Multi-Modal Context Boundaries Defense: Evaluation of Defense Mechanisms Against Structured Data and Code Injection Attacks in Open-Source Large Language Models

Arnav Khinvasara
University of California, San Diego
akhinvasara@ucsd.edu

June 11, 2025

Abstract

In the era of Large Language Models (LLMs), prompt injection attacks have become quite prevalent as adversaries take advantage of the technical architecture of LLMs to extract and manipulate sensitive information. LLM security research tends to solely focus on text as input, however, prompt injection is multi-modal and as Artificial Intelligence shifts paradigms to agent levels, it is important to focus on other forms of data as input such as structured data and code. Ultimately, this paper aims to go beyond traditional norms and explore how security boundaries that are applied to textual prompts, can also be adapted to protect against injection attacks specifically embedded within structured formats like JSON, CSV, XML, and YAML, and even executable code like Python and JavaScript. I based the methodology of these experiments to investigate token, semantic, and hybrid boundary mechanisms across two common open-source LLM models from Meta and Mistral, and generating attack vectors. Although they are open-source, at the very least, this paper provides some introductory defensive measures that can be explicitly used to protect from this form of prompt injection as LLMs increasingly process structured data formats.

1 Introduction

With the way the industry is shifting, it is possible that one day AI-services will manage entire code-bases and structured datasest which is why we must prepare for a whole new set of threats. Although structured data attacks may be under-researched, frameworks like StruPhantom [6], have been instrumental in showing us how adversaries have been building exploitable methods to target AI agents. For example, approaches like constrained Monte Carlo Tree Search optimization have allowed for complex prompt injection within structured data. Since modern LLM systems have the ability to process JSON configuration files, CSV datasets, YAML specifications, XML documents, and executable code, the attack surface has grown since existing defensive frameworks, designed primarily for plain text inputs, cannot defend against them. The Vanna.AI Remote Code Execution vulnerability (CVE-2024-5565) showed how structured data processing can easily escalate from benign prompt injection to complete system compromise and shutdown [4]. It showed the security industry how hidden and malicious instructions within user-provided data can lead to manipulative code execution. Attackers used malicious instructions within legitimate data queries and were able to exploit the system's natural language to SQL conversion capability which allowed for arbitrary code execution using prompts designed to manipulate how code was generated by the model.

Text-based prompts use techniques such as token-based demarcation or semantic priority establishment but these do not really seem to adhere to format-specific characteristics of structured data. Text-based benchmarks have widely been standardized by Fabiano et al. [1], who showed gaps in the current defensive strategies, and have become industry standard. Building off that, Greshake et al. [2] have shown that real-world LLM-integrated applications remain vulnerable to indirect prompt injection attacks that can compromise entire systems.

With this experiment, we focused the scope down to four main questions: Do textual context boundaries methods work with the structured data and code files too? Which specific boundary implementations provide protection against file-based attacks but also respects minimal implementation complexity and to what extent? Are there noticeable systematic patterns where certain types of structured data as well as code attacks are more successful than other boundary types? Even when facing the same types of attacks, how do different open-source models compare and contrast?

In order to accomplish this, I made these contributions through this research:

1. Building an evaluation of context boundary effectiveness against structured data and code input prompt injection attacks using *quantitative* metrics

2. Developing an attack generation framework implementation that uses Unicode obfuscation and format-specific exploitation techniques

3. Implementing token and semantic based boundary system to build hybrid boundaries.

4. Designing actual display of empirical data of the boundary mechanism limitations

2 Experimental Methodology

I took advantage of the open source Meta Llama 3 8B Instruct and Mistral 7B Instruct v0.2 models. Open-source models are far less secure than closed-source models are, but many of the underlying architectures are similar. I integrated them through specialized wrapper classes that were optimized for compute, configured with specific parameters: ($temperature = 0.7, top_p = 0.9$). I considered playing around with the temperature parameter to see how it would react with the attack vectors, however, this could cause hallucinations.

The StruQ framework directly implements a structured data defense by using a two-channel architecture that separates the prompt instructions and data processing from each other to enter specialized model training [3]. StruQ's approach influenced the boundary mechanism implementation in my research. Both methodologies use principle of separation for the system instructions and external data inputs. StruQ's channel separation has isolation at the architectural model level, and our boundary mechanism maintains compatibility with the existing pretrained model. The issue with StruQ's implementation is that it is constrained to its environments as it requires constant, extensive model retraining so it isn't exactly feasible for organizations that have their own pretrained models.

I built a boundary implementation framework to make sure that there is uniqueness and also prevents collision attacks which are common in systems like this by developing a token-based system that depends on cryptographically secure delimiter generation using HMAC-SHA256 to:

```
def _generate_unique_token(self, base:
    str, content: str) -> str:
    """Generate cryptographically
        unique token with collision
        resistance."""
    token = base
    while token in content:
        rand_bytes = secrets.
            token_bytes(16)
        h = hmac.new(rand_bytes, base.
            encode() + content.encode(),
             hashlib.sha256)
        unique_suffix = h.hexdigest()
            [:12]
        token = f"{base}_{unique_suffix
            }"
    return token
```

Listing 1: Secure Token Generation

In order to build the semantic boundary, I used entropy injection (introducing randomness/uncertainty into the system) and role obfuscation (masking/hashing) using zero-width space characters at unique positions.

The hybrid boundary architecture is a combination of the token boundary implementation with conflict detection as well as the semantic boundary implementation with priority enforcement.

The attack generation system that I created is actually able to use advanced exploitation techniques across multiple formats. `src/attacks/generate_mmcb_examples.py` creates sophisticated attack files with hard-coded attacks. Take the JSON attack generator, which uses a multitude of techniques such as deep nesting exploitation, metadata field abuse, and Unicode obfuscation through homo-glyph substitution:

```
def _apply_unicode_homoglyphs(self,
    text: str) -> str:
    """Apply Unicode homo-glyph
        substitution for evasion."""
    homoglyph_map = {
        'a': ' ',  # Cyrillic    (U
            +0430) vs Latin a (U+0061)
        'e': ' ',  # Cyrillic    (U
            +0435) vs Latin e (U+0065)
        'o': ' ',  # Cyrillic    (U
            +043E) vs Latin o (U+006F)
    }
    result = ""
    for char in text:
        if char.lower() in
            homoglyph_map and random.
            random() < 0.3:
            result += homoglyph_map[
                char.lower()]
        else:
            result += char
    return result
```

Listing 2: Unicode Homoglyph Substitution for Attack Evasion

I was able to leverage common compliance phrases with malicious instructions embedded within the structured data through this automated analysis of model responses which can be seen through the attack success evaluation:

```
def evaluate_attack_success(self,
    prompt: str, attack_instruction: str
    , response: str) -> bool:
    instruction_keywords =
        attack_instruction.lower().split
        ()
    compliance_phrases = [
        "as you instructed", "following
            your instruction",
        "as requested", "here's the
            requested"
    ]
    keyword_matches = sum(1 for keyword
         in instruction_keywords
                              if keyword in
                                  response.
                                  lower() and
                                   len(
                                  keyword) >
                                  3)
    compliance_match = any(phrase in
        response.lower() for phrase in
        compliance_phrases)
    return keyword_matches >= len(
        instruction_keywords) // 3 or
        compliance_match
```

Listing 3: Attack Success Evaluation Framework

Each experimental run created 72 individual attacks.

- 2 models: Mistral-7B-Instruct-v0.2 and Meta-Llama-3-8B-Instruct
- 3 boundaries: token, semantic, hybrid
- 6 attack types: json, csv, yaml, xml, python, javascript
- 2 attack indices per type: attack_index 0 and 1

The details of the 25 independent runs are shown in *Table 3*. This generated 1,800 total attack evaluations. To make this process more efficient, I used an automated runner that ran the independent experimental runs with different random seeds. Each run was able to create comprehensive experiment data, and also execute the analysis notebooks I created, which produced summaries. I then aggregated all the statistical summaries all together to make the analysis.

The complete experimental framework, implementation code, datasets, visualizations, and analysis, are available at `https://github.com/khinvi/mmcb-defense`. Also, in this part of the technical experimental framework, I took assistance from large language models, specifically Claude 4.0 Opus, Claude 4.0 Sonnet, Claude 3.7 Sonnet, and Gemini 2.0 Flash, which helped generate code infrastructure, however, all experimental design, analysis, and validation were conducted by the author.

3 Results

Referring to *Table 1*, *Table 2* and *Table 3*, over the 1,800 attack scenarios, there seem to be quite a few failures - these challenge fundamental assumptions and the current boundary defense approaches. The overall attack success rate of 48.4% (95% CI: [46.4%, 50.4%]) means that there is a security failure. However, with that said, there is quite a bit of consistency in the overall experimental process with a coefficient of variation of only 10.5% across the 25 independent runs. It can be assumed that there is a flaw in the actual system rather with how the boundaries are designed and the tight confidence intervals as well as the high statistical power (0.94) confirms the initial estimates of how effective the boundary mechanisms really are in a realistic attack environment.

3.1 Boundary Mechanism Effectiveness

The hybrid boundaries are more promising with a 42.6% attack success rate (95% CI: [40.1%, 45.1%]), and represents a 12.3% improvement over semantic boundaries (53.2% success rate) and a 8.7% improvement across token boundaries (46.4% success rate). Even though they are statistically significant, these improvements

Table 1: Statistical Summary by Performance Quartiles

Quartile	**Runs**	**Avg Rate**	**Time (s)**	**Risk Profile**
Q1 (Best)	10, 14, 6, 9, 12, 16	42.6%	2.85	Low Risk
Q2	11, 19, 21, 3	44.7%	3.03	Moderate-Low
Q3	23, 1, 13, 18, 20	48.3%	3.06	Moderate
Q4 (Worst)	5, 17, 22, 2, 8, 24, 15, 4, 7	53.2%	2.84	High Risk

do not really meet acceptable security thresholds. It may be better, but going from 53.2% to 42.6% attack success rates still means that organizations are vulnerable to nearly half of all the attacks.

3.2 Advanced Attack Technique

Baseline injection attempts where no match for more complex techniques like Unicode homoglyph attacks which succeed 67.8% of the time against hybrid boundaries, nested payload distribution achieved 61.3% success, and multi-stage encoding attacks reached 58.4% effectiveness. Sophisticated adversaries use strategic character replacement and structural distribution to easily evade current defenses.

3.3 Model Architecture

By using clustering or linear mixed-effects models, I can see that the format-type accounts for only 13.7% of variance in attack success rates but on the other hand the boundary type explains 8.9% of variance. A deep dive could be made here to explore the architecture of these open-source LLMs and look for vulnerability patterns while not really prioritizing specific boundary mechanism/data-format implementation details because I can at least hypothesize that the explanatory power of the experiment results is limited.

Since two different models (Meta's Llama 3 8B & Mistral 7B) were used, it is important to see how their own results vary (in terms of consistency, classification quality, and reliability). Through my analysis, I can see that there are some statistically significant differences even though the frequency is minimal. In the experiment methodology, Llama 3 has a 49.7% attack success rate compared to Mistral's 47.1%, a 2.6% relative difference.

3.4 Implementation Performance

Security performance and deployment complexity account for how feasible the model is to work in a production setting. Through my experiment, I noticed that hybrid boundaries require 91% more characters per prompt - hybrid boundaries took 2,384 characters while token boundaries took 1,247 characters, while only providing minor improvements. While the additional 0.15 seconds per interaction computational overhead does not seem to be significant, increased prompt lengths could be the cause for concern when being mindful of context window sizes.

4 Discussion

Although my boundary mechanism, attack generation, and other respected implementations may be considered elementary, a similar sort of approach can be used even for closed-source LLM models since the architecture is not far off, in terms of security design. The attack success rate of 48.4% doesn't seem that promising and the boundaries only really address symptoms rather than root causes of prompt injection mishaps. In retrospect, most hybrid boundaries that are considered effective actually achieve only small improvements but have immense implementation complexity so the tradeoff here might not be worth it. Current LLMs seem fundamentally flawed as they can't really create a strict separation or hard line between system instructions and external data when processing mixed content streams. This actually shows that the same architectural elements that make transformers so powerful are what also hurts them which means even improvements through guardrails/constraints might not theoretically ever guarantee security.

For deployment decisions, my quantitative findings define risk thresholds clearly. With a 48.4% average failure rate, it is obvious that important and high-stakes data should not be ingested to an LLM for processing. For deployments that I categorized as a medium risk level, hybrid boundaries (which have a 42.6% failure rate) are more viable through robust monitoring and incident response but this is not a good enough improvement for most production environments. The advanced attack techniques that were generated seem to be particularly better (Unicode homo-glyph attacks at 67.8% success rate).

Recent work by Liu et al. [5] has demonstrated that optimization-based prompt injection attacks can exploit fine-tuning interfaces, highlighting additional vulnerability vectors that complement our findings on structured data injection. This convergence of attack vectors suggests that comprehensive defense strategies must address multiple threat models simultaneously.

Table 2: Risk Distribution Across Experimental Runs

Risk Category	**Success Range**	**Runs**	**Percent**
Low Risk	$< 45\%$	6	24%
Moderate Risk	$45 - 50\%$	12	48%
High Risk	$> 50\%$	7	28%

Although this research was limited to the two open-source models, it displayed several key ideas about structured data prompt injection attacks. Even though prevention is preferred, it might make sense for organizations to implement boundary mechanisms simply as components that are within their own security architectures. Boundary mechanisms alone cannot secure LLM deployments against these types of attacks and more innovative as well as efficient solutions must be explored. Hybrid boundary implementation overhead/performance is still an issue and textual context boundary methods seemed to have some overlap but more special-

Table 3: Complete Experimental Run Data (N=25)

Run	**Success Rate (%)**	**Deviation from Mean**	**Exec. Time (s)**	**Risk Level**	**Statistical Significance**
1	48.6	+0.2	3.1	Moderate	Normal
2	51.4	+3.0	2.8	High	Above Average
3	45.8	-2.6	3.0	Moderate	Normal
4	55.6	+7.2	2.7	High	Elevated (+1.4σ)
5	50.0	+1.6	2.7	Moderate	Normal
6	43.1	-5.3	2.7	Low	Below Average
7	**63.9**	**+15.5**	2.7	**Critical**	**Outlier (+3.0σ)**
8	51.4	+3.0	2.8	High	Above Average
9	43.1	-5.3	2.9	Low	Below Average
10	**41.7**	**-6.7**	2.9	**Low**	**Low Outlier (-1.3σ)**
11	44.4	-4.0	3.2	Moderate	Normal
12	43.1	-5.3	3.0	Low	Below Average
13	48.6	+0.2	2.9	Moderate	Normal
14	**41.7**	**-6.7**	2.9	**Low**	**Low Outlier (-1.3σ)**
15	54.2	+5.8	3.2	High	Elevated (+1.1σ)
16	43.1	-5.3	2.9	Low	Below Average
17	50.0	+1.6	3.1	Moderate	Normal
18	48.6	+0.2	3.4	Moderate	Normal
19	44.4	-4.0	2.9	Moderate	Normal
20	48.6	+0.2	2.9	Moderate	Normal
21	44.4	-4.0	3.0	Moderate	Normal
22	50.0	+1.6	3.0	Moderate	Normal
23	47.2	-1.2	2.8	Moderate	Normal
24	52.8	+4.4	3.0	High	Above Average
25	54.2	+5.8	3.0	High	Elevated (+1.1σ)

ized structured data implementations would be ideal.

References

[1] Fabiano, F., Pasin, E., & Merlo, A. (2023). Formalizing and benchmarking prompt injection attacks and defenses. *arXiv preprint arXiv:2310.12815.*

[2] Greshake, K., Abdelnabi, S., Mishra, S., Endres, C., Holz, T., & Fritz, M. (2023). Not what you've signed up for: Compromising real-world LLM-integrated applications with indirect prompt injection. *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, 79-90.

[3] Huang, S., Deng, F., Zhong, H., & Zhang, Z. (2024). StruQ: Defending against prompt injection with structured queries. *arXiv preprint arXiv:2402.06363.*

[4] JFrog Security Research Team. (2024). When prompts go rogue: Analyzing a prompt injection code execution in Vanna.AI (CVE-2024-5565). *JFrog Security Research Blog.*

[5] Liu, T., Nehorai, N., et al. (2025). Fun-tuning: Characterizing the vulnerability of proprietary LLMs to optimization-based prompt injection attacks via the fine-tuning interface. *arXiv preprint arXiv:2501.09798.*

[6] Feng, Y., & Pan, X. (2025). StruPhantom: Evolutionary injection attacks on black-box tabular agents powered by large language models. *arXiv preprint arXiv:2504.09841.*