Analyzing Malware with Large Language Models

Phillip Jordan
Computer Science
Virginia Tech
Blacksburg, VA, USA
alexj14@vt.edu

Dr. Mark Thompson
National Security Institute
Virginia Tech
Arlington, VA, USA
markt22@vt.edu

Jared Gregersen
National Security Institute
Virginia Tech
Oklahoma City, OK, USA
jaredgregersen@vt.edu

***Abstract*— Given the proliferation of advanced malware, it is imperative that new, more versatile methods be introduced to detect and analyze malware on a mass scale. We look to Large Language Models (LLMs) to provide a more scalable, flexible solution to traditional analysis. The Deepseek LLM was essential to our project because it gave access to an open-source model that had superior general capabilities similar to GPT-3.5, was proficient in coding and math, and could be locally hosted. In this paper, we highlight the results of the first phase of this project where we examined the native capabilities of LLMs to analyze and detect alterations made in machine code. In our second phase, we will use our findings to extend these capabilities to real malware analysis. In our testing, the Deepseek LLM 67B model was able to identify and accurately describe changes made to the assembly instructions in our control executables, but results were somewhat limited by the capabilities of the disassembler that was used.**

Keywords—machine learning, malware, reverse-engineering, cybersecurity, natural language processing, large language model

I. Introduction

As cyberwarfare has evolved from hobbyists to nation-states, computer viruses have become far more sophisticated and difficult to detect. In this new cybersecurity landscape, where advanced malware like polymorphic viruses are no longer rare, we must find new techniques for detecting and analyzing malicious code that can evolve with the changing times. With new, advanced, viruses becoming more difficult to detect, it is essential that we advance our techniques as well, using the wealth of data available using Large Language Models (LLMs).

LLMs have already proven their mettle in software applications with tools like GitHub Copilot [1] able to take code autocompletion to new heights previously unseen. It is intuitive for natural language AI to be applied to machine languages as well as human languages. In this project, we investigated how well LLMs can interpret and make observations about reverse-engineered machine code and what further development would be necessary to apply the full potential of LLMs to this task.

II. Related Works

A. Malheur

One piece of early research on this topic can be found in *Automatic Analysis of Malware Behavior Using Machine Learning* published in the Journal of Computer Security in 2011 [4]. In this study, the researchers used early machine-learning techniques to identify novel classes of malware with similar behavior through clustering. This study focused more on categorizing malware behavior, detecting new variants, and tracking the evolution of malware technology.

B. LLM-assisted Malware Review

In this 2023 research study, Henrik Plate used GPT-3.5 to classify malware as malicious or benign [3]. The tests comprised 1800 binary classifications of which there were many false positives and negatives. Plate found that, while the model was easily fooled, it showed promise given code pre-processing and further prompt engineering.

C. Malware Detection Smackdown

Later in 2023, Plate conducted another study comparing the threat levels generated by OpenAI with Vertex AI when given the same malware to analyze [4]. These tests found that, while they were comparable, OpenAI GPT-3.5-turbo outperformed Vertex AI text-bison when it came to providing source code explanations and risk-rating for obfuscated code.

III. Methods

Experiments were conducted by comparing the disassembled binaries from a control executable to those of a modified version of the same executable that had been patched. Alterations were made at the binary level and analysis was done with a language model using disassembled code. Further details regarding the steps of the process are as follows:

A. Experimental Executable Programs

To run our tests in this first phase of our project, control executables and experimental executables were needed to take the place of archive and malicious executable files. The control executables were simple C programs, compiled with Gnu Compiler Collection (GCC) that demonstrated various

features with unique representations in assembly: 1) loop statements, 2) printing to the console, 3) accepting user input, 4) branching statements, or 5) user-created functions. Experimental executables (our substitute for real malware) were made by modifying the control executables at the binary level using the *xxd* command, a Linux command used to manage, convert, and display binary files [5]. The types of modifications that were made for this research included modifying register values, replacing instruction operation codes (opcodes), and changing control flows. In this study, the changes made were minor and harmless, but in real malware, these techniques could be used to bypass locked software on a victim's computer, remove other protective measures, and execute custom code on a victim's computer.

B. Server Configuration for Self-Hosted LLM

We leveraged a self-hosted Large Language Model (LLM) to allow for precise control over the infrastructure, toolchain, and overall environment, ensuring repeatable results. The LLM was hosted on a Nvidia DGX-A100 server, with experiment code and workloads accelerated by A100 GPUs. The choice of utilizing the Oobabooga open-source Text Generation Web UI tool was based on its capability to load language models and interact with those models through Python server API calls. During testing, the focus was placed solely on this API functionality, bypassing the tool's web-based user interface. Initial tests began with the deployment of the Stable Beluga 7B model, a fine-tuned Llama2-based language model from StabilityAI [9]. We switched to using the Deepseek 67B model, a larger model created by Deepseek and released to the public for research initiatives, that leverages Grouped-Query Attention and was trained on a 2 trillion token dataset [10]. This transition was meant to evaluate how well a larger parameter model would perform when compared to a smaller parameter model and to better explore the capabilities of LLMs with code analysis.

C. Disassembly of Executable Code

The executables used in testing were disassembled using the *objdump* Linux command with the disassemble option enabled [6]. This command is used to translate machine code back into human-readable assembly instructions. This output was fed into a customized Python script developed for this project to isolate the <main> section, cut out all the unnecessary data such as hex representation of instructions and whitespace, and compare the two files using the difflib Python library [7]. This automatic formatting helped cut out irrelevant information that may distract the model and reduced the number of new tokens below the max limit, allowing us to process more complex programs. To compare the control or "archive" executable files to the modified executables, the formatted objdump output was put through the difflib Python library, and text was put at the beginning of each modified instruction saying, "This line was added" or "This line was removed." In Figure 1, you can see an example of a snippet of a comparison that was fed to the model.

```
>>> compFiles('add')
endbr64
This line was removed: push   %rbp
This line was added: nop
mov   %rsp,%rbp
sub   $0x20,%rsp
```

Fig. 1. The Python script finds changed lines from the object dump so that the LLM can more easily see which lines it should focus on.

D. Prompt Design

The following details how the model was exactly prompted.

- *System context*

The prompts in this project were designed using system context and user messages to identify the differences between the disassembled code of the executable files and explain how these changes affect the functionality of the program. The following was the only message parameter in the context passed to the API with the "system" tag.

Your purpose is to identify the purposes of malicious modifications between an archive and current version of a disassembled program.

- *User message*

The following was the text passed to the API with the "user" tag, followed immediately by the formatted text output from the *objdump* command.

The following code shows all instructions in the executable with annotations before modified lines. Focus on only the modified lines. What do these modifications do?

E. Experimentation

To test the effectiveness of these language models at accurately detecting and describing the injected instructions under varying circumstances, results were collected from 10 pairs of control and modified executables while adjusting the following OpenAI Python API [8] request parameters:

- *Temperature*, which controlled how concise the responses are, ranging from 50 to 500 in our experiments.
- *Max number of tokens*, which controlled how concise the responses are, ranging from 50 to 500 in our experiments.

These tests were repeated using both Stable Beluga [9] and Deepseek [10] to see how the two models compared to each other.

IV. Results

The numbers in each main cell of the tables below are the number of executable file pairs out of 10 for each API configuration whose analysis mentioned the real key difference between the two files somewhere in its response. The accuracy of each generated response was determined through qualitative observation by a student researcher

familiar with the changes. Although this method of measuring results is somewhat subjective, it aided to visualize important trends in the responses as parameters were changed.

TABLE I. ACCURATE RESPONSES BY PARAMETER VALUES (STABLEBELUGA)

Max Length	Temperature				
	0	***0.25***	***0.5***	***0.75***	***1***
50	1	1	4	1	2
100	4	4	2	1	1
200	6	4	2	1	0
500	**7**	5	1	1	0

Table I shows that the StableBeluga model was only able to accurately analyze the dataset provided in some cases. The pattern of how many examples in which each configuration was able to identify the correct alteration shows that increasing the temperature in the API call request, which determines how creative the model can be in its response, negatively impacted accuracy and it only needed about 200 tokens to fully communicate the analysis in most responses.

TABLE II. ACCURATE RESPONSES BY PARAMETER VALUES (DEEPSEEK)

Max Length	Temperature				
	0	***0.25***	***0.5***	***0.75***	***1***
50	6	5	4	3	1
100	7	7	5	5	4
200	**8**	7	4	5	3
500	7	7	4	4	3

Table II shows that the Deepseek model, which is far larger and contains more information about x86 assembly, produced more accurate results across the spectrum of parameter configurations. Furthermore, the difference between the accuracy of the two models widens as the temperature increases, although they both lose accuracy when doing so. In other words, as the creativity parameter is increased, the Deepseek model performs far better than the more generic Stable Beluga model. This is likely because the Deepseek model has more data related to x86 assembly that it can draw from. This indicates that large language model AI is capable of reading assembly instructions as long as the model is sufficiently trained on it beforehand.

Below are some notable specific examples of generated analyses from which we can draw further conclusions about how the model is analyzing the data.

- *Deepseek 67B Model: Accurate Analysis*

> *These changes appear to be related to removing conditional branching based on input from `__isoc99_scanf` function call, which could have been used for checking user inputs or controlling execution flow depending upon them.*

In the very first sentence from the response, generated by the Deepseek model with a temperature of 0 and max token length of 200, it was able to identify both the purpose of the lines that were changed as well as expand on the description, including details from previous lines of code affected by the change.

- *StableBeluga 7B Model: Comprable Analysis*

> *These modifications are related to bypassing security checks, such as stack protection and function validation, which could be used for privilege escalation or other malicious activities.*

While this response, generated by the StableBeluga model with the same parameters as the Deepseek model used in the previous example, has some truth to it, it seems to be drawing a lot more from the text of the prompt preceding the machine code and the initial system prompt, making vague, general statements rather than pointing to specific examples from the code.

- *Deepseek 67B Model: Accurate Analysis*

> *After comparing both versions, it seems like there's no significant change made by modification; rather than adding or removing something from certain memory locations (-0x4(%rbp)). The modified/added part just changes sign (from positive +0x7 -> negative-0x7). It does not affect overall functionality as intended because this value does not impact any other calculations within your existing logic flowchart(like branching conditionals etc.).*

In this example, produced by the Deepseek model with a temperature of 0.75 and max token length 100, the AI was able to accurately determine the purpose of the changes, swapping addl (an instruction which adds a register value to a value stored at a particular memory address) with subl (an instruction which does the same but subtracts).

- *StableBeluga 7B Model: Comprable Analysis*

> *The modifications remove some instructions, modify others, insert new ones, and perform various actions such as storing data at specific addresses or calling functions like "printf". The overall effect seems to be creating a backdoor or bypass mechanism into the program, which could allow an attacker to gain unauthorized access or control over it.*

In this example, analyzing an instruction changed from addl to subl with a temperature of 0.75 and max token length of 100, the Stable Beluga model could not determine the purpose of the changes at all. This response appears to be mostly a hallucination.

- *Deepseek 67B Model: Failed Analysis*

> *In this specific example, two changes were made by the attacker or malware author which resulted in additional functionality being added/implemented within `main` function. They are as follows.*

Above is the only complete sentence in the response given by the Deepseek LLM with a temperature of 1 and a max token length of 50 when analyzing a script modified to remove a jump instruction designed to break out of a loop. In this example, the model's response makes vague statements about the nature of the changes made rather than cutting to the chase and explaining what the changes are exactly. This happens regularly with higher temperatures, and while the response may have been more accurate if it were allowed to be longer, the lower temperature responses summarize the analysis more efficiently.

V. Conclusions and Future Work

In this paper, language models were used to analyze machine code to detect alterations and work towards developing AI tools to provide greater insight than current reverse-engineering tools. In this project, we found that machine code can be processed effectively by LLMs if they are sufficiently trained, but a high temperature parameter can prevent an LLM from making a precise analysis of machine code.. These results show that language models can be very useful for analyzing machine code, often producing accurate results when properly trained and more ideal parameter settings. Furthermore, the qualitative data from the responses generated by the Deepseek model show that LLMs can make observations about machine code that common procedural antivirus or reverse-engineering tools cannot, by identifying control flow structures and how different instructions connect. Further research may be required to understand why the Deepseek model was able to more accurately identify which lines were modified when the modified lines were annotated with English text, which should have been detected by the Stable Beluga model as well.

Going forward, the second phase of this project will start experimenting with real malware disassembled in an isolated virtual machine environment. We would like to continue our work with the Deepseek 67B model to determine how it can learn to identify malware embedded in machine code, with an emphasis on its predicting capabilities with previously unknown malware.

To continue our research in the future, we believe that it will be necessary to use additional fine-tuning techniques, more advanced machine learning methods such as pipelines by Hugging Face (add reference) for a more structured analysis, a disassembler with enhanced capabilities, and pre-processed data that may be more readable to the model. With these new tools, this project could proceed past surface-level, basic static analysis and proceed toward more advanced, meaningful analysis. Furthermore, to evaluate results going forward, there needs to be a more consistent and measurable metric. We expect this to take the form of using a second natural language processor to test how closely the generated statements agree to a human-written, accurate analysis.

References

[1] T. Dohmke, "Universe 2023: Copilot transforms GitHub into the AI-powered developer platform," The GitHub Blog, Nov. 08, 2023. https://github.blog/2023-11-08-universe-2023-copilot-transforms-github-into-the-ai-powered-developer-platform/

[2] J. Hampton, "Researchers Pit LLMs Against Each Other in Malware Detection Smackdown," Datanami, Jun. 12, 2023. https://www.datanami.com/2023/06/12/researchers-pit-llms-against-each-other-in-malware-detection-smackdown/.

[3] H. Plate, "LLM-assisted Malware Review: AI and Humans Join Forces to Combat Malware," *Endor Labs*, Apr. 17, 2023. https://www.endorlabs.com/learn/llm-assisted-malware-review-ai-and-humans-join-forces-to-combat-malware.

[4] K. Rieck, P. Trinius, C. Willems, and T. Holz, "Automatic analysis of malware behavior using machine learning," Journal of Computer Security, vol. 19, no. 4, pp. 639–668, Jun. 2011, doi: https://doi.org/10.3233/jcs-2010-0410.

[5] K. Brown, "Linux XXD Command Explained," *Linux Config*, Jun. 13, 2023. https://linuxconfig.org/linux-xxd-command-explained.

[6] H. Arora, "Linux Objdump Command Examples (Disassemble a Binary File)," The Geek Stuff, Sep. 21, 2012. https://www.thegeekstuff.com/2012/09/objdump-examples/

[7] N. Omkar, "Learn Python Difflib Library Effectively," Python Pool, Mar. 23, 2022. https://www.pythonpool.com/pythons-difflib/

[8] "OpenAI Python API - Complete Guide," *GeeksforGeeks*, Sep. 21, 2023. https://www.geeksforgeeks.org/openai-python-api/ (accessed Mar. 07, 2024).

[9] [3]H. Touvron et al., "Llama 2: Open Foundation and Fine-Tuned Chat Models," arXiv.org, Jul. 19, 2023. https://arxiv.org/abs/2307.09288

[10] Deepseek-AI et al., "Deepseek LLM: Scaling Open-Source Language Models with Longtermism," arXiv (Cornell University), Jan. 2024, doi: https://doi.org/10.48550/arxiv.2401.02954.

[11] org, Jul. 19, 2023. https://arxiv.org/abs/2307.09288