Semantic Preprocessing for LLM-based Malware Analysis

Benjamin Marais, Tony Quertier, and Grégoire Barrué

Orange Innovation, Cesson-Sevigne, France
`benjamin.marais@orange.com`
`tony.quertier@orange.com`
`gregoire.barrue@orange.com`

Abstract. In a context of malware analysis, numerous approaches rely on Artificial Intelligence to handle a large volume of data. However, these techniques focus on data view (images, sequences) and not on an expert's view. Noticing this issue, we propose a preprocessing that focuses on expert knowledge to improve malware semantic analysis and result interpretability. We propose a new preprocessing method which creates JSON reports for Portable Executable files. These reports gather features from both static and behavioral analysis, and incorporate packer signature detection, MITRE ATT&CK and Malware Behavior Catalog (MBC) knowledge. The purpose of this preprocessing is to gather a semantic representation of binary files, understandable by malware analysts, and that can enhance AI models' explainability for malicious files analysis. Using this preprocessing to train a Large Language Model for Malware classification, we achieve a weighted-average F1-score of 0.94 on a complex dataset, representative of market reality.

Keywords: Malware Analysis· Data Preprocessing· Large Language Model· Cybersecurity.

Introduction

While Artificial Intelligence is now an usual tool in the domain of Malware detection and Malware classification [7], the training of the models is still one of the main issues, because of the representation of the data. Some works try to solve this problem by extracting more information from the samples, like EMBER [2], while other works try to change the representation of the files, for instance by transforming them into images [18], in order to train a convolutional neural network, or directly by analyzing the byte sequence of the binary file [23]. In the same way as other feature extraction methods [26], EMBER provides lots of features for each sample. However, it can be hard to understand which are the most impactful [19]. There are other techniques for feature extraction [6], some of them coupling static with behavioral analysis [32], but they all seem to suffer from the fact that they are not easy to understand, and thus do not allow some result explainability. Note that EMBER recently released a new version, providing more detailed reports in the PE file format [9].

This lack of relevant representation results in a bad classification of malware categories, and a poor explainability of the results. We need to add to these reports more contextual information. The aim is to get benefits from the summarization capabilities of the Large language Models to extract the most important information from the reports. Indeed, Large Language Models (LLM) offer a new opportunity in lots of domains, and Malware detection and classification is obviously one of them. The ability of these models to learn from semantic reports, coupled to the attention mechanism, is very useful. LLMs are used for instance to try to understand code [28], or even to disassemble obfuscated code [25].

In this article, we want to provide a new preprocessing for PE file feature extraction, which uses both static and behavioral analysis in order to create a JSON report of a PE file sample, containing detailed information. We illustrate the relevance of this preprocessing by using it to classify Malware categories thanks to LLM models. This classification task is actually more complex than the usual detection task, and some works try for instance to use image classification [12], or the features extracted from EMBER [9], [13]. While EMBER provides a standardized set of static PE features for malware classification, it does not include any contextual information derived from known rule-based threat signatures. Our approach complements existing datasets ([8], [10], ...) by incorporating both packer signature detections and YARA rule matches. These additional features aim to capture higher-level semantic signals and enhance interpretability, which are crucial for threat hunting and advanced malware triage. One advantage of our preprocessing is that the reports can be understood directly by an

analyst, since they contain concrete information. This article is organized as follows. In Section 1 we present our dataset, the format of our files and the preprocessing that we use to create our reports. Then, Section 2 gathers information about the experiments that we run with our reports, by feeding a Large Language Model for our classification task, and details the results that we obtained. Finally, we conclude about the advantages of using our complete preprocessing to train a LLM and discuss about future works.

1   Methodology

In this section, we present the methodology of our paper by describing the different fundamental blocks. To begin with, we detail the PE format and the dataset used in our work. Then, we describe the preprocessing proposed for our experimentation with the features that we selected and how we extract them from the binary files.

1.1   File format and dataset

Portable Executable Format Malicious files remain an important cybersecurity threat, accounting for over 60% of successful cyberattacks in 2024 [20,11]. Among them, malware based Portable Executable (PE) format represents more than 95% of malicious files in circulation [4], and are commonly use by threat actor to hide and diffuse malicious payload and commit exploits [14].

The PE format is a file format inherited from the COFF (Common Object File Format) format mainly used by UNIX systems. This format was created in 1993 by Microsoft [16] to be used with Windows operating system files. It is mainly used for DLLs (.dll) and executable (.exe) files. As shown in Figure 1, a PE file is divided into two parts: the header and the sections. The header describes the file and its contents. It contains, among other things, information such as the file creation date, information on file loading into memory and the number of sections. Each section is described by a specific header that includes its name, size and location in virtual memory. Sections generally contain the executable code (.text), the variables used and their default values (.data) or information on the functions (DLLs) required to execute the file (.idata).

By studying the information contained in a file's header and sections, as well as its behavior (and purpose), it is possible to determine whether a file is potentially malicious or not.
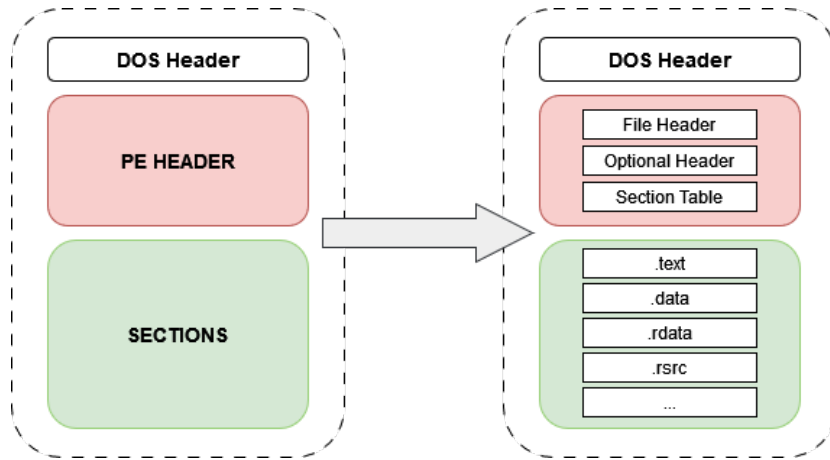

Fig. 1. Simplified architecture of a PE file

Dataset The files used for our work come from an open source dataset, available online. We selected the BODMAS dataset [30] for malicious files. A particularity of this dataset is that the types (or categories) of the files are described in the metadata provided by the authors. In this dataset we isolated 8 categories, which were the most represented in this dataset, the others not having enough

samples to train a model to classify them. The different categories are: trojan, worm, ransomware, backdoor, information stealer, downloader, dropper, and virus. They differ by their behavior once a computer is infected; for example a ransomware will encrypt the computer's data to force the owner to pay, whereas a trojan can hide its malicious behavior in order to steal data from the infected computer.

We chose to use a relatively big dataset (around 57,000 samples) compared to some other works [18,24], in order to ensure a good training of our model. This involves having under-represented malware categories because it is sometimes harder to find some types of malware and easier to find some other types, but in our opinion this allows to have a realistic distribution of the different categories. Besides, if we wanted to have a perfectly balanced dataset, we would have been forced to drastically reduce the size of our dataset, thus deteriorating the quality of the training.

Table 1. BODMAS' malware repartition

File Categories	Number of instance
Trojan	$29,972$
Worm	$16,697$
Backdoor	$7,331$
Downloader	$1,031$
Ransomware	821
Dropper	715
Infostealer	448
Virus	192

1.2   Preprocessing and Features Description

Based on our malware analysis experience, on expert knowledge, and inspired by work as EMBER [2], we propose the following preprocessing to convert a binary file into a convenient form to use it with machine learning or deep learning models. We extract substantial information from the file, generally used for static malware detection and analysis, and format this information into a report as a JSON file. As EMBER, we mainly focus on extracting static features contained in binary files, because they are the main characteristics used by malware analysts in this field. However, we also extract information based on rules or signatures to enrich our JSON reports. These rules or signatures give behavioral (MITRE ATT&CK and MBC) or obfuscation (packing) information. In the remainder of this section, we will describe in more details the characteristics extracted from the binary files. For each subsection of the JSON report, we will provide examples based on a variant of the Wannacry ransomware [17], whose sha256 is *ed01ebfbc9eb5bbea545af4d01bf5f1071661840480439c6e5babe8e080e41aa.*

Global information The first section of a JSON report provides global information on the analysis file as the file name, signature (sha256, md5) the file type, the target OS, the compilation date, the file size and the file entropy. This information describes the file, and for instance a weird compilation date or a hight entropy can be indicators of a suspicious behavior. However, it cannot be used to decide whether a file is a benign file or a malware, and it needs to be completed with all the information explained below.

Section information The second part of the report gives information related to the sections of the file.

In PE files, the section table describes how the program is organized in memory. Each section contains data or code needed during runtime or linking. A common section includes *.text* (executable code), *.data* (initialized data), *.rdata* (read-only data) and *.rsrc* (resources like icons or dialog). Each section has a set of characteristics, with a flag describing the section's attributes, such as whether it is executable, readable, writable or contains code, as described in table 2.

It is common to base malware analysis on section analysis [21,29]. For example, changing section names is a common technique used to evade detection or hinder reverse engineering [1,22]. In the

```
"global_information": {
    "filename": "wannacry.exe",
    "sha256": "ed01ebfbc9eb5bbea545af4d01bf5f1071661840480439c6e5babe8e080e41aa",
    "md5": "84c82835a5d21bbcf75a61706d8ab549",
    "file_type": "PE32 executable (GUI) Intel 80386 for MS Windows",
    "compilation_date": "2010-11-20 10:05:05",
    "file_size_ko": 3514,
    "file_entropy": 7.995,
    "target_os": "Windows(95)[I386, 32-bit, GUI]",
    "date_of_analysis": "2025-05-01"
```

Fig. 2. Global file information

Table 2. Some section characteristics

Flag Name	Description
MEM EXECUTRE	Section contains executable code
MEM WRITE	Section is writable
MEM READ	Section is readable.
CNT CODE	Section contains code
CNT INITIALIZED DATA	Section contains initialized data

other hand, some section names are a characteristic of evasion tools such as UPX[1] that compress the original code and data into custom sections like *.upx1* or *.upx2*. Additionally, malware may assign unusual characteristics to a section, such as making the *.rsrc* executable, which may indicate a hidden malicious payload in this section. These irregularities, in naming and characteristics, are red flags and often indicate packing, obfuscation or code injection techniques used to bypass antivirus engines or harden reverse engineering.

In our JSON report, as described in figure 3, we keep, among other things, the name, the size, the signature (sha256) and the characteristics of the sections.

Imported functions The Import Address Table (IAT) is a critical structure within a PE (Portable Executable) file that lists all the external functions a program relies on, typically from Windows DLLs like kernel32.dll, user32.dll, or advapi32.dll. Each entry in the IAT corresponds to a function that will be resolved and loaded by the Windows loader at runtime, pointing to the actual memory address of the function in the relevant DLL. The IAT contains both the names of the imported functions and the DLLs they come from. During static analysis, the IAT provides valuable insight into a binary's behavior. For instance, imports by ordinal instead of by name can be suspicious [3], as it is a common tactic used to evade detection or obfuscate a malicious intent. Analysts also look for malicious patterns in imported functions, such as the presence of API calls often used in code injection or process hollowing, for example. A cluster of the imports contained can strongly suggest the binary may be performing one of the actions described in Table 3. Therefore, scrutinizing the IAT is a key step in identifying potential threats during static malware analysis.

In the next section, we extract information related to the IAT. As described in Figure 4 We first give the imphash, a file signature related to the imported functions, and the number of clear and ordinal imported functions. Then we give the list of all imported functions, sorted by library (kernel32.dll for example). If a function is imported by ordinal, we retrieve, if possible, its name.

Packing signatures The fourth section, illustrated in Figure 5 is related to the obfuscation, in particular if the file is packed. Packing code is a common technique used by threat actor and malware editors to hide malicious payload and disable or prevent static analysis [27]. Even if it is a technique used by software editors for intellectual property reasons, it is also a indicator of suspicious for malicious files analysis.

---

[1] https://upx.github.io/

```
"section_information": {
    "number_of_sections": 4,
    "sections_desscription": [
        {
            "name": ".text",
            "sha256_section": "7e02fab4c69f500a381269cf00d273702d910366a6cb8979a0def828
            "id_section": 0,
            "entropy": 6.598759307021768,
            "file_ratio": 0.007698681526806527,
            "raw_address_start": "0x00001000",
            "raw_address_end": "0x000079B0",
            "section_size": 27056,
            "virtual_address": "0x00001000",
            "virtual_size": 27056,
            "characteristics": [
                "CNT_CODE",
                "MEM_EXECUTE",
                "MEM_READ"
            ]
        },
        {
            "name": ".rdata",
            "sha256_section": "7a7c8335535965af42e8e56457d63e07666afef48208f156fd4e81f5
            "id_section": 1,
            "entropy": 6.691364173966162,
            "file_ratio": 0.006952032342657343,
            "raw_address_start": "0x00008000",
            "raw_address_end": "0x0000DF70",
            "section_size": 24432,
            "virtual_address": "0x00008000",
            "virtual_size": 24432,
            "characteristics": [
                "CNT_INITIALIZED_DATA",
                "MEM_READ"
            ]
        },
        {…
        },
        {…
        }
    ]
```

Fig. 3. File section information

Table 3. Some risky windows API calls from [31]

Exploit	Windows API Calls
Code injection	CreateRemoteThread, OpenProcess, VirtualAllocEx, WriteProcessMemory, EnumProcesses
Dynamic DLL loading	LoadLibrary, GetProcAddres
Memory scrapping	CreateToolhelp32Snapshot, OpenProcess RadProcessMemory, EnumProcess
Unpacking/self-injection	ViruatlAlloc, VirtualProtect
Execute a program	WinExec, ShellExecute, CreateProcess
Query artifact	CreateMutex, CreateFile FindWindow, GetModuleHandle, RegOpenKeyE

We use open-source tools as DIEC[2], PEid[3] or PyPackerDetecte[4] to detect signatures of packer tools. Based on these different tools, we extract signatures of potential packers, and other indications of potential packing. We define the packing label l_p as following :

$$l_p = \frac{\sum_{i=1}^{n} w_i l_{pi}}{\sum_{i=1}^{n} w_i} \tag{1}$$

---

[2] https://github.com/horsicq/Detect-It-Easy
[3] https://github.com/packing-box/peid
[4] https://github.com/packing-box/pypackerdetect

```
"imports_information": {
    "imphash": "68f013d7437aa653a8a98a05807afeb1",
    "number_of_imports": 114,
    "number_of_ordinals": 0,
    "has_no_imports": false,
    "imports": [
        {
            "libname": "kernel32.dll",
            "imports": [
                {
                    "impname": "getfileattributesw",
                    "is_ordinal": false,
                    "ordinal_value": 0
                },
                {
                    "impname": "getfilesizeex",
                    "is_ordinal": false,
                    "ordinal_value": 0
                },
                {
                    "impname": "createfilea",
                    "is_ordinal": false,
                    "ordinal_value": 0
                },
                {
                    "impname": "initializecriticalsection",
                    "is_ordinal": false,
                    "ordinal_value": 0
                },
                {
                    "impname": "deletecriticalsection",
                    "is_ordinal": false,
                    "ordinal_value": 0
                },
                {
                    "impname": "readfile",
                    "is_ordinal": false,
                    "ordinal_value": 0
                },
                {
                    "impname": "getfilesize",
                    "is_ordinal": false,
                    "ordinal_value": 0
                },
```

Fig. 4. IAT information

where $l_{pi} \in \{0, 1\}$ is the packing label returned by each packing detection tools, with $l_{pi} = 1$ if the tools detect a packer signature or indication of packing. Each sub-label l_{pi} is weighted by a weight w_i defined by the efficiency of the tools for packing detection.

Finally, based on the signature returned by packing detection tools, we also return the potential packing software used on the file.

Mitre Att&ck and MBC information Then the last section is related to Mitre Att&ck and MBC identification.

The MITRE ATT&CK[5] framework is a globally accessible knowledge base of adversary tactics and techniques based on real-world observations. It provides a structured approach to understanding how cyberthreats operate across the attack lifecycle. For malware analysis, especially in categorizing threats like trojans and ransomware, ATT&CK helps analysts to map malicious behaviors to specific techniques such as credential dumping, command and control, or data encryption for impact. By aligning observed malware behaviors with ATT&CK techniques, security professionals can gain deeper insight into the malware's functionality, objectives, and potential mitigation. This behavioral mapping is critical for threat intelligence, detection engineering, and incident response, allowing defenders to anticipate attacker moves and improve their defensive posture.

The Malware Behavior Catalog (MBC)[6] is a structured framework developed to describe and classify the behaviors of malware in a consistent and standardized way. Unlike general threat frame-

---

[5] https://attack.mitre.org/
[6] https://github.com/MBCProject/mbc-markdown

```json
"packing_information": {
    "packing_score": 0.11,
    "packing_label": "not_packed",
    "packer_name": null,
    "log": [
        {
            "tool_name": "DIEC",
            "packed": false,
            "weight": 4,
            "detected_packers": []
        },
        {
            "tool_name": "PEiD",
            "packed": false,
            "weight": 3,
            "detected_packers": []
        },
        {
            "tool_name": "Import",
            "packed": false,
            "weight": 1,
            "suspect_imports": [
                "kernel32.dll.virtualalloc",
                "kernel32.dll.heapalloc",
                "kernel32.dll.createprocessa",
                "kernel32.dll.findresourcea",
                "kernel32.dll.sizeofresource",
                "kernel32.dll.readfile",
                "kernel32.dll.virtualfree",
                "kernel32.dll.loadlibrarya"
            ]
        },
        {
            "tool_name": "PPD",
            "packed": true,
            "weight": 1,
            "signatures": "PEID signature: Microsoft Visual C++ v6.0, Microsoft Visual
        }
    ]
```

Fig. 5. Packing signatures

works, MBC focuses specifically on how a malware operates, including capabilities such as data theft, persistence, evasion, and execution. It complements MITRE ATT&CK by diving deeper into malware-centric techniques and patterns, making it especially valuable for categorizing malware categories like trojans, ransomware, and spyware. By mapping observed behaviors to MBC entries, analysts can better understand a malware sample's functionality, compare it with known behaviors, and enrich threat intelligence. This detailed behavioral insight supports reverse engineering, detection development, and attribution efforts, enhancing the overall malware analysis process.

Using CAPA[7], we extract MITRE ATT&CK and MBC information and integrate it into our JSON report, as described in Figure 6. The purpose of including this information is to provide to malware analyst behavioral information about a malicious file, and to improve malware analysis by deep learning models by giving a better description of the files.

2   Experiments

In this section, we present the results of our malware classification approach based on transformers architecture. This is a much more complicated use-case than malware detection. We do not address the detection task here as we believe it does not require the use of transformers, and it has been addressed previously in other works with good results [2,7,15] .

We have chosen the BERT base uncased model [5], a transformers model pretrained on a large corpus of English data in a self supervised fashion using a masked modeling objective, because it is a very versatile model with only 110 million parameters. One limitation of Bert is in the number of 512 tokens, but by selecting the most relevant parts of the reports, we end up with an encoding where the average number of tokens is 403, the median number is 350 and only $15/percent$ of the data is greater

---

[7] https://github.com/mandiant/capa

```
"mitre_mbc_information": {
    "mitre_id": [
        {
            "id": "T1222",
            "technique_name": "File and Directory Permissions Modification ",
            "tactic_name": "DEFENSE EVASION",
            "description": "Adversaries may modify file or directory permissions/attrib
        },
        ...,
        {
            "id": "T1543.003",
            "technique_name": "Create or Modify System Process::Windows Service ",
            "tactic_name": "PERSISTENCE",
            "description": "Adversaries may create or modify Windows services to repeat
        }
    ],
```

Fig. 6. MITTRE ATT&CK and MBC

than 512 tokens. We consider this a good compromise, given the quality of the Bert model and its speed of training. Using a small model enabled us to perform our tests on an Apple M4 Pro macbook. We also did some tests by fine-tuning deepseek models with 1.5B parameters on a 4090 RTX, but the computation time was too long and the results not so satisfying.

2.1 Training / testing

With this model, we are able to process and analyze JSON reports associated to each malware. Transformers, with their ability to capture complex contextual relationships in textual data, are a very well-adapted solution for this classification problem. The following results highlight the effectiveness of this approach and provide insights into the strengths of transformer based malware classification, if we have the adapted preprocessing. As the classes are not balanced, we focus on the F1-score, but we give other metrics for information. The train/test split is as usual 80/20, and we train our model on 10 epochs with a batch size of 16. We classify 8 categories, ranked from 0 to 7, in the following order: trojan, worm, ransomware, backdoor, information stealer, downloader, dropper, and virus.

2.2 Results / analysis

In order to give a first illustration of our results concerning the classification task, we compute the confusion matrices for the train set in Figure 7 and for the test set in Figure 8. We can see that the model is really good at identifying the most represented categories. It is less clear for the under-represented categories, but the results are still good. The only category where the classification seems not to be efficient is the dropper category, where the model tends to identify droppers as trojans. It is not very surprising, and can be explained because droppers (also named trojan-dropper) are a type of trojan that has been designed to install a malware into a computer. For example, F-Secure or Kaspersky categorize it as trojan-dropper.

To get a more precise idea of our model's performances, we present in Tables 4 and 5 the accuracy, F1-score and recall of the train and test sets respectively. We also specify the number of sample of each class in these tables, and compute the average scores of the classification task.

The classification results obtained using the BERT model demonstrate strong performances across most malware categories, even the under-represented, with a F1-score greater than 0.92 (resp. 0.83) for all classes except one during the training (resp. testing). As illustrated by the Tables 4 and 5, the model achieves very high accuracy and F1-score in distinguishing all classes except droppers with minimal misclassification. Unsurprisingly, under-represented categories are harder to identify for the model, but it could be fixed if we manage to increase the number of samples from these categories.

Overall, these results highlight the potential of leveraging language models associated with very complete reports on malware for automated classification. It is thus important to create such reports, gathering as much information as possible, from static and behavioral analysis.
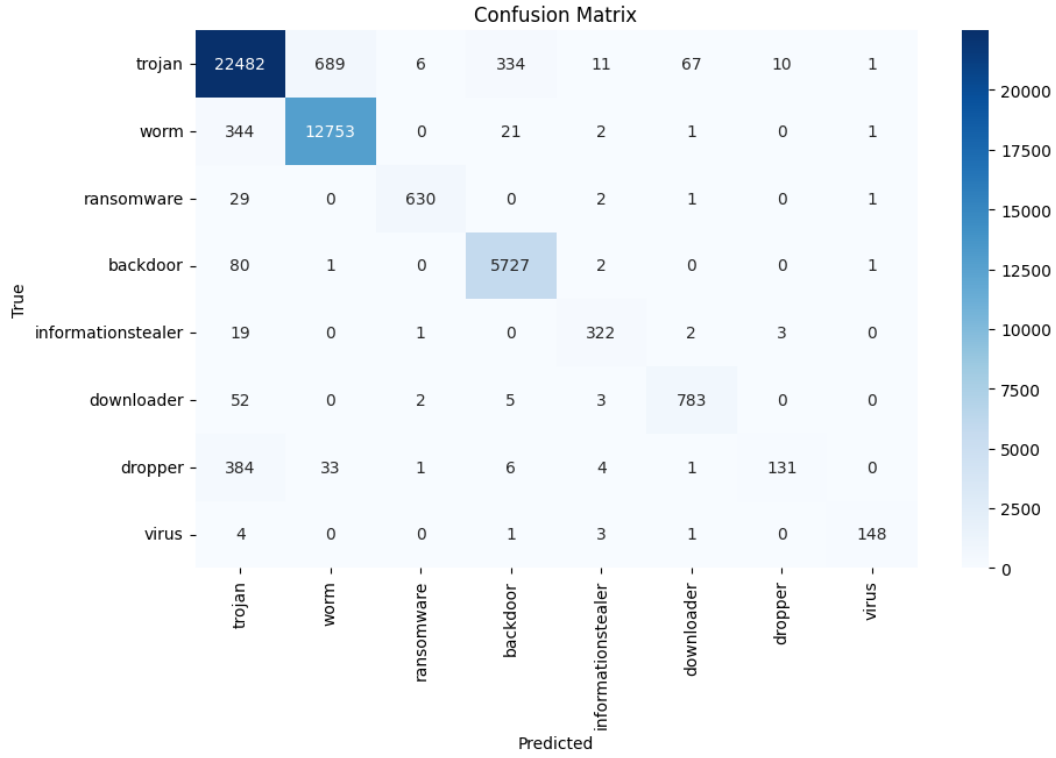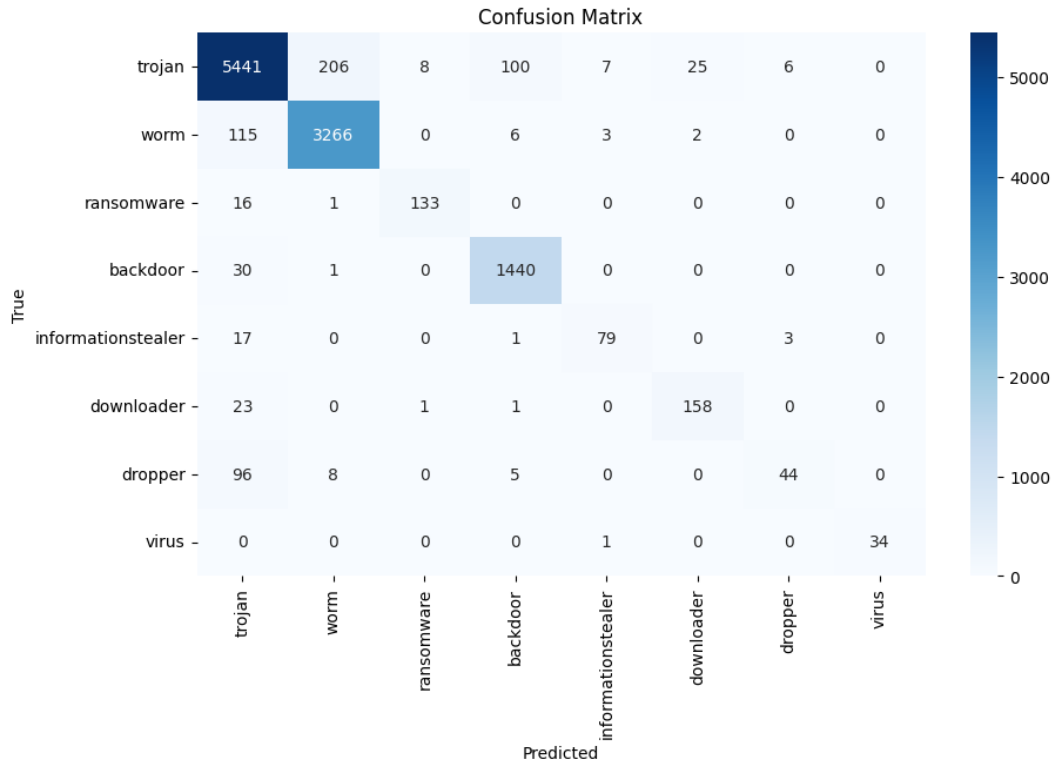
Fig. 7. Confusion matrix for the train dataset.

Fig. 8. Confusion matrix for the test dataset.

Conclusion and future works

In a nutshell, we present in this article a new preprocessing for PE malware files, which allows semantic analysis via Large Languages Models. Our preprocessing generates JSON reports gathering static and

Table 4. Training classification report

Class	Accuracy	Recall	F1-score	Support
0	0.96	0.95	0.96	23600
1	0.95	0.97	0.96	13122
2	0.98	0.95	0.97	663
3	0.94	0.99	0.96	5811
4	0.92	0.93	0.93	347
5	0.91	0.93	0.92	845
6	0.91	0.23	0.37	560
7	0.97	0.94	0.96	157
Global				
Accuracy			0.95	45105
Macro avg	0.94	0.86	0.88	45105
Weighted avg	0.95	0.95	0.95	45105

Table 5. Testing classification report

Class	Accuracy	Recall	F1-score	Support
0	0.95	0.94	0.94	5793
1	0.94	0.96	0.95	3392
2	0.94	0.89	0.91	150
3	0.93	0.98	0.95	1471
4	0.88	0.79	0.83	100
5	0.85	0.86	0.86	183
6	0.83	0.29	0.43	153
7	1.00	0.97	0.99	35
Global				
Accuracy			0.94	11277
Macro avg	0.91	0.84	0.86	11277
Weighted avg	0.94	0.94	0.94	11277

behavioral features, such as packer signatures and YARA rules. Our preprocessing gathers concrete features, that can be read and understood directly by analysts. We used a BERT model in order to classify eight categories of malware, on a realistic dataset where some categories are under-represented. The classification results are very good and encouraging, showing that our preprocessing is suitable for semantic analysis.

This work opens several perspectives to enhance our performances. First, static and behavioral analysis would benefit to be completed by dynamic analysis, which will be incorporated in the reports, giving an even more complete preprocessing for PE files. We also want to refine our classification by detecting the category but also the family of a malware. In this aim it will be suitable to increase the size of our dataset, gathering more samples from the diverse categories and thus the families. This will probably increase the performances of our model, which might be able to differentiate more precisely close categories such as droppers and trojans for example. We also might be able to classify more categories and more families. Finally, the big advantage of LLMs is the use of attention mechanisms. We ran some tests in this direction but it was not conclusive for our classification use-case, but it will be interesting to use it for instance to summarize the behavior of a sample based on its report. For the classification use-case, we will try to provide some explainability based of our knowledge concerning the information contained in the report: for example some Mitre Att&ck id may be specific to the ransomware category.

Note that our feature extraction and report generation method is currently a prototype which will be enhanced according to the previous perspectives, and released publicly for open research. We believe that this new complete preprocessing is a very good tool for AI-assisted Malware classification, as it gathers in the same place all the information about a file, allowing very modular analysis if someone wants to use only specific features. We expect this contribution to help solving the issue of bad data representation for Malware classification.

References

1. Anderson, H.S., Kharkar, A., Filar, B., Evans, D., Roth, P.: Learning to evade static pe machine learning malware models via reinforcement learning (2018), https://arxiv.org/abs/1801.08917
2. Anderson, H.S., Roth, P.: Ember: an open dataset for training static pe malware machine learning models. arXiv preprint arXiv:1804.04637 (2018)
3. Aubert, M.: Pestudio standard (2016), https://medium.com/@aubsec/pestudio-standard-f2ada4e8564, accessed: 2025-05-28
4. AV-Test: Av-atlas (2025), https://portal.av-atlas.org/malware, accessed: 2025-05-28
5. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: Bert: Pre-training of deep bidirectional transformers for language understanding (2019), https://arxiv.org/abs/1810.04805
6. El Neel, L., Copiaco, A., Obaid, W., Mukhtar, H.: Comparison of feature extraction and classification techniques of pe malware. In: 2022 5th International Conference on Signal Processing and Information Security (ICSPIS). pp. 26–31. IEEE (2022)
7. Gaber, M.G., Ahmed, M., Janicke, H.: Malware detection with artificial intelligence: A systematic literature review. ACM Comput. Surv. **56**(6) (Jan 2024). https://doi.org/10.1145/3638552, https://doi.org/10.1145/3638552
8. Harang, R., Rudd, E.M.: Sorel-20m: A large scale benchmark dataset for malicious pe detection (2020), https://arxiv.org/abs/2012.07634
9. Joyce, R., Miller, G., Roth, P., Zak, R., Zaresky-Williams, E., Anderson, H., Raff, E., Holt, J.: Ember2024 – a benchmark dataset for holistic evaluation of malware classifiers (06 2025). https://doi.org/10.48550/arXiv.2506.05074
10. Joyce, R.J., Raff, E., Nicholas, C., Holt, J.: Maldict: Benchmark datasets on malware behaviors, platforms, exploitation, and packers (2023), https://arxiv.org/abs/2310.11706
11. Kapersky: The cyber surge: Kaspersky detected 467,000 malicious files daily in 2024 (2024), https://www.kaspersky.com/about/press-releases/the-cyber-surge-kaspersky-detected-467000-malicious-files-daily-in-2024, accessed: 2025-05-28
12. Khadilkar, A., Stamp, M.: Image-based malware classification using qr and aztec codes (2024), https://arxiv.org/abs/2412.08514
13. Loi, N., Borile, C., Ucci, D.: Towards an automated pipeline for detecting and classifying malware through machine learning (2021), https://arxiv.org/abs/2106.05625
14. Mandiant: Portable executable file infecting malware is increasingly found in ot networks (2021), https://cloud.google.com/blog/topics/threat-intelligence/pe-file-infecting-malware-ot/?hl=en, accessed: 2025-05-28
15. Marais, B., Quertier, T., Morucci, S.: Ai-based malware and ransomware detection models (2022), https://arxiv.org/abs/2207.02108
16. Microsoft Corporation: Inside windows: An in-depth look into the win32 portable executable file format (2019), https://learn.microsoft.com/en-us/archive/msdn-magazine/2002/february/inside-windows-win32-portable-executable-file-format-in-detail, accessed: 2025-05-28
17. Mohurle, S., Patil, M.: A brief study of wannacry threat: Ransomware attack 2017. International journal of advanced research in computer science **8**(5), 1938–1940 (2017)
18. 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. pp. 1–7 (2011)
19. Oyama, Y., Miyashita, T., Kokubo, H.: Identifying useful features for malware detection in the ember dataset. In: 2019 seventh international symposium on computing and networking workshops (CANDARW). pp. 360–366. IEEE (2019)
20. Positive Technologie: Cybersecurity threatscape: Q2 2024 (2024), https://global.ptsecurity.com/analytics/cybersecurity-threatscape-2024-q2, accessed: 2025-05-28
21. Quertier, T., Barrué, G.: Use of multi-cnns for section analysis in static malware detection. arXiv e-prints pp. arXiv–2402 (2024)
22. Quertier, T., Marais, B., Morucci, S., Fournel, B.: Merlin – malware evasion with reinforcement learning (2022), https://arxiv.org/abs/2203.12980
23. Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., Nicholas, C.: Malware detection by eating a whole exe (2017), https://arxiv.org/abs/1710.09435
24. Ronen, R., Radu, M., Feuerstein, C., Yom-Tov, E., Ahmadi, M.: Microsoft malware classification challenge (2018), https://arxiv.org/abs/1802.10135
25. Rong, H., Duan, Y., Zhang, H., Wang, X., Chen, H., Duan, S., Wang, S.: Disassembling obfuscated executables with llm. arXiv preprint arXiv:2407.08924 (2024)
26. Sun, B., Li, Q., Guo, Y., Wen, Q., Lin, X., Liu, W.: Malware family classification method based on static feature extraction. In: 2017 3rd IEEE International Conference on Computer and Communications (ICCC). pp. 507–513. IEEE (2017)

27. Vigna, G., Balzarotti, D.: When malware is packing heat. ENIGMA 2018, January 16-18, 2018, Santa Clara, CA, USA (2018), copyright Usenix. Personal use of this material is permitted. The definitive version of this paper was published in ENIGMA 2018, January 16-18, 2018, Santa Clara, CA, USA and is available at :

28. Wang, Y., Le, H., Gotmare, A.D., Bui, N.D., Li, J., Hoi, S.C.: Codet5+: Open code large language models for code understanding and generation. arXiv preprint arXiv:2305.07922 (2023)

29. Xiao, M., Guo, C., Shen, G., Cui, Y., Jiang, C.: Image-based malware classification using section distribution information. Computers & Security **110**, 102420 (2021). https://doi.org/https://doi.org/10.1016/j.cose.2021.102420, https://www.sciencedirect.com/science/article/pii/S0167404821002443

30. Yang, L., Ciptadi, A., Laziuk, I., Ahmadzadeh, A., Wang, G.: Bodmas: An open dataset for learning based temporal analysis of pe malware. In: 2021 IEEE Security and Privacy Workshops (SPW). pp. 78–84. IEEE (2021)

31. Zeltser, L.: Tips for reverse-engineering malicious code, https://zeltser.com/reverse-engineering-malicious-code-tips/, accessed: 2025-05-28

32. Zhang, J., Qin, Z., Yin, H., Ou, L., Zhang, K.: A feature-hybrid malware variants detection using cnn based opcode embedding and bpnn based api embedding. Computers & Security **84**, 376–392 (2019)