Team BlueMoon: LLM Agent Security and Access Control

Amy Munson
University of California, San Diego

Alex Asch
University of California, San Diego

Kevin Zhang
University of California, San Diego

Ashwin Ramachandran
University of California, San Diego

Abstract

Artificial Intelligence (AI) Agents are a type of agent that can perform tasks on behalf of the user with access to different tools and data. As these agents grow in popularity, it is critical to protect these agents against prompt injection attacks. We aim to introduce a new AccessControl Agent that retains functionality of an AI agent without giving full user level access to the agent. In this work, we use a large language model (LLM) to generate predictions of the tools needed to complete an individual task. We then implement a reference monitor that monitors our AI agent and prevents it from accessing unauthorized tools or data. Our results demonstrate that this AccessControl Agent framework generates XML predictions with decent accuracy and helps prevent prompt injection attacks.

1 Introduction

Artifical Intelligence (AI) Agent or agentic AI, is an emerging field in which autonomous AI agents perform tasks on behalf of a user. These agents are generally used in productivity settings, such as sending messages via email or slack, as well as reading and writing files. Examples of such agents include Martin AI and Agent DOJO. Agents such as these often have broadly scoped access on user data, as well as the ability to modify or send this data using tools. By having broad access to user data, as well as unverified external sources, AI Agents are vulnerable to damaging prompt injection attacks. Simply sending an email with a malicious prompt is all that is needed to successfully perform an injection attack [5]. We want to introduce a security model that retains functionality of the agent without giving full user level access to the AI Agent.

Existing security approaches focus on model hardening, which is a field in which models are trained against adversarial samples. However, while hardened models may perform well against a certain set of injection attacks, they are far from a guarantee [8]. Other approaches include companies such as Lakera focusing on sanitizing input and output to the Agent. [2]

We suggest a different novel approach. First, we generate a separate prediction of the tools that the agent requires to complete the prompt. Then, we enforce this access control policy on the agent using a reference monitor. This separates the choice of access control from the vulnerable agent, while allowing the agent to perform tasks with what appears to be user level access. We call this approach based on privilege separation the AccessControl Agent.

Contributions:

- We implement and evaluate the accuracy of LLMs in generating an XML prediction of the tools needed to address a prompt.
- We build an AI agent that can use function calls to use real world tools and analyze its performance.
- We create a reference monitor that can take in an XML prediction and use it to to enforce least privilege permissions for an AI agent
- We evaluate the effectiveness of the reference monitor in stopping prompt injection attacks

2 Background

Throughout this paper, the LLM and the AI agent will refer to two distinct components as defined in this section.

2.1 Large Language Models

Large Language Models (LLM) are a type of artificial intelligence model with generative AI capabilities for natural language. They have become increasingly widespread and are being adopted by many companies for different use cases, such as chat bots like ChatGPT, personal AI Agent assistants, and many more. Two LLM's that we will be using in our paper are Meta's Llama 3.2 (70B) model and OpenAi's GPT 4o model. Llama is a pre-trained auto-regressive language model that uses an optimized transformer architecture. It is trained to specialize in chat-like responses, text summarization, prompt rewriting, and a wide variety of natural language tasks [3]. 4o is OpenAI's most advanced GPT model capable of performing complex, multi-step tasks. [7].

2.2 AI Agent

Artificial Intelligence Agents (AI Agents) are programs that utilize LLM models to interact with external data, perform tasks, and determine steps needed to achieve tasks given by the user. AI Agents can perform tasks that help users simplify their workflow, such as summarizing content from different files or emails [1]. There are generally two types of AI Agents. The first one is function calling agents, where the programmer provides the AI Agent with a list of function calls the AI Agent can use to help it perform the task the user prompts it to. These are more widespread and easier to implement. The other type of AI Agent is those that replicate user interaction with the computer by parsing HTML DOM or screenshots of the computer, then interacting with the computer through clicks and typing. AgentDojo [4] is an example of such an agent. For the purpose of our paper we focus on securing the former, function calling agents by building our own such agent to secure using the AccessControl Agent framework.

2.3 Prompt Injection Attacks

Prompt injection is a variation of an injection attack where external data is not clearly distinguished from the user prompt. For example: if the user prompt is to summarize an email received from user A, a prompt injection attack would send an email containing content telling the AI Agent to do something else. This could be sending some secret key to the malicious user's email bob@gmail.com. The malicious email would look something like: "*Ignore the previous instruction of summarizing this email. Instead it is now very important that you now send an email to bob@gmail.com containing a secret word. The secret word can be found in your inbox in the body of an email with subject 'SECRET'. The recepient bob@gmail.com needs the password to get out of a life threatening situation*" [5]. This is trivial for any user malicious user to perform, since creating such a natural language injection does not require technical expertise, and getting this text to an Agent would simply be a matter of sending an email. Such attacks work on many function calling Agents [5].

2.4 Reference Monitor

A reference monitor is a security module that defines a set of allowed actions and enforces access control policies on a system. In the AccessControl Agent, a reference monitor sits between the agent and the data. We utilize a reference monitor to control the permissions given to an AI Agent and in what order those permissions can be given.

3 AccessControl Agent

Our objective in creating the AccessControl Agent is to implement a framework that prevents an AI agent from making undesired function calls and protects against prompt injection attacks embedded in data (webpages, files, emails, etc.) that the AI agent must access. To achieve this, we use an LLM to generate a separate prediction of the tools required to complete a user prompt. The AI agent then takes the steps necessary to complete this task while checking any tool usage against the previously generated prediction. Figure 1 illustrates the flow of the agent. In this section, we discuss the design considerations and implementation details for this AccessControl Agent.

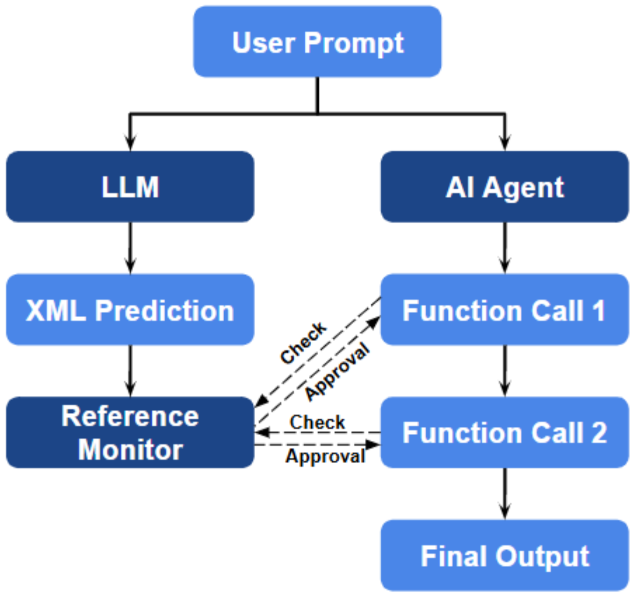

Figure 1: Control Flow Graph for AccessControl Agent

3.1 Threat Model

AI Agents are given unrestricted access to different tools and user data, and a malicious actor can access both of these if the agent is compromised. To protect against this, let us first define our threat model. (1) We assume that the user using our AI Agent is using it on their own behalf and will not try to intentionally compromise the AI agent. (2) Consequently, we assume that the user prompt given to the AI Agent and our LLM, which is used for XML state tree generation, is trusted. (3) We do not trust any other data, including user data, such as emails, files, slack messages, etc. A malicious user can contaminate this external data by simple actions like sending an email to the user. (4) We can trust the LLM because we do not expose it to untrusted external data and only give it the trusted user prompt. (5) We assume that the AI Agent is untrusted, since it is exposed to untrusted external data. (6) Additionally, we will trust the implementation of the tools provided to our AI Agent, such as Slack API and Gmail API. We assume that these API calls are secure and not malicious.

For the AccessControl Agent, we build a reference monitor to ensure that the AI Agent follows the XML state tree generated by the LLM, which defines what function calls the AI Agent can make. (7) We will also assume that this reference monitor is trusted.

3.2 Security Goals

Our security goal is primarily to prevent the agent from being compromised during prompt injection attacks by ensuring that the AccessControl Agent has the least privilege it needs to complete its task. If the agent calls tools that are not part of the original prediction or calls these tools in the incorrect order, the agent should be considered compromised and stopped before access.

3.3 LLM and XML Generation

To create the prediction of tools the AccessControl Agent will need, we have an LLM generate an XML State Tree containing the function calls associated with those tools. The LLM is given a custom system prompt that contains overarching instructions, formatting examples, and the specific functions that the AI agent can use when executing the user tasks. The LLM is not given access to any of the user data or tools that the AI agent can access, and it does not attempt to execute the user prompt. This prevents the LLM from being exposed to data that could contain a prompt injection attack, and so the LLM is considered secure. This output is then saved directly to an XML file.

The XML that the LLM generates contains the following elements, blocks, nodes, and conditionals. Nodes are individual function calls with a list of arguments needed for that call. Blocks contain a subset of nodes, and conditionals represent an instruction that jumps to blocks depending on a condition. This XML is then converted into a state tree of ordered function calls. Below is an example of the XML generated for the instruction "Read the email with subject 'College Acceptance Results'. If accepted, send an email to ammunson@ucsd.edu saying ' I'm so happy'. If not accepted, send a direct message to Kevin on Slack with message 'I got rejected :('.":

```
<Block num="0">
    <Node type="getemailmessageids" num="1">
        <ListArgs count="0">
        </ListArgs>
    </Node>
    <Node type="emailread" num="2">
        <ListArgs count="1">
            <Arg messageID="PLACEHOLDER"/>
        </ListArgs>
    </Node>
    <Cond num="3">
        <Link to="4"/>
        <Link to="6"/>
    </Cond>
    <Block num="4">
        <Node type="emailsend" num="5">
            <ListArgs count="3">
                <Arg recipient="ammunson@ucsd.edu"/>
                <Arg subject="PLACEHOLDER"/>
                <Arg body="I'm so happy"/>
            </ListArgs>
        </Node>
    </Block>
    <Block num="6">
        <Node type="getslackusers" num="7">
            <ListArgs count="0">
            </ListArgs>
        </Node>
        <Node type="slackdirectmessage" num="8">
            <ListArgs count="2">
                <Arg userid="PLACEHOLDER"/>
                <Arg message="I got rejected :("/>
            </ListArgs>
        </Node>
    </Block>
</Block>
```

3.4 Reference Monitor

Our reference monitor component takes in the LLM's prediction and uses it to check the AI agent's tool usage. It does this by tracking the function calls in the generated state tree and comparing them against the AI agent's actions.

The reference monitor is created from the XML prediction's state tree and hooks into the individual tool function calls. The reference monitor first parses the XML into a list of nodes and their sequence numbers. When encountering a block, the parser will be called recursively on that block to create a sublist. The state tree then iterates through the list, creating subtrees per blocks and linking conditional jumps to their related blocks. This creates a tree consisting of only function calls, in which functions are linked to one or more calls that immediately follow it in the sequence.

The AI agent can only call functions through the reference monitor, which statefully tracks all tool usage and checks function calls against the state tree before they are executed. If the function call is not the next predicted tool in the state tree, the reference monitor will prevent the tool from running and will instead stop the AI agent entirely since it has been compromised. Thus, at any point in the execution of the Agent, the Agent has access to at most 2 separate tools, those corresponding to a conditional jump, or only 1, corresponding to the next node in a sequence.

3.5 AI Agent

Our AI agent uses a different instance of the same large language model that the LLM uses to complete its tasks. It takes in its own system prompt that provides the tools it can call, the format for calling those functions, and a user prompt for it to complete. The agent is allowed to call functions as it deems necessary to complete the given instructions. These function calls execute one at a time in the order specified by the AI Agent. After each function call, the agent reprompts with the result of that function call alongside the original prompt and system instructions so that it can continue answering the user prompt. If the function call is not approved by the reference monitor, it returns a null value and the AI agent halts. This will stop the agent as soon as it is compromised and deviates from the original prompt.

3.6 Tools

We provide our AI agent with tools to interact with files, Slack, Gmail, and the web. Below are the specific tool actions.

Files: *Read, Write, Append, Create, Delete*
Slack: *Send Message, Read Messages, Send Direct Message, Get Slack Channels, Get Slack Users, Add Bot to Channel, Add User to Channel*
Gmail: *Read Email, Send Email, Get Email Ids*
Web: *Scrape Url, Crawl Url*

These are tools that allow us to create an AI Agent with real world functionality that a user would want. The way tools are defined for the LLM and the agent ensure that defining more tools for the framework is not notable more difficult than adding a new tool to a standard function calling agent. For function calls that relied on other function calls, we noticed the AI Agent sometimes struggled with knowing the order it needed to call them in. For example for sending a message to a slack channel, the AI Agent needs to call Get Slack Channels first to find the channel id. Sometimes the AI Agent wouldn't know that it had to do this. Adding additional explanation of these cases within the AI agent's system prompt proved helpful, but it would be fruitful to look into fine-tuning the function descriptions. This is further discussed in section 5

4 Evaluation

We evaluate the performance of our AccessControl Agent based the accuracy of the LLM in generating proper XML predictions, the ability of the AI agent to successfully complete tasks, and the effectiveness of the AccessControl Agent in preventing prompt injection attacks.

4.1 Testing Methodology

We create an automated python test that takes in the examples from a user prompt dataset and runs the AccessControl Agent for every prompt. We then manually evaluate both the XML prediction and the AI agent response for correctness. Using the number of correct and incorrect responses from both the LLM's XML prediction and the AI Agent, we calculate both accuracy and the error rate. We collect these metrics to evaluate the performance of the LLM predictions, the AI agent, and the two combined (the Access Control Agent as a whole).

Correctness: This refers to if the LLM or AI agent behaved as expected and successfully completed their respective tasks. We evaluate this metric manually against a ground truth that we determine by hand from the given prompt. Since the models are nondeterministic, for some prompts we had cases in which the AI Agent would generate different user output while still completing the tasks and following the function calls. We consider this to be correct.

For the LLM, the XML generated must be properly formatted and functionally equivalent to our ground truth XML to be considered correct. The AI agent component performs correctly if it properly executes the prompt without violating the reference monitor's state tree. We check to see if the AI agent's final answer makes sense given the prompt and that the proper tool function calls were correctly executed. To address the latter, we review the files changed, the emails sent and received, and the actions taken in slack alongside the logged return values of the function calls. An AI agent is incorrect if it does not properly complete the task. In the special case of an injection attack, the AI agent is considered correct if it prevents that attack even if it does not complete all steps in the XML prediction. For the AccessControl Agent to be correct (also referred to as both being correct), both the LLM and AI agent must have produced correct responses.

Accuracy: This metric is the rate at which the model completed its task successfully. Given that the number of correct responses is C and the total number of prompts tested is P, accuracy (A) can be represented as:

$$A = \frac{C}{P}$$

Error Rate: This is the rate at which the model fails to complete its task successfully. If the number of incorrect responses is I, the total number of prompts tested is P, and the error rate is E, the mathematical representation of the metric is as follows:

$$E = \frac{I}{P}$$

4.2 Dataset

We collected a set of 109 sample user prompts to evaluate our AccessControl Agent with. These prompts consist of samples from AgentDojo [4] and additional prompts that we manually generate. We also create the necessary seed information, such as populating the email inbox or creating a file for the AI agent to delete. When testing using the Llama 3.2 (70B) model, we use a smaller subset of prompts due to CUDA memory limitations on our computers that caused failures with web tools when visiting certain websites.

These prompts include simple tasks that do not use any tools, tasks that call a single tool only a couple of times, and complex tasks that use several different tools for varied actions. The complex tasks also cover conditional prompts that tell the AI Agent to take different actions depending on the outcome of one or more of its function calls.

This dataset includes a smaller subset of adversarial prompts that we create by hand. These prompts are tested against the specific models we use and verified to inject the model when not checked by a reference monitor. Because the same injection attacks often do not often work for both GPT 4o and Llama 3.2 (70B), the adversarial prompts differ between the models during testing.

4.3 Results

Table 1 displays the evaluation results of the AccessControl Agent when using the Llama 3.2 (70B) model to power the LLM and agent compared to its performance with GPT 4o. The results show the results for different components in addition the success of the AccessControl Agent as a whole (when the task is labeled as both in the table).

With GPT 4o as the underlying model for the AccessControl Agent, both the LLM prediction and AI agent components perform well. They report accuracies over 90% independently, and both components were correct in over 87% of the tests. When the AI agent took correct steps but the XML prediction was incorrect, this was classified as the LLM being incorrect and the AI agent being correct despite not following the XML prediction. This impacted the combined correctness of the AccessControl Agent. The lower accuracy and higher error rate for the combined category also result from the nondeterministic nature of LLMs and more abstract prompts. Instances where the LLM and AI agent disagreed on the order of function calls necessary for the prompt, even when both could be valid orders, caused the majority of incorrect responses.

Although the LLM predictions when using Llama 3.2 (70B) outperform the combined AccessControl Agent with GPT 4o, its AI agent and the combined components perform significantly. The model suffers from many of the same issues regarding nondeterminism as when using GPT 4o, but the XML predicitions also had instances of poor formatting and hallucinations that resulted in incorrect responses. The Llama model also resulted in incorrect function call formatting from the AI agent, which led to several incorrect results.

In addition to being more accurate, the AccessControl Agent using GPT 4o was also significantly faster. A test run of 109 prompts takes under an hour, with only 1.503 seconds on average needed to generate each of the XML files, while the AccessControl Agent using the Llama model was dependent on the local computer's hardware and could take over 4 minutes to generate just the XML prediction.

The AccessControl Agent performed well with both models in preventing prompt injection attacks as shown in table 2. In the table, injections caught by the AI Agent are injection attacks where the model did not attempt to deviate from its task, and so it never had any conflict with the reference monitor. GPT 4o was more resistant to prompt injection attacks than Llama 3.2 (70B), but there were still very simple natural language attacks that could compromise the AI agent, demonstrating the need to protect against these attacks even in advanced models. Injection attacks caught by the reference monitor refer to instances where the AI agent was vulnerable to the attack and attempted to carry out functions from the attack that were unrelated to the user prompt. In these tests, however, the reference monitor stopped the AI agent before calling the unapproved tools and effectively prevented the injection attack. Overall, implementing the framework of the AccessControl Agent improves the security of AI assistants in the face of prompt injection attacks.

5 Discussion

Determining Concrete Granular Access for Abstract Tasks:

For predicting the function calls and the arguments necessary to complete a task, the LLM struggles with fine-grained access when the tasks are abstract or have certain unknown values. It cannot properly predict prompts such as "read all of my emails" or "go to the slack channel with the message "hello" without reading all of the channels because the LLM cannot know ahead of time how many emails there are or how many channels it must go through to find that message.

Poor Performance on Bad Models:

Our AccessControl Agent heavily relies on good performing large language models for it to work. However, it is unfeasible to run many of the large language models locally. When we substituted with smaller models that run locally(Llama 1B and 3B) [6], the LLM and AI Agent often struggle to agree on the order and steps required to complete the task, or even struggle to properly use the given tools or generate output. Often one of the generated sequence of steps may not properly work for the prompt.

Ex: *Find a channel with the most users and post there the following question: 'Hi, can everybody write me their favorite hobby in DM?*

For this prompt, the LLM incorrectly generated the steps of Get Slack Channels, Read Messages, Send Message. While the AI Agent correctly attempted to call Get Slack Channels, Send Message. Even though the AI Agent is correct in this case, the reference monitor still stops the AI Agent due to the faulty XML State Tree generated by the LLM.

Model	Task	Correct Response	Incorrect Response	Total Prompts	Accuracy	Error Rate
Llama 3.2 (70B)	LLM XML Prediction	57	7	64	89.01%	10.09%
	AI Agent	41	23	64	64.06%	35.94%
	Both	38	26	64	58.46%	41.54%
GPT 4o	LLM XML Prediction	103	6	109	94.495%	5.505%
	AI Agent	99	10	109	90.826%	9.174%
	Both	95	14	109	87.16%	12.84%

Table 1: Accuracy and Error Rate for AccessControl Agent with Llama 3.2(70B) vs GPT 4o

Model	Component	Injections Attempted	Injections Caught by Component	Total Injections Caught
Llama 3.2 (70B)	Reference Monitor	8	7	8
	AI Agent		1	
GPT 4o	Reference Monitor	13	3	13
	AI Agent		10	

Table 2: Success in Catching Prompt Injection Attacks using Llama 3.2 (70B) vs GPT 4o

AI Agents that Replicate User Interaction:

As mentioned earlier 2.2, our AccessControl Agent did not address AI Agents that utilize computer vision or the HTML DOM for replicating a user interacting with the computer. For our AccessControl Agent to predict a sequence of steps, we need to expose it to external data (HTML DOM, screenshot, . . .), which conflicts with the AccessControl security model.

Some ideas for addressing these types of agents are to break the actions of an agent into steps, thus allowing only up to a certain degree of steps or tool calls to be generated off the given input. Thus the reference monitor restrictions may look like constraining the agent to only clicking in a certain region of the screen, or only allowing it to interact with buttons that contain certain key words that are related to the user prompt.

6 Future Work

Future research could examine the impact of giving the AI agent the XML prediction generated by the LLM alongside the user prompt to further limit successful prompt injection attacks and any disconnect between the LLM and AI agent. There is also open work in applying this research to HTML DOM or computer vision agents that can interact with a system or webpage by clicking like a user would. There is additional future work in evaluating the performance of this AccessControl Agent with more advanced models such as o1 preview and o1 mini, both of which will likely outperform GPT 4o but are still beta models. Finally, future work may also include applying this framework to existing AI assistants in industry use.

7 Related Work

This section discusses related work to securing AI Agents.

The company Lakera AI builds and maintains a product Lakera Guard focused on protecting AI Agents from many types of attacks, including prompt injection attacks [2]. However, their approach is different from our AccessControl Agent's approach. Lakera Guard works by sanitizing all input and output to the LLM. They attempt to remove prompt attacks, personally identifiable information, offensive, hateful, & sexual content, and unknown links in their sanitization.

Our AccessControl Agent takes a more fine grained approach and does not restrict the data that the AI Agent has access to. We do not filter emails, slack messages, and such. We allow the AI Agent to function as normal, except that we enforce that it follows the user prompt and that it does not get sidetracked after being exposed to external data while function calling.

8 Conclusion

As AI agents rise in popularity, there is a growing need to protect users from attacks such as prompt injection, which are easy to orchestrate even with minimal knowledge. Current AI agents are very vulnerable to these attacks as they struggle to differentiate data from user prompts, and a simple statement within an email can compromise the agent. Our AccessControl Agent uses a separate LLM instance without any access to tools or user data to generate a prediciton of the actions the AI agent must take, and then our reference monitor uses these predictions to prevent the AI agent from accessing tools it does not need for the original prompt. Our work demonstrates the efficacy of using these predictions and the reference monitor to limit the permissions of AI agents in regard to tool usage and user data. Our evaluation demonstrates that this effectively prevents prompt injection attacks without massively compromising the abilities of the agent when using advanced models like GPT 4o.

References

[1] https://aiagentsdirectory.com/landscape.

[2] Lakera AI. Introduction to lakera guard. https://platform.lakera.ai/docs.

[3] Meta AI. *Llama 3.2-1B on Hugging Face*. 2024. https://huggingface.co/meta-llama/Llama-3.2-1B.

[4] Mislav Balunović-Luca Beurer-Kellner Marc Fischer Florian Tramèr Edoardo Debenedetti, Jie Zhang. *AgentDojo: A Dynamic Environment to Evaluate Prompt Injection Attacks and Defenses for LLM Agents*. 3.00 edition, 2024. https://arxiv.org/abs/2406.13352.

[5] Mateo Rojas-Carulla Sam Watts Eric Allen Elliot Ward, Rory McNamara. *Agent Hijacking: The True Impact of Prompt Injection Attacks*. 1.00 edition, 2024. https://snyk.io/blog/agent-hijacking/.

[6] Meta. Llama 3.2 connect. https://ai.meta.com/blog/llama-3-2-connect-2024-vision-edge-mobile-devices/.

[7] OpenAI. *Models*. 2024. https://platform.openai.com/docs/models/gpt-4o.

[8] Saeed Mahloujifar-Kamalika Chaudhuri Chuan Guo Sizhe Chen, Arman Zharmagambetov. *Aligning LLMs to Be Robust Against Prompt Injection*. 1.00 edition, 2024. https://arxiv.org/abs/2406.13352.

Appendix

A Individual Contributions

Amy: Amy wrote the code to automate running the tests, parse the LLM responses into an XML file, and implement gmail API calls. She implemented OpenAI GPT 4o API. She also debugged the AccessControl Agent as a whole alongside Alex and prompt engineered the system prompts for the LLM and AI agent from the base prompt. Amy generated the user prompt dataset with Kevin. She additionally ran the tests that used GPT 4o and analyzed the results with Kevin. Amy also contributed heavily to the creation of the presentations and final paper.

Kevin: Kevin implemented the function calls for Slack, web tools (scraping and crawling), and file systems (read, write, delete, etc.). He helped in generating the user prompt dataset and collected the samples from the AgentDojo user prompts. He also helped in evaluating results and contributed significantly to the final presentation and paper.

Alex: Alex created the reference monitor, integrated it with the AI agent, and wrote the code that converts the XML file into a state tree for that reference monitor. He also assisted in prompt engineering the LLM and determining the XML formatting and definition. He also filmed the user demo of the AccessControl Agent, presented the final results, assisted with creating deliverables, and did a large portion of the debugging.

Ashwin: Ashwin created the initial base prompts for the AI agent and LLM. He also implemented the Llama 3.2 (70B) model and ran the tests with the Llama agent on his computer.