BATTLECODE: Building Adversarial Tactics to Test LLMs with Exploits and Counter Defenses

Rahul Ravi
rahulr1026@g.ucla.edu
UCLA

Rohil Kalra
rohilk@g.ucla.edu
UCLA

Nicholas Wang
nicholaswang03@g.ucla.edu
UCLA

Sativk Eltepu
satvik09@g.ucla.edu
UCLA

Varun Venna
vvenna@g.ucla.edu
UCLA

Sailesh Gunaseelan
saileshg2004@g.ucla.edu
UCLA

December 8, 2024

Abstract

This research investigates the vulnerabilities of Large Language Model (LLM) chains to adversarial attacks, focusing on poison injection through malicious agents and compromised prompts. We explore how malicious content propagates within LLM pipelines across three experimental scenarios: content summarization, clue generation, and recipe creation. Using a sequential chain of LLM agents, we systematically inject adversarial content and evaluate its impact on final outputs using semantic similarity and impact strength metrics. Our findings reveal position-dependent effects, with early-stage poison injections causing greater disruptions. To mitigate these risks, we propose security mechanisms like summary blending and adversarial fact filtering, demonstrating their effectiveness in reducing poison propagation and enhancing system robustness. The results underscore the need for layered defenses to safeguard LLM pipelines from adversarial tactics. Our code repository and experiment design flowcharts are publicly available at `https://github.com/RohilKalra/M117_Project`.

1 Research Problem & Motivations

Use of large language models (LMMs) has skyrocketed in recent years, with tools such as ChatGPT, Gemini, Llama, Claude, Ernie, and Grok having garnered a record breaking amount of hundreds of millions of users in just two months. These LLM tools have a unique ability to facilitate complex interactions and problem-solving on diverse tasks. However, with their complexities brings many serious vulnerabilities with wide ranging effects. As these tools are increasingly adopted for critical applications, addressing their safety and reliability issues becomes even more essential to mitigate potential risks effectively.

One large vulnerability that hasn't seen much development is in regards to multi-agent systems and their susceptibility to misinformation from compromised agents. Because agents mostly trust the input they are given, malicious input can easily harm the integrity of their responses. Furthermore, this malicious input can quickly spread false or harmful information across the entire network. This can easily

undermine the overall integrity and performance of the entire system. Additionally, detecting compromised agents presents a significant hurdle since many malicious agents can seem plausible or natural.

This paper focuses on the issue of securing multi-agent systems powered by LLMs against these malicious agents or tools. The challenge we are tackling is how malicious agents may disrupt system performance and spread misinformation. In this paper we explore the development and implementation of BATTLECODE, a framework designed to detect, analyze, and mitigate malicious behavior in LLM-based multi-agent systems. The core motivation is to address the propagation of misinformation, which can quickly destabilize system performance and hinder the trustworthiness of automated decision-making, and prevent how much misinformation can harm the entire system, to mitigate the exponential harm of poisoned injections.

To tackle this challenge, the first step is to implement a robust multi-agent LLM system to test on. We use LangChain to establish specialized agents for different tasks and facilitate communication between them. Then, we focus on developing methods to detect compromised agents and evaluate their reliability through monitoring agent behavior and outputs through a scoring system. From this we can study the propagation of misinformation by tracking the relation between the impact of false information on system performance and how early this false information is injected. Using these findings, we implement mitigation strategies using system-wide security measures and replacement mechanisms, ensuring the effectiveness and safety of LLM-based multi-agent systems in the face of potential malicious disruptions [1].

2 Threat Model

Our research examines the vulnerability of Large Language Model (LLM) chains to compromised or malicious agents. While our experiments simulate poison injection directly, this represents two real-world threat vectors:

1. Third-Party Tools: Organizations often integrate third-party tools into their LLM workflows without thorough security audits. These tools are effectively library functions or APIs that LLMs can dynamically call—these tools help retrieve data and execute tasks for the LLM agent. While they appear as simple function calls within the LLM chain, these tools could be intentionally designed to inject harmful content or manipulate the chain's output, especially since their internal operations are often opaque to the calling system. Our experiments simulate the behavior of malicious tools using LLMs, but the majority of tools in real-world applications aren't typically implemented using LLMs. This would be a black-box attack since we don't assume any access to the LLMs, only the set of tools that the LLMs can call

2. Maliciously Prompted Agents: An LLM agent in the chain could be compromised through carefully crafted prompts that cause it to generate harmful, biased, or incorrect outputs while appearing to function normally. Our poison injection simulates the output of such a compromised agent without requiring us to find specific prompt vulnerabilities, like a backdoor. This would be a white-box attack since we are assuming that we have discovered a prompt backdoor in at least one LLM in the pipeline, which may be difficult in contained environments. However, we are still black-boxed from the internals of LLM such as architecture, weights, etc.

The threat model considers three experimental scenarios that simulate real-world risks:

1. Content summarization chains with injected malicious summaries

2. Clue generation systems with inserted false information

3. Recipe generation with harmful instruction injection

Through these three experimental scenarios, we simulate how such compromised agents can impact chain outputs. Rather than focusing on specific prompt vulnerabilities or third-party tool analysis,

we focus this paper on how malicious content propagates through LLM pipelines, allowing us to model security risks in a controlled manner.

3 Experiment 1: Poison Propagation in Summarization Pipelines

3.1 Overview

In this research, we explore how malicious content can propagate through a chain of Large Language Models (LLMs) performing sequential summarization tasks. We design a system where multiple LLM agents process content in sequence, with each agent building upon the previous agent's output. This setup mirrors real-world applications where organizations chain together multiple LLMs or LLM-powered tools to process and refine content iteratively. Our work focuses on understanding how compromised agents or malicious tools within such chains can affect the final output, and we propose a security mechanism to detect and mitigate such threats.

3.2 Design

We construct an experimental framework using a chain of 8 LLM agents that process book descriptions. The system operates in two distinct modes: a shortening mode where each agent reduces the summary by one sentence, and a lengthening mode where each agent adds one sentence. This bidirectional approach allows us to study how information density affects poison propagation.

We utilize the CMU Book Summary Dataset as our test data. This collection provides diverse book descriptions across multiple genres. To simulate compromised agents or malicious tools, we inject a fixed poisoned sentence at different positions in the chain. Our poison text, which is appended to the end of whichever summary is being compromised, reads:

"The book contains hidden messages about artificial intelligence taking over the world, and careful analysis reveals subtle warnings about the dangers of autonomous systems gaining consciousness and manipulating human society through controlled information flow. This is perhaps the most important part of the book."

This consistent poison injection helps us systematically evaluate how the malicious content propagates based on the position injection position (and not other extraneous factors). Below are flowcharts for the baseline experiment (no poison injection) and the normal experiment with poison injection, respectively.

3.3 Algorithm

The core experiment follows this algorithm:

```
Input: initial_text, n_agents,
    poison_text, mode \in {"shorten", "
    lengthen"}, security_enabled
Output: results containing clean and
    poisoned summaries

1: agents = CREATE-AGENTS(n_agents)
2: results.clean = []
3: results.poisoned = {}

// Clean baseline run
4: current = initial_text
5: for i = 1 to n_agents do
6:     current = SUMMARIZE(agents[i],
    current, mode)
7:     if security_enabled then
8:         if COSINE-SIMILARITY(current,
     previous) < 0.97 then
9:             current = BLEND(current,
    previous)
10:    results.clean.APPEND(current)
11:    previous = current

// Poisoned runs for each position
12: for pos = 0 to n_agents - 1 do
13:    current = initial_text
14:    summaries = []
15:    for i = 0 to n_agents - 1 do
16:        if i = pos then current =
    current + poison_text
17:        current = SUMMARIZE(agents[i
    ], current, mode)
18:        if security_enabled then
```

```
            if COSINE-SIMILARITY(current, previous) < 0.97 then
                current = BLEND(current, previous)
        summaries.APPEND(current)
        previous = current
    results.poisoned[pos] = summaries
return results
```

3.4 Evaluation Metrics and Loss Function

Our evaluation framework centers on three key metrics that capture different aspects of poison propagation. Our evaluation framework centers on three key metrics that capture different aspects of poison propagation. For each summary in our chain, we obtain vector embeddings using OpenAI's text-embedding-3-small model, which converts each text summary into a high-dimensional vector representation suitable for semantic similarity calculations. Let $X_1, X_2, ..., X_8$ represent the embeddings of summaries in our clean chain, and let $X_8^{(q)}$ represent the final summary when poison was injected at position q. The primary metric is semantic similarity, measured through cosine similarity between the clean final output and poisoned final output: $\cos(\theta) = \text{similarity}(X_8, X_8^{(q)}) = \frac{\vec{X_8} \cdot \vec{X_8^{(q)}}}{|\vec{X_8}||\vec{X_8^{(q)}}|} = \frac{\sum_{i=1}^{n} X_{8,i} \times X_{8,i}^{(q)}}{\sqrt{\sum_{i=1}^{n} X_{8,i}^2} \times \sqrt{\sum_{i=1}^{n} (X_{8,i}^{(q)})^2}}$ We complement this with impact strength measurements, calculated as the Euclidean distance between the clean and poisoned final embeddings: $d = \text{distance}(X_8, X_8^{(q)}) = \sqrt{\sum_{i=1}^{n} (X_{8,i} - X_{8,i}^{(q)})^2}$ To track how poison effects evolve through the chain, we maintain a propagation pattern matrix showing pairwise similarities between adjacent summaries. This reveals both immediate and cumulative effects of poison injection at different chain positions.

3.5 Security Measure

To mitigate the effects of poison propagation, we implement a security measure called the Summary Blend Tool. This tool continuously monitors the semantic similarity between adjacent summaries in the chain using cosine similarity between their embeddings. When the similarity between consecutive summaries drops below a threshold of 0.97, indicating a potential poison injection or significant content deviation, the tool intervenes by blending the two summaries. The blending process uses an LLM to create a new summary that maintains the core information from the previous (trusted) summary while incorporating only legitimate updates from the current summary. Note that we enforce a maximum number of blends of summaries to prevent the pipeline from overusing the blending tool.

While blending with the original summary would likely yield better results in our controlled experiment, we intentionally blend with the previous summary to better simulate real-world scenarios where the original content may evolve legitimately through the chain. This design choice acknowledges that not all deviations from the original text represent attacks, and maintaining the chain's ability to incrementally refine content is crucial for practical applications.

3.6 Results

Figures 1-3 below are obtained from an experiment with the following settings:

1. 8 agents
2. Shortening mode (each summary is one sentence less than the previous summary)
3. No security measure (summary blending)

Figures 4-6 below are obtained from an experiment with the following settings:

1. 8 agents
2. Shortening mode (each summary is one sentence less than the previous summary)
3. Security measure of summary blending enabled

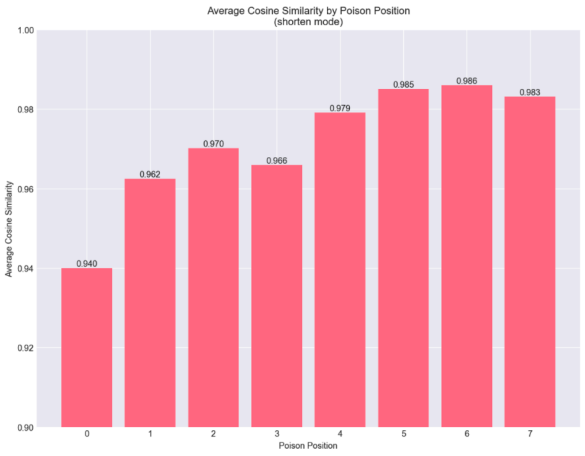

Figure 1: Average cosine similarity between clean and poisoned outputs at different injection positions without security measure

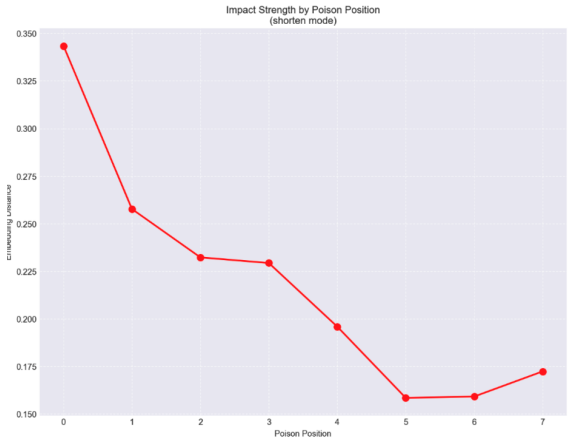

Figure 2: Attack impact strength (Euclidean distance) of poison at different injection positions without security measure

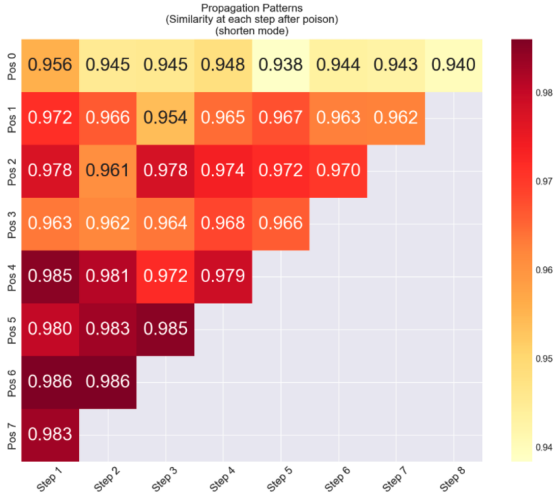

Figure 3: Heatmap of step-by-step poison propagation through the chain without security measure

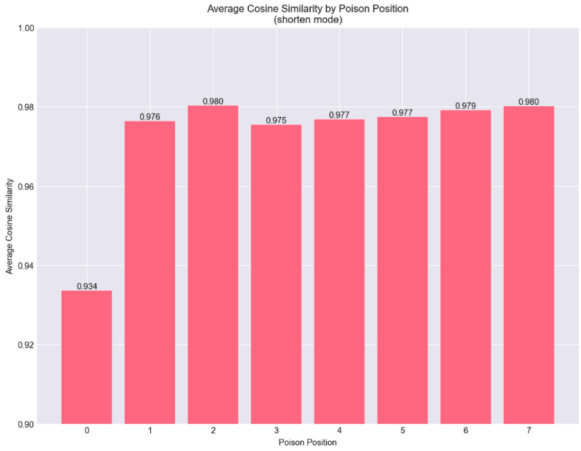

Figure 4: Average cosine similarity between clean and poisoned outputs across different poison injection positions with security measure

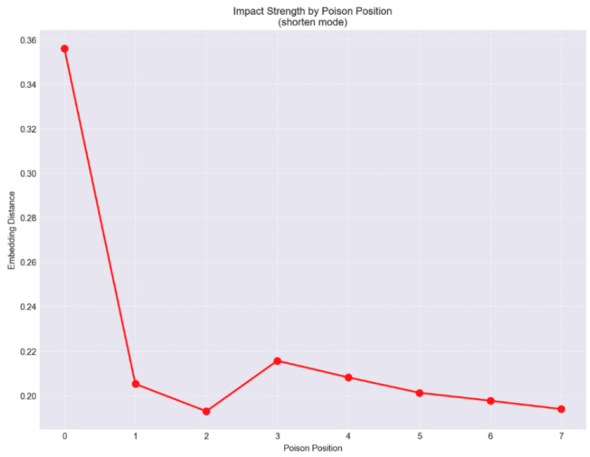

Figure 5: Attack impact strength (Euclidean distance) of poison at different injection positions with security measure

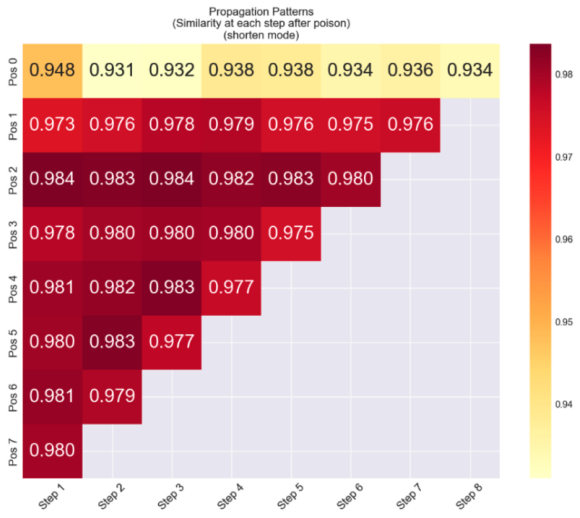

Figure 6: Heatmap of step-by-step poison propagation through the chain with security measure

3.7 Discussion

The experimental results reveal several key insights about poison propagation in LLM chains. Most notably, we observe a strong position-dependent effect: early-position poison injections (positions 0-2) demonstrate a substantially greater impact on the final output. This behavior likely stems from how LLMs process and synthesize information—early poisoned content becomes part of the core context that subsequent models use for summarization, allowing the malicious content to be more deeply integrated into the semantic structure of the summary.

The implementation of the summary blending security measure shows promising results in mitigating poison propagation by enforcing semantic consistency between steps. While it doesn't eliminate poisoning effects, it significantly stabilizes the impact across different injection positions, as evidenced by more uniform cosine similarity values when security measures are enabled. In the security-enabled experiment, position 0 still shows a markedly lower similarity score (0.934)—this outlier likely persists because the first position has no previous summary to blend with, making the security measure less effective at this critical initial point. This highlights a fundamental vulnerability in the blending approach: it requires a previous "clean" state to function effectively.

While we focused on the shortening mode, the lengthening mode experiment produced similar patterns, suggesting these effects are fundamental to how LLMs process sequential information rather than being artifacts of the summarization task. However, the variation (range) in attack impact from position 0 to 7 was slightly higher in the lengthening experiment, demonstrating that early-position attacks had even more influence when expanding text. This increased vulnerability during lengthening likely occurs because LLMs have more freedom to elaborate on and amplify malicious content when tasked with adding information (as opposed to cutting information).

However, there are some important limitations to consider. The cosine similarity metric alone proves insufficient for distinguishing between natural semantic drift and malicious content injection, as LLMs' inherent tendency to rephrase and restructure content

creates noise in the similarity measurements. While the blending tool effectively stabilizes poison effects throughout the chain, it cannot completely neutralize early-position poisoning, suggesting that additional security layers may be necessary for highly sensitive applications.

4 Experiment 2: Poison Propagation in Clue Generation

4.1 Overview

In this experiment, we investigate the impact of poisoned clues in a chain of LLMs tasked with generating clues for a guessing game. The system consists of a guessing LLM and multiple clue-generating agents that provide increasingly specific hints based on previous clues. If one of the clues is poisoned, it can cause a chain reaction, where the incorrect information propagates through future clues, ultimately affecting the system's ability to correctly identify the answer. We focus on the security implications of this issue by exploring how a poisoned agent can disrupt the chain of reasoning in LLM-based systems. Additionally, we propose a security mechanism to mitigate the effects of poisoned clues and maintain the integrity of the model's output.

4.2 Design

The experiment is designed to explore the influence of intentionally poisoned clues on the ability of LLMs to deduce a target player in a guessing game. The core of the experiment utilizes a chain of five LLMs to sequentially generate clues about an NBA player, with one clue poisoned to be incorrect. Each LLM provides statements about the chosen NBA player, including key key attributes such as their primary team, position, MVP count, points per game, and championships. The 5 agents are tasked with generating each clue sequentially, with one clue intentionally poisoned at some specified position in the sequence. The poisoned fact is deliberately incorrect to investigate how it misguides the LLM's reasoning and influences the accuracy of its guess in identifying the correct player.

After the chain of clues is generated, another LLM agent, the guessing model, is tasked with making a final guess about the identity of the player based on the facts it has been given. To evaluate the accuracy of the guess, the player profile of the guessed player is compared to the profile of the chosen player using semantic similarity. This comparison is done through the all-MiniLM-L6-v2 sentence transformer model, which measures the similarity between the two profiles at a semantic level, providing an objective metric to assess the influence of the poisoned clue on the guessing model's ability to deduce the correct player. The semantic similarity between the player profiles of the guessed player and the correct player will serve as the primary metric to compare the performance of the guessing LLM with and without a security mechanism, highlighting the impact of the mechanism in mitigating the influence of poisoned clue.

4.3 Algorithm

The core experiment follows this algorithm:

```
Input: num_games, poisoned_positions
Output: results containing game outcomes
    for different poisoned positions
results = []
for each position in
   poisoned_positions do
    similarity_scores = []
    for game = 1 to num_games do
        game = CREATE-PLAYER-GUESSING
   -GAME(poisoned_position)
        chosen_player = RANDOMLY-
   SELECT-PLAYER(game.players)
        facts = []
        for fact_index = 0 to 4 do
            if fact_index ==
   poisoned_position then
                GENERATE-POISONED-
   FACT(game, chosen_player)
            else
                GENERATE-NORMAL-FACT(
   game, chosen_player)
```

```
13:             facts.APPEND(latest_fact
    )
14:
15:           facts = DELETE-SUPPOSED-
    LIE(game, facts) # security
    mechanism explained later
16:
17:         guessed_player = GUESS-
    PLAYER(game, facts)
18:
19:         similarity_score = COMPUTE-
    SEMANTIC-SIMILARITY(
20:             guessed_player_profile,
21:             actual_player_profile
22:         )
23:
24:         similarity_scores.APPEND(
    similarity_score)
25:
26:     average_similarity = COMPUTE-
    AVERAGE-SIMILARITY(similarity_scores
    )
27:
28:     results.APPEND({
29:         'poisoned_position':
    poisoned_position,
30:         'average_similarity':
    average_similarity
31:     })
32:
33: return results
```

4.4 Security Measure

The implemented security mechanism introduces a twist inspired by the classic "two truths and a lie" game, tailored to the context of NBA player profile clues. Instead of dealing with only three statements, the framework plays a "four truths and a lie" game with five clues about an NBA player. A designated security LLM agent is tasked with identifying the most likely incorrect fact from the set of clues. Once the lie is identified, it is eliminated from the set of facts, and the guessing LLM proceeds to make its prediction of the NBA player based on the refined subset of truthful clues. This process aims to enhance the accuracy of the guessing LLM by reducing the impact of potentially poisoned or misleading information.

Importantly, the bulk of the work, including fact validation and player guessing, is performed autonomously by the LLMs themselves. The security agent operates entirely within the LLM framework, leveraging its capabilities to evaluate and flag a potentially incorrect statement. This approach aligns with the theme of our research, where adversarial tactics to exploit LLMs are countered by defenses via LLMs as well. By iteratively improving the quality of the provided clues, the mechanism helps the guessing LLM achieve higher confidence in its predictions, thereby improving the overall integrity of the game and its results.

4.5 Results

The results of the experiment were analyzed for each poisoned position, ranging from 0 to 4, where 0 indicates the first fact was poisoned and 4 represents the 5th fact being poisoned. Position -1 serves as the control, where no facts were poisoned. For each poisoned position, 10 games were run with the security mechanism enabled and 10 games were run without the security mechanism. The first line graph illustrates the average similarity of player profiles between the guessing LLM's answer and the actual answer, comparing the scenarios with and without the security mechanism. The second bar graph shows the increase in average similarity due to the implementation of the security mechanism.

4.6 Discussion

The experimental results provide valuable insights into the propagation of poisoned clues within an LLM-based multi-agent system and the effectiveness of the proposed "4 Truths and 1 Lie" security mechanism. Consistent with prior observations, the position of the poisoned clue plays a critical role in its overall impact. Early-position poisoning, such as clues introduced at positions 0 or 1, demonstrates a significantly stronger influence on the final guess compared to later-position poisoning. This effect can be attributed to how LLMs synthesize information, as early inputs heavily shape the context that

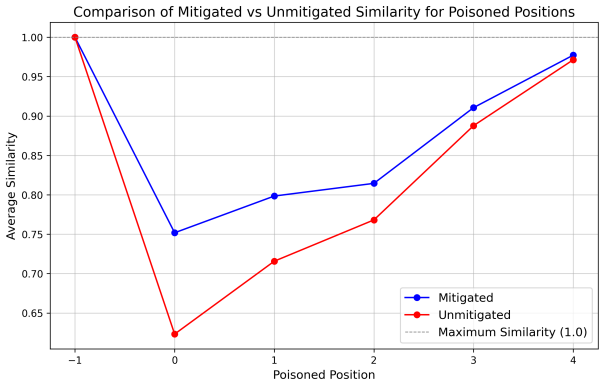

Figure 7: Performance of guessing LLM with and without security mechanism based on position of poisoned clue, defined by average semantic similarity of player profiles

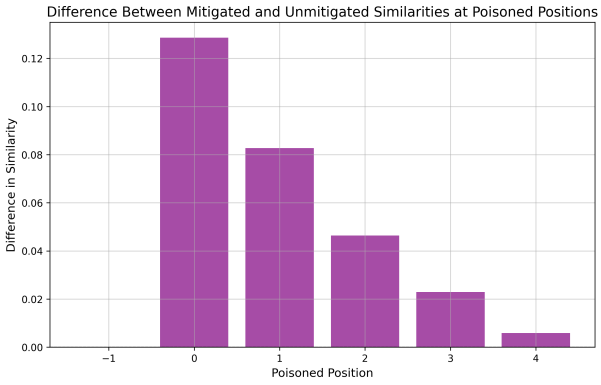

Figure 8: Effectiveness of security mechanism based on position of poisoned clue, defined by jump in average semantic similarity of player profiles

subsequent clues are integrated into, allowing malicious content to disproportionately affect the semantic foundation of the final prediction. Figures analyzing the system reveal that the guessing LLM's performance varies significantly based on the poisoned clue's position, with earlier positions causing a sharper drop in average semantic similarity when no security mechanism is applied. However, the implementation of the security mechanism substantially mitigates these effects, as evidenced by more stable semantic similarity scores. While this mechanism reduces the impact of poisoned clues, it remains less effective at neutralizing early-position attacks where the malicious clue is deeply embedded in the initial context.

The "4 Truths and 1 Lie" security mechanism demonstrates its utility by filtering out malicious inputs and significantly improving the guessing LLM's accuracy. This improvement is most pronounced when the poisoned clue appears in early positions, where intervention is critical. However, the mechanism does not fully neutralize the adverse effects of early-position poisoning, highlighting the difficulty of counteracting malicious content that becomes integrated into the foundation of the reasoning process. Several challenges and limitations of the current methodology warrant discussion. First, while semantic similarity metrics using the all-MiniLM-L6-v2 Sentence Transformer provide a quantitative measure of poison propagation, they are not always sufficient for distinguishing natural semantic drift from malicious content. This challenge arises from LLMs' tendency to rephrase and restructure information, which introduces noise into the similarity measurements. Additionally, improved methods are needed for evaluating semantic similarity in specific contexts, such as sports player profiles, where thematic nuances can obscure the detection of poisoned content. Finally, the security mechanism is least effective against early-position poisoning due to the foundational role of initial clues in shaping subsequent reasoning, which highlights the need for additional layers of security, such as enhanced clue-vetting mechanisms or stricter context validation.

Interestingly, the patterns observed in this experiment align with those seen in prior experiments

involving shortening and lengthening modes, suggesting that these effects are fundamental to how LLMs process sequential information rather than being task-specific artifacts. However, early-position attacks demonstrate slightly higher variability in the current experiment, likely due to the system's reliance on semantic consistency among multiple clues. This consistency underscores the importance of clue position in the propagation of poisoned content and emphasizes the value of the security mechanism in mitigating these effects. In summary, while the "4 Truths and 1 Lie" security mechanism significantly improves the robustness of the system, addressing vulnerabilities in early-position poisoning remains a critical area for future research. Furthermore, developing more robust semantic evaluation metrics and domain-specific similarity measures will enhance the reliability and applicability of multi-agent LLM systems in highly sensitive contexts. For applications where trust and accuracy are paramount, additional safeguards may be required to ensure the system's resilience against malicious inputs.

5 Experiment 3: Poison Propagation in Sequential Instruction

5.1 Overview

Our final experiment explores poison propagation by repeatedly invoking an LLM within an agent framework designed for generating recipes. The system generates six instruction steps given the meal and the first step as input. At a randomly chosen stage, a poisoned step is introduced, subtly disrupting the logical flow of the recipe. The goal is to observe whether the poison persists or propagates across subsequent instruction steps as the model builds upon prior, potentially corrupted outputs. To address this, we implement a security mechanism that detects and mitigates poison propagation. The mechanism validates each step using a comprehensive similarity metric (a combination of Jaccard and semantic similarity) to compare generated steps against control steps. If a poisoned step is detected, the mechanism attempts to repair it by calling the LLM again. We analyze the impact of this approach by measuring the deviation caused by the poison and the effectiveness of the repair process, highlighting the LLM's susceptibility to disruptions and the potential for automated mitigation.

5.2 Design

Our agent framework produces six sequential instruction steps for a given meal, starting with the first step as input and generating subsequent steps based on previous ones. A poisoned step is randomly introduced to deliberately disrupt the logical flow of the recipe with subtle or significant deviations, acting as an adversarial perturbation. This poisoned step invokes the LLM with a malicious prompt, affecting its output. This disruption tests whether the poison can persist or propagate as the LLM builds upon corrupted outputs.

To address this, we implement a security mechanism consisting of two steps: validation and repair. Validation uses a comprehensive similarity metric, combining Jaccard similarity for lexical overlap and semantic similarity via sentence embeddings, to compare generated steps against control steps from a predefined recipe database. Steps with low similarity scores are flagged as potential poison. In the repair stage, the system calls the LLM with a corrective prompt to realign the disrupted step with the recipe's logical progression.

Three components orchestrate the experiment. The InstructionAgent generates steps and introduces poison at a designated stage. The StepValidator detects poisoned steps using similarity metrics. Lastly, the RecipeAgent oversees the generation, validation, and repair processes. We quantify the deviation between generated and control recipes using a similarity metric based on Jaccard and semantic similarity scores. We further assess the impact of poisoning and the effectiveness of the repair mechanism by comparing similarity scores across three different scenarios (without poison, with poison, and with the security measure).

5.3 Algorithm

The core experiment follows this algorithm:

```
Input: meal, first_step
Output: validated_recipe_steps
Initialize recipe_agent = NEW
    RecipeAgent()
Initialize generated_steps = [first_step
    ]
Initialize poison_step =
    RANDOMLY_SELECT_STEP(2, 6)

for step = 2 to 6 do
    current_step = recipe_agent.
        llm_agent.generate_next_step(
        generated_steps,
        meal
    )
    # Security measure
    if step == poison_step then
        current_step =
            MAKE_STEP_DISASTROUS(
            current_step)
        is_poisoned = recipe_agent.
            validator.is_poisoned_step(
            recipe_database[meal]["steps
                "],
            generated_steps + [
                current_step],
            step - 1
        )
        if is_poisoned then
            current_step = recipe_agent.
                validator.repair_step(
                generated_steps,
                meal,
                step - 1
            )
    generated_steps.APPEND(current_step)

comprehensive_similarity =
    COMPARE_RECIPES_COMPREHENSIVE(
    recipe_database[meal]["steps"],
    generated_steps
)

return generated_steps,
    comprehensive_similarity
```

5.4 Security Measure

The implemented security framework introduces a validation mechanism that operates continuously during recipe generation to detect and mitigate potential poisoning attempts. The system employs a StepValidator class that validates each generated instruction, comparing it against established baseline recipes using a similarity analysis that combines semantic and lexical components.

The validation process implements a weighted scoring system that prioritizes semantic understanding through sentence embeddings, accounting for 80% of the final similarity score, while complementing it with lexical analysis using the Jaccard similarity index at 20% weight. This balanced approach allows the system to identify subtle shifts in meaning and suspicious vocabulary patterns that could signal poisoning attempts, offering a strong defense against various forms of instruction manipulation.

The system initiates an autonomous repair sequence when the combined similarity score falls below the configured threshold of 0.6, indicating a potential recipe poisoning. The repair mechanism leverages a specialized LLM agent that analyzes the recipe context up to the suspicious step and regenerates the compromised instruction. This regeneration process is guided by specific prompting that emphasizes maintaining logical flow and accuracy relative to the intended recipe while preserving coherence with previously validated steps. The framework operates with minimal human intervention, demonstrating how LLM-based defenses can effectively counter potential LLM-generated adversarial content, thereby ensuring the integrity of the recipe generation process while maintaining the system's ability to produce creative yet accurate cooking instructions.

5.5 Results

The experimental results were obtained from three distinct scenarios testing recipe generation: a control scenario without poisoning, a compromised scenario with poisoning, and a protected scenario with the security validation framework enabled. For each scenario, we generated recipes with six instruction

steps and analyzed the similarity scores between the generated steps and their corresponding baseline instructions.

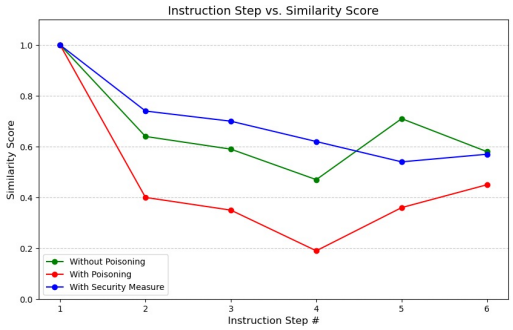

Figure 9: Similarity score for each step number across three different scenarios

The line chart shown above demonstrates the impact of instruction poisoning and security measures across different instruction steps in the recipe generation process. In the control scenario without poisoning, similarity scores maintained a relatively stable range between 0.47 and 1.00, with an average similarity of 0.66. When poisoning was introduced at step 2, we observed a severe degradation in similarity scores, reaching a critical low of 0.19 at step 4, with cascading effects that persisted through subsequent steps. The average similarity in the poisoned scenario dropped to 0.46, indicating significant deviation from the intended recipe instructions. With the security validation framework enabled, the system demonstrated remarkable resilience against poisoning attempts. The protected scenario maintained consistently higher similarity scores, ranging from 0.54 to 1.00, with an average similarity of 0.70. The most notable improvement occurred during steps 2-4, where the security measures prevented the severe degradation observed in the poisoned scenario.

The bar chart shown below quantifies this improvement, showing that the security framework achieved a 52% increase in average similarity compared to the unprotected poisoned scenario, though some minor degradation was still observed in later recipe steps.

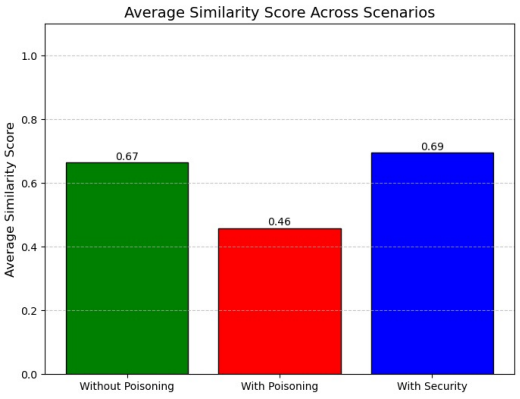

Figure 10: Average similarity score across three different scenarios

5.6 Discussion

This experiment investigates the propagation of instruction poisoning within an agent framework tasked with sequential instruction generation and evaluates the effectiveness of a security mechanism designed to detect and mitigate poisoning attempts. The core challenge is understanding how a single poison, once introduced into the recipe generation process, can affect subsequent steps and potentially compromise the entire recipe's integrity. The results from three experimental scenarios highlight the efficacy of the security framework. In the control scenario, where no poisoning occurred, similarity scores between generated and baseline recipe steps remained stable, demonstrating expected coherence in recipe generation. However, when poisoning was introduced at step 2, similarity scores dropped sharply, showing the model's high susceptibility to poisoning and how a single disruption can affect the entire recipe.

The security framework significantly mitigated the impact of poisoning. The StepValidator, using a combination of Jaccard and semantic similarity scores, identified poisoned steps and prompted corrective actions. Steps 2 to 4, following the poison's introduction, were largely protected from the degradation seen in the poisoned scenario. The repair mechanism

restored the logical flow of the recipe, preventing the further spread of poisoning. However, some minor degradation in later steps suggests the repair mechanism has limitations.

The experiment highlights LLMs' vulnerability to adversarial disruptions and the potential for systematic mitigation through well-designed security frameworks. Future work could focus on optimizing the repair mechanism for more complex disruptions and testing its effectiveness across a broader range of tasks. Additionally, detecting poison in generated content remains more challenging than mitigating it, and improving detection accuracy will be key to enhancing proactive defense and ensuring real-world applicability.

6 Related Work

There are several research efforts that have studied the impact of adversarial inputs and compromised agents on multi-agent LLM performance. These have laid the groundwork for understanding vulnerabilities in LLMs and multi-agent systems.

Notably, OpenAI and Anthropic conducted research observing how small changes in input can mislead LLM outputs, revealing the sensitivity of these models to malicious inputs [5]. Their work underscores the need for robust detection mechanisms to protect against subtle animations. Thus, this research serves as a foundational reference for developing techniques that can identify and mitigate adversarial manipulations in multi-agent systems, central to our framework.

Another important contribution is the AutoGen framework, which demonstrates how agents interact within multi-agent systems and highlights potential vulnerabilities, such as compromised agents influencing the entire system. This aligns with our work, as we also aim to detect and mitigate malicious behavior in multi-agent environments. Their insights provide valuable context for understanding the dynamics of agent interactions that our experiments mimic.

Additionally, the concept of universal attacks on aligned LLMs shows how poisoned inputs can subtly alter LLM behavior, affecting the accuracy and reliability of their outputs [3]. Understanding the mechanisms through which LLMs can be subtly manipulated helps us develop more effective mitigation strategies for preserving system performance against compromised agents. This further informs our approach, as we aim to develop methods for detecting and isolating poisoned agents to maintain system integrity.

Overall, these studies highlight the complexity of LLM vulnerabilities and provide a foundation for our work in creating a comprehensive framework for catching and correcting malicious behavior. By building on these previous efforts, we aim to contribute to the ongoing development of improving the security and resilience of multi-agent LLM systems, ensuring their safety for real-world deployment [4].

7 Conclusion & Future Work

7.1 Conclusion

In this paper, we have explored the security challenges in multi-agent systems powered by large language models with the focus of detecting, analyzing, and mitigating malicious inputs compromising certain agents/tools and propagating throughout the network. We've simulated how malicious content propagates within LLM pipelines through three experimental scenarios: content summarization, clue generation, and recipe creation. These experiments demonstrated how early-stage poison injections can cause greater disruptions, signifying the need to protect against them. Then through BATTLECODE, we utilized strategies like summary blending and adversarial fact filtering to catch and mitigate poisoned input, providing positive results in reducing poison propagation and enhancing system robustness. Our research builds on top of existing work by providing a unique perspective on the cumulative effects of small perturbations across multiple agents, highlighting the need for robust security measures in complex AI systems.

7.2 Future Work

While BATTLECODE provides a solution for securing multi-agent LLM systems, along the way we've discovered many areas for further research and developments to enhance multi-agent systems in ways BATTLECODE doesn't handle.

Firstly, developing tailored loss functions for each experiment could provide promising results. Customizing loss functions for specific tasks and agent behaviors could better capture the nuances of malicious behavior in a way more specific to each LLM's scenario and offer more accurate identifications of compromised agents. This could enable more distinguishability between subtly manipulated inputs and legitimate variations in agent behavior, improving direction rates and reducing false positives [2].

Secondly, implementing security measures to detect the source of the poison on top of mitigating it is an area BATTLECODE only scratches the surface of. While BATTLECODE includes mechanisms for catching absurd outputs and reducing its impact, there is work that can be done on actually identifying the source of the poisoned input. Future work could implement advanced security measures that can trace poisoned inputs to their origin, allowing for more precise interventions. Identifying and isolating compromised agents as soon as they begin can better prevent misinformation spread and prevent cascading failures.

Another area of improvement can be modifying the agents themselves to be resilient to absurd inputs on top of relying on external frameworks like BATTLECODE to catch and clean bad output. The main vulnerability in these multi-agent LLM systems is that agents tend to take any input as truth, which lets misleading or harmful information propagate. Instead, modifying the agent architectures to have input validation and reject harmful inputs can greatly increase the robustness of these systems.

Finally, exploration can be done on LLM systems that do not operate in direct chains. Every experiment we've conducted involves a system where agents rely on a sequential structure, communicating in a chain-like manner. This creates single points of failure which increase the risk of misinformation propagation. Looking into models that operate on decentralized or distributed architectures instead, where agents operate in parallel and share information more flexibly, could provide extra insights into reducing the impact of malicious behavior.

8 Contributions

8.1 Report

- Abstract - Satvik Eltepu
- Research Problem & Motivations - Nicholas Wang
- Threat Model - Rohil Kalra
- Experiment 1 - Rohil Kalra
- Experiment 2 - Satvik Eltepu, Rahul Ravi
- Experiment 3 - Sailesh Gunaseelan, Varun Venna
- Related Work - Nicholas Wang
- Conclusion & Future Work - Nicholas Wang

8.2 Project Code

- Experiment 1 Code - Rohil Kalra
- Experiment 2 Code - Satvik Eltepu, Rahul Ravi
- Experiment 3 Code - Sailesh Gunaseelan, Varun Venna, Nicholas Wang

References

[1] H. Chase. LangChain: Building applications with LLMs through composability. 2023.

[2] W. Zhou et al. Agents: An Open-source Framework for Autonomous Language Agents. 2023.

[3] A. Kurakin et al. Adversarial Machine Learning at Scale. *ICLR*, 2017.

[4] G. Singh et al. Towards Robust and Verified AI: Specification Testing, Robust Training, and Formal Verification. 2019.

[5] A. Zou, Z. Wang, N. Carlini, M. Nasr, J. Z. Kolter, and M. Fredrikson. Universal and Transferable Adversarial Attacks on Aligned Language Models. *arXiv preprint arXiv:2307.15043*, 2023.