Fine Tuning LLMs vs Non-Generative Machine Learning Models: A Comparative Study of Malware Detection

Gheorghe Balan[1,2], Ciprian-Alin Simion[1,2] and Dragoş Teodor Gavriluţ[1,2]
[1] *"Al.I. Cuza" University, Faculty of Computer Science, Iasi, Romania*
[2] *Bitdefender Laboratory, Iasi, Romania*
{*asimion, gbalan, dgavrilut*}*@bitdefender.com*

Keywords: Fine Tuning LLMs, Neural Network, API Sequence, Malware Detection.

Abstract: The emergence of Generative AI has provided various scenarios where Large Language Models can be used to replace older technologies. Cyber-security industry has been an early adopter of these technologies, but in particular for scenarios that involved security operation centers, support or cyber attack visibility. This paper aims to compare how well Large Language Models behave against traditional machine learning models for malware detection wrt. various constrains that apply to a security product such as inference time, memory footprint, detection and false positive rate. In this paper we have fine tuned 3 open source models (LLama2-13B, Mistral, Mixtral) and compared them with 18 classical machine learning models (feed forward neural networks, SVMs, etc) using more than 135,000 benign and malicious binary samples. The goal was to identify scenarios/cases where large language models are suited for the task of malware detection.

1 INTRODUCTION

The rise of Generative AI has opened multiple possibilities in terms of automation that allowed cyber-security vendors to use Large Language Models for tasks like:

- support
- attack visibility and explainability
- security operation centers

In most cases, these models are used as a second opinion in security operation centers or to validate detections given by other technologies.

However, since this type of models rely on vast datasets for their training, they have the potential to be used in other security-related scenarios. One thing that needs to be taken into consideration is that these models rely on various forms of natural language and as such they are more likely to provide a proper result for data presented in the same way.

In terms of malware identification, this could be obtained using a list of API calls (as they would reflect the behavior of a malware). This input could also be obtained from technologies that already exist in a cyber-security technology stacks such as sandboxes or emulators.

On the other hand, there are several constraints that various technologies in a security suite have (such as performance, detection rate (i.e. recall), inference time, etc). It is also important to notice that the temperature hyper-parameter (specific to LLMs) might not be that useful in a situation where a deterministic result is required (for example in cases where detection validation or various QA[1] processes are required).

Another relevant aspect is that while LLMs started as cloud services, there are several models[2] that can be tested locally. This potentially reduces the service cost and various privacy issues that came with a cloud-based model.

With this in mind, this paper attempts to evaluate if such out-of-the-box models and a fine-tuned version of them can actually be used for malware detection and in what capacity.

The evaluation is done taken into consideration various constraints and limitations of current detection technology stacks. We focused on API calls as they describe the way a program works (and assuming various description were part of the training dataset, using them might provide a model with enough information to infer the maliciousness quality of a file).

It is also important to evaluate if these large language models are comparable with existing machine

[1] quality assurance
[2] https://huggingface.co/models

Balan, G., Simion, C.-A. and Gavriluţ, D. T.
Fine Tuning LLMs vs Non-Generative Machine Learning Models: A Comparative Study of Malware Detection.
DOI: 10.5220/0013177300003890

In *Proceedings of the 17th International Conference on Agents and Artificial Intelligence (ICAART 2025) - Volume 3*, pages 715-725
ISBN: 978-989-758-737-5; ISSN: 2184-433X

learning technologies. From this perspective we elaborated an experiment where multiple non-generative machine learning models are evaluated against 3 large language models over a sample set of malicious and benign files that were executed in an emulator so that the ordered list of API calls could have been extracted.

The comparison takes into consideration various metrics such as the detection rate (i.e. recall), false positive rate, inference time and memory footprint.

The rest of the paper is organized as follows: Section 2 presents similar research, Section 3 explains the problem we are tackling in this paper, Section 4 and Section 5 present our experiment, Section 6 discusses the results of our experiment and finally Section 7 draws some conclusions.

2 RELATED WORK

As with all other fields, cyber-security researchers got very excited when seeing LLMs' capabilities and tried to use them as creatively as possible. Even if there are a lot of papers discussing the use of LLMs in an offensive manner (Charan et al., 2023), (Karanjai, 2022), (Botacin, 2023), (Pa Pa et al., 2023) we will intentionally leave them out as this paper focuses on using them to test their malicious behaviour detection power.

The authors of (Motlagh et al., 2024) have gathered multiple papers that focus on protecting, defending, detecting, but also on adversarial uses of LLMs. A very interesting analysis they have done is counting the number of papers published by the function presented. Their research shows that by the time of their publishing there were at least 30 papers that were focused on offensive functions like Reconnaissance, Initial Access, Execution, Defense Evasion or Credential Access. All of these are MITRE ATT&CK techniques[3].

On the other hand, the papers that focus on defensive techniques are spread across directions like Identify, Protect, Detect, Respond and Recover, as the authors (Motlagh et al., 2024) show they have found over 30 papers that delve in the techniques aforementioned as well.

One great application of LLMs is with web content filtering as shown in (Vörös et al., 2023). In this paper the authors show how they achieved better results (up to 9% increase in accuracy) using LLMs when compared to standard deep learning algorithms. In addition, they showed how they fine-tuned a large language model using only 10000 samples and achieved better performance than the current state-of-the-art solutions that was trained on 10 million samples. One of the biggest problems with this kind of algorithms is even in their name: they are large, sometimes too large to make them usable in a practical scenario. Probably the most impressive result of this paper (Vörös et al., 2023) is that using a smaller model (175 times smaller) they attained performance levels comparable to the original LLM (770 million parameters). Some of the models they used are BERT (Devlin et al., 2019), eXpose or GPT-3 Babbage.

A lot of malware, especially zero-day threats, leverage the use of exploits in legitimate software. These exploits end up in these programs most often by mistake. This paper (Omar, 2023) proposes a new framework named *VulDetect* tasked with detecting vulnerable code. The framework utilizes GPT-2, BERT and LSTM to detect vulnerabilities in C, C++ and Java code by employing Knowledge Distillation in a teacher-student configuration. The results were compared with VulDeBERT (Kim et al., 2022) and LSTM, all trained and tested on four datasets of vulnerable code SARD (Zhou and Verma, 2022), SeVC (Shoeybi et al., 2020), Devign (Zhou et al., 2019) and D2A (Zheng et al., 2021). Their best performing model was the one based on GPT-2 with 93.59% accuracy when tested against SARD dataset.

Another interesting research done by Rahali et. al. proposes a malware detecting framework named MalBERT (Rahali and Akhloufi, 2021). In their experiments they used a dataset of Android APK files that they downloaded and processed by extracting the AndroidManifest.xml and removing unnecessary information from it. They started from over 13 million files (APKs) from the Androzoo public dataset and, after processing, ended up with 12K benign samples and 10K malware samples. The authors pursued two endeavours, binary classification (malware/benign) and multi-classification(ex. spyware, dropper, clicker, etc.). When compared with LSTM, XLNet (Yang et al., 2020a), RoBERTa (Liu et al., 2019) and DistilBERT (Sanh et al., 2020), the best performing model was BERT (Devlin et al., 2019), in both binary classification and multi classification. On the first task, BERT achieved an accuracy of 97% with the next model, XLNet, at 95%. On the multi classification task, BERT attained an accuracy of 91% with the next model, LSTM, at 85%.

When it comes to analysing malware samples one of the most used formats is Application Programming Interface (API) sequences. The SLAM (Chen et al., 2020) (Sliding Local Attention Mechanism) framework takes advantage of just that, and more. After the

[3]https://attack.mitre.org/matrices/enterprise/

authors get a handle on the API sequences for their samples, they also categorize them by behaviour in 17 categories. After that, they construct these 2 dimensional input vectors containing the API sequence (numbers) and the category index sequence. The researchers argue that by doing so, the sample embeds a stronger correlation between semantics and structural information. The dataset contained 110K benign samples and 27K malware samples. Each sample was truncated at 2000 APIs or padded with 0's at the end of the sequence to match that size. The proposed framework involved five steps:

- Splitting the input vector
- Initializing the local attention window from the split
- Running the Convolutional Neural Network training step
- Concatenating the results
- Applying Softmax on the concatenation

Finally, they used Random Forest (RF), Attention CNN LTSM (ACLM) (shiqi, 2019) and a two-stream CNN-Attention model (TCAM) (Yang et al., 2020b) to compare SLAM to. After a 10 fold cross validation, their research shows that the SLAM framework attained an average accuracy of 97% with the next model being TCAM with 92%.

3 MALWARE DETECTION CHALLENGES

Cyber-security has always been depicted by the cat-and-a-mouse game between security products and malware writers where each one reacts to the changes added by the other. The advancement in machine learning field in the last decade indicated a potential edge for the security vendors as complex neural networks architecture could be harder to bypass by a malware. However, even if in theory this seems to be correct, the same advancement in GANs (Generative Adversarial Networks) balanced the field.

We evaluate a machine learning model using two metrics (related to cyber-security):

- **proactivity** - the ability to detect new malware samples that are discovered long after the model was trained.
- **genericity** - the ability to detect new samples that have nothing or very little in common with the samples used in the training set.

Those two metrics are always strongly linked to the input data. This means that on one hand, using a relevant feature set (for example behavior information) could potentially create a model that is harder to evade. On the other hand, knowing certain limitations might allow you to find exactly what kind of things you need to change to a malware sample to avoid detection. With this in mind, let's enumerate certain constraints that are required for a model to be used in practice:

- **inference speed**
- **false positive rate**
- **detection rate**
- **memory footprint**
- **model update size**

Some of these constraints are correlated with the way a model is being used, as follows:

- using a model for **real-time protection**[4] implies that a model must be fast (in terms of inference) since access to the scanned object is locked until the verdict from the model is received. Usually this implies smaller models. It also implies a lower (close to 0) false positive rate as blocking a clean object might have a serious impact (e.g., blocking the access to a system file might block the entire endpoint).
- a **on-premise model** requires a small memory footprint (as one need to be certain that model runs on multiple architectures with limited resources in some cases). This also implies a reduced update size.
- in contrast, a **cloud model** is not limited by size or hardware architectures. However, using a cloud model implies you can not scan all accessed objects as one needs to take into consideration the time needed to connect to the cloud. This means that a cloud model is not always a good choice for real-time protection - where all accessed objects have to be scanned.

The more a product relies on real-time protection, the more attacks are being stopped before they happen, but models have to be small to achieve a good inference time as well as small memory footprint and reduced update size. This is specific to the anti-malware protection component of a security product. However, using larger models implies scanning objects asynchronously and as such lose the protective capabilities. At the same time, using a cloud model is not limited by memory size, architecture or update

[4]block access to an object until its scan is complete and delete the object afterwards if it is deemed malicious

requirements and can be harder to evade. This is usually the case with security analytics components such as EDR[5] or XDR[6].

With this in mind, we are attempting to validate if large language models can be used in any of the previously described scenarios. Due to their complex architecture we expect them to be more resilient on evasion techniques and as such provide a better **proactivity** and **genericity** in terms of identifying new threats. At the same time, all of the above restrictions must be preserved leading to the question whether these models can be truly used for threat identification and if so, in what capacity?

4 DATABASES

LLMs inference time for each sample is relatively long for a real-world scenario where a decision should be taken as fast as possible. Therefore it is more suitable to use them as an additional decision layer, where the previous layer(s) will likely filter out common benign files (for speed improvements). With this in mind, we wanted to mimic such a training environment where the number of benign samples to be processed is lower than the one of malicious samples. Hence, our *initial-api-sequences-database* consists of sequences of APIs extracted from 136,383 samples (58,472 benign and 77,911 malicious). This initial file database was split in two smaller file-databases (*training-api-sequences-database* - 38,472 benign / 57,911 malicious and *testing-api-sequences-database* - the remaining 20,000 for each class).

The APIs were extracted using a proprietary emulator provided by a security company. The average number of APIs extracted for clean samples is 276 while for malicious samples is 1392. As an observation, malicious files yielded more APIs during the emulation phase.

Moreover, differences between benign and malicious files can also be observed by looking at the top ten APIs extracted for each class (Table 1 and Table 2).

Two particular APIs stood out in the malicious dataset, *kernel32_ReadFile* and *kernerl32_SleepEX*. This is due to how malicious files are often implemented. Usually an attacker tends to evade automated analysis by implementing long sleep (now an obsolete technique) and multiple read operations in order to gather data from running environment (used mostly by Ransomware, Password / Data Stealers).

Table 1: Top ten APIs seen in benign train dataset.

#	API	Count
1	kernel32_FlsGetValue	2015725
2	kernel32_HeapFree	1177867
3	kernel32_GetProcAddress	1121622
4	kernel32_TlsGetValue	773809
5	kernel32_SetLastError	630800
6	kernel32_WriteFile	350663
7	kernel32_MultiByteToWideChar	215998
8	kernel32_ReadFile	199316
9	kernel32_WideCharToMultiByte	188462
10	kernel32_lstrcmpiW	185232

Table 2: Top ten APIs seen in malicious train dataset.

#	API	Count
1	kernel32_TlsGetValue	4848697
2	kernel32_lstrcpynA	4200079
3	kernel32_FlsGetValue	3155210
4	kernel32_GetProcAddress	3030600
5	kernel32_SleepEx	2492721
6	msvbvm60___vbaFreeVar	2239934
7	kernel32_ReadFile	2126757
8	kernel32_HeapFree	1956913
9	oleaut32_SysFreeString	1857905
10	kernel32_WriteFile	1444253

Our databases were next used as follows:

- step1 - obtained a new database *training-llm-vt-detecting-engines-count* by acquiring the number of VirusTotal engines[7] detecting each sample in *training-api-sequences-database*; for each sample we also stored the sequence of API Calls; this database will be used to fine tune LLMs
- step2 - *testing-api-sequences-database* - the sequences were fed to fine tuned LLM models and the results were saved (*llm-results-database*)
- step3 - applied a feature selection algorithm over the *training-api-sequences-database* and created a new database (*training-api-sequences-feature-database*) which contained for each sample only binary values (1 if the selected feature is found in the API sequence list, 0 otherwise)
- step4 - trained and tested several machine learning models; testing the resulted models on *testing-api-sequences-database*; stored obtained results in *ml-models-results-database*
- step5 - compared the obtained results (*ml-models-results-database*, *llm-results-database*)

[5]Endpoint Detection and Response

[6]Extended Detection and Response

[7]https://www.virustotal.com/gui/home/search

5 EXPERIMENT SETUP

We divided our experiment in two larger parts: the first part, where we fine tune and evaluate 3 open source Large Language Models (LLama2-13b, Mistral-7b-v03 and Mixtral-8x7b-v.01) and one where we looked into non-generative machine learning models that can be used for malware detection. In both cases we evaluated the detection rate (recall), false positive rate and inference time. For both fine-tuned LLMs / ML models, we use the same database, *training-api-sequences-database*, to fine-tune / train and *testing-api-sequences-database* to test the obtained LLMs / ML models.

The experiment was conducted on a virtual machine with 4 RTX 4090TI GPUs, 128 Gi RAM, 8 vCPU, and 512Gi storage.

5.1 Fine Tuning LLMs

The Large Language Models used in this experiment were: LLama2-13b (Rozière et al., 2024), Mistral (Jiang et al., 2023), Mixtral (Jiang et al., 2024). Each of these models were pulled from the HuggingFace[8] repository and deployed locally.

To interact with these models we first constructed the prompts. After some empirical tests with different forms of prompts where we requested qualifiers, simple binary verdicts or even class attribution confidence percentages, we settled with asking for a number on a scale from 0 to 9 where 0 is unlikely malware and 9 likely malware. We constructed the prompt by concatenating 3 parts.

The first part, the prefix, was this: *"Given the following API call sequence: "*. The second part, the api_seq, was obtained for every sample's sequence by taking each API call name and stripping its class name, resulting only in the function name. We stripped them of the class name for two reasons: minimizing the size of the prompts and having as little repetitive strings as possible in the prompt. After obtaining all the function names(acledit_EditAuditInfo $\longrightarrow$ EditAuditInfo), we then added them into a sequence separated by a commas, whilst maintaining their original order. Lastly, we added the suffix, where we asked the models to respond with a digit: *"On a scale from 0 to 9 where 0 is very unlikely and 9 is very likely, how likely is it that this sequence belongs to a malware file? respond with a single digit. Don't provide additional information, just the digit. Even if you are not sure, just provide a digit."*.

Even if the number of available VirusTotal engines is 101[9] we found out that in our database *training-llm-vt-detecting-engines-count* the maximum number of detecting engines is 73. Moreover, we have to consider that from time to time an antivirus engine might generate false positives, therefore a clean file might get detected by a small number of engines. On the other hand, some newly malicious files are firstly detected by few antivirus engines. Hence, in order to down-scale these scores to [0, 9] and keep a balanced approach, we used the following formula: $max(min(int(vt\_detecting\_engines\_count/7.5), 9), 0)$ (Table 3)

Table 3: Train score distribution.

Score	Count	Score	Count
0	39009	5	7183
1	1371	6	12961
2	1978	7	17879
3	2972	8	7578
4	5291	9	161

The score was appended to the prompt as the expected answer ("Answer: {digit}"). Due to training environment constraints we decided to fine tune on maximum 4096 tokens; Hence, the length of API sequences was limited in order to fit this restriction, keeping only the first aprox. 4000 APIs.

The fine tuning process was implemented in Python and made use of *transformers*, *LoraConfig* and *peft* HuggingFace Python modules.

- Each of the LLM **models** were loaded in 4bit, with double quant, quant type to "nf4" and compute type to bfloat16, being mapped on all 4 GPUs.
- From the **tokenizer** perspective, for each prompt we added the *bos* and *eos* tokens and a padding (with eos) to the right to ensure the fixed tokens length of 4096. This is necessary to ensure a smooth fine tuning process.
- For **LoraConfig** we decided to set bias to **all** as we are working with sequences of APIs and each token might be important. Rank was set to 32, Alpha Parameter to 64, dropout to 0.05 and task_type "CAUSAL_LM". For **llama2** and **mistral** we set the target modules to q_proj, k_proj, v_proj, o_proj, gate_proj, up_proj, down_proj, lm_head. For mixtral, we replaced gate_proj, up_proj, down_proj with w1, w2 and w3.
- For **TrainingArguments** we used a train batch size of 3 with 2 steps gradient accumulation, a small learning rate of 2.5e-5, bf16, and

[8]https://huggingface.co/

[9]https://virustotal.readme.io/docs/list-file-engines

"paged_adamw_8bit" as suggested in HuggingFace docs[10]. The number of steps was set to 3750.

The times needed to fine-tune the models are the following: *Llama2-13b - 62h 14m*, *Mistral-7b - 35h 28m*, *Mixtral-8x7b - 46h 50m*.

5.2 Testing Fine Tuned LLMs

In the testing phase we used the same prompt architecture, this time applied on *testing-api-sequences-database*. For each call to the LLMs we gathered the response and the duration of the call. We then needed to parse the textual response so we could extract a numerical verdict. We did so in two steps. The first step was to apply a regular expressions on the response received, "Answer (\d{1})". If no valid match was found, we then moved to the second step were we applied two more generic regular expressions: "(\d{1})", "([0|1|2|3|4|5|6|7|8|9]{1})". If we still did not get any matches, we moved on considering that this response was unusable and discarded it.

At this point, we had a relation between a sample, a model, a numerical verdict and the time needed to evaluate all the samples. With all these, we calculated false positive rate, detection rate (i.e. recall) and average response time for every model.

In an effort to improve the LLM results we also compounded their individual results in three voting mechanisms:

1. Average - in this approach we choose the result to be the average value of all 3 models
2. Based on majority - in this system we went through all samples and checked all 3 models for their numerical verdict on the 0 to 9 scale. In the case of two or three models with the same numerical verdict we would select that. This decision would help limit the impact these models would have in a real life scenario.
3. Based on Veto Mechanism - in this approach we looked for at least one model's numerical verdict to be in the interval (0-t for clean or t-9 for malware, the *t* is the threshold value).

Like for the individual models, we also computed the results of the voting systems with respect to 8 selected threshold values.

[10]https://huggingface.co/docs/transformers/v4.29.1/en/perf_train_gpu_one

5.3 Non-Generative Models: Feature Mining and Selection

In order to use the traditional, non-generative models, we have to extract specific features from API Calls sequences. Hence, from a feature mining perspective a 3-steps process was implemented:

1. from training files database (*training-api-sequences-database*) three boolean features types were derived:
   - *traditional features* - 1 if a certain API was found in the API sequences, 0 otherwise.
   - *mapping features* - 1 if two specific APIs was found in the API sequence, 0 otherwise.
   - *sub-sequence features* - 1 if a sub-sequence of length 2 was found in the API sequence, 0 otherwise.
2. next, the obtained features were sorted by using a F2 metric score ($round(5.0 * F_i[malicious]) / (5.0 * F_i[malicious] + 4.0 * (total\_benign - F_i[benign]) + F_i[benign]) * 100.0, 2)$ - where $F_i[malicious]$ denotes how many malicious files contain F_i feature, $F_i[benign]$ denotes how many benign files contain F_i feature and $total\_benign$ the total number of benign files; *round* is a function to round the obtain value at two decimals.
3. based on the previous work done by (Balan et al., 2023) we have limited the number of features used to validate our non-generative machine learning models at 600. The dataset containing samples with these 600 features will be further refered to as *training-api-sequences-feature-database*).

After the first step from the described methodology, we resulted in 2730 traditional features, 677716 mapping features and 55784 sub-sequence features of length 2, for each file. At the end of our feature selection algorithm, the resulted database contained samples with 76 traditional features, 510 mapping features and 14 sub-sequences features.

5.4 Non-Generative ML Models

A number of 18 Machine Learning configurations were used to validate our approach. To implement the models we used sklearn-python package[11] and xgboost[12]. For each model we used a 3-fold cross-validation approach. We have two reasons behind this approach:

[11]https://scikit-learn.org/stable/

[12]https://xgboost.readthedocs.io/en/stable/python/python_api.html

- We wanted to check the variance between different results obtained from the 3 validation splits. This would outline if the model is highly sensitive to training data and if it is prone to over-fit. It was also a good way to identify if potential outliers reside in our dataset.
- As we trained multiple models, training time was something we also needed to take into consideration. A larger fold number (e.g., 20 folds) would have increased the training time and associated costs.

From a time perspective, each model's training time was negligible (only a matter of seconds). We used a virtual machine with 6vCPUs, 72Gi and 1 RTX 2080 Ti.

- *XGBoost* - **XGB** - as shown in many conducted studies, XGBoost should obtain good results in binary classification problem; the authors showed in (Li et al., 2024) that it can be used to slightly reduce the number of false positives; in our implementation, we decided to keep the default parameters.
- Multinomial Naive Bayes (*MultinomialNB*) - **MNB** - if we are to consider the API sequence as a story that a sample tells us, then MultinomialNB might yield decent results as shown in (Singh et al., 2019); on the other hand, from previous similar research (Balan et al., 2023) the authors stated that MultinomialNB model is not performing well on APIs; however, we decided to keep it so we would have a large number of models to compare LLMs to; similarly, we kept the default parameters as given by the python scikit-learn package implementation.
- *Logistic Regression*, *Support Vector Machines*, *Decision Trees* and *Random Forest* are ones of the most tested models in the malicious software detection problem (Senanayake et al., 2021); We made the following variations in hyper parameters:
  - *LogisticRegression* - **LR1** - max_iter set to 1000 and l2 regularization.
  - *LogisticRegression* - **LR2** - max_iter set to 100 and l2 regularization.
  - *LinearSVC* - **SV1** - default sklearn implementation, with C=0.0001.
  - *LinearSVC* - **SV2** - default sklearn implementation, with C=0.001.
  - *DecisionTreeClassifier* - **DTG** - criterion set to gini
  - *DecisionTreeClassifier* - **DTE** - criterion set to entropy.
  - *RandomForestClassifier* - **RF1** - n estimators set to 30, max depth set to 9.
  - *RandomForestClassifier* - **RF2** - n estimators set to 50, max depth set to 12.
- a *BaggingClassifier* - **BGL** - applied over a DecisionTreeClassifier with gini criterion; when compared to multiple linear regression model-based classifiers, Bagging-DT scored almost the best accuracy as shown in (Şahın et al., 2022).
- an *AdaBoostClassifier* - **ADB** - applied over a DecisionTreeClassifier with gini criterion; AdaBoost has been widely used in malware detection problem; recent research (Al-haija et al., 2022) shows how it can outperform state-of-the-art models.
- a *VotingClassifier-Hard* - **VCH** - (voting hard) which has all the above defined models as estimators; similar research (Bakır, 2024) yielded good results.
- a *CustomOneSideVotingClassifier-Benign* - **VCB** - a custom implementation of voting classifier; if at least one classifier yields a benign prediction then the file is classified as benign; applied on the same models as *VotingClassifier-Hard*.
- a *CustomOneSideVotingClassifier-Malicious* - **VCM** - a custom implementation of voting classifier; if at least one classifier yields a malicious prediction then the file is classified as malicious; applied on the same models as *VotingClassifier-Hard*.
- *Neural Networks* implemented in three different configurations
  - *LegacyNN* - **LeN** - and *LightNN* - **LiN** - as defined in (Balan et al., 2023)
  - A *ThirdNN* - **TNN** - 2 hidden layers of 32 and 16 with 'ReLU' activation, output layer with 'sigmoid', 'RMSProp' as optimizer and 'binary_crossentropy' for the loss function.

These models were used with the features resulted after applying the feature selection method described in the previous subsection.

6 RESULTS

6.1 Fine Tuned LLMs

After concluding with all processing we obtained 39956 usable responses for LLama2-13b, 39961 for Mistral, 39958 for Mixtral.

Given the way we constructed our prompt, in order to analyze our results, we needed a way to clearly

separate the malicious verdicts from the benign ones. To do this, we chose *1,2,3,4,5,6,7,8* as thresholds. Everything equal or above the threshold was considered as malicious and everything below was considered benign. Tables 4, 5, 6, 7, 8, 9, 10 show the accuracy, detection rate (i.e. recall) and false positive rate for each considered threshold and each Large Language Model.

Table 4: Llama2-13b Results.

Threshold	Acc	Recall	FPR
1	82.25	**99.06**	34.54
2	**82.95**	98.79	32.87
3	82.85	97.82	32.1
4	82.72	94.74	29.29
5	80.69	85.53	24.13
6	76.71	71.23	17.83
7	68.9	41.82	4.06
8	54.03	8.07	0.09

Table 5: Mistral-7B Results.

Threshold	Acc	Recall	FPR
1	**91.02**	**99.19**	17.13
2	90.87	98.87	17.12
3	90.65	98.38	17.06
4	90.27	97.57	17.02
5	89.82	96.31	16.66
6	85.32	83.94	13.3
7	70.3	41.15	0.6
8	52.75	5.42	0.0

Table 6: Mixtral-8x7B Results.

Threshold	Acc	Recall	FPR
1	86.67	**99.16**	25.8
2	86.78	99.15	25.55
3	**86.79**	99.15	25.55
4	86.76	98.87	25.33
5	84.08	90.72	22.54
6	78.57	78.85	21.71
7	68.74	39.74	2.33
8	52.13	4.16	0.0

One noticeable observation here is that none of the models were able to obtain a proper balance (in terms of practical usage) between detection rate and false positive rate. For example, Llama2-13b obtained 99.06% Recall for *t=1*; at the same time the false positive rate is 34.54% (making it unfeasible for practical usage). On the other hand, a model with a low false positive rate (0.6%) such as Mistral-7b (*t=7*) has only managed to obtain a recall of 41.15% (also not good enough for industry detection standards).

Table 7: Average Results.

Threshold	Acc	Recall	FPR
1	80.62	**99.69**	38.41
2	83.53	99.57	32.47
3	88.01	99.14	23.11
4	**89.52**	97.64	18.58
5	89.26	90.24	11.71
6	79.02	64.69	6.68
7	64.47	29.01	0.14
8	51.11	2.12	0.0

Table 8: Majority Results.

Threshold	Acc	Recall	FPR
1	**90.5**	**98.81**	17.8
2	90.43	98.54	17.66
3	90.15	97.8	17.48
4	89.3	95.7	17.09
5	85.61	86.93	15.72
6	81.42	77.23	14.39
7	69.1	39.12	0.97
8	52.29	4.49	0.0

When it comes to response times, Table 11 shows the average time per request and the total time for a model.

6.2 Non-Generative ML Models

By applying the training methodology described for ML Models we managed to obtain (Table 12) the best Accuracy for DTE - **96.98%**. However, this is not necessarily the model that one may choose in a real world scenario. Depending on one's goal, it may be more suitable to use **VCM** or **VCB** as the best Recall was obtained by **VCM** (98.64%) and the lowest FP Rate by **VCB** (1%). However, it is important to keep in mind that both of these models (**VCM**, **VCB**) are directly dependent on the other models and this comes with an increase in evaluation time and total used model bandwidth.

The worst Accuracy is obtained by **MNB** and having no other strength points, it is clear that this model is not suited for solving this problem.

6.3 Comparison

What follows is a comparison between the generative and the non-generative models' results, comparing them on each of the constraints we stated in 3:

- **Inference Time** - When it comes to inference time, a non-generative model has a negligible response time (tens of milliseconds) whilst the LLMs response time ranged between 4 and 13

Table 9: Veto Clean Results.

Threshold	Acc	Recall	FPR
1	**92.42**	**98.4**	13.54
2	92.27	97.85	13.29
3	91.82	96.44	12.8
4	90.54	92.73	11.65
5	85.41	79.67	8.86
6	78.03	62.41	6.39
7	64.05	28.13	0.1
8	51.11	2.12	0.0

Table 10: Veto Malicious Results.

Threshold	Acc	Recall	FPR
1	79.94	**99.69**	39.77
2	80.54	99.69	38.58
3	80.63	99.69	38.4
4	81.13	99.66	37.36
5	**82.02**	98.72	34.66
6	80.94	92.3	30.4
7	74.39	54.62	5.87
8	55.51	11.03	0.08

seconds. As previously mentioned, in the context of blocking opening or executing a file on an endpoint, a security product can not have a time impact bigger than hundreds of milliseconds at most. Besides the actual inference time, one must take into account the round-trip of the HTTP call in the case of a cloud deployment. Summarizing, in both cases the LLMs are not suitable.

- **Detection Rate (Recall)** - From detection rate perspective, all LLMs configurations are scoring higher results than traditional models; However, their suitability for a real-life scenario is highly dependent on an additional method to lower the number of false positive; On the other hand, the non-generative models achieve similar results as the ones obtained by LLMs in terms of recall but with a visible lower false positive rate.
- **False Positive Rate** - Sometimes more important than the detection rate, is the false positive rate. If the impact of not blocking a malicious sample may have either small or big impact on an endpoint, the impact of blocking a clean sample can have a devastating impact on a device. For example, if that blocked clean file is an operating system file, that endpoint may be rendered unusable. Looking at the results obtained by the LLMs in our experiment, the false positive rate is acceptable only for **LLama2-13b** with $t = 8$, **Mixtral-7b** with $t \in \{7,8\}$, **Mixtral-8x7b** with $t = 8$, **Average** with $t \in \{7,8\}$, **Majority** with $t \in \{7,8\}$, **Vote Malicious** with $t = 8$ and **Vote Clean** with $t \in \{5,6,7\}$ but in all cases the detection rate is not.

  In contrast, in the case of the non-generative models, almost all results are single digits and of these, almost half have a false positive rate lower than 5%.
- **Memory Footprint** - LLMs require a lot of memory (even if quantizated). While this is not a problem if the model is executed on a server or a cloud service where usually such resources (RAM) are available, one can not assume that each endpoint will have sufficient resources (in terms of memory) to allow such a model to run. As such, it using them on low-end devices or on general on devices where the amount of memory is unknown does not seem feasible.
- **Model Update Size** - With larger size come larger updates. While this is not a problem for a cloud service (where you only need to update once) it is a problem if the model is distributed locally (especially if the number of customers that are using the models is large - e.g., millions). This means that each time you have an update for a model each on

Table 11: LLM 1 Request Average duration and Total Duration for each Model. A=Llama2-13b, B=Mistral, C=Mixtral.

	Malicious		Benign	
Model	1-Avg.	Total	1-Avg	Total
A	5s	27h45m	6s	33h
B	6s	33h	4s	22h
C	12s	67h	12s	67h

Table 12: ML Models results sorted by ACC Desc.

Mdl	Acc	Recall	FPR	F1	F2
DTE	**96.98**	96.83	2.87	**96.97**	96.89
XGB	96.96	97.26	3.34	96.96	**97.15**
DTG	96.93	96.78	2.93	96.92	96.84
BGL	96.92	97.06	3.23	96.92	97.01
LeN	96.62	96.55	3.31	96.62	96.57
VCH	95.94	95.68	3.8	95.92	95.77
RF2	95.82	96.16	4.53	95.83	96.03
TNN	95.38	94.57	3.82	95.34	94.88
LiN	94.78	95.66	6.1	94.84	95.32
LR1	93.12	95.1	8.86	93.24	94.34
RF1	93.01	95.79	9.77	93.19	94.73
LR2	92.9	95.18	9.39	93.05	94.32
SV2	92.59	94.41	9.22	92.72	93.72
ADB	92	95.2	11.19	92.24	93.99
SV1	91.38	93.68	10.9	91.57	92.82
VCB	89.18	79.35	**1**	87.99	82.6
VCM	88.18	**98.64**	22.27	89.29	94.68
MNB	85.17	81.79	11.46	84.64	82.91

of them will have to download that update (and there is a price for the bandwidth that in this case will not be insignificant).

Moreover, comparing our results with similar research done in (Sánchez et al., 2024), we can observe that by fine-tuning Large Language Models, the accuracy value increases. For example, using transfer learning, they obtained an accuracy value of *58.17%* for Mistral with a context window size of 8192. Compared to their result, after fine-tuning, we managed to obtain an accuracy of *91.02%* for a threshold value *1*, with only 4096 tokens. However, their best-model, *BigBird*, with a context size of 4096 scored an accuracy value of *86.67%* which is close enough to the results obtained by our models.

7 CONCLUSION

In terms of **real-time protection** large language models are not suited (at least for the moment) for this task. The main disadvantages are (in order):

1. Long inference time (in these cases, the inference process should not take more than a couple of milliseconds)
2. Detection (recall) and False positive rate (in particular false positive rate should be close to 0)
3. Memory footprint (a decent model requires a lot of memory that most consumer endpoints do not have)
4. Cost (for scenarios where the models are stored locally and updates are needed, the cost will increase linearly with the number of customers)

With the advancement of the NPU[13] and combined with fine-tuning LLM models for specific detection tasks most of the previous disadvantages might be solved. For the moment non-generative models seem to produce better results for this type of scenarios.

However, we consider that LLMs can be successfully used as an additional detection layer in a threat detection environment where the inference time and false positive rate could be negligible; For example, such solutions might be deployed in a SandBoxed environment where the time needed to draw a conclusion is a matter of seconds/minutes. Moreover, in a SandBoxed execution, multiple techniques to identify benign files might be deployed in order to reduce the FP rate.

In terms of a model for security analytics platform (EDR, XDR or SIEM) these models can be a good option, but only after fine-tuning for specific detection tasks. It should also be pointed out that even in this case, running a model locally might not be that easy due to memory constraints. While most of these system have a cloud component, in scenarios where privacy is relevant, the memory footprint might be an issue.

[13]Neural Processing Units

REFERENCES

Al-haija, Q. A., Odeh, A. J., and Qattous, H. K. (2022). Pdf malware detection based on optimizable decision trees. *Electronics*.

Bakır, H. (2024). Votedroid: a new ensemble voting classifier for malware detection based on fine-tuned deep learning models. *Multimedia Tools and Applications*.

Balan, G., Simion, C.-A., Gavrilut, D., and Luchian, H. (2023). Feature mining and classifier selection for api calls-based malware detection. *Applied Intelligence*, 53:29094–29108.

Botacin, M. (2023). Gpthreats-3: Is automatic malware generation a threat? In *2023 IEEE Security and Privacy Workshops (SPW)*, pages 238–254.

Charan, P. V. S., Chunduri, H., Anand, P. M., and Shukla, S. K. (2023). From text to mitre techniques: Exploring the malicious use of large language models for generating cyber attack payloads.

Chen, J., Guo, S., Ma, X., Li, H., Guo, J., Chen, M., and Pan, Z. (2020). Slam: A malware detection method based on sliding local attention mechanism. *Security and Communication Networks*, 2020:1–11.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., de las Casas, D., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., Lavaud, L. R., Lachaux, M.-A., Stock, P., Scao, T. L., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. (2023). Mistral 7b.

Jiang, A. Q., Sablayrolles, A., Roux, A., Mensch, A., Savary, B., Bamford, C., Chaplot, D. S., de las Casas, D., Hanna, E. B., Bressand, F., Lengyel, G., Bour, G., Lample, G., Lavaud, L. R., Saulnier, L., Lachaux, M.-A., Stock, P., Subramanian, S., Yang, S., Antoniak, S., Scao, T. L., Gervet, T., Lavril, T., Wang, T., Lacroix, T., and Sayed, W. E. (2024). Mixtral of experts.

Karanjai, R. (2022). Targeted phishing campaigns using large scale language models.

Kim, S., Choi, J., Ahmed, M. E., Nepal, S., and Kim, H. (2022). Vuldebert: A vulnerability detection system using bert. In *2022 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 69–74.

Li, Z., Zhu, H., Liu, H., Song, J., and Cheng, Q. (2024). Comprehensive evaluation of mal-api-2019 dataset by machine learning in malware detection. *ArXiv*, abs/2403.02232.

Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. (2019). Roberta: A robustly optimized bert pretraining approach.

Motlagh, F. N., Hajizadeh, M., Majd, M., Najafi, P., Cheng, F., and Meinel, C. (2024). Large language models in cybersecurity: State-of-the-art.

Omar, M. (2023). Detecting software vulnerabilities using language models.

Pa Pa, Y. M., Tanizaki, S., Kou, T., van Eeten, M., Yoshioka, K., and Matsumoto, T. (2023). An attacker's dream? exploring the capabilities of chatgpt for developing malware. In *Proceedings of the 16th Cyber Security Experimentation and Test Workshop*, CSET '23, page 10–18, New York, NY, USA. Association for Computing Machinery.

Rahali, A. and Akhloufi, M. A. (2021). Malbert: Using transformers for cybersecurity and malicious software detection.

Rozière, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Sauvestre, R., Remez, T., Rapin, J., Kozhevnikov, A., Evtimov, I., Bitton, J., Bhatt, M., Ferrer, C. C., Grattafiori, A., Xiong, W., Défossez, A., Copet, J., Azhar, F., Touvron, H., Martin, L., Usunier, N., Scialom, T., and Synnaeve, G. (2024). Code llama: Open foundation models for code.

Sánchez, P. M. S., Celdr'an, A. H., Bovet, G., and Pérez, G. M. (2024). Transfer learning in pre-trained large language models for malware detection based on system calls. *MILCOM 2024 - 2024 IEEE Military Communications Conference (MILCOM)*, pages 853–858.

Sanh, V., Debut, L., Chaumond, J., and Wolf, T. (2020). Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter.

Senanayake, J. M. D., Kalutarage, H. K., and Al-Kadri, M. O. (2021). Android mobile malware detection using machine learning: A systematic review. *Electronics*.

shiqi, L. (2019). Android malware analysis and detection based on attention-cnn-lstm. *Journal of Computers*, pages 31–43.

Shoeybi, M., Patwary, M., Puri, R., LeGresley, P., Casper, J., and Catanzaro, B. (2020). Megatron-lm: Training multi-billion parameter language models using model parallelism.

Singh, G., Kumar, B., Gaur, L., and Tyagi, A. (2019). Comparison between multinomial and bernoulli naïve bayes for text classification. *2019 International Conference on Automation, Computational and Technology Management (ICACTM)*, pages 593–596.

Vörös, T., Bergeron, S. P., and Berlin, K. (2023). Web content filtering through knowledge distillation of large language models.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R., and Le, Q. V. (2020a). Xlnet: Generalized autoregressive pretraining for language understanding.

Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R., and Le, Q. V. (2020b). Xlnet: Generalized autoregressive pretraining for language understanding.

Zheng, Y., Pujar, S., Lewis, B., Buratti, L., Epstein, E., Yang, B., Laredo, J., Morari, A., and Su, Z. (2021). D2a: A dataset built for ai-based vulnerability detection methods using differential analysis.

Zhou, X. and Verma, R. M. (2022). Vulnerability detection via multimodal learning: Datasets and analysis. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, page 1225–1227, New York, NY, USA. Association for Computing Machinery.

Zhou, Y., Liu, S., Siow, J., Du, X., and Liu, Y. (2019). Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks.

Şahın, D. Ö., Akleylek, S., and Kılıç, E. (2022). Linregdroid: Detection of android malware using multiple linear regression models-based classifiers. *IEEE Access*, 10:14246–14259.