode 4-gram	1811490 $\rightarrow$ 10003	0.9899
Content Complexity	6	0.9811
Section Size	846 $\rightarrow$ 25	0.9775
Section Permission	9	0.9701
Import Library	570 $\rightarrow$ 300	0.9393
File Size	3	0.9352
API 4-gram	402972 $\rightarrow$ 5000	0.5796

Among individual features, opcode 4-grams provided the highest accuracy of 99.08%, meaning static disassembly does not invoke many negative impacts on opcode 4-grams. They are effective both in dynamic and static analysis, but their extraction requires much time and computational resources. The original dimension of opcode 4-grams before feature selection is the largest (1408515). Content complexity, PE section sizes and PE section permissions achieved 98.11%, 97.75% and 97.01% accuracy respectively, which are satisfactory considering they are low dimensional representations. Import libraries did not perform very well, but the prediction paths generated by a Decision Tree of import libraries can provide functionality comparisons between malware families, like Fig. 2. Other features except API 4-grams cannot do this. At the beginning, we expected that the API sequences would be an effective feature in static disassembly analysis, as it does in dynamic behavior analysis. Unexpectedly, the API 4-gram is the worst. Its accuracy is only 57.96% and involves a 5000 dimensional feature vector representation. Our result shows that API sequences may only be applicable to dynamic behavior analysis. The data errors caused by static disassembly extraction have a very negative effect on feature validity.

Among integrated features, the combination of PE section sizes, PE section permissions and content complexity is almost the best with 99.40% accuracy and 40 dimensions. If all features were used, the accuracy was only improved by



0.08%, while the number of dimensions increased dramatically to 10343. Additionally, the highest accuracy of 99.48% we achieved is still lower than 99.97% in [10]. If interpretability is not considered, the combination of Convolutional Neural Networks and gray-scale images they used is obviously an excellent model.

## 5 LIMITATIONS OF STATIC DISASSEMBLY

The dataset contains only static text. In general, it negatively affects classification accuracy. We identified three specific problems. They mainly affect the extraction of API sequences and import libraries, and do not have serious impacts on the three new features we proposed.

### 5.1 Lazy Loading

In the process of extracting import libraries, only the libraries in the *Import Table* can be extracted, which is a structure in PE Headers used to import external APIs. These libraries will be automatically loaded when malware starts. In order to make malicious behavior more hidden, developers can use lazy loading to load a library just before it is about to be used. Lazily loaded libraries cannot be extracted from static disassembly scripts. As shown in Fig. 1, top libraries are ubiquitous and have no special significance for malware classification. A reasonable speculation is that sensitive libraries are lazily loaded and PE Headers only contain regular libraries.

### 5.2 Name Mangling

Compared with import libraries, the API sequence is more negatively affected. We found two reasons. The first is *Name Mangling*. It allows different programming entities to be named with the same identifier, like C++ overloading. Compilers can select the appropriate function based on parameters. It is convenient for programmers. Internally, compilers need different identifiers to distinguish them. Name mangling adds noise to the API  $n$ -gram extraction. For the same or similar functions, we may extract more than one name. A theoretical solution is to convert mangled names back to the same original name. However, in practice, it is challenging to develop converters for every possible compiler and language. Moreover, some compilers do not disclose their detailed name mangling mechanism.

### 5.3 Jump Thunk

*Jump Thunk* is the second reason for the poor performance of API sequences. Many compilers generate a jump thunk, a small code snippet, for each external API, then convert all calls to the API into calls to its jump thunk. This mechanism can provide an interface proxy. But it makes API sequences inaccurate when we used linear scanning to extract external API calls. Theoretically, we can recognize jump thunks and match them to external APIs. But thunks' names are random and their contents may be more complex than jump instructions.

## 6 PRACTICAL APPLICATION

As discussed in [2], the similarity between previous and future malware will degrade over time due to function updates and polymorphic techniques. Polymorphic techniques can automatically and frequently change identifiable characteristics like encryption types and code distribution to make malware unrecognizable to anti-virus detection. To solve this, we designed an automatic malware classification workflow to apply and enhance our classifier in practice with IDA Pro’s Python development kit, as shown in Fig. 7. The source code is available on the GitHub<sup>4</sup> and makes available as practical features the following contributions

### 1. Data Generation

In general, analysts can only collect raw executable malware, not disassembly scripts like those provided in the dataset. To generate new training data, we developed an IDA Pro script that can be run from the command line with IDA Pro’s parameters `-A` and `-S`, which launch IDA Pro in autonomous mode and make it run a script. For each executable file, it produces disassembly instructions and hexadecimal machine code, relying on IDA Pro’s disassembler. These two output files are in the same format as the files used for training in the dataset.

### 2. Automatic Classification

We used another automatic machine learning library TPOT to search for the best model for the feature combination of PE section sizes, PE section permissions and content complexity. We think this combination maintains a good balance between accuracy and the number of dimensions. TPOT achieved 99.26% accuracy, slightly lower than `auto-sklearn` (99.40%). Unlike `auto-sklearn`, TPOT uses Genetic Programming to optimize models [4]. Once the search is complete, it will provide Python code for the best pipeline. `auto-sklearn` does not have a similar function. With the fitted model, we developed an IDA Pro classifier plug-in. When an analyst opens a malware sample with IDA Pro, the plug-in will produce the required raw input files before features are calculated and the classification performed as in Fig. 8.

### 3. Manual Classification

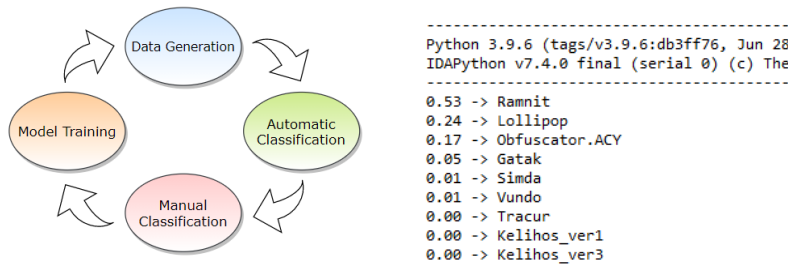
Although automatic classification is very useful, the result may be inaccurate or in doubt therefore the plugin provides a simple means for analysts to perform in-depth analysis manually to determine a sample’s exact family.

### 4. Model Training

With sufficient output files and labels of the latest samples, the classifier can be retrained and strengthened either manually or in an automated fashion.

Our model was trained by these nine malware families only, so if an input sample does not belong to them, the model will get an incorrect result or classify the sample into the family that is most similar to its actual type. But theoretically, these features are applicable to more families if more datasets are available.

<sup>4</sup> <https://github.com/czs108/Microsoft-Malware-Classification>



**Fig. 7.** The automatic malware classification workflow **Fig. 8.** The IDA Pro classifier plug-in

## 7 CONCLUSION AND FUTURE WORK

This paper demonstrates how novel, highly discriminative features of relatively low dimensionality when combined with automatic machine learning approaches can provide highly competitive classification accuracy for malware classification. Compared with traditional manual analysis, machine learning can provide a fast and accurate classifier after training on the latest malware samples. It does not rely on an understanding of code. Unlike API and opcode  $n$ -grams, which aim to match specific malicious operations, our features focus more on macroscopic information about malware. In theory, these features are more compatible with multiple operating systems and not susceptible to code encryption. One shortcoming is that they cannot offer detailed understanding of malicious behaviors like API sequences. Analysts must combine multiple features in order to perform more in-depth analysis. In addition, the negative limitations and effects of static text are more severe than we thought, especially for API  $n$ -grams. It is challenging to extract exact API sequences from disassembly scripts with linear scanning.

We conclude with a number of open avenues for research that might reduce the negative effects of static disassembly and improve machine learning models for malware processing:

- Remove regular libraries from the import library feature. Machine learning models are forced to use only sensitive libraries to classify samples. Note a potential problem here is that only a tiny number of sensitive libraries may be extracted.
- Although many C/C++ compilers exist, there are not many commonly used versions. We can consider developing name demangling for common compilers and renaming APIs using our defined convention.
- The core of a disassembly script is assembly instructions. So assemblers may be helpful to perform code analysis to determine the correspondence between APIs and jump thunks.

## Bibliography

- [1] Feurer, M., Klein, A., Eggenberger, K., Springenberg, J., Blum, M., Hutter, F.: Efficient and robust automated machine learning. In: Cortes, C., Lawrence, N.D., Lee, D.D., Sugiyama, M., Garnett, R. (eds.) *Advances in Neural Information Processing Systems* 28, pp. 2962–2970. Curran Associates, Inc. (2015), <http://papers.nips.cc/paper/5872-efficient-and-robust-automated-machine-learning.pdf>
- [2] Gibert, D., Mateu, C., Planes, J.: The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications* **153**, 102526 (2020). <https://doi.org/https://doi.org/10.1016/j.jnca.2019.102526>, <https://www.sciencedirect.com/science/article/pii/S1084804519303868>
- [3] Kalash, M., Rochan, M., Mohammed, N., Bruce, N.D.B., Wang, Y., Iqbal, F.: Malware classification with deep convolutional neural networks. In: 2018 9th IFIP International Conference on New Technologies, Mobility and Security (NTMS). pp. 1–5 (2018). <https://doi.org/10.1109/NTMS.2018.8328749>
- [4] Le, T.T., Fu, W., Moore, J.H.: Scaling tree-based automated machine learning to biomedical big data with a feature set selector. *Bioinformatics* **36**(1), 250–256 (2020)
- [5] Microsoft Corporation: PE format. Available at <https://docs.microsoft.com/en-us/windows/win32/debug/pe-format> (2021/05/10) (2021)
- [6] Nahid, M., Mehdi, B., Hamid, R.: An improved method for packed malware detection using PE header and section table information. *International Journal of Computer Network and Information Security* **11**, 9–17 (09 2019). <https://doi.org/10.5815/ijcnis.2019.09.02>
- [7] Nataraj, L., Karthikeyan, S., Jacob, G., Manjunath, B.S.: Malware images: Visualization and automatic classification. In: *Proceedings of the 8th International Symposium on Visualization for Cyber Security. VizSec '11*, Association for Computing Machinery, New York, NY, USA (2011). <https://doi.org/10.1145/2016904.2016908>, <https://doi.org/10.1145/2016904.2016908>
- [8] Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., Nicholas, C.: Malware detection by eating a whole EXE (2017)
- [9] Ronen, R., Radu, M., Feuerstein, C., Yom-Tov, E., Ahmadi, M.: Microsoft malware classification challenge. *ArXiv abs/1802.10135* (2018)
- [10] Simonyan, K., Zisserman, A.: Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014)
- [11] Symantec Corporation: Internet security threat report. Tech. rep., Symantec Corporation (2019)
- [12] Zheng, R., Fang, Y., Liu, L.: Homology analysis of malicious code based on dynamic-behavior fingerprint (in Chinese). *Journal of Sichuan University (Natural Science Edition)* **53**(004), 793–798 (2016)