# A Lean Transformer Model for Dynamic Malware Analysis and Detection

Tony Quertier\*, Benjamin Marais\*, Grégoire Barrué\* Stéphane Morucci\*, Sévan Azé\*, Sébastien Salladin<sup>†</sup>

\*Orange Innovation,Rennes, France <sup>†</sup>Orange DSEC, Rennes, France mails : firstname.name@orange.com

Abstract—Malware is a fast-growing threat to the modern computing world and existing lines of defense are not efficient enough to address this issue. This is mainly due to the fact that many prevention solutions rely on signaturebased detection methods that can easily be circumvented by hackers. Therefore, there is a recurrent need for behavior-based analysis where a suspicious file is ran in a secured environment and its traces are collected to reports for analysis. Previous works have shown some success leveraging Neural Networks and API calls sequences extracted from these execution reports.

Recently, Large Language Models and Generative AI have demonstrated impressive capabilities mainly in Natural Language Processing tasks and promising applications in the cybersecurity field for both attackers and defenders.

In this paper, we design an Encoder-Only model, based on the Transformers architecture, to detect malicious files, digesting their API call sequences collected by an execution emulation solution. We are also limiting the size of the model architecture and the number of its parameters since it is often considered that Large Language Models may be overkill for specific tasks such as the one we are dealing with hereafter. In addition to achieving decent detection results, this approach has the advantage of reducing our carbon footprint by limiting training and inference times and facilitating technical operations with less hardware requirements. We also carry out some analysis of our results and highlight the limits and possible improvements when using Transformers to analyze malicious files.

*Index Terms*—malware detection and analysis, dynamic analysis, Large Language Model, cybersecurity, artificial intelligence

#### INTRODUCTION

The emergence of generative AI [1] and transformers [2] opens up a lot of new opportunities in the field of cybersecurity [3], for both attackers and defenders. Numerous papers have highlighted various ways in which this new technology can be used to automate attacks, such as the creation of malware [4] or phishing websites [5] for instance, but also to improve Red Teams tooling [6] or accelerate mitigations.

In the context of dynamic malware analysis, where a malicious file is executed in a secure environment and information reports are gathered to identify potential misbehaving activities, researchers focus on AI to deal with the problems of detection and classification [7]. The question of how to deal with the information contained in these reports is crucial, and several works [8]–[11] have identified the importance of API (Application Programming Interface) calls and their arguments.

Recently, Trizna *et al.* leveraged "Speakeasy" [12], a Windows emulation tool that makes it possible to collect dynamic traces during the analysis of malware and benign Windows PE files with minimal temporal and computational costs [13]. These traces (i.e. reports) are then processed to train machine learning models for malicious behavior detection. Trizna publicly released two of these models: "Quo Vadis" [13] and "Nebula" [14].

In this article, we are building upon Trizna works with a focus on a Small Language Model (SLM) to limit our carbon footprint and to reduce training times and computing resources [15]. Even if larger Language Models tend to perform better [16], it has also been showed that a particular attention needs to be devoted to the ratio between the size of the models and the size of the training data [17], especially when dealing with very specific tasks.

Our model aims at identifying malware based on API names and their arguments extracted from Speakeasy emulation reports and tokenized in a custom and optimized way. We compare our results with Nebula [14], and also train our model on other datasets to investigate our SLM generalization capabilities.

Note that our definition of an Language Model may

differ from others since from our point of view, a Language Model encompasses Encoder-Only, Encoder-Decoder and Decoder-Only architectures.

The paper is organized as follows. In Section I we detail our datasets, the different preprocessing steps and the architecture of our SLM, Encoder-Only model. Section II presents several results, while we detail some limitations in Section III. Finally, Section IV gives some insights into future works in order to improve our model.

# I. METHODOLOGY

Motivated by the fact that LLMs require significant computing resources, one side objective is to implement a Small Language Model. For specific tasks, it has been shown that it is not always optimal to implement and train a Large Model with a massive amount of data. Leaner Models can match Large Models performance when they are trained with high quality data [18], [19]. Focusing on low demanding models has several benefits both from a carbon footprint perspective and from a usability point of view. Since our models are expected to be operated by Orange technical teams, low-demanding energy and resources models are easier and faster to deploy to production.

We build an Encoder-Only model, based on the architecture defined in [2]. We combine Speakeasy emulation reports provided by Quo Vadis and JSON reports generated from two academic datasets and also from our own proprietary corpus. API calls are extracted from these reports, normalized and tokenized using a custom algorithm. Associated tokens are then input into the model for training. This model outputs Portable Executable (PE) files [20] prediction labels based on a maliciousness score. Figure 1 provides an overview of this detection chain.

## Datasets

To investigate the relevance of a Transformer-based machine learning model achieving malware detection digesting emulation reports, we leverage several datasets in this work. The first one is the freely available Quo Vadis dataset [13] that has been generated using Speakeasy emulation tool and some manual labeling by a professional threat intelligent team. This dataset contains JSON reports of malware and benign execution emulations, with information on API calls, file usages, network traffic and registry accesses, among others. In this work, we are focusing only on API names and their arguments. This dataset has also the advantage of being already split into training and validation corpuses to foster reproducible



Fig. 1: Overview of our detection chain.

results. Our model is thus trained on this Quo Vadis train corpus and compared to previous academic results [13], [14]. We also experiment our methodology on the Bodmas dataset provided by [21], with extra benign files extracted from PEMachineLearning dataset [22]. This combined dataset is referred to as "B&PEML" which stands for "Bodmas and PEMachineLearning". It finally contains 57,293 malicious files and 81,322 benign files in raw PE format, collected between August 2019 and September 2020. As it is a well-balanced dataset and widely used in the literature [23]-[25], it makes sense to evaluate our model on JSON reports generated by Speakeasy on this specific dataset. Comparisons with existing and future works can also be made to benchmark performance. Finally, we use a third internal dataset composed of benign and malicious PE files collected internally during the first months of 2024. This last source of data will give us insights into our model generalization capabilities when facing recent PE files. In this document, we refer to this internal dataset as the OBMID-24 dataset where this acronym stands for "Orange Benign and Malware Internal Dataset 2024".

Several samples from these datasets produced unusable reports because of too few reported API calls. This is due to the fact that Speakeasy stops its process when it encounters an API call that is not present in its emulation library. In order to prevent some bias related to these early stops, we discard all the reports under 4 KBytes, because they generally contain fewer than five instructions. Table I reports the approximate number of samples per dataset after this filtering.

TABLE I: Approximate number of retained samples per dataset.

	benign	malware
Quo Vadis	33k	59k
B&PEML	42k	39k
OBMID-24	10k	5k

# Cleaning and normalization

To facilitate reports processing by our model, we first perform API names and arguments cleaning and normalization. Similarly to Trizna *et al.*. [14], we filter out some unnecessary symbols and replace some fields by generic placeholders in order to reduce variability, tokens list size and irrelevant data. For instance, all url addresses are replaced by the  $\langle url \rangle$  placeholder. Table II summarizes several placeholders used during this cleaning and normalization process. This preprocessing limits the creation of pointless tokens, since a tokenizer could for example split an url address into multiple useless parts that may hinder model training and inference efficiencies.

TABLE II: List of some placeholders used to normalize our data.

Arguments	Tags
http://	〈url〉
C:\\	$\langle path \rangle$
Long string (100+)	(string)
Google\\Temp	(google)
Error 0x80004005	〈error〉

#### Tokenization

Tokenization is a crucial step and must not be overlooked. Even if their work focuses on source code, Jimenez *et al.* [26] report that differences between tokenizers are of practical importance since the use of different tokenizers may lead to contradictory conclusions. Put another way, a careful selection of tokenizers is required for a given specific task.

Since API names from Speakeasy reports are constant and finite in number, it does not make a lot of sense to tokenize these names in order to capture different meanings according different contexts. Authors in [27] found that preventing the tokenizer from decomposing API names can help improve Pre-Trained Models performance. Adding API names as specific tokens to their dictionary makes their model focus more on API sequences rather than on API names alone.

VMware found similar results in [28] when using common tokenizers algorithms for API names and/or API calls arguments. A WordPiece tokenizer applied to VMware product names, technical jargon and multicompound words leads to a "sub-token soup" that hinders Natural Language Processing efficiency.

In the Quo Vadis dataset, we identified about 2,500 unique API names that could all be added to our dictionary making it as large as dictionaries used in other academic works. For instance, Nebula [14] leverages a 50k tokens dictionary while Neurlux [9] relies on a 10k tokens dictionary. However, due to our focus on a limited resources model, we decided to associate specific tokens to top API names only to limit the size of our dictionary.

We complement this static mapping with different tokens computed by a WordPiece tokenizer [29], [30] on remaining API names, their associated libraries and all API arguments. WordPiece tokenizer is an efficient solution to alleviate some out-of-vocabulary issues when dealing with rare API names or unseen API parameters. We also double check manually that our dictionary does not exhibit sub-optimal tokenization on all API names and libraries. Our final dictionary size is close to 5,500 tokens.

We present an example of our preprocessing method applied to a malicious file in Figure 2.

# Encoder

We build an Encoder-Only model to classify PE files, based on Speakeasy emulation reports. We decided not to rely on fine-tuning some existing pre-trained Encoder-Only models, like BERT [31] and its derivatives, for two main reasons: (i) API names and arguments have very different characteristics from the language that BERT model or alike has processed during its training steps and (ii) we want to focus on a relatively modest machinelearning architecture to limit our carbon footprint by reducing time and resources consumption.

Our model is very similar to the Transformer architecture as detailed in [2]. We are using an embedding layer and some positional encoding before the encoder layers. The encoder layers are composed of multi-head attention (MHA) sublayers, followed by feed-forward neural networks (FFNNs). After each sublayer, we add residual learning [32] as well as a normalization step to









Fig. 2: An example of preprocessing applied to a malicious file.

ensure a better learning and a better context understanding. Encoder layers output is then aggregated using a global average pooling and then passed through a fullyconnected neural layer and returns the output prediction of the label. Figure 3 details the different layers of our Encoder-Only model.

According to [14], it could be more suitable to use a smaller embedding size and increase the number of heads in the MHA layer. So we set the embedding size to 32, and we set the number of heads for each MHA layer to 8. We also choose to restrict the size of the input sequences given to the model to 512 tokens. This choice is justified by our analysis of the sequences length in both our datasets, presented in Figure 4, which shows that very few sequences contain more than 500 tokens (only 12% in the B&PEML dataset).



Fig. 3: Architecture of our Encoder-Only model.

#### **II. RESULTS**

#### A. Tests on the different datasets

We first train our model on Quo Vadis dataset only with a train-test separation as defined by their authors. Note that the testset provided by Quo Vadis is composed of 3 months-old data compared to its training data, as explained in [13]. We retain 80% of the train dataset for training and 20% for validation. Our model achieves 98% (resp. 97%) accuracy on the train (resp. validation) dataset. Table III summarizes the F1-score and the accuracy result computed on the test corpus of datasets detailed in Section I.

TABLE III: Results on different datasets using our model trained on Quo Vadis dataset only.

	F1-score	Accuracy
Quo Vadis	0.87	0.872
B&PEML	0.76	0.73
OBMID-24	0.385	0.76

Results are pretty decent on the Quo Vadis dataset but quite disappointing on the two other datasets. On OBMID-24, our model performs poorly which can be interpreted as a low generalization capability or a possible overfitting problem. Figure 5 shows the confusion matrices for these tests. There are also a lot of malware files tagged as benign files, which is not acceptable in operational conditions.

In the next experiment, we train our model on B&PEML dataset with train-validation-test split of 70%-15%-15%, stratified sampling, and early stopping. This model gives train and validation accuracies of 99%.



(a) Number of tokens per sequence on Quo Vadis dataset.

(b) Number of tokens per sequence on B&PEML dataset.

Fig. 4: Number of tokens per sequence on both datasets.



(b) Test on the OBMID-24 dataset.

Fig. 5: Confusion matrices for training on the Quo Vadis dataset.

Results on B&PEML dataset are very good, but both F1-score and accuracy decrease significantly when inference is conducted on other datasets, which suggests a potential overfitting problem as in the first experiment. Figure 6 shows the confusion matrices for this second experiment.

Our last test consists in training our algorithm on both B&PEML and Quo Vadis datasets. After observing no significant changes in results, we choose to also use

TABLE IV: Results on different datasets using our model trained on B&PEML dataset only.

	F1-score	Accuracy
Quo Vadis	0.70	0.70
B&PEML	0.99	0.99
OBMID-24	0.63	0.85



(b) Test on the OBMID-24 dataset.

Fig. 6: Confusion matrices for training on the B&PEML dataset.

OBMID-24 as the validation subset. We get a train accuracy of 98.5%, and a validation accuracy of 95%. We test

TABLE V: Results with our model trained on the Quo Vadis and B&PEML datasets and validated on OBMID-24 datasets.

	F1-score	Accuracy
Quo Vadis	0.88	0.88
B&PEML	0.99	0.99

this model on B&PEML and on QuoVadis, with the same test samples as for the two previous experiments. Results are summarized in Table V. Note that we sidestep the overfitting problem by using validation data comprising very recent files, that were not processed by the model during training.

This training increases the performance of our model for both datasets, even if there is still a significant amount of false negatives according to Figure 7 when testing the model on Quo Vadis. We also tested this training on both datasets without the validation step on recent files, which did not give us significant improvements.



Fig. 7: Confusion matrix for training on the Quo Vadis and B&PEML datasets, validation on the OBMID-24 dataset and testing on Quo Vadis dataset.

#### B. Data analysis

In order to have a better understanding of our results, we conducted a deeper analysis of the different datasets used for our experiments. We compute the most recurrent trigrams per classes for both Quo Vadis and B&PEML datasets, to see if the distribution is the same or if some differences between classes could lead to some bias or to some relevant information. Figure 8 presents the trigram occurrences in both datasets.

We can observe on Figure 8a that the trigram *kernel32* getprocaddress 0x7000000 is the most common trigram in the malware samples of the Quo Vadis train set,

while it is only the third common trigram in the benign samples, with a third of the occurrences of the first trigram in this case. As a consequence, this trigram may be interpreted by our model as the main information related to the maliciousness of a file. In Figure 8b, we see that this API call is the most common in both malware and benign samples on B&PEML dataset. We suspect that this specific trigram might induce a bias when training our model on the Quo Vadis dataset, because the benign distribution of trigrams is not the same as its malware counterpart, for common API instructions. This potential bias could explain the aforementioned bad generalization capability when training our model on Quo Vadis and testing it on other datasets.

## C. Complementary analysis

We ran other tests on our model in order to challenge our findings. We trained our model with the API calls without the arguments, as shown in Figure 9a. When training on Quo Vadis, the results of [14] are confirmed with a 9% reduction in F1-score for the test on Quo Vadis, but on B&PEML F1-score drops to 20%. We also notice that the majority of binaries were classified as benign which is a major drawback. When training on B&PEML, the conclusions are fairly similar. The B&PEML test loses a few percentages but remains at 97% in terms of F1-score and accuracy. Results on Ouo Vadis dataset are somehow similar but on the OBMID-24 dataset, results drop to 72% in F1-score and 53% in accuracy. These results suggest that API calls arguments have a strong contribution and are therefore needed to generalize on data that differ from the training dataset.

We have also experimented a third preprocessing method based on the sequence of API calls, as shown in Figure 9b, adding tags such as [START] and [END] or [SEP] to separate API calls and arguments from each other. However, both modifications did not demonstrate any improvement from a performance standpoint. Worse, by adding the [SEP] tag every seven tokens on average and maintaining a length of 512 tokens, we are losing approximately ten instructions per report.

We ran Nebula code available on Github<sup>1</sup>, directly on the B&PEML dataset. The results are rather disappointing with a 40% F1-score and an accuracy of 57%, suggesting a possible generalization problem. It would be interesting to dig deeper in future works to gain a better understanding of these results.

<sup>1</sup>Nebula Github: https://github.com/dtrizna/nebula, July 2024



#### (a) Trigram occurrences in Quo Vadis.



(b) Trigram occurrences in B&PEML.

Fig. 8: Trigram occurrences for both dataset on the train samples.

['kernel32', '.', 'getmodulehandlea',
'user32', '.', 'getkeyboard', '##type',
'kernel32', '.', 'getcommandlinea',
'kernel32', '.', 'getstartupinfoa',
'kernel32', '.', 'getcurrentthreadid',
'kernel32', '.', 'getsystemdirectorya',
'kernel32', '.', 'initializecriticalsection',
'kernel32', '.', 'localalloc',
'kernel32', '.', 'virtualalloc',
'kernel32', '.', 'localalloc',
'kernel32', '.', 'virtualalloc',
<pre>'kernel32', '.', 'getmodulefilenamea', 'kernel32', '.', 'copyfilea']</pre>

(a) Data format after preprocessing without arguments in the sequences of API calls.

I	'[START]', 'kernel32', '.', 'getmodulehandlea', "(['", '0x0', "'])", '[SEP]',
	'user32', '.', 'getkeyboard', '##type', "(['", '0x0', "'])", '[SEP]',
	'kernel32', '.', 'getcommandlinea', '([])', '[SEP]',
	'kernel32', '.', 'getstartupinfoa', "(['", '0x1211f', '##58', "'])", '[SEP]',
	'kernel32', '.', 'getcurrentthreadid', '([])', '[SEP]',
	'kernel32', '.', 'getsystemdirectorya', "(['<", 'path', ">',", "'", '0x100', "'])", '[SEP]',
	'kernel32', '.', 'initializecriticalsection', "(['", '0x40', '##f5', '##b0', "'])", '[SEP]',
	'kernel32', '.', 'localalloc', "(['", '0x0', "',", "'", '0xff8', "'])", '[SEP]',
	'kernel32', '.', 'virtualalloc', "(['", '0x0', "',", "'", '0x10000', '##0', "',", "'", ])", '[SEP]',
	'kernel32', '.', 'localalloc', "(['", '0x0', "',", "'", '0x64', '##4', "'])", '[SEP]',
	'kernel32', '.', 'virtualalloc', "(['", '0x50000', "',", "'", '0x4000', "',", "'",])", '[SEP]',
	'kernel32', '.', 'getmodulefilenamea', "(['", '0x0', "',", "'<", 'path', ">',", "'", '0x105', "'])", '[SEP]',
	'kernel32', '.', 'copyfilea', "(['<", 'path', ">',", "'<", 'path', ">',", "'", '0x0', "'])", '[END]']

(b) Data format after adding tags as [START], [END] and [SEP]. Fig. 9: Two other preprocessing methods used in our experimentation.

## **III.** LIMITATIONS

Speakeasy is a valuable security solution since (i) it is capable of generating sandbox-like reports quickly and easily with a lot of quality information, (ii) it is an opensource project with a permissive software license (MIT license) and (iii) it is still active at the time of writing (July 2024). Nevertheless, it suffers a few limitations. As mentioned by their authors, it cannot fully emulate some files [33] since windows API are too many to be all emulated. A given emulation may also fail due to a missing expected environment by the sample under analysis. This comprises for instance the absence of specific files or registry keys and also unexpected data as returned by the emulation engine.

Our experiments faced these limitations with an impact on our computed model. Indeed, even if very few sequences from Speakeasy reports are longer than 500 tokens, we can see on Figure 4 that on both B&PEML and Quo Vadis datasets lots of sequences have no more than 50 tokens: 18% on Quo Vadis and 30% on B&PEML. Another meaningful metric is that 53% of sequences exhibit less than 150 tokens, due to the fact that the emulation tool stops when it encounters an API call that is not emulated. This leads to low detailed reports that are less valuable when training our model.

From an attacker perspective, this limitation is useful to break any Speakeasy analysis. An attacker would just have to call an unsupported API at the beginning of its malicious payload to bypass a detection algorithm, which would then analyze an insufficient detailed emulation report. Mitigation is still possible since emulated API handlers can be added by defining a Python function with the correct name and arguments even if this kind of mitigation is an unbalanced cat-and-mouse game. As a consequence, a detection model based on Speakeasy reports must not be used as the only line of defense in the context of PE files dynamic analysis.

# IV. DISCUSSION AND FUTURE WORKS

Our approach using an SLM for malware detection allows us to use AI in a responsible way, and forces us to get a better knowledge about our data in order to implement more precise preprocessings and identify valuable features.

According to our experiments, our model can really take advantage of being trained on a combination of datasets, while being challenged in its validation steps with more recent samples. We still need to conduct more investigations to improve our false-negatives ratio. We think this could be done for instance by cleaning and normalizing input data even more during the preprocessing step.

Some additional data analysis are also required to make sure that the distributions of API calls of our malware and benign samples are similar, in order to prevent any bias in the data.

The next step in our work consists in combining static and behavioral analysis into an "hybrid" model. It would first use a machine learning model based on static features to identify the maliciousness of a sample [34] then further analyze it using the Encoder-Only model in case this sample has not been classified with sufficient confidence. We expect this hybrid model to be more accurate than any model that would use either static or behavioral features. This hybrid model would also have the advantage of being rather small, which matches with our low carbon footprint side objective.

Even if Speakeasy suffers some limitations (cf. Section III), it still remains a very interesting tool that should benefit from being complemented by other dynamic analysis techniques. Combining and correlating security information from such other techniques is a promising research field for some future works.

We are also curious to assess the robustness of this Encoder-based model and of our future hybrid detection model when challenged with offensive tools such as Gym-Malware [35], Malfox [36], or our own adversarial model MERLIN [37]. These tools are capable of evading some static antivirus detection engines by altering some key characteristics of a given PE file. It would be interesting to evaluate the impacts of such alterations on both Speakeasy emulation reports and on our detection results.

## CONCLUSION

In this paper, we designed a Small Language Model leveraging an Encoder-only architecture, for malware detection based on behavior analysis. We deliberately limited the size of our model for carbon footprint considerations and ease of deployment in an operational context. These restrictions allowed us to identify several key factors for this task, such as the importance of an optimal tokenizer, the efficiency of the emulation tool that reports dynamic behavior and as usual, data quality. Our results are pretty decent but suggest that some improvements should be conducted to enhance the performance of our detection chain. Besides, comparing our results on several datasets seems to indicate a bias for the Quo Vadis dataset, which could explain some bad generalization capabilities of our model. Finally, a good practice to avoid overfitting and maintain good generalization capabilities could be to rely on very recent samples during the validation step.

## Acknowledgments

This work is supported by Orange FSI funding program coordinated by Adrien Servouze and Jean-Marc Blanco and also Orange Innovation SECMA research project led by Sok-Yen Loui. We would like to thank them for their valuable support.

#### REFERENCES

- Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, and Jianfeng Gao. Large language models: A survey, 2024.
- [2] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.
- [3] Farzad Nourmohammadzadeh Motlagh, Mehrdad Hajizadeh, Mehryar Majd, Pejman Najafi, Feng Cheng, and Christoph Meinel. Large language models in cybersecurity: State-of-theart, 2024.
- [4] Yin Minn Pa Pa, Shunsuke Tanizaki, Tetsui Kou, Michel Van Eeten, Katsunari Yoshioka, and Tsutomu Matsumoto. An attacker's dream? exploring the capabilities of chatgpt for developing malware. In *Proceedings of the 16th Cyber Security Experimentation and Test Workshop*, pages 10–18, 2023.
- [5] Julian Hazell. Spear phishing with large language models. arXiv, 2305, 2023.
- [6] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. Pentestgpt: An Ilm-empowered automatic penetration testing tool. arXiv preprint arXiv:2308.06782, 2023.
- [7] Matthew G. Gaber, Mohiuddin Ahmed, and Helge Janicke. Malware detection with artificial intelligence: A systematic literature review. ACM Comput. Surv., 56(6), jan 2024.
- [8] Durre Zehra Syeda and Mamoona Naveed Asghar. Dynamic malware classification and api categorisation of windows portable executable files using machine learning. *Applied Sciences*, 14(3), 2024.
- [9] Chani Jindal, Christopher Salls, Hojjat Aghakhani, Keith Long, Christopher Kruegel, and Giovanni Vigna. Neurlux: Dynamic malware analysis without feature engineering, 2019.
- [10] Zhaoqi Zhang, Panpan Qi, and Wei Wang. Dynamic malware analysis with feature engineering and feature learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(01):1210–1217, Apr. 2020.
- [11] Xiaohui Chen, Zhiyu Hao, Lun Li, Lei Cui, Yiran Zhu, Zhenquan Ding, and Yongji Liu. Cruparamer: Learning on parameter-augmented api sequences for malware detection. *IEEE Transactions on Information Forensics and Security*, 17:788–803, 2022.
- [12] Mandiant. Speakeasy: Windows kernel and user mode emulation. https://github.com/mandiant/speakeasy.
- [13] Dmitrijs Trizna. Quo vadis: Hybrid machine learning metamodel based on contextual and behavioral malware representations, 2022.
- [14] Dmitrijs Trizna, Luca Demetrio, Battista Biggio, and Fabio Roli. Nebula: Self-attention for dynamic malware analysis, 2023.
- [15] Sunita Tiwary. Small is the new big: the rise of small language models. https: //www.capgemini.com/insights/expert-perspectives/ small-is-the-new-big-the-rise-of-small-language-models, July, 22 2024. Published by Capgemini.
- [16] Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models, 2020.
- [17] Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driess-

che, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models, 2022.

- [18] Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks are all you need, 2023.
- [19] Yuanzhi Li, Sébastien Bubeck, Ronen Eldan, Allie Del Giorno, Suriya Gunasekar, and Yin Tat Lee. Textbooks are all you need ii: phi-1.5 technical report, 2023.
- [20] Matt Pietrek. An in-depth look into the win32 portable executable file format, part 2. *MSDN Magazine, March*, 2002.
- [21] Limin Yang, Arridhana Ciptadi, Ihar Laziuk, Ali Ahmadzadeh, and Gang Wang. BODMAS: An Open Dataset for Learning based Temporal Analysis of PE Malware. In *Proceedings* -2021 IEEE Symposium on Security and Privacy Workshops, SPW 2021, pages 78–84, 2021.
- [22] Michael Lester. Pratical Security Analytics PE Malware Machine Learning Dataset. https://practicalsecurityanalytics. com/pe-malware-machine-learning-dataset/.
- [23] Bhargav Chowdary Rayankula. An Evaluation and Performance study on BODMAS dataset for Malware Analysis. PhD thesis, Dublin, National College of Ireland, 2023.
- [24] Tran Hoang Hai, Vu Van Thieu, Tran Thai Duong, Hong Hoa Nguyen, and Eui-Nam Huh. A proposed new endpoint detection and response with image-based malware detection system. *IEEE Access*, 11:122859–122875, 2023.
- [25] Qikai Lu, Hongwen Zhang, Husam Kinawi, and Di Niu. Selfattentive models for real-time malware classification. *IEEE Access*, 10:95970–95985, 2022.
- [26] Matthieu Jimenez, Cordy Maxime, Yves Le Traon, and Mike Papadakis. On the impact of tokenizer and parameters on n-gram based code analysis. In 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), pages 437–448, 2018.
- [27] Mohammad Abdul Hadi, Imam Nur Bani Yusuf, Ferdian Thung, Kien Gia Luong, Jiang Lingxiao, Fatemeh H. Fard, and David Lo. On the effectiveness of pretrained models for api learning. page 309–320, 2022.
- [28] Rick Battle. Weaknesses of wordpiece tokenization. https://medium.com/@rickbattle/ weaknesses-of-wordpiece-tokenization-eb20e37fec99.
- [29] Mike Schuster and Kaisuke Nakajima. Japanese and korean voice search. In *International Conference on Acoustics, Speech* and Signal Processing, pages 5149–5152, 2012.
- [30] Hugging Face. Summary of the tokeniers. https://huggingface. co/docs/transformers/tokenizer\_summary.
- [31] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805, 2018.
- [32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.
- [33] Mandiant. Limitations. https://github.com/mandiant/speakeasy/ blob/master/doc/limitations.md.
- [34] Benjamin Marais, Tony Quertier, and Stéphane Morucci. Albased Malware and Ransomware Detection Models. In *Conference on Artificial Intelligence for Defense*, Actes de la 4ème Conference on Artificial Intelligence for Defense (CAID 2022), Rennes, France, November 2022. DGA Maîtrise de l'Information.

- [35] H. S Anderson, Bobby Filar, and Phil Roth. Evading Machine Learning Malware Detection. *BlackHat DC*, page 6, 2017.
- [36] Fangtian Zhong, Xiuzhen Cheng, Dongxiao Yu, Bei Gong, Shuaiwen Song, and Jiguo Yu. MalFox: Camouflaged Adversarial Malware Example Generation Based on C-GANs Against Black-Box Detectors. pages 1–14, 2020.
- [37] Tony Quertier, Benjamin Marais, Stéphane Morucci, and Bertrand Fournel. Merlin – malware evasion with reinforcement learning, 2022.