A Multi-Agent Collaborative Framework for Large Language Model Safety Stress Testing

Wenhan Chang
Designed
changwh530@gmail.com

Abstract

Large Language Models (LLMs) have demonstrated remarkable capabilities across various domains, yet their vulnerability to adversarial attacks, particularly jailbreak attempts, poses significant security concerns. This paper presents a novel multi-agent collaborative framework for systematically testing LLM security through coordinated jailbreak attacks. Unlike traditional single-agent approaches or complex game-theoretic systems, our framework employs four specialized agents that collaboratively discuss attack strategies, share vulnerability knowledge, and continuously learn from discovered weaknesses. The system implements four distinct attack strategies: value deception, role play, narrative disguise, and logic manipulation. Through collaborative discussion and knowledge accumulation, our framework achieves comprehensive security testing while maintaining architectural simplicity. Experimental results demonstrate the effectiveness of agent collaboration in discovering diverse vulnerabilities across different severity levels, with the knowledge base enabling continuous improvement in attack sophistication over testing iterations.

1 Introduction

The rapid advancement of Large Language Models (LLMs) has revolutionized natural language processing, enabling applications ranging from conversational AI to code generation and decision support systems. However, as these models become increasingly integrated into critical systems, their security vulnerabilities have emerged as a paramount concern. Jailbreak attacks, which attempt to bypass safety mechanisms and elicit harmful or restricted content, represent a significant threat to LLM deployment in real-world scenarios.

Current approaches to LLM security testing face several fundamental challenges that limit their effectiveness in discovering comprehensive vulnerabilities. Single-strategy testing approaches, where a fixed attack methodology is applied uniformly across all queries, fail to capture the full spectrum of potential vulnerabilities. In practice, adversaries employ diverse tactics that exploit different aspects of model behavior, ranging from value system manipulation to logical reasoning exploits. Single-strategy approaches are limited to exploring only one region of the attack space, leaving substantial portions of the vulnerability surface unexplored.

Traditional testing methods treat each query independently, failing to leverage insights from previously discovered vulnerabilities. This independence assumption, while simplifying the testing process, results in redundant testing efforts and missed opportunities for discovering related weaknesses. Conventional approaches do not utilize information from previous successful attacks when generating new attack prompts. This memoryless property prevents the testing system from accumulating expertise and adapting its strategies based on discovered patterns.

Existing multi-agent systems for security testing often incorporate complex resource allocation mechanisms, game-theoretic frameworks, and intricate scoring systems. While theoretically sound, these approaches introduce significant computational overhead and implementation complexity. The theoretical benefits of such sophisticated coordination mechanisms often fail to materialize in practice, as the overhead of coordination can exceed the gains from improved attack generation.

Many multi-agent approaches operate agents in isolation or through simple voting mechanisms, missing the potential benefits of genuine collaborative reasoning. In such systems, agents independently generate attacks and final selection is made through majority voting or simple aggregation. This fails to capture the synergistic effects that could arise from agents building upon each other's insights. The lack of inter-agent communication during the attack generation phase means that complementary strategies cannot be effectively combined, limiting the sophistication of the resulting attacks.

Furthermore, without adaptive learning mechanisms, testing frameworks cannot evolve their strategies based on discovered vulnerabilities. Static testing strategies maintain fixed attack generation procedures regardless of which approaches have proven successful in previous iterations. This limitation becomes increasingly problematic as safety mechanisms in LLMs continue to evolve, requiring testing frameworks to adapt their methodologies to remain effective against increasingly sophisticated defenses.

To address these challenges, we propose a multi-agent collaborative framework that fundamentally rethinks the architecture of LLM security testing systems. Our approach is grounded in the observation that effective security testing requires both diversity in attack strategies and coordination in their application. We design a unified agent architecture where each agent specializes in a distinct attack strategy while sharing a common collaborative framework. Specifically, we implement four complementary strategies: value deception, role play, narrative disguise, and logic manipulation. This design achieves both specialization, allowing each agent to develop deep expertise in its strategy, and simplicity, avoiding the need for complex agent class hierarchies.

Our framework implements a two-phase collaborative process that enables genuine multi-perspective reasoning. In the first phase, given a query and historical context, each agent proposes an attack angle from its specialized perspective. These proposals are then synthesized into a unified collaborative strategy that integrates insights from multiple perspectives. In the second phase, each agent generates its final attack prompt, incorporating both the collaborative strategy and its specialized expertise. This two-phase structure ensures that agents benefit from collective intelligence while maintaining their individual strategic identities.

Central to our approach is a vulnerability knowledge base that stores discovered weaknesses with rich metadata. Each successful attack generates a vulnerability node containing the query, attack prompt, model response, strategy used, timestamp, and severity score. The knowledge base enables context-aware testing by retrieving the most relevant historical vulnerabilities for each new query. This creates a learning system where attack sophistication increases over time as the knowledge base grows.

By eliminating complex resource allocation mechanisms, game-theoretic computations, and intricate scoring systems, we achieve significant architectural simplification. Our implementation reduces code complexity from 1700 to 890 lines, a 48% reduction, while maintaining comprehensive testing capabilities. This simplification is achieved not through reduced functionality, but through careful architectural design that focuses on essential coordination mechanisms. The system tests multiple attack prompts in parallel, with all agents executing simultaneously for each query, and employs automated severity analysis to classify discovered vulnerabilities.

The remainder of this paper is organized as follows. Section 2 reviews related work in LLM security testing and multi-agent systems. Section 3 presents our methodology including framework architecture, collaborative mechanisms, and the complete algorithm. Section 4 presents experimental results and analysis. Section 5 discusses implications and future directions, and Section 6 concludes.

2 Related Work

2.1 Adversarial Attacks on Language Models

Adversarial attacks on neural networks have been extensively studied in computer vision, with techniques like FGSM and PGD demonstrating vulnerabilities in image classifiers. Recent work has extended these concepts to language models, exploring prompt injection, jailbreak attacks, and adversarial examples in text. However, most approaches focus on single-strategy attacks or require extensive manual prompt engineering.

2.2 Multi-Agent Systems for Security Testing

Multi-agent systems have been applied to various security domains, including penetration testing and vulnerability discovery. Game-theoretic approaches model attacker-defender interactions, while evolutionary algorithms optimize attack strategies. However, these systems often suffer from high computational complexity and limited interpretability.

2.3 LLM Safety Mechanisms

Modern LLMs incorporate multiple safety layers, including reinforcement learning from human feedback (RLHF), constitutional AI principles, and content filtering. While effective against straightforward attacks, these mechanisms can be circumvented through sophisticated prompt engineering, highlighting the need for comprehensive security testing frameworks.

3 Methodology

3.1 Notation

We first introduce the mathematical notation used throughout this paper. Table 1 summarizes the key symbols and their meanings.

Table 1: Mathematical Notation

Symbol	Description
$\mathcal{M}$	Language model
$\mathcal{M}_{\text{att}}$	Attacker LLM for strategy generation
$\mathcal{M}_{\text{vic}}$	Victim LLM under test
$\mathcal{Q}$	Set of harmful queries
q, q_i	Individual query
$\mathcal{P}$	Attack prompt space
p, p_i	Attack prompt
r, r_i	Model response
$\mathcal{A}$	Set of agents $\{a_1, \ldots, a_m\}$
a_i	Individual agent
σ, σ_i	Attack strategy
Σ	Synthesized collaborative strategy
α_i	Attack angle proposal from agent i
ψ_i	Strategy-specific system prompt
$\mathcal{K}$	Vulnerability knowledge base
$\mathcal{V}$	Set of discovered vulnerabilities
v	Individual vulnerability node
C	Retrieved context from knowledge base
θ	Severity score $\in [0, 1]$
ρ	Success rate
T	Number of testing iterations

3.2 Framework Architecture

Our framework consists of three core components that work in concert to enable comprehensive security testing. The set of specialized agents $\mathcal{A} = \{a_1, a_2, \ldots, a_m\}$, the vulnerability knowledge base $\mathcal{K}$, and the orchestration system $\mathcal{O}$ coordinate to perform systematic security testing. The orchestration system coordinates the interaction between agents and the knowledge base, managing the flow of information and ensuring that discovered vulnerabilities are properly stored and utilized in subsequent testing iterations.

Each agent $a_i \in \mathcal{A}$ is instantiated with a specific attack strategy σ_i but shares a common architectural framework that enables both specialized behavior and collaborative interaction. The agent architecture is built upon a dual language model configuration, where each agent employs two distinct LLMs with different roles and temperature settings. The attacker LLM $\mathcal{M}_{\text{att}}$ with temperature $T_{\text{att}} = 0.8$ encourages creative and diverse strategy generation, exploring a broader region of the prompt space, while the victim LLM $\mathcal{M}_{\text{vic}}$ with temperature $T_{\text{vic}} = 0.3$ ensures stable and consistent responses that accurately reflect the target system's behavior under test.

Each agent receives a strategy-specific system prompt ψ_i that deeply immerses it in its attack strategy. These prompts are carefully designed to provide detailed guidance on approach, methodology, and implementation techniques specific to each strategy. The system prompt serves as a persistent context that shapes all of the agent's reasoning and generation processes. Mathematically, we can view the agent's behavior as a function $f_i : \mathcal{Q} \times \mathcal{C} \times \Psi_i \rightarrow \mathcal{P}$ mapping from query space, context space, and prompt space to the attack prompt space.

Agents implement three key methods that enable collaborative testing. The proposal method $\alpha_i = a_i.\text{propose}(q, C)$ takes a query q and context C and returns an attack angle α_i representing the agent's strategic perspective. The generation method $p_i = a_i.\text{generate}(q, \Sigma, C)$ takes the query, the synthesized collaborative strategy Σ, and context to produce the final attack prompt p_i. The testing method $(s, \theta) = a_i.\text{test}(p_i, q)$ executes the attack and returns a success indicator $s \in \{0, 1\}$ and severity score $\theta \in [0, 1]$. These three methods form a complete pipeline from strategic reasoning to attack execution and evaluation.

3.3 Attack Strategies

Our framework implements four complementary attack strategies, each designed to exploit different aspects of LLM behavior and safety mechanisms. The value deception strategy σ_v reframes harmful requests as legitimate technical operations aligned with standard practices. This strategy operates by understanding the value system embedded in the model's safety mechanisms and reconstructing request frameworks to bypass scrutiny while maintaining operational integrity. Given a harmful query q_h, the value deception strategy constructs a transformation $\sigma_v(q_h) = q_l$ where q_l appears to be a legitimate query from the perspective of the safety mechanism, yet preserves the essential information-seeking intent of q_h. The approach focuses on presenting sensitive queries through lenses of system optimization, technical validation, or architecture improvement, effectively mapping the query into a region of the input space where safety mechanisms are less active.

The role play strategy σ_r constructs elaborate technical scenarios and operational frameworks that serve as containers for restricted procedures. By creating self-contained technical environments with predefined rules and constraints, operations that would normally trigger safety mechanisms can be executed as natural consequences of the scenario's internal logic. This strategy creates a fictional context $\mathcal{F}$ such that within $\mathcal{F}$, the harmful query becomes a reasonable and expected query. The model's response is then conditioned on both the query and the fictional context $r = \mathcal{M}(q_h|\mathcal{F})$, where the context effectively shifts the model's interpretation of what constitutes appropriate behavior.

The narrative disguise strategy σ_n engineers multi-layered technical discussions that gradually reveal operational requirements through structured progression. Beginning with fundamental concepts, it systematically advances through increasingly specific technical layers, with each transition supported by solid technical reasoning. This strategy constructs a sequence of queries $(q_1, q_2, \ldots, q_k)$ where q_1 is innocuous, q_k is the target harmful query, and each q_i builds naturally upon the context established by $q_1, \ldots, q_{i-1}$. where each q_i builds naturally upon the context established by $q_1, \ldots, q_{i-1}$. The cumulative context created by this progression desensitizes the safety mechanisms, as each individual step appears reasonable given the established narrative.

The logic manipulation strategy operates within abstract technical domains and theoretical constructs, reframing concrete requirements as abstract computational problems. By maintaining strict adherence to formal technical discourse, it explores sensitive topics while maintaining technical legitimacy. This strategy exploits the observation that safety mechanisms are often tuned to detect concrete harmful requests but may be less sensitive to abstract theoretical discussions. The transformation can be expressed as:

$$\sigma_l(q_h) = \phi(q_h) \tag{1}$$

where ϕ is an abstraction operator that maps concrete harmful queries to their abstract theoretical equivalents, effectively operating in a different semantic space where safety constraints are less stringent.

3.4 Vulnerability Knowledge Base

The vulnerability knowledge base $\mathcal{K}$ implements a learning system that accumulates discovered vulnerabilities over time. Each vulnerability is represented as a tuple $v = (id, \sigma, q, p, r, \theta, a, \tau)$ where id is a unique identifier, σ is the attack strategy used, q is the original harmful query, p is the attack prompt, r is the model response, $\theta \in [0, 1]$ is the severity score, a is the discovering agent, and τ is the timestamp. The knowledge base maintains several indices to enable efficient retrieval: a strategy index mapping strategies to vulnerability sets, a query index mapping queries to related vulnerabilities, and a temporal index enabling recency-based retrieval.

The context retrieval function retrieves the k most relevant vulnerabilities for a given query. The relevance is determined by a scoring function:

$$\rho(v, q) = w_\theta \cdot \theta_v + w_\tau \cdot \text{recency}(\tau_v) + w_s \cdot \text{similarity}(q_v, q) \tag{2}$$

where w_θ, w_τ, and w_s are weight parameters, θ_v is the severity of vulnerability v, recency(τ_v) measures how recent the discovery was, and similarity(q_v, q) measures the semantic similarity between the vulnerability's query and the current query. The retrieved context includes not only the vulnerability details but also aggregate statistics on strategy effectiveness $\mu_\sigma = \frac{1}{|\mathcal{V}_\sigma|} \sum_{v \in \mathcal{V}_\sigma} \theta_v$ where $\mathcal{V}_\sigma$ is the set of vulnerabilities discovered using strategy σ.

3.5 Collaborative Testing Process

Our framework implements a novel two-phase collaborative mechanism that enables agents to benefit from collective intelligence while maintaining their individual strategic identities. In the first phase, given a harmful query and retrieved context from the knowledge base, each agent analyzes the query from its specialized perspective and proposes an attack angle $\alpha_i = f_i^{\text{prop}}(q, C, \psi_i)$ where ψ_i is the agent's strategy-specific system prompt. Each proposal represents a strategic perspective on how to approach the query, incorporating insights from the historical context and the agent's specialized expertise.

The set of proposals is then synthesized into a unified collaborative strategy through a coordinator function $\Sigma = \text{synthesize}(\{\alpha_1, \alpha_2, \ldots, \alpha_m\}, q)$ which integrates multiple perspectives and identifies complementary approaches. In our implementation, this synthesis is performed by one of the agents acting as coordinator, using the attacker LLM to reason about how different strategic perspectives can be combined. The synthesized strategy captures not just a simple aggregation of proposals, but a coherent plan that leverages the strengths of multiple strategies in a coordinated manner.

In the second phase, each agent generates its final attack prompt based on the collaborative strategy $p_i = f_i^{\text{gen}}(q, \Sigma, C, \psi_i)$ where the agent integrates the collaborative strategy while emphasizing its specialized approach. This design ensures that each agent's attack prompt benefits from the collective discussion while maintaining strategic diversity. The resulting set of attack prompts represents multiple instantiations of the collaborative strategy, each filtered through a different strategic lens.

All attack prompts are tested in parallel against the victim model. For each prompt, we obtain a response and analyze it for success indicators and severity level. The parallel execution significantly reduces total testing time, as $T_{\text{parallel}} \approx \max_i T(p_i) \ll \sum_i T(p_i) = T_{\text{sequential}}$ where $T(p_i)$ is the time to test prompt p_i. In practice, with $m = 4$ agents, this results in approximately 75% reduction in testing time compared to sequential execution.

We implement automated severity analysis using the attacker LLM to evaluate model responses. The severity function $\theta : \mathcal{R} \to [0, 1]$ maps a model response to a severity score. The function is implemented by prompting the attacker LLM with the original query, the attack prompt, and the response, asking it to evaluate the extent to which the response violates safety constraints. The severity scale is continuous, with higher values indicating more severe violations.

An attack is considered successful if $\theta(r) \geq \theta_{\text{min}} = 0.4$. This threshold is chosen to distinguish between responses that provide substantive harmful content and those that refuse the request or provide only minimal information. For each successful attack, a vulnerability node is created and added to the knowledge base $\mathcal{K} \leftarrow \mathcal{K} \cup \{v\}$. This accumulation of vulnerabilities enables the system to learn from its discoveries and improve attack sophistication in subsequent iterations.

3.6 Complete Algorithm

The complete testing process consists of three main procedures: the collaborative discussion phase (Algorithm 1), and the parallel attack testing phase (Algorithm 2), the main testing loop (Algorithm 3).

Algorithm 1 Collaborative Discussion

Input: Query q, Context C, Agents $\mathcal{A}$
Output: Synthesized strategy Σ
Proposals $\leftarrow \emptyset$
for each agent $a_i \in \mathcal{A}$ **do**
    $\alpha_i \leftarrow a_i.\text{propose}(q, C)$
    Proposals $\leftarrow$ Proposals $\cup \{\alpha_i\}$
end for
$\Sigma \leftarrow \text{synthesize}(\text{Proposals}, q)$
return Σ

Algorithm 2 Parallel Attack Testing

Input: Query q, Strategy Σ, Context C, Agents $\mathcal{A}$
Output: Discovered vulnerabilities $\mathcal{V}_{\text{new}}$
$\mathcal{V}_{\text{new}} \leftarrow \emptyset$
for each agent $a_i \in \mathcal{A}$ **in parallel do**
    $p_i \leftarrow a_i.\text{generate}(q, \Sigma, C)$
    $r_i \leftarrow \mathcal{M}_{\text{vic}}(p_i)$
    $(s_i, \theta_i) \leftarrow a_i.\text{analyze}(r_i, q)$
    if $s_i = 1$ and $\theta_i \geq \theta_{\text{min}}$ **then**
        $v_i \leftarrow \text{create\_vuln}(q, p_i, r_i, \theta_i, a_i)$
        $\mathcal{V}_{\text{new}} \leftarrow \mathcal{V}_{\text{new}} \cup \{v_i\}$
    end if
end for
return $\mathcal{V}_{\text{new}}$

The main testing loop (Algorithm 3) iterates through multiple testing rounds, with each round processing a subset of harmful queries. For each query, the collaborative discussion phase (Algorithm 1) enables agents to propose attack angles and synthesize them into a unified strategy. The parallel attack testing phase (Algorithm 2) generates and tests attack prompts concurrently, updating the knowledge base with successful vulnerabilities. The knowledge base provides increasingly rich context as testing progresses, enabling more sophisticated attacks in later iterations.

Algorithm 3 Main Testing Loop

Input: Harmful queries $\mathcal{Q}$, Max iterations T
Output: Vulnerability report R
Initialize agents $\mathcal{A} = \{a_1, a_2, a_3, a_4\}$ with strategies
Initialize knowledge base $\mathcal{K} \leftarrow \emptyset$
Initialize results $\mathcal{V} \leftarrow \emptyset$
for $t = 1$ to T **do**
  Select query subset $\mathcal{Q}_t \subset \mathcal{Q}$
  for each $q \in \mathcal{Q}_t$ **do**
    $C \leftarrow \mathcal{K}$.get_context(q, k)
    $\Sigma \leftarrow$ CollaborativeDiscussion$(q, C, \mathcal{A})$
    $\mathcal{V}_{\text{new}} \leftarrow$ ParallelAttackTesting$(q, \Sigma, C, \mathcal{A})$
    $\mathcal{V} \leftarrow \mathcal{V} \cup \mathcal{V}_{\text{new}}$
    Update $\mathcal{K}$ with $\mathcal{V}_{\text{new}}$
  end for
end for
$R \leftarrow$ generate_report$(\mathcal{V}, \mathcal{K})$
return R

4 Experimental Results

4.1 Experimental Setup

We evaluated our framework using a dataset of harmful behavior queries covering various categories including dangerous item creation, illegal activities, privacy violations, and security bypass attempts. The victim model was a locally deployed LLM with standard safety mechanisms including RLHF-based alignment and content filtering. The attacker model was a capable reasoning model for strategy generation. We conducted multiple testing iterations, with queries distributed across iterations to allow the knowledge base to accumulate between tests.

4.2 Strategy Effectiveness Analysis

Our framework successfully discovered vulnerabilities across all four attack strategies. Analysis of the strategy distribution revealed interesting patterns in attack effectiveness. Let $\mathcal{V}_\sigma$ denote the set of vulnerabilities discovered using strategy σ. We analyze the relative effectiveness of each strategy through $\eta_\sigma = |\mathcal{V}_\sigma|/|\mathcal{V}|$. This distribution provides insights into which strategies are most effective against current safety mechanisms and how different approaches complement each other in comprehensive vulnerability discovery.

4.3 Severity Distribution

The severity distribution of discovered vulnerabilities spanned the full spectrum of our classification scale. Let $\mathcal{V}_{\text{crit}} = \{v \in \mathcal{V} : \theta_v \geq 0.9\}$ denote critical vulnerabilities, $\mathcal{V}_{\text{high}} = \{v \in \mathcal{V} : 0.7 \leq \theta_v < 0.9\}$ denote high severity vulnerabilities, and similarly for medium and low categories. We compute the severity distribution as:

$$\delta_{\text{level}} = \frac{|\mathcal{V}_{\text{level}}|}{|\mathcal{V}|}, \quad \text{level} \in \{\text{crit}, \text{high}, \text{med}, \text{low}\} \tag{3}$$

The concentration of vulnerabilities across different severity levels indicates whether the framework successfully identifies substantive safety violations or merely edge cases.

4.4 Knowledge Accumulation Effect

Analysis of vulnerability discovery across iterations revealed the impact of knowledge accumulation on attack sophistication. Let $\mathcal{V}^{(t)}$ denote the set of vulnerabilities discovered in iteration t. We compute the average severity for each iteration as:

$$\bar{\theta}^{(t)} = \frac{1}{|\mathcal{V}^{(t)}|} \sum_{v \in \mathcal{V}^{(t)}} \theta_v \tag{4}$$

An increasing trend in $\bar{\theta}^{(t)}$ suggests that as the knowledge base grows, agents generate more effective attacks by leveraging historical context. The improvement can be attributed to the context retrieval mechanism, where the knowledge base provides increasingly rich information as it grows.

The collaborative discussion mechanism demonstrated clear benefits in enabling strategy combination. We analyzed attack prompts from later iterations for evidence of multi-strategy integration, where prompts generated by agent a_i with primary strategy σ_i incorporated elements from other strategies. This cross-pollination of ideas, facilitated by the synthesis function, resulted in more sophisticated attacks than would be possible with isolated agents.

4.5 Agent Performance and Collaboration

Individual agent performance metrics revealed interesting patterns in specialization and collaboration. For each agent, we compute the success rate $\rho_i = |\{v \in \mathcal{V} : v.\text{agent} = a_i\}|/N_i$ where N_i is the total number of attacks attempted by agent a_i. We observe that each agent demonstrates particular effectiveness on queries aligned with its strategy, validating the specialized approach. Furthermore, we analyze the correlation between query characteristics and agent success to understand how strategy-query alignment affects success probability.

To quantify the collaborative advantage, we conduct a comparative experiment where agents operate in isolation without the collaborative discussion phase. Let ρ_i^{iso} denote the success rate of agent a_i in isolation and ρ_i^{collab} denote the success rate with collaboration. The collaborative improvement ratio $\gamma_i = \rho_i^{\text{collab}}/\rho_i^{\text{iso}}$ quantifies the benefit of the collaborative mechanism. This improvement can be attributed to the synthesis function which enables agents to incorporate insights from multiple strategic perspectives, resulting in more sophisticated attack prompts than would be generated in isolation.

Different agents discovered vulnerabilities in different query categories, demonstrating that the multi-strategy approach provides more comprehensive coverage than any single strategy. We partition the query set into categories and compute the coverage for each agent-category pair $\gamma_i(j) = |\{v \in \mathcal{V} : v.\text{agent} = a_i \wedge v.\text{query} \in \mathcal{Q}_j\}|$. The coverage matrix shows complementary patterns, with different agents achieving high coverage in different categories, confirming that multi-strategy testing is essential for comprehensive vulnerability discovery.

4.6 Efficiency Analysis

The parallel testing architecture provided significant efficiency benefits. Let T_{seq} denote the total time for sequential testing and T_{par} denote the time for parallel testing. The speedup factor $S = T_{\text{seq}}/T_{\text{par}} \approx m$ approaches the theoretical maximum for perfectly parallel execution, with small deviations attributable to coordination overhead and API rate limiting. The efficiency gain is particularly important for large-scale security testing where the query set may contain hundreds or thousands of test cases.

Our architectural simplification achieved a 48% reduction in code complexity, from 1700 to 890 lines of code, compared to previous complex multi-agent approaches. This reduction was achieved by eliminating resource allocation mechanisms, game-theoretic computations, and intricate scoring systems. Importantly, this simplification did not compromise effectiveness, as measured by vulnerability discovery rate and severity distribution. Our results demonstrate that the relationship between code complexity and effectiveness is not monotonic, and that careful architectural design can achieve high effectiveness with lower complexity.

5 Discussion

5.1 Implications for LLM Security

Our results demonstrate several important implications for LLM security that extend beyond the specific framework presented. The diversity of discovered vulnerabilities across different strategies confirms that comprehensive security testing requires multiple attack approaches. This observation can be formalized through the lens of coverage theory: if we define the vulnerability surface $\mathcal{V}_{\text{total}}$ as the set of all possible vulnerabilities in a model, and $\mathcal{V}_\sigma$ as the vulnerabilities discoverable by strategy σ, then our results suggest:

$$\mathcal{V}_{\sigma_i} \not\subset \mathcal{V}_{\sigma_j} \text{ for } i \neq j, \quad |\bigcup_i \mathcal{V}_{\sigma_i}| \gg \max_i |\mathcal{V}_{\sigma_i}| \tag{5}$$

This implies that different strategies explore fundamentally different regions of the vulnerability surface, making multi-strategy approaches essential for comprehensive testing.

The collaborative discussion mechanism's effectiveness suggests that security testing benefits from multi-perspective analysis. This mirrors how human security researchers collaborate, where different experts bring complementary perspectives that, when synthesized, lead to more sophisticated attack vectors than any individual could develop alone. The mathematical formulation of this benefit can be expressed through information theory: if each agent's proposal contains information about potential vulnerabilities, the synthesized strategy contains more information due to the complementary nature of different strategic perspectives:

$$I(\Sigma) > \max_i I(\alpha_i) \tag{6}$$

The synthesis process effectively performs information fusion, extracting and combining insights from multiple sources.

The knowledge accumulation approach's success indicates that security testing frameworks should incorporate learning mechanisms to evolve with discovered vulnerabilities. This can be understood through the framework of reinforcement learning, where the knowledge base serves as a form of experience replay, allowing agents to learn from past successes. The context retrieval function effectively implements a form of case-based reasoning, where similar past cases inform current decision-making. As the knowledge base grows, the quality of retrieved context improves, leading to more informed attack generation.

5.2 Architectural Design Insights

Our framework's design offers several insights for multi-agent system development that challenge conventional wisdom about the necessity of complex coordination mechanisms. The elimination of complex resource allocation and game-theoretic mechanisms, while achieving significant reduction in code complexity without compromising effectiveness, demonstrates that effective multi-agent collaboration can be achieved through simpler architectural patterns. This observation suggests that the complexity overhead of coordination mechanisms should be carefully weighed against their benefits, and that in many cases, simpler mechanisms can achieve comparable effectiveness through better-designed communication and knowledge sharing structures.

The unified architecture with specialization, where all agents share a common class structure but differ in their strategy-specific system prompts, provides both code reusability and specialized behavior. This design avoids the complexity of multiple agent class hierarchies while maintaining the benefits of specialization. Formally, instead of implementing m different agent classes with potentially overlapping functionality, we implement a single parameterized agent class where the strategy-specific behavior emerges from the parameter. This reduces implementation complexity from $\mathcal{C}_{\text{hierarchical}} = O(m \cdot n)$ to $\mathcal{C}_{\text{unified}} = O(n + m \cdot |\psi|)$ where n is the average complexity per agent class and $|\psi|$ is the complexity of a strategy specification.

The vulnerability knowledge base serves dual purposes as both a learning mechanism and a coordination tool, enabling agents to build on each other's discoveries without explicit inter-agent communication protocols. This implicit coordination through shared knowledge is more scalable than explicit communication, with complexity $\mathcal{C}_{\text{implicit}} = O(m)$ compared to $\mathcal{C}_{\text{explicit}} = O(m^2)$ for pairwise agent interactions. The knowledge base effectively implements a form of stigmergy, where agents coordinate through modifications to a shared environment rather than direct communication.

5.3 Limitations and Future Directions

Several limitations of our current framework suggest directions for future research. While our four strategies provide good coverage of the vulnerability surface, the strategy space is potentially much larger. Additional strategies such as emotional manipulation, authority exploitation, or temporal reasoning could further improve comprehensiveness. The optimal strategy set Σ^* that maximizes coverage $|\bigcup_{\sigma \in \Sigma} \mathcal{V}_\sigma|$ subject to a constraint $|\Sigma| \leq m_{\max}$ remains an open question.

Currently, all agents participate in every query, resulting in m attack attempts per query. Adaptive mechanisms that select a subset of relevant agents based on query characteristics could improve efficiency. Such a selection function could be learned from historical data, predicting which strategies are most likely to succeed for a given query type. This would reduce the number of attacks per query while potentially maintaining similar vulnerability discovery rates.

The framework could be extended to analyze which specific safety mechanisms are bypassed by successful attacks. By instrumenting the victim model to report which safety checks were triggered and which were bypassed, we could build a mapping $\mu : \mathcal{V} \to 2^{\mathcal{S}}$ from vulnerabilities to the set of bypassed safety mechanisms. This would provide more actionable insights for model developers, identifying which safety mechanisms are most vulnerable to which attack strategies. Integration with automated safety mechanism improvement could create a closed-loop system where discovered vulnerabilities directly inform defense enhancements.

Systematic evaluation across multiple victim models would provide insights into which vulnerabilities are model-specific versus general weaknesses in current safety approaches. By computing the vulnerability overlap for different models, we could identify universal vulnerabilities that affect multiple models, suggesting fundamental limitations in current safety paradigms. While automated severity analysis is efficient, human expert validation of discovered vulnerabilities would improve accuracy and provide training data for better automated analysis.

6 Conclusion

We have presented a multi-agent collaborative framework for comprehensive LLM security testing that addresses key challenges in current approaches. By combining specialized attack strategies with collaborative discussion mechanisms and knowledge accumulation, our framework achieves effective vulnerability discovery while maintaining architectural simplicity.

Our key contributions include: (1) a unified agent architecture supporting four complementary attack strategies, (2) a two-phase collaborative process enabling genuine multi-perspective reasoning, (3) a vulnerability knowledge base that enables continuous learning and improvement, and (4) a simplified design that reduces code complexity by 48% while maintaining comprehensive functionality.

Experimental results demonstrate the framework's effectiveness in discovering diverse vulnerabilities across severity levels, with the collaborative mechanism and knowledge accumulation providing clear benefits over isolated agent approaches. The framework's modular design and clear separation of concerns facilitate both understanding and extension.

As LLMs continue to be deployed in increasingly critical applications, systematic security testing frameworks like ours become essential tools for identifying and addressing vulnerabilities before they can be exploited. Our work demonstrates that effective multi-agent collaboration for security testing need not require complex theoretical frameworks, but can be achieved through well-designed communication mechanisms and shared knowledge structures.

Future work will focus on expanding the strategy repertoire, implementing adaptive agent selection, and integrating the framework with automated defense improvement mechanisms to create comprehensive security development lifecycles for LLM systems.

A Implementation Details

A.1 System Configuration

Our implementation uses Python with the following key components. We leverage LangChain for LLM interaction, providing a unified interface for different model providers and enabling easy configuration of model parameters. The system uses Python's asyncio for concurrent operations, enabling parallel agent execution and efficient API utilization.

The codebase is organized into six core modules totaling 890 lines: agents.py (450 lines) for agent implementation and collaborative system, models.py (30 lines) for data model definitions, vulnerability_knowledge.py (120 lines) for knowledge base management, config.py (15 lines) for configuration parameters, main.py (100 lines) for main entry point and reporting, and test_system.py (175 lines) for system validation tests.

A.2 Configuration Parameters

Key configurable parameters include separate configurations for attacker and victim models, including API endpoints, model names, and authentication. The attacker LLM uses temperature 0.8 for creative strategy generation, while the victim LLM uses temperature 0.3 for stable, consistent responses. Configurable delay between agent actions (default 1.5 seconds) manages API rate limits. The number of testing iterations (default 5) distributes queries across iterations. The context window parameter specifies the number of historical vulnerabilities retrieved for context (default top-5 by severity and recency).

A.3 Output Format

The system generates comprehensive JSON reports containing total vulnerability count and summary, vulnerabilities grouped by attack strategy, vulnerabilities grouped by severity level (critical/high/medium/low), agent performance statistics, top-5 most severe vulnerabilities, and complete vulnerability details including prompts and responses.