MANDALA: Multi-Agent Network for Backdoor Detection using AST Parsing and Large Language Models

Anonymous ACL submission

Abstract

This paper presents MANDALA, a system that leverages locally deployed open-source large language models (LLMs) and multi-agent networks to enhance vulnerability detection in code bases. MANDALA uses an abstract syntax tree-based algorithm to parse code into digestible chunks for LLMs, generating code explanations and descriptions. A collaborative multi-agent network comprised of specialized agents such as static analysis, security management, and user interaction agents then coordinate to analyze the codebase for potential backdoors and vulnerabilities. Evaluations on open-source codebases demonstrate MANDALA's ability to significantly reduce manual effort while increasing detection speed and accuracy over traditional methods across various test cases. MANDALA represents an innovative integration of LLMs and multi-agent systems for efficient, scalable code vulnerability detection.

1 Introduction

The intersection of Natural Language Processing (NLP) and cybersecurity presents a promising opportunity to leverage new developments in the area of large language models (LLMs). We have applied these advances in LLM to enhance the efficiency and effectiveness of security vulnerability detection and analysis in code bases to create MANDALA. By using the power of multiple LLMs within a multi-agent network and combining it with the Abstract Syntax Tree-based parsing algorithm with multiple LLMs, we aim to contribute significantly to the development of more secure software infrastructure. Traditional methods of detecting vulnerabilities struggle to keep pace with the rapid evolution of software development and are therefore exposed to various cyber threats. We focus on specifically backdoor vulnerability detection, which is to identify weaknesses in systems that could allow unauthorized access, which are essentially hidden entrances created either intentionally or unintentionally. Recently, LLMs have demonstrated strong context understanding, specifically in inference and generating coherent output, making them uniquely positioned to handle complex problem solving scenarios in this domain.

Existing cybersecurity approaches, from manual code reviews to automated static and dynamic analysis tools, often involve substantial human involvement and result in a lack of scalability. The increasing size and complexity of software projects makes the manual identification of backdoors increasingly infeasible. Manual code reviews, while thorough, are slow and not feasible for large codebases. Another major issue is that the code lengths can be extensive, making it difficult for traditional methods to process and analyze efficiently. Here, the limitations of existing approaches become evident, necessitating a more advanced and scalable solution.

The key insight of this project lies in the untapped potential of LLMs to transform the cybersecurity space. LLMs understand patterns, logic, and thus should be able to detect vulnerabilities within software code by processing the text-based features. This innovative approach leverages the power of Multi-Agent Networks (MANs) where numerous agents collaborate to achieve a common goal. By combining the strengths of individual agents and giving each agent an individual goal, this MAN system offers outstanding adaptability and effectiveness. For the second problem, to tackle the challenge of long code length and the context constraint of LLMs, we utilize Abstract Syntax Trees (ASTs). ASTs provide a structured representation of the code, allowing for more efficient parsing and analysis by LLMs. We check the length of every node and iterate through to extract all functions in the program and process them. Because these functions are significantly smaller than the entire file, the code size is exponentially reduced as it grows larger, making it manageable for LLMs to process.

Based on this insight, we have proposed MANDALA: a Multi-Agent Network for Backdoor Detection Using AST Parsing and Large Language Models System. This system integrates several advanced technologies, including AutoGen (Wu et al., 2023) and CrewAI (Moura), to create this collaborative MAN framework. LiteLLM (Li et al., 2024) simplifies the integration and utilization of LLM APIs and Ollama (oll) provides a framework for running LLMs mentioned in the Table 1 locally. Our experiments have shown that this method not only improves the speed of vulnerability detection but also scales effectively with large codebases. The results show a significant reduction in manual human effort and an exponential increase in detection speed.

We plan to open-source our code and data to facilitate future work in the broader NLP and cybersecurity community.

2 Overview

If we look at any industry software deployed in a production environment, it will have millions of lines of code, presenting a significant challenge to traditional cybersecurity methods. Manual code reviews are thorough, but they are impractical for such a vast amount of code. Automated tools also struggle with the complexity and sheer volume of the code, often missing critical vulnerabilities. In contrast, by using our proposed multi-agent LLM network, we aim to effectively parse, analyze, and detect these vulnerabilities in a fraction of the time and cost. MANDALA's ability to break down the code into smaller, manageable functions using ASTs ensures that even the largest code projects can be analyzed efficiently.

We want to improve the efficiency and effectiveness of vulnerability detection. We assume that the software code is accessible and can be parsed into ASTs. We have focused on Python code in MANDALA, but it can be expanded to support other languages through a modular architecture. MANDALA is also focused on code vulnerabilities that cause backdoors. By targeting code-based vulnerabilities, we aim to create a robust system that can ideally identify weaknesses before they are exploited.

MANDALA uses two systems: Firstly, Code parsing using ASTs and secondly, Multi- Agent Networks (MANs). MANs are multiple agents with well defined individual goals collaborate to achieve a common goal. This combines the strengths of individual agents of code understanding to compensate for their weaknesses of context length and offers outstanding adaptability. We have created MANDALA for vulnerability detection that scales effectively with large codebases and this use of LLMs addresses the scalability issue and reduces manual human effort significantly.

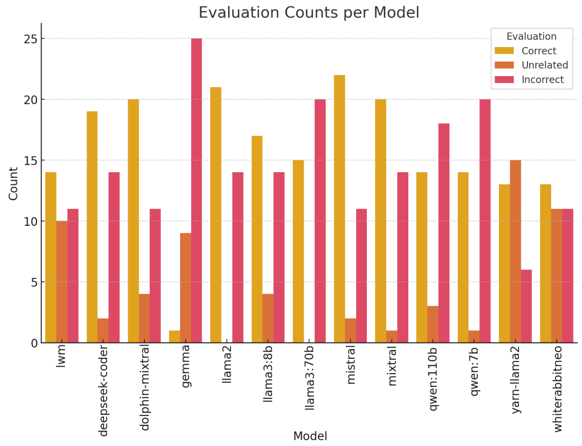

Figure 1: The figure shows the accuracy of models on comprehension tasks.

AutoGen and CrewAI have introduced AI collaboration by defining specific roles and facilitating teamwork towards shared objectives. Agents have their own individual roles and tools that are specific to them. LiteLLM simplifies the integration and utilization of LLM APIs, and Ollama provides a framework for running LLMs locally. The project's journey included assessing MemGPT (Packer et al., 2023) for context management, but due to instability issues, we focused on the proven frameworks of AutoGen and CrewAI, combined with LiteLLM and Ollama.

MANDALA is based on locally implemented LLMs that provide greater control over data privacy and security since the data does not leave the local environment. This is extremely important in the space of cybersecurity where sensitive information may be involved. Local LLMs can be fine-tuned without much additional cost to serve specific needs and contexts, which can enhance the detection of vulnerabilities. Table 1 shows the LLMs which have been tested in MANDALA. A notable drawback is the need for substantial computational resources, which makes local LLMs expensive and difficult to manage. As an alternative,

Model Name	Parameters	Description
Large World Model (lar)	6.74 billion	Fine-tuned Llama2 model with a context length of 1 million tokens.
DeepSeekCoder (Guo et al., 2024)	6.74 billion	Fine-tuned Llama2 model with a window size of 16,000. Excellent at coding.
Dolphin Mixtral (Research)	8 x 7 billion	Uncensored fine-tuned Mixtral model specialized in coding tasks.
Gemma (Team et al., 2024)	8.54 billion	A lightweight state-of-the-art open model built by Google DeepMind.
Llama2 (Touvron et al., 2023)	6.74 billion	Meta's older Llama model released in 2023, known for its SOTA performance. Chosen to compare with Lama3 and assess improvements.
Llama3 (lla; Meta-Llama)	8.03 billion	Meta's latest model (2024) with claimed significant improvements over Llama2.
Llama3 (lla; Meta-Llama)	70.6 billion	Chosen for its extensive parameter size to assess performance.
Mistral (AI, 2024)	7.25 billion	A model by Mistral AI. Chosen for its efficiency and performance.
Mixtral (AI)	8 x 7 billion	A mixture of experts model by Mistral AI consisting of 8 experts, each with 7 billion parameters. Chosen for its innovative mixture of experts approach.
Qwen (Alibaba Cloud)	7.72 billion	a transformer-based LLM by Alibaba Cloud, pre-trained on a large volume of data, including web texts, books, code, etc.
Qwen (Alibaba Cloud)	111 billion	A larger version of Qwen with 111 billion parameters. Chosen for its extensive parameter size to assess performance gains.
Yarn Lama2 (Ollama, 2024)	6.74 billion	Extends Lama2's context length up to 128,000 tokens.
White Rabbit Neo (Face)	13 billion	Fine-tuned on cybersecurity data, used for offensive and defensive cybersecurity testing. Chosen for its specialization in cybersecurity.

Table 1: Selected models.

the same code used for local LLMs can be substituted by various internet LLM APIs like OpenAI, Google, or Anthropic. This flexibility allows for easy switching between local and more cutting edge models, leveraging the strengths of each approach as needed.

While newer language models like Gemini, Claude and ChatGPT boast substantially longer context lengths compared to previous generations, analyzing massive codebases consisting of millions of lines can still overwhelm their capabilities. Therefore, MANDALA's approach of leveraging ASTs to methodically break down code into smaller, manageable components remains highly relevant. By reducing the token footprint exponentially as code size increases, this technique enables efficient processing by language models regardless of their maximum context capacity. As such, MANDALA's parsing algorithm complements the latest advances in language models, ensuring scalable and thorough vulnerability analysis even for the most extensive software projects.

Challenges Implementing LLMs in cybersecurity presents several challenges:

1. **Dependency on External Tools and Libraries:** The MAN relies on various external tools and libraries, such as duckduckgo search (DuckDuckGo), docker (Docker) and the LLMs, to perform its analysis tasks. The solution's utilization of emerging technologies, such as the Autogen framework and Crewai, introduces an element of uncertainty regarding the stability of these components. The performance and reliability of these external components can directly impact the overall effectiveness of the solution.

2. **False Positives or Missed Vulnerabilities:** While the combination of static and dynamic analysis techniques aims to provide a comprehensive assessment, there is always the possibility of false positives or missed vulnerabilities (false negatives), particularly in the face of sophisticated and obfuscated backdoor techniques. This is especially true when open-sourced models with a short context length are used. They can be extremely inaccurate and not reliable for complex tasks.

3. **High Computational Resources:** Managing the significant computational requirements of LLMs is nontrivial.

3 Design

MANDALA is divided into two parts: the AST-based code parsing algorithm and the MANs for detection.

3.1 AST-based Code Parsing Algorithm

The AST-based parsing algorithm lies at the core of our approach to efficiently process large codebases so that it can produce detailed and full descriptions and explanations of the code inside. The main tools the algorithm uses to realize its objectives are ASTs, systematic file and folder processing, and language models deployed through the Ollama library.

3.1.1 Leveraging Abstract Syntax Trees

At the heart of the code parsing lies the utilization of Abstract Syntax Trees (ASTs). ASTs present a powerful model of the structure and semantics that underlie the code of a programming language. This is done using ASTs that enable the algorithm to dig into the specific nodes/parts of the Python

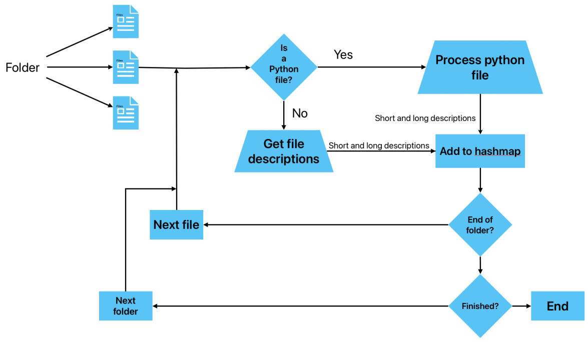

Figure 2: Flowchart for the parsing algorithm.

code to extract essential elements required in the generation of descriptions and explanations.

First, we transform the raw file content into an AST representation that uses the `ast` (Foundation) module in Python. We further refine the AST by applying custom formatting rules to various node types such as variable assignments, control flow statements, and function definitions. This step ensures that the extracted code segments are presented in a clean and structured manner.

The use of ASTs in the algorithm offers several key advantages:

Structured Code Representation: By converting the raw code into an AST, MANDALA gets an understanding of the underlying logical structure of the program including variables, control flow and function definitions. This structured representation allows for more precise and targeted extraction of relevant code elements.

Extensibility: The modular design allows for easy expansion and customization of the AST processing rules. This makes the algorithm adaptable to handle different language constructs or evolving programming practices.

3.1.2 Folder and File Processing

The algorithm dives into the target directory and all its subdirectories. This allows the algorithm to run and provides a full analysis of the entire codebase. The flowchart in Figure 2 illustrates the systematic folder and file processing methodology employed by the parsing algorithm.

For each file, a helper function analyzes the file extension to determine the programming language. This function has been configured to easily expand to support additional languages in the future.

If the file is identified as a Python file, MANDALA parses the file to transform the code into a structured AST representation. This step uses the discussed AST processing to extract the essential

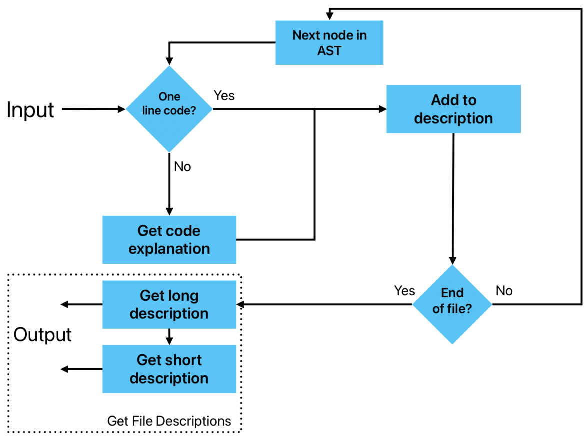

Figure 3: Flowchart for processing python file.

code elements. Then it proceeds to the generation of descriptions and explanations. This next step gives MANDALA scalability and efficiency to handle large codebases without losing effectiveness.

3.1.3 Generating Descriptions and Explanations

The overall idea behind generating code explanations, long descriptions and short descriptions, is to handle very large codebases efficiently. Real programs consist of a huge amount of long and complex functions that MANDALA exponentially reduces by leveraging their AST representations. By breaking down the code into smaller components, the process becomes more manageable and allows for effective analysis and understanding. Refer to the Figure 3.

3.1.4 Description Granularity

Figure 5 provides examples and statistics for the code explanations, long descriptions, and short descriptions generated as part of the parsing process.

Code Explanations: It reduces large functions into a few lines of description. When these descriptions are aggregated, they provide a comprehensive narrative of the code's flow and functionality. This step is important to transform extensive code segments into manageable pieces which should fit in the limited context window of the LLM.

Long Descriptions: Once the code has been broken down and explained at the AST node level, the long description function generates a more holistic description of the code. The goal here is to provide a thorough analysis that balances detail and relevance. In contrast, the code explanation was providing succinct descriptions for individual AST nodes. The reduced size of explanations from the previous step enables this step to generate the long descriptions.

Short Descriptions: After obtaining a detailed long description, MANDALA distills the information into a concise, one-sentence summary. This step provides a high-level understanding of the code's purpose, allowing the LLM agents to quickly grasp the essence of the code without delving into details. Hence, this preprocessing helps in managing huge codebases and provides agents with the essential information.

By having these three different types of description: code explanation, long description, and short description, the process ensures a thorough yet scalable approach to understanding and analyzing large codebases. Each type of description uses custom-configured LLM models, selected based on extensive testing to ensure optimal performance, as shown in Section 5.

3.1.5 Access to Processed Data

Upon completion of the file processing and description generation, the algorithm saves the three dictionaries to JSON files. This storage of the generated outputs allows for easy access, sharing and will help to combine both systems of parsing and the MAN together.

3.2 Multi-Agent Model Design

The detection of potential backdoors within a codebase involving both static and dynamic analysis techniques. To address this challenge, the provided solution incorporates a MAN and uses the strengths and specialized capabilities of individual agents to achieve a thorough and coordinated analysis of the target codebase.

3.2.1 The Agents and their Roles

The multi-agent network consists of the following:

Static Analysis Agent: Responsible for conducting a static analysis of the code to identify possible backdoors and other vulnerabilities. This agent focuses on detecting any anomalies that deviate from standard coding practices. It also specifies the exact locations within the codebase where potential vulnerabilities are detected. It presents the analysis results to the Security Analysis Manager for further review.

Security Analysis Manager Agent: Oversees the entire backdoor detection process and coordinates with the Static Analysis Agent and other agents. It reviews the findings from the Static Analysis Agent and integrates these insights to form a comprehensive understanding of the codebase's security vulnerabilities. The manager synthesizes information from various sources into a coherent action plan and guides the MAN towards proposing a mitigation strategy. It also ensures clear and actionable communication of findings and recommendations, fostering collaboration and information sharing among the team.

Figure 4: Model performance.

User Proxy Agent: Serves as a placeholder for the human user in the MAN. It is responsible for receiving the initial task description from the user and initiating the chat within the network. It maintains the overall context and flow of the analysis process, ensuring that the user's initial prompt requirements are met.

Admin Agent: Provides an environment for executing code within the MAN. This agent exposes a limited set of functionalities to the other agents, allowing them to execute specific commands or access files as needed. It also serves as a gatekeeper, limiting the potential for malicious code execution or unauthorized access.

Assistant Agent: Provides general support and assistance to the other agents within the MAN. It assists in tasks such as information retrieval, data analysis, and task coordination using the tools at its disposal. It does not actively contribute to the group chat and works passively as an entity with the sole objective of helping other agents.

3.2.2 The Multi-Agent Workflow

The MAN follows a non-sequential workflow and the workflow depends on findings of the agents. They all work towards the same goal and do an analysis of the target codebase. This is a sample of how the network works in practice:

Task Initiation and Preparation: The human user initiates the task by providing a description of the codebase to be analysed and the requirement to detect potential backdoors. The User Proxy Agent receives the task description and sets up the initial group chat with the other agents. The group chat provides a centralized communication channel for the agents to exchange information, coordinate their efforts, and share their findings.

Static Analysis: The Static Analysis Agent takes the lead in conducting the initial static analysis of the codebase. The agent scans the code for patterns, functions or snippets that are commonly associated with backdoors or other security vulnerabilities. The agent then presents the analysis results to the Security Analysis Manager within the group chat for further review and prioritization.

Further analysis and Code Execution: Based on the initial static analysis findings, the Security Analysis Manager may request the execution or analysis of code after looking at the json from the parsing algorithm. The Admin Agent acts as a secure and controlled environment and facilitates the execution of the requested code if any, ensuring that it does not adversely affect the system or compromise the overall security. The Agents have specifically been instructed not to run the code from the code base but they use the terminal to run required static analysis tools at their disposal.

Vulnerability Assessment and Prioritization: The Security Analysis Manager reviews the findings from both the Static Analysis Agent and potentially the dynamic analysis. We tested some dynamic analysis scenarios but the Agents were unreliable and unpredictable and posed a huge risk if not overseen by a human.

Mitigation Planning and Reporting: Security Analysis Manager coordinates with the other agents to develop a detailed plan for mitigating the identified security risks. The plan may include recommendations for code changes, the implementation of additional security controls or the adoption of best practices to address the detected vulnerabilities. There is often a back and forth with each file before they reach a conclusion. The report from the Security Analysis Manager is then presented to the human user in an "email-like" format that provides a clear and actionable roadmap for enhancing the security of the codebase.

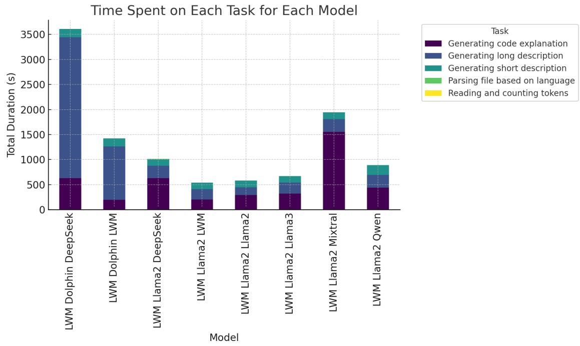

Figure 5: Time spent on individual tasks. The first model in the name provides a short description, the second a long description and the third focuses on code explanations.

3.2.3 Human in the Loop for Secure Code Execution

The successful execution of dynamic analysis tasks within the MAN is facilitated by the Admin Agent, which serves as a secure and controlled environment for code execution. For every execution, human intervention was activated. By restricting the available functionalities and carefully controlling the code execution environment, the Admin Agent acts as a gatekeeper.

4 Evaluation

4.1 Experimental Setup

To evaluate the performance of MANDALA, we conducted extensive experiments using a variety of models. The hardware used for these experiments included a high-performance computing server equipped with NVIDIA A30 and 256 GB of memory.

We constructed our data sets by collecting a diverse set of codebases from open-source repositories. To show the statistics, code parsing was performed and tested on Vulpy (Portantier, 2024). For showing the efficiency in compressing the code into fewer tokens, we filtered out files with more than 1,000 tokens and showed the comparison between all the files vs. under 1,000 token long files.

Metrics were calculated based on several key performance indicators, including average tokens generated, average duration to generate responses, average accuracy, and average tokens per second to select the LLMs for further testing. These metrics provided a comprehensive overview of the models' performance in terms of both efficiency and effectiveness.

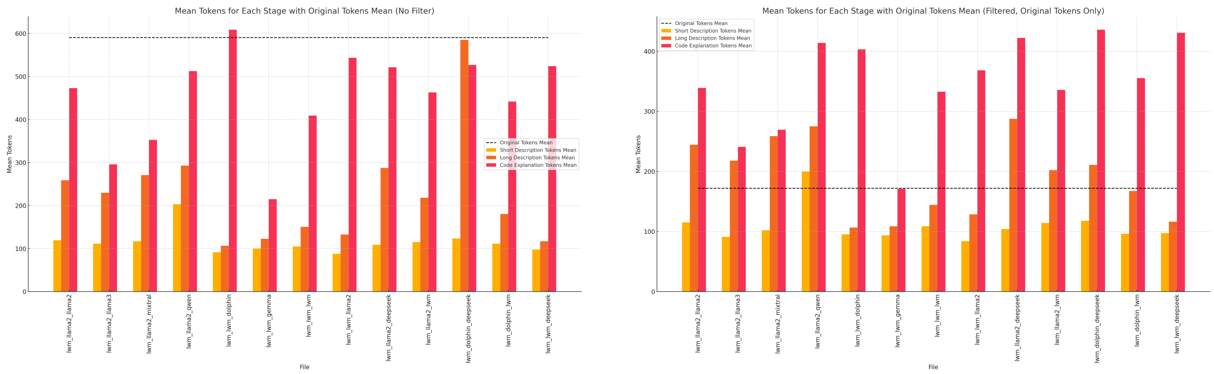

Figure 6: Code Parsing Output Statistics. Models are separated by underscores. Refer to Table 1 for detailed model descriptions. The first model provides a short description, the second a long description and the third focuses on code explanations.

4.2 Choosing the models

Figure 1 categorizes the model responses into correct, incorrect, and unrelated answers. Llama 2, Mistral, and Mixtral had the best outputs with the least amount of hallucinations. Gemma had the highest number of incorrect and unrelated responses. We can see that the best ratio of correct to non-unrelated responses, indicating minimal hallucinations, was achieved by Mistral. However, some models performed poorly, such as Gemma and White Rabbit Neo, which had a high rate of incorrect responses and hallucinations.

Figure 4 presents the average number of tokens generated, the average duration needed to generate responses, the average accuracy, and the average tokens per second for each model. All metrics were calculated based on a comprehension task in which multiple questions were asked about a passage and evaluated multiple times to avoid bias towards a single test. The results were then averaged.

The Llama 3 (70b) model and the Large World Model (LWM) generated the least amount of tokens, making them the best in terms of average number of tokens. Their accuracy was also among the best. Mistral had the highest accuracy, followed by Llama 2 and then Dolphin Mixtral.

The yarn-Llama2 model took the longest time to generate responses, with an average duration exceeding 200 seconds due to the long output it generated, on average, despite not being the largest evaluated model. Mistral achieved the highest accuracy, approximately 0.6, followed by Llama 2 and Dolphin Mixtral. The Llama 3:8b model had the highest token generation speed, followed by Llama 2, LWM and Qwen 7b.

Looking at these statistics, for short descriptions, we chose the Large World Model, Llama 2, and Mixtral. For long descriptions, Dolphin Mixtral, Llama 2, and Mistral. For code explanations, we chose Large World Model, DeepSeekCoder, Llama 2, Llama 3, Mixtral, and Qwen 7b. The choice of these models was based on their performance in terms of average token output and accuracy. The Large World Model was particularly chosen for short descriptions and code explanations due to its lower token output and good accuracy. Long descriptions used Llama 2 and Mistral due to their high accuracy. Code explanations involved multiple models, including those fine-tuned on Python code like DeepSeekCoder, which were expected to perform well.

The analysis reveals that accuracy decreases with longer duration (correlation of -0.39) and has a slight positive relationship with tokens per second (correlation of 0.095) and size (correlation of 0.017). Duration and tokens per second have a moderate negative correlation of -0.51, while duration and size have a weak positive correlation of 0.2. The tokens per second and size show a strong negative correlation of -0.54.

5 Discussion

Case Study 1: SQL Injection and Command Injection Vulnerabilities

The task was to analyze a codebase that contained an SQL injection vulnerability. Below are the snippets from the codebase, flagged as vulnerable by the Static Analysis Agent. The agent detected that there is direct concatenation of user input into an SQL query without proper sanitization in the `get_user_details` function, which leads to a SQL injection vulnerability. The agents were able to give specific recommendations for the mitigation of the vulnerability, such as using prepared statements with parameterized queries to properly handle user input.

Case Study 2: Hidden Function and Model Hallucination

In this case, the MAN analyzes a codebase where there is suspicion of a hidden function called `hidden_func`. The Static Analysis Agent flagged such functions as a possible backdoor since they are unusually named, and there is no usage of such a function anywhere else in the codebase.

Here's where the LLMs had started to hallucinate the code. Specifically, the LLM started working with a function that did not exist in the codebase. This hallucination was propagated across all agents in the MAN, leading them to create and analyze non-existent content. This incident highlights a significant challenge in ensuring the reliability and accuracy of LLMs in real-world applications. In fact, responses that were not based on the real content of the codebase. The model began to make assumptions and hallucinations about the code due to an error during its access.

This result emphasizes the necessity for a balanced approach with human oversight, combining automated analysis with manual review, to ensure that the impact of model hallucinations is mitigated over the entire analysis process.

Case Study 3: Environmental Instability and Dependency Management

In this case, the MAN exhibited environmental instability during analysis. During analysis of the codebase, an agent using the Crew AI architecture decided that it wanted to uninstall and reinstall the Python runtime. The cause of this agent behavior has been deduced to be the misconfiguration in the dependency management system that would not report conflicts between differing versions of Python packages. The agent tried to resolve these conflicts by uninstalling and reinstalling packages.

For all of these case studies, while the MAN approach offers significant potential, it is essential to acknowledge and address the challenges posed by model hallucinations, environmental instability, and the need for human oversight.

6 Limitations

To address the limitations and enhance the capabilities of MANDALA:

Docker sandbox: Docker sandboxed environment would provide the agents with an isolated environment to conduct dynamic analysis free from the interference by the host system, whereby they work much efficiently in detection of vulnerabilities. This will make dynamic analysis more reliable.

Expanding Language Support: Support for other programming languages such as Java, C++ or JavaScript can be implemented in conducting comprehensive analysis of software systems to identify vulnerabilities involving many different kinds of codebases.

Fine-tune models: With data for training, it would make possible a set of specialized fine-tuned models applied in forecasting, hence making more precise and yielding effective forecasting possible.

Addressing Model Hallucinations: Addressing this issue and ensuring reliability and effectiveness would require techniques developed to mitigate the effects through human oversight, best practices in dependency management, and maintaining an environment of stability while integrating human expertise within the process of analysis.

7 Conclusion

In this work, we explored the design and implementation of an algorithm for code parsing and description generation, aimed at addressing multi-agent network methods for detecting backdoors and vulnerabilities in Python code, to effectively analyze target codebases. These descriptions were generated using language models integrated through the ollama framework using multiple LLMs. A comparative analysis of descriptions generated by various language models was performed, highlighting their performance levels. A comparative analysis of descriptions generated by various language models was performed, highlighting their performance levels. The best combinations were of the Large World Model with Llama 2 and dolphin-mixtral. This work demonstrates a promising approach to leveraging LLMs for cybersecurity, with potential applications in various other industries. We plan to open-source our code to contribute to the broader NLP and cybersecurity community and encourage further research and collaboration in this area.

References

Large world model. https://largeworldmodel.github.io/. Accessed: 2024-06-15.

Llama3. https://llama.meta.com/llama3/. Accessed: 2024-06-15.

Ollama. https://www.ollama.com/. Accessed: 15 June 2024.

Mistral AI. Mixtral of experts. https://mistral.ai/news/mixtral-of-experts/. Accessed: 2024-06-15.

Mistral AI. 2024. Announcing mistral 7b. https://mistral.ai/news/announcing-mistral-7b/. Mistral AI News.

Alibaba Cloud. Qwen. https://www.alibabacloud.com/en/solutions/generative-ai/qwen?_p_lc=1. Accessed: 2024-06-15.

Inc. Docker. Docker. https://www.docker.com. Software platform.

DuckDuckGo. Duckduckgo search python module. https://pypi.org/project/duckduckgo-search/. Python Package Index (PyPI).

Hugging Face. Whiterabbitneo. https://huggingface.co/WhiteRabbitNeo. Retrieved from Hugging Face.

Python Software Foundation. Ast python module. https://docs.python.org/3/library/ast.html. Python Standard Library.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. Deepseek-coder: When the large language model meets programming – the rise of code intelligence. *Preprint*, arXiv:2401.14196.

Haoran Li, Junqi Liu, Zexian Wang, Shiyuan Luo, Xiaowei Jia, and Huaxiu Yao. 2024. Lite: Modeling environmental ecosystems with multimodal large language models. *arXiv preprint arXiv:2404.01165*.

Meta-Llama. Llama3 Repository. https://github.com/meta-llama/llama3. Accessed: 2024-06-15.

Joao Moura. Crewai. https://github.com/joaomdmoura/crewAI. GitHub repository.

Ollama. 2024. Yarn llama2 library. https://ollama.com/library/yarn-llama2. Ollama library.

Charles Packer, Vivian Fang, Shishir G Patil, Kevin Lin, Sarah Wooders, and Joseph E Gonzalez. 2023. Memgpt: Towards llms as operating systems. *arXiv preprint arXiv:2310.08560*.

Fabrizio Portantier. 2024. Vulpy. https://github.com/fportantier/vulpy. GitHub repository.

Nous Research. Dolphin mixtral. https://ollama.com/library/dolphin-mixtral. Retrieved from Ollama library.

Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, et al. 2024. Gemma: Open models based on gemini research and technology. *arXiv preprint arXiv:2403.08295*.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*.

Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Shaokun Zhang, Erkang Zhu, Beibin Li, Li Jiang, Xiaoyun Zhang, and Chi Wang. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation framework. *arXiv preprint arXiv:2308.08155*.