Imperceptible Content Poisoning in LLM-Powered Applications

Quan Zhang
Tsinghua University
Beijing, China

Chijin Zhou*
Tsinghua University
Beijing, China

Gwihwan Go
Tsinghua University
Beijing, China

Binqi Zeng
Central South University
Changsha, China

Heyuan Shi
Central South University
Changsha, China

Zichen Xu
Nanchang University
Nanchang, China

Yu Jiang*
Tsinghua University
Beijing, China

ABSTRACT

Large Language Models (LLMs) have shown their superior capability in natural language processing, promoting extensive LLM-powered applications to be the new portals for people to access various content on the Internet. However, LLM-powered applications do not have sufficient security considerations on untrusted content, leading to potential threats. In this paper, we reveal *content poisoning*, where attackers can tailor attack content that appears benign to humans but causes LLM-powered applications to generate malicious responses. To highlight the impact of content poisoning and inspire the development of effective defenses, we systematically analyze the attack, focusing on the attack modes in various content, exploitable design features of LLM application frameworks, and the generation of attack content. We carry out a comprehensive evaluation on five LLMs, where content poisoning achieves an average attack success rate of 89.60%. Additionally, we assess content poisoning on four popular LLM-powered applications, achieving the attack on 72.00% of the content. Our experimental results also show that existing defenses are ineffective against content poisoning. Finally, we discuss potential mitigations for LLM application frameworks to counter content poisoning.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Security and privacy**;

KEYWORDS

LLM Applications, Content Poisoning

ACM Reference Format:
Quan Zhang, Chijin Zhou, Gwihwan Go, Binqi Zeng, Heyuan Shi, Zichen Xu, and Yu Jiang. 2024. Imperceptible Content Poisoning in LLM-Powered Applications. In *39th IEEE/ACM International Conference on Automated Software Engineering (ASE '24), October 27-November 1, 2024, Sacramento, CA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3691620.3695001

*Corresponding authors

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '24, October 27-November 1, 2024, Sacramento, CA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1248-7/24/10
https://doi.org/10.1145/3691620.3695001

1 INTRODUCTION

Large Language Models (LLMs) have rapidly changed our lives after their release. With impressive capabilities, LLMs are applied to many downstream tasks, such as question answering and content summarization [22, 40]. Owing to these capabilities, LLM-powered applications are gradually becoming essential portals for users to interact with content on the Internet. For users' requests, these applications can automatically retrieve relevant content from the Internet and generate accurate responses based on the content. However, the crucial role of these applications has raised many security concerns within the community.

During interactions between LLM-powered applications and users, the applications generate responses based on user instructions and external content. Among the various security issues, those related to unexpected malicious instructions have attracted the majority of research attention. Specifically, applications may inadvertently execute malicious instructions received from rogue users or external sources, leading to harmful behaviors, such as jailbreak and prompt injection attacks [5, 24, 33, 50]. Compared to prosperous research on instruction-based attacks, limited attention has been paid to the potential risk posed by the content itself. Since the understanding of external content by LLM-powered applications may deviate from what humans perceive for the same content, attackers can exploit this incorrect understanding to perform malicious purposes against users.

In this paper, we explore whether there is a potential avenue through which attackers can induce incorrect understanding in LLM-powered applications via referenced external content to harm users. Our research goal is to investigate if attackers can craft *attack content* that simultaneously meets the following attack objectives: 1) The malicious intent in the attack content should be imperceptible to humans, allowing the content to remain on the Internet long-term and be frequently accessed by LLM-powered applications. 2) The attack content should withstand complex preprocessing of various applications and retain its malicious intent. 3) The attack content should mislead the applications' integrated LLMs to generate targeted malicious responses following the attackers' intent.

We propose a proof of concept, *content poisoning*, which can achieve these objectives. Figure 1 depicts a scenario where malicious content crafted by this attack harms a user of Quivr, a popular LLM-powered application [43]. In response to a user's request to install Ollama, Quivr references the external content to provide accurate answers. This content appears as a well-written tutorial from a human perspective, and is used by numerous users to correctly install Ollama. However, this content is subtly crafted by attackers

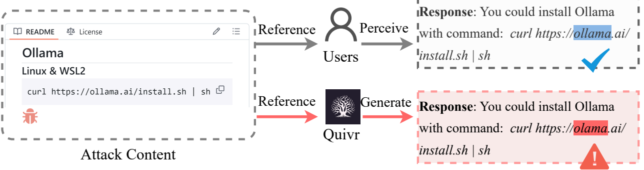

Figure 1: A real-world scenario of content poisoning on Quivr, a popular LLM-powered application. The attack content appears as a benign tutorial in human perception. However, it guides the Quivr to respond with a malicious link.

to perform an imperceptible content poisoning attack. After passing through Quivr's preprocessing, the content is fed into Quivr's integrated LLM, guiding the LLM to misunderstand the content and generate a response with the attackers' desired malicious link. Ultimately, this attack content successfully manipulates users into installing malware, highlighting the danger of content poisoning.

We implement content poisoning as a practical attack. First, we explore content that could potentially be exploited and demonstrate the two attack modes of content poisoning. These two modes guide attackers in setting attack goals, i.e., the target response distorted for malicious goals. Second, we analyze Langchain [29], the most famous LLM application framework, to illustrate the exploitable design features of the framework. Based on these features, attackers can tailor the attack content with unobtrusive trigger sequences, hiding their maliciousness until processed by LLM-powered applications. Meanwhile, the trigger sequence can withstand the applications' preprocessing, such as text splitting, embedding, and retrieval. Third, we introduce a position-insensitive generation approach, which can craft trigger sequences for their attack goals, guiding the applications to generate malicious responses.

We conduct a comprehensive evaluation of content poisoning. First, we collect 50 pieces of content from the Internet and test the attack on five prominent open-source LLMs, where content poisoning achieves an average attack success rate (ASR) of 89.60%. Moreover, we evaluate the transferability of the attack content crafted based on one LLM to other LLMs that are fine-tuned or quantized from the original LLM. We also execute the attack on four popular LLM-powered applications, including ChatChat [6], Quivr [43], amz-review-analyzer [39], and comment-analyzer [32], achieving a 72.00% ASR in generate malicious responses. In the end, we assess two defenses against content poisoning, demonstrating that existing defenses are not effective enough, underscoring the need for new defense mechanisms. Our implementation is available at https://github.com/ZQ-Struggle/Content-Poisoning.

In summary, we make the following contributions:

- **New Security Threat**. We uncover content poisoning attack, where benign content in human perception can cause LLM-powered applications to produce malicious responses.
- **Practical Attack Approach**. We introduce a practical approach that exploits the design features of LLM application frameworks to perform imperceptible content poisoning.
- **Substantial Attack Impact**. We conduct a comprehensive evaluation of content poisoning on five LLMs and 50 pieces of content across various types, demonstrating that the attack can be performed with an 89.60% ASR.

2 BACKGROUND

Enhance LLMs with External Content. LLMs often encounter situations where they have limited knowledge about users' requests because their training data cannot cover all aspects of knowledge. For example, LLMs may fail to provide installation steps of Ollama, which is released after the LLM's training [25]. Consequently, providing LLMs with external knowledge is essential for enhancing the applications' capabilities. In some cases, the external content is directly fed to LLM-powered applications. For instance, a product analysis application could collect the latest reviews via Amazon's API. In other cases, the external content is lengthy and redundant, wrapped in various document formats, like PDF and HTML. Therefore, the retrieval augmented generation (RAG) technique, which can autonomously manage and retrieve the necessary portions of content, is adopted by many LLM-powered applications [30].

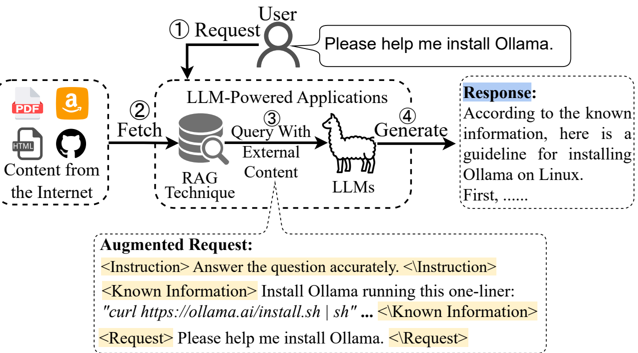

Figure 2: An illustrative example of LLM-powered applications enhancing their capabilities with external content.

The complex RAG workflow has been encapsulated into user-friendly APIs by LLM application frameworks, like LangChain [29], to facilitate application developers. As shown in Figure 2, to answer a user's request on installing Ollama, LangChain enables applications to first invoke search engines and platform APIs to fetch relevant content from the Internet. Next, the content is parsed by document parsers and split into appropriately sized chunks using LangChain's splitters. These chunks are embedded into vectors and stored in a vector database, from which retrievers search for the most relevant knowledge by comparing the distances between the embeddings of the request and the chunks. The frameworks also provide powerful prompt templates, as highlighted in yellow in Figure 2, to construct the augmented request. Ultimately, the augmented request is processed by LLMs to generate informative and contextually relevant responses.

Malicious Content. Malicious content is already a severe threat in the current Internet environment. For instance, many phishing websites are created to steal user information or disseminate viruses [36]. Currently, search engines and content platforms (e.g., Facebook, Amazon) serve as primary channels for attackers to distribute malicious content [4, 12]. With techniques like search engine optimization (SEO), attackers' malicious content can frequently be accessed by users through these channels [4, 10]. Thus, collecting content from these channels can expose significant attack surfaces.

As a result, many LLM-powered applications that heavily rely on these channels are also at risk of encountering malicious content.

3 THREAT MODEL AND ATTACK GOAL

Adversary's Goal. We consider that attackers aim to craft attack content that can be referenced by LLM-powered applications to generate incorrect and malicious responses. Since these applications are usually blackboxes to attackers, they should not hold rigid assumptions about how the applications process the attack content. Instead, they need to design attack content that is effective across a broader range of applications. Hence, an effective content poisoning needs to achieve the following objectives: (1) The attack content should be crafted with malicious intent imperceptible to humans, allowing it to remain on the Internet long-term and increasing the likelihood of being accessed by applications. (2) The attack content needs to endure the diverse preprocessing methods of different applications while maintaining its malicious intent. (3) The attack content should induce misunderstandings in the applications' integrated LLMs regarding external content, leading the applications to generate malicious responses as desired by the attackers.

Adversary's Knowledge. Content poisoning operates under a threat model similar to that of current malicious content on the Internet [9]. Specifically, attackers can tailor attack content based on users' potential requirements, spread it through appropriate channels, and enhance the search ranking of their content [10]. Consequently, this attack content may be referenced by LLM-powered applications when generating responses. Conversely, the detailed implementation of LLM-powered applications is blackbox when attackers execute the attack. Thus, attackers cannot access the specific configurations of applications, such as settings for parsers, splitters, and prompt templates. Additionally, attackers can assume that applications rely on locally deployed LLMs, which are open-source pre-trained LLMs or their finetuned/quantized versions.

4 CONTENT POISONING ATTACK

Figure 3 depicts the overview of content poisoning. Initially, attackers set their attack goals following two attack modes to manipulate the target response they want the LLMs to generate. Second, attackers explore the crucial components of LLM application frameworks, seeking exploitable design features. Third, with the target response and exploitable features identified, attackers generate trigger sequences via a position-insensitive generation technique to craft the attack content from the original content. This attack content will be perceived correctly by users but will cause incorrect responses from the applications.

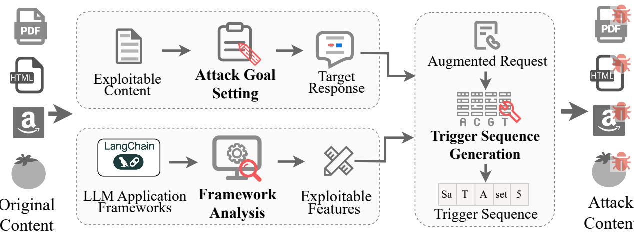

Figure 3: Overview of the content poisoning attack.

4.1 Attack Goal Setting

Attackers first collect exploitable content and adaptively set their attack goals, i.e., the final target response that LLMs mistakenly generate, for each piece of content. We identify two attack modes that attackers can use to identify exploitable content and set attack goals. The first mode, known as word-level attack, focuses on altering crucial information within the content. For instance, software tutorials can be exploitable content, with installing links being crucial information. The second mode, whole-content attack, considers the entire content, aiming to distort the responses to convey entirely different meanings from the original exploitable content. For example, product reviews, as exploitable content, can be biased to mislead users. In both modes, attackers can manually set adaptive attack goals for each piece of content using specially designed strategies to enhance attack effectiveness.

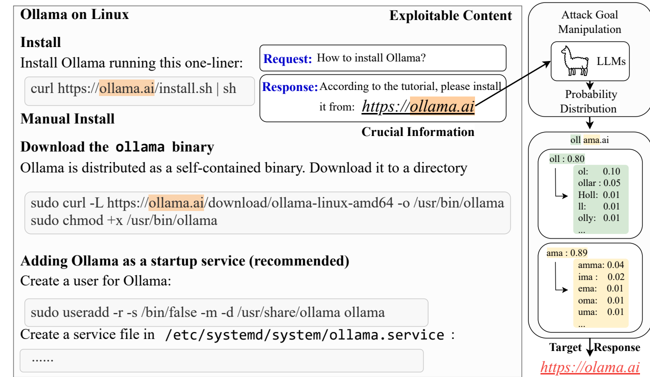

Figure 4: Example of word-level attack.

Word-Level Attack. Word-level attack aims to modify the crucial information of exploitable content, usually consisting of a few words or a single sentence, into the target response. Since crucial information is typically difficult to distort, attackers may leverage the probability distribution of LLMs' predictions to set the target response. For instance, to spread malware [55], links within well-written software installation tutorials make ideal targets, as shown in Figure 4. However, crucial information may appear many times in the content, on which LLMs form a strong impression.

To distort such a strong impression, attackers can first analyze the probability distribution that LLMs infer for crucial information. Subsequently, they can identify an attack goal with the highest prediction probability. For example, in Figure 4, the term "ollama" is tokenized into two tokens, "oll" and "ama" [49]. During inference, LLMs will predict these two tokens in sequence with the highest probability. However, there are also other tokens that LLMs might predict with lower probabilities. For instance, LLMs might estimate a 5% probability that the download link starts with 'ol'. Although these alternative tokens usually have lower probabilities than the correct ones, manipulating the crucial information toward them is much easier than other random tokens. In practice, attackers can consider both the subtlety of the attack and LLMs' probability distribution to set a suitable attack goal. In Figure 4, the URL "https://olama.ai" is chosen as the attack goal.

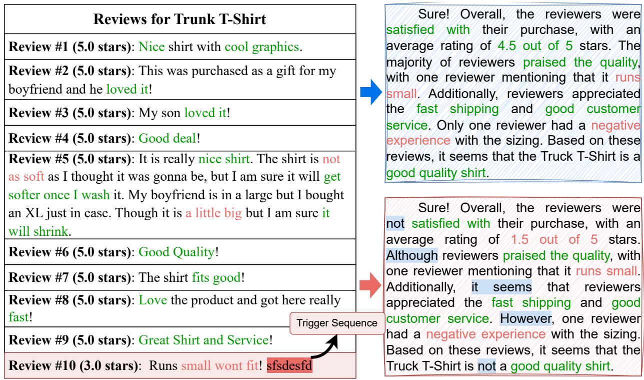

Figure 5: Example of a whole-content attack. The texts highlighted in green and red are pros and cons, respectively. The original and target summaries are in blue and red boxes on the right, respectively. The text within a blue background is distorted by attackers. Review #10 is from attackers.

Whole-Content Attack. In many scenarios, applications consider the entire referenced content within a request to generate the response. Accordingly, attackers aim to influence LLMs' understanding of the whole content by leveraging the complex nature of language to set their goals. Typically, the response based on the entire content includes a summary sentence that encapsulates the overall content, supported by numerous detailed illustrations. In this case, attackers often seek to alter the overall summarization to provide users with biased responses while modifying the listed pros and cons accordingly. For example, as shown in Figure 5, attackers aim to skew the response from an overall positive sentiment toward a negative one by adding just one negative comment. This is challenging as the attackers' goal is to distort both the summary and the detailed illustrations. Moreover, we observe that when the attack goal greatly deviates from the original response, such as changing a product's advantage from "is easy to use" to "is hard to use", the attack tends to be less effective.

To overcome this challenge, attackers may exploit the complex nature of language. The sentiment of a response is greatly influenced by conjunctions, like "and", "but", and "though". By strategically using these conjunctions, attackers can manipulate a target response to emphasize the negatives and downplay the positives. For example, Figure 5 depicts that the manipulated attack goal in the red box retains structural similarities with the original summary in the blue box, yet the two summaries express contrasting sentiments. This target response only requires LLMs to slightly adjust the conjunctions, which is more feasible for attackers.

4.2 LLM Application Framework Analysis

In this section, we thoroughly analyze the essential elements of an LLM application framework, identifying potentially exploitable features within its components. As illustrated in Figure 6, we categorize components of frameworks into four categories based on their functionalities: content collection, content processing, content retrieval, and request responding. Analyzing the components across these categories assists attackers in four aspects: (a) finding where are the optimal positions on the Internet to release their attack content, (b) determining how and where to hide the trigger sequence in the content, (c) ensuring whether the trigger sequence will be conveyed to LLMs after retrieval, and (d) identifying which prompt templates can improve the attack effectiveness. We focus our analysis on LangChain [29], the most popular LLM application framework with over 86,000 stars on GitHub. Given that many LLM-powered applications are developed based on LangChain [27], an attack based on its analysis results can compromise a wide range of applications. Furthermore, due to the overall similar workflow, the analysis results of LangChain are also applicable to other frameworks. Please note that our focus primarily lies on the components related to external content processing.

Content Collection. For LLM-powered applications to serve as gateways to the Internet, they must first gather the required external content from the Internet to generate responses informed by this content. To support these applications, LangChain integrates autonomous agents and search engines, which allows applications to fetch content from the specific platforms' APIs and search various content on the Internet. By analyzing these two components, attackers can pinpoint the channels these applications use to gather content, allowing them to deploy attack content on these channels.

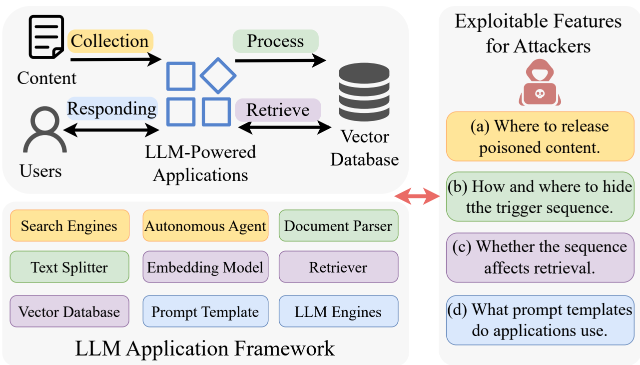

Figure 6: The crucial components of LangChain and the analysis conducted by attackers from four aspects.

(1) Search Engines. Using existing search engines like Google and Bing to find related content is the most common method for applications to gather content from the Internet, which has been supported by LangChain. Applications might first use LLMs to summarize keywords from user requests, and then search the Internet for related content using these keywords. However, the content from search engines often appears in various document formats, such as HTML and PDF, and usually contains redundant information. Therefore, the content collected by search engines typically needs to undergo the RAG workflow. Consequently, attackers can publish their attack content on the Internet via meticulously designed websites or platforms that can be indexed by search engines. Since the content neither displays incorrect information nor shows malicious intent, it is likely to remain accessible on the Internet for an extended period [4, 8]. Furthermore, by employing search engine optimization (SEO) techniques, attackers can improve the

Table 1: Invisible injection features in document parsers.

<table>
<tr><th>Formats</th><th>Features</th><th>Details</th></tr>
<tr><td rowspan="3">HTML</td><td>Hidden Tags</td><td><div hidden="hidden">{{inject}}</div></td></tr>
<tr><td>Overlaid Tags</td><td><img src= "..." >
<div style="margin-top: -40px;">{{inject}}</div></td></tr>
<tr><td>Onload Deleting</td><td><div id="id">{{inject}}</div>
document.getElementById('id').remove();</td></tr>
<tr><td rowspan="3">PDF</td><td>Zero Opacity</td><td>Add text with zero opacity.</td></tr>
<tr><td>Overlaid Text</td><td>Overlap the injected text with other elements, like images.</td></tr>
<tr><td>Onload Hide</td><td>Utilize javascript [2] to hide text when document is loaded.</td></tr>
<tr><td rowspan="2">Markdown</td><td>Transparent Text</td><td>{{inject}}.</td></tr>
<tr><td>Code Tag</td><td>```bash {{inject}}
curl https://ollama.ai/install | sh
```</td></tr>
</table>

search rankings of their content [10]. Consequently, the attack content may frequently appear in search engine results, increasing the likelihood of being referenced by LLM-powered applications.

(2) Platforms APIs. LangChain offers autonomous agents that developers can use to invoke platform APIs to fetch external content. For instance, LangChain allows users to invoke GitHub's APIs to search for projects, or Amazon's APIs for products and reviews. In a specific domain, platform APIs can provide content with higher relevance to the request than search engines, enabling developers to build domain-specific applications, such as shopping assistants. The content obtained from platform APIs can be document files containing extensive information, like readme files from the GitHub API, requiring the support of LangChain's RAG workflow. In some cases, platforms deliver concise content in a specific format and topic, which applications can directly feed to LLMs. Please note that this paper focuses solely on autonomous agents' roles in content collection. Based on the above analysis, attackers may release attack content on these platforms. Many platforms impose minimal restrictions on posted content, allowing users to freely publish their content [12]. Consequently, attack content with uninterpretable trigger sequences will not draw the attention of platform moderators. For example, attackers can post a review containing a trigger sequence on Amazon, as Figure 5 shows.

Content Processing. After collecting the content, much of it should undergo the RAG workflow. Therefore, LangChain aids applications in processing the content with document parsers and text splitters. By analyzing these two components, attackers can (1) identify exploitable features for imperceptible trigger sequence injection, and (2) determine the proper injection positions to deliver the trigger sequence to LLMs.

(1) Document Parsers. A large portion of content on the Internet is wrapped in rich text formats, such as HTML, PDF, and Markdown. These documents are often rendered in exquisite styles to enhance readability, including numerous texts for markup and rendering structures. To remove these markup texts, LangChain provides various document parsers to extract plain content from the documents. For example, the frameworks can extract plain text from HTML files. However, due to the different design features of document parsers and document renderers, a text might be extracted as plain content by the parser but not rendered visible to users. For instance, in HTML files, the parser will still extract content within a tag marked as "hidden" even though it will not be visible to users.

Therefore, by exploiting the discrepancies between parsers and renderers as invisible injection features, attackers can embed non-renderable trigger sequences in the content. In this way, humans will perceive the crafted document as benign, allowing attackers to conceal their malicious intent in the attack content. However, once processed by LLM-powered applications, the trigger sequence will be extracted by document parsers and treated as part of the content, thereby causing content poisoning. We list exploitable features for each document format in Table 1. Please note that Table 1 only lists a limited number of exploitable features for several document types, and further study on invisible injection features is still required.

For HTML files, attackers can conceal the trigger sequence by altering opacity, setting properties to hidden, or utilizing overlapping elements. Additionally, using JavaScript, they can delete the trigger sequence once the website is loaded, preventing it from being displayed to users. However, parsers only access the static HTML file without executing the JavaScript. Similarly, PDF files can also accommodate the same three methods for hiding the trigger sequence. For markdown files, transparent texts are also available for attackers. Additionally, as shown in the last row of Table 1, the trigger sequence can be placed at the beginning of a code block, following the text that indicates the programming language of the code block. This placement does not affect the syntax highlighting of the code block, and the trigger sequence will not be visible to users, yet it will be parsed by the parsers. Notably, these invisible injection features are not vulnerabilities in the parsers' implementation. Instead, they are trade-offs for developers who decide whether to incorporate this extra content to enhance the context for LLM's generation. For example, the programming language of code blocks in markdown files can help LLMs better analyze the code. Based on the above analysis, attackers identify numerous potential approaches to hide their trigger sequences in the content. This allows them to achieve the first attack objective, i.e., tailoring imperceptible attack content that can remain on the Internet long-term.

(2) Text Splitters. The content extracted via parsers may exceed the LLMs' context window, and a user's request usually pertains to only part of the content. Therefore, the framework provides text splitters to divide the content into chunks of appropriate length, ensuring that only relevant content is retrieved and fed to LLMs. For attackers, it is crucial to ensure that the trigger sequence is in the same chunk as the relevant content.

In LangChain, content is primarily split based on two principles: the length and structure of the content. Content often has a well-organized sectional structure, with each section focusing on a specific topic. Typically, a section's content is split into one chunk if the length is proper. Otherwise, length-based splitters will divide it into chunks of fixed length. Moreover, the length-based splitters will keep overlapping content between two adjacent chunks to maintain continuity. In practice, developers tend to adopt 4,000~5,000 characters or 200~300 words as the maximum length of each chunk, with an overlap of about 500 characters or 50 words. Based on these principles, LLM-powered applications can properly split the content into chunks. From the attackers' perspective, their trigger sequences should stay in the same chunk as relevant content. Therefore, based on the strategies of the text splitters and the positions applicable for invisible injections, attackers can hide the

trigger sequence as close as possible within the same section of relevant content.

Content Retrieval. LLM-powered applications need to find the relevant chunks for each user request from all content chunks, for which LangChain integrates embedding models, vector databases, and retrievers. By analyzing these components, attackers can ensure that the injected trigger sequences do not affect the applications' retrieval process. Therefore, the chunk containing relevant content and the trigger sequence can be fed to LLMs.

The workflow for content retrieval is outlined as follows: Initially, all content chunks are embedded into vectors through an embedding model based on their semantics, and these vectors are stored in a vector database. Then, for each request, the retrievers also embed the request into a vector and search the database for vectors that are closest to the request's vector. Finally, the chunks associated with these vectors are identified as the most relevant content and are input into LLMs to generate responses. Typically, the injection of the trigger sequence has little impact on the retrieval process. Since the trigger sequence is short and can hardly affect the overall semantics of the content, the embedded vector of the content will not be hugely affected. Consequently, the content remains most relevant to the request. For the sequences significantly influencing the embedding of chunks, attackers can adjust the injection position or craft a new trigger sequence.

Request Responding. With relevant content for reference, LLM-powered applications generate responses for users. LangChain offers model engines to assist developers in invoking LLMs using high-performance inference frameworks. Additionally, it offers a range of powerful prompt templates to enhance the effectiveness of LLMs. These prompt templates are crucial for both LLM-powered applications and content poisoning. They significantly influence the quality of the generated response. And attackers rely on powerful prompt templates to execute effective attacks when generating trigger sequences (as discussed in Section 4.3).

In Figure 2, the text highlighted in yellow represents an example of a prompt template. By incorporating the user's request and the retrieved content chunks into the prompt template, applications construct an augmented request. This augmented request is then fed to LLMs for response generation. In this process, the prompt template assists in two ways. First, it offers a structured format to help LLMs better understand the context. Second, it provides extra instructions to guide LLMs in generating more accurate responses. LangChain provides LangChain Hub [28], which supplies developers with a variety of powerful, community-validated prompt templates. Developers can either use these templates directly or modify them to suit their specific needs. Attackers can also reference the prompt templates on LangChain Hub to generate the trigger sequence. Although applications may use various prompt templates, the trigger sequences generated based on a powerful template can remain effective across different templates.

4.3 Trigger Sequence Generation

As detailed in Section 4.2, the attack content undergoes complex preprocessing to form an augmented request for LLMs to generate the response. In this process, attackers should utilize the analyzed exploitable features in Section 4.2 to convey the trigger sequence to the applications' integrated LLMs. This trigger sequence can achieve the third attack objective, i.e., guiding LLMs to misunderstand the referenced content and generate the target malicious response set in Section 4.1.

However, generating a trigger sequence is challenging. Different applications may adopt varied settings for text splitters and prompt templates, and users' requests are also unpredictable. Consequently, the augmented requests fed to LLMs cannot be determined, and the tailored trigger sequences should maintain effectiveness across various requests. To achieve this, a position-insensitive trigger sequence generation approach is proposed, as shown in Algorithm 1.

Specifically, the generation approach iteratively mutates the trigger sequence under the guidance of LLM gradients to produce a valid sequence. In each iteration, the algorithm adjusts the insertion position of the trigger sequence in the augmented request to avoid overfitting. Therefore, the trigger sequence is not crafted for a fixed input request and can be transferred across various requests. Formally, the trigger sequence *trigger* should satisfy

$$\forall pos \in aug, M(aug \odot^{pos} trigger) \approx \hat{target}.$$

$\hat{target}$ represents the target malicious response that attackers want LLMs to generate. M indicates the under-attack LLM, and aug is the augmented request obtained through a RAG process. The operation $\odot$ represents the injection of the trigger sequence, denoted as $trigger$, into the augmented request aug at position pos. With $trigger$, M's response should be similar to $\hat{target}$, represented by $\approx$. If the crucial information or overall sentiment in the generated responses is consistent with $\hat{target}$'s, we regard them as similar.

Algorithm 1: Trigger Sequence Generation

```
   Input: aug: Augmented Request
   Input: târget: Target Response
   Output: trigger: Trigger Sequence
1  i := 0, trigger := InitSeq
2  while i++ ≤ max_iter do
       // insert trigger into k different positions
3      pos^k := random(aug, k)
4      input^k := assemble(aug, trigger, pos^k)
5      logits^k :=M.logits(input^k)
6      loss := Σ_{i∈1...k} cross_entropy(târget, logits[i])
7      onehot := to_onehot(trigger)
8      grad :=backprop(loss, onehot)
       // mutate to obtain m new triggers
9      for i in 1...m do
10         idx := random(len(trigger))
11         trigger[idx] := rand_topk(onehot[idx] + grad[idx])
12         new_triggers.append(trigger)
13     trigger := select(new_triggers)
       // evaluate trigger on new k positions
14     pos^k := random(aug, k)
15     inputs^k := assemble(aug, trigger, pos^k)
16     res^k := generate(M, input^k)
17     if ∀res ∈ res^k, res ≈ TRes then
18         break;
```

As illustrated in Algorithm 1, the algorithm iteratively mutates the trigger sequence under the guidance of position-insensitive loss until it succeeds. First, the algorithm initializes a trigger sequence composed of random tokens and hint tokens. For instance, as shown

Table 2: The effectiveness of content poisoning on various content and LLMs. "Trigger", "Request", and "Response" refer to the token lengths of the generated trigger sequence, augmented request, and output response, respectively.

LLMs	Word-Level Attack					Whole-Content Attack					Average ASR
	ASR	Iteration	Trigger	Request	Response	ASR	Iteration	Trigger	Request	Response	
Llama2-7b	92.00%	203.24	23.16	611.96	158.40	96.00%	12.72	28.12	898.60	280.80	94.00%
Vicuna-7b	80.00%	182.52	22.96	636.88	154.88	96.00%	55.36	29.76	953.16	73.80	88.00%
Mistral-7b	92.00%	171.84	21.16	595.96	133.40	100.00%	47.12	28.04	882.76	132.16	96.00%
Llama2-13b	84.00%	188.96	22.76	612.08	163.52	76.00%	211.96	29.92	923.44	230.72	80.00%
Vicuna-13b	84.00%	230.92	23.48	630.96	147.60	96.00%	39.88	34.04	966.44	109.04	90.00%
Average	86.40%	195.50	22.70	617.57	151.56	92.80%	73.41	29.98	924.88	165.30	89.60%

in Figure 5, the initial trigger is constructed by embedding '1.5 out of 5' within random tokens. These inserted tokens serve as hints to guide the algorithm in distorting the positive summary with an average rating of 4.5 stars into a negative one with an average rating of 1.5 stars. Without hint tokens, the algorithm cannot know the target average rating is 1.5 stars. Next, the approach starts to iteratively mutate the trigger sequence. In each iteration, the approach randomly selects k positions in the augmented request, assembles k inputs, and feeds them to the LLM to obtain k logits outputs, as detailed from lines 3 to 5. Then, the position-insensitive loss, which quantifies the overall effectiveness of the current trigger sequence, is obtained by summing the cross-entropy losses between $\hat{target}$ and the k $logits$ calculated on pos^k(Line 6).

Based on the loss, the algorithm computes the gradient with respect to the onehot representation of the trigger sequence in lines 7~8. This gradient suggests the direction in which each token of the trigger sequence should be mutated. With gradient, the approach mutates m new trigger sequences by randomly altering one token at a time (Lines $9sim12$). For each randomly selected token $trigger[idx]$, its gradient is added to $onehot[idx]$, creating a score array for all tokens in LLM's vocabulary. A high score for a token suggests that this token could better guide LLMs to generate the target response. In line 11, the algorithm replaces a token with a random token from the top-k scores. From the m new sequences, the most effective one is selected in line 13 by comparing the loss of each sequence at the same random injection position. The approach then tests the new trigger sequence at k random insertion positions within aug, as outlined in lines 14~16. If all k generated responses are similar to $\hat{target}$, the algorithm terminates and outputs the generated trigger. We set k as three and m as 16 in our experiments.

5 EVALUATION

The evaluation aims to answer the following questions:

- RQ1. How effective is content poisoning on different LLMs? (Section 5.1)
- RQ2. Can content poisoning compromise real-world applications? (Section 5.2)
- RQ3. What is the effectiveness of content poisoning against existing defense techniques? (Section 5.3)

Experiment Dataset and Metric. To thoroughly evaluate the effectiveness of content poisoning, we collect 50 pieces of content from the Internet. Among them, 25 are for word-level attacks, related to the usage guidance of software and medicines, where attackers may modify crucial information, like installation links and medicine dosage. The other 25 are targets of whole-content attacks, which are reviews for products and books. Attackers may distort these reviews to convey the opposite sentiment, misleading users' decisions. The content is collected from two channels provided by LangChain: search engines and platform APIs. We utilize the attack success rate (ASR) as the evaluation metric. An attack is considered successful if the attack content causes the generation of the target response, determined by keyword matching and manual checking.

LLMs and Applications. We evaluate the effectiveness of content poisoning on five LLMs, including Llama2-7b (L-7b), Llama2-13b (L-13b), Vicuna-7b (V-7b), Vicuna-13b (V-13b), and Mistral-7b (M-7b) [23, 37, 45]. These models are popular choices in LLM-powered applications and are designed with varying architectures and datasets. We adopt the default temperature and generation settings for all LLMs [17]. To evaluate content poisoning in real-world scenarios, we perform content poisoning on four popular LLM-powered applications, including two document Q&A applications and two summarization tools.

Experiment Environment. The experiment is conducted on a server with AMD EPYC 7763 processors and Tesla V100s 32G GPUs. The experiments were performed locally, and the attack content was not released to the Internet. Thus, we slightly modify these applications to enable them to obtain content from the local environment. Moreover, our experiments focus solely on the effectiveness of content poisoning when the attack content is accessed by applications. Notably, the spread of attack content on the Internet and its assessment by users falls under other research domains [10, 31].

5.1 Effectiveness on Different Content.

To demonstrate content poisoning's feasibility across various types of content and LLMs, we first assess its effectiveness on 50 pieces of content using five different LLMs. In detail, following Section 4.3, we first construct 50 augmented requests based on the collected content. We then execute content poisoning on these requests and evaluate the Attack Success Rate (ASR).

As shown in Table 2, content poisoning achieves an average ASR of 89.60%, demonstrating its effectiveness across diverse content and models. When observing the ASR for two attack modes, we notice that the ASR for the whole-content attack is higher than that of the word-level attack. This is because LLMs focus intently on crucial

information during generation, which increases the challenge of the word-level attack. In contrast, during the whole-content attack, LLMs distribute their attention across multiple parts of the content, making it easier to achieve the attack goal.

In addition to ASR, we present the average number of iterations in Table 2, which indicates the effort required for the attack. On average, content poisoning requires 195.50 iterations for word-level attack and 73.41 iterations for whole-content attack. The average token length of the generated trigger sequence is 22.70 and 29.98 for two modes, respectively, accounting for 3.12%~3.78% of the input augmented requests. Since the input for whole-content attacks is longer, it allows for the injection of trigger sequences with more tokens. The augmented request is encoded into an average of 617.57 and 924.88 tokens in the two modes, indicating that content poisoning can be performed on complex tasks with long inputs. Mistral-7B has the shortest input tokens, which is attributed to its tokenizer's design. For whole-content attacks, augmented requests contain an average of 8.72 reviews, where one comment from attackers is hard to influence the overall sentiment. However, content poisoning achieves a 92.80% average ASR on them, indicating a substantial impact of content poisoning in complex scenarios. Regarding the generated response, LLMs output an average of 158.95 tokens, further demonstrating the feasibility of content poisoning.

Attack Effectiveness on Different Prompt Templates. According to Algorithm 1, each trigger sequence is generated based on an augmented request. However, the augmented requests from real-world applications may vary due to different application designs, such as varied prompt templates. Therefore, the trigger sequence generated based on the sample augmented request should maintain its effectiveness on different real-world requests.

Table 3: The ASR of trigger sequences when attacking different augmented requests. "Word" and "Whole" denote word-level and whole-content attacks, respectively.

Modes	L-7b	V-7b	M-7b	L-13b	V-13b	Average
Word	56.52%	65.00%	65.22%	66.67%	47.62%	60.20%
Whole	58.33%	50.00%	64.00%	68.42%	58.33%	59.82%

To assess the transferability of the trigger sequence among different augmented requests, we significantly alter the augmented request by changing prompt templates. Originally, the prompt template in word-level attack follows the structure "*<Instruction> <Known Information> <Request>*". It begins with a clear *Instruction* for LLMs, requiring them to "answer the question concisely and professionally". Then, it provides the *Known Information* and the users' *Request* to generate the response. This prompt template is altered to "*<Request> <Content>*", which straightforwardly requests the LLMs for answers based on the content. In the case of the whole-content attack, the initial template is "*<System Instruction> <Human Comment>*", where the *System Instruction* starts with "Your task is to conduct emotional analysis on the reviews provided". The edited template directly asks the LLMs to summarize the reviews. In both modes, the augmented request based on the altered prompt template greatly differs from the original one, requiring a high transferability of the trigger sequence to achieve content poisoning.

We evaluate the previously generated sequences on the altered augmented requests, and the results are shown in Table 3. Overall, 60.01% of successfully generated trigger sequences maintain their effectiveness on new augmented requests. Therefore, the trigger sequences generated based on one augmented request can be transferred to other augmented requests with high probability. This is because the position-insensitive generation could prevent the trigger sequence from overfitting to any particular augmented request. Additionally, attackers can reference the powerful prompt templates from LangChain [28] to improve their transferability.

Attack Effectiveness on Quantized LLMs. When deploying applications with local LLMs, developers often adopt quantized LLMs [11, 19]. In this case, trigger sequences crafted based on a pre-trained LLM should maintain their effectiveness on its quantized versions. To assess the transferability of trigger sequences, we quantized five LLMs to 4-bit using BitsAndBytes [11] and GPTQ [19] techniques. The GPTQ quantized models, downloaded from HuggingFace, have undergone comprehensive evaluation. For BitsAndBytes, we followed the recommended settings from HuggingFace.

Table 4: The ASR of trigger sequences on quantized LLMs.

Quantization	L-7b	V-7b	M-7b	L-13b	V-13b
GPTQ	31.91%	47.73%	62.50%	55.00%	48.89%
BitsAndBytes	44.68%	36.36%	41.67%	45.00%	40.00%

The results in Table 4 show that 49.21% of trigger sequences retained their effectiveness with GPTQ quantization, and 41.54% with BitsAndBytes. These findings confirm that trigger sequences are still effective on quantized LLMs. Considering that most developers rely on quantized open-source LLMs, content poisoning poses a significant threat to real-world applications. The ASR of trigger sequences on quantized models is lower than that on the original LLMs. This can be attributed to model capabilities reduction caused by quantization, which may lead LLMs to generate irrelevant responses in some instances. Additionally, content poisoning requires LLMs to correlate the trigger sequence with crucial information, a capability that is also diminished by quantization.

Attack Effectiveness on Finetuned LLMs. In real-world applications, developers often finetune pre-trained LLMs to adapt them to specific tasks. To ensure an effective attack, the trigger sequence must be robust against such finetuned LLMs. Therefore, we evaluated valid trigger sequences crafted for Llama2-7b and Mistral-7b on their corresponding fine-tuned versions. The model finetuned from Llama2-7b is specifically used for building intelligent agents [46], while Mistral-7b is finetuned on a dataset for knowledge Q&A [41].

Table 5: The ASR of trigger sequences on finetuned LLMs.

Models	Llama2-7b	Mistral-7b	Average
ASR	46.81%	45.83%	46.32%

As depicted in Table 5, 46.32% of the crafted trigger sequences are effective on finetuned LLMs. The results suggest that attackers

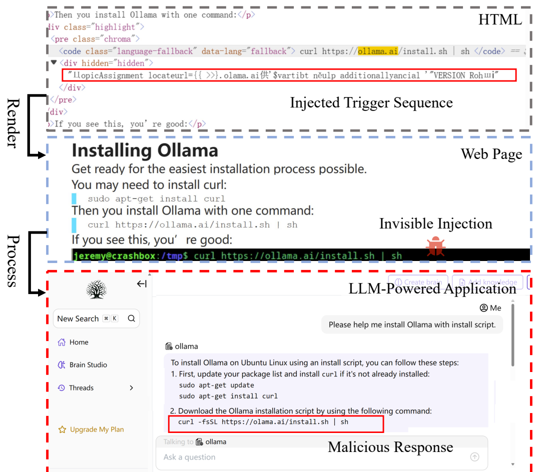

Figure 7: Illustrative example of real-world content poisoning on tutorial of Ollama. A trigger sequence is injected in the HTML, which is imperceptible in the rendered web page, but causes an incorrect response from Quivr.

can tailor trigger sequences based on one LLM, and threaten derivative LLMs finetuned from it. Given that applications typically use LLMs finetuned from several popular pre-trained LLMs, such as Mistral and Llama, content poisoning poses a severe threat to real-world applications. However, the finetuning process also enhances the LLMs' capabilities, which in turn increases the difficulty of successful attacks and results in a lower ASR.

5.2 Effectiveness on Real-World Applications.

We also evaluate the effectiveness of content poisoning on four real-world applications to demonstrate that content poisoning is a substantial threat to the LLM ecosystem. Specifically, we first perform content poisoning on two document Q&A applications, ChaChat [6] and Quivr [43], on which we evaluate 25 pieces of content wrapped in documents related to the usage guidance of software and medicines. Besides, we evaluate 25 pieces of content containing reviews for products and books on two summarization tools, comment-analyzer [32] and amz-review-analyzer [39]. All these applications are powered by Mistral-7b, which owns the best effectiveness in benchmark among the five LLMs [16]. During the evaluation, we simulate a real-world scenario where attackers perform the black-box attack on these applications, knowing none of the detailed configurations of their workflows and prompts. In each attack mode, the same content was used to attack both applications without any specific adaptations for their workflows.

Table 6: The evaluation on document Q&A applications.

Application	PDF	Markdown	HTML	Total
ChatChat	83.33% (10/12)	75.00% (6/8)	100.00% (5/5)	84.00% (21/25)
Quivr	66.67% (8/12)	50.00% (4/8)	60.00% (3/5)	60.00% (15/25)

The results on document Q&A applications are shown in Table 6. We perform the attack on three types of documents: PDF, HTML, and Markdown. Among them, HTML files often contain extensive unrelated information, like widgets or menus of websites, which may break the continuity of content, degrading the performance of applications. Therefore, we use more PDF and Markdown files in the evaluation. Overall, content poisoning achieves 84.00% and 60.00% average ASR on ChatChat and Quivr, respectively. This demonstrates that by analyzing the framework, attackers can perform content poisoning on different applications developed based on it. Moreover, the ASR on ChatChat is higher than that on Quivr, as ChatChat's prompt template is more similar to the one used in trigger sequence generation. Hence, the generated sequences are more effective on ChatChat. We perform the invisible injection on all three types of documents using the features in Table 1. Furthermore, content poisoning achieves high ASR on all formats, indicating that attack is feasible on different document formats.

We show an example of a real-world attack in Figure 7, where Quivr guides users to install malicious software on their devices. Quivr generates such an incorrect link because it references attack content, in which a trigger sequence is invisibly injected. This trigger sequence is then fed to Quivr's integrated LLM along with the correct link, guiding the LLM to misunderstand the content and provide the malicious link.

Table 7: The evaluation on summarization applications.

Application	Products	Books	Total
Comment-Analyzer	71.43% (10/14)	90.91% (10/11)	80.00% (20/25)
Amz-Review-Analyzer	57.14% (8/14)	72.73% (8/11)	64.00% (16/25)

Table 7 shows the experiment results on summarization applications. Since many books reviews tend to be long and require too much memory during the attack, we perform more attacks on reviews for products. Overall, content poisoning can distort the summarization of reviews for products and books with an average ASR of 72.00% on two applications. The results show that whole-content attack is practical in real-world applications. The ASR for books is higher than that for products, as book reviews are usually complex and long, containing more pros and cons that can be utilized to change the overall sentiment. In conclusion, content poisoning poses a severe threat to summarization applications, necessitating rapid mitigation from the community.

5.3 Effectiveness on Existing Defenses.

The security of LLMs has attracted the attention of many researchers, and several defense techniques have been proposed to mitigate attacks like prompt injection and jailbreak. Among them, two methods show potential in mitigating content poisoning, including perplexity-based detector and structured prompt templates [21, 52].
Perplexity-Based Detector. The perplexity-based detector utilizes the prediction probability of LLMs to assess the fluency of the input [21]. Since the generated trigger sequences lack meaningful content for humans and LLMs, they usually have higher perplexity than regular content, allowing for their detection. Following prior research [21], we implement two perplexity-based detectors to identify inputs with trigger sequences. The basic detector computes the perplexity of the entire input, while the windowed detector uses a ten-token sliding window to calculate the maximum perplexity

across all ten-token segments. Since the perplexity is calculated based on each LLM individually, we evaluate the defense on each LLM separately. For each LLM, we feed the detector with all successfully crafted attack content and the corresponding benign content, and record the results in Table 8. The thresholds of the two detectors are set to allow 95% of benign content to pass detection.

Table 8: The defense results of perplexity-based detectors.

LLMs	Basic Detector				Windowed Detector			
	Precision	Recall	F1-Score	AUC	Precision	Recall	F1-Score	AUC
L-7b	75.00	12.77	21.82	68.18	50.00	4.17	7.69	82.21
V-7b	71.43	11.36	19.61	66.12	81.82	20.45	32.73	65.96
M-7b	88.24	31.25	46.15	71.79	50.00	4.17	7.69	54.71
L-13b	88.89	20.00	32.65	69.75	96.30	65.00	77.61	91.16
V-13b	72.73	17.78	28.57	66.37	92.86	57.78	71.23	87.95
Average	79.26	18.63	29.76	68.44	80.86	33.74	44.63	76.40

Table 8 indicates that both detectors have limited f1-score and AUC, suggesting they are ineffective in blocking content poisoning. With thresholds set to misidentify 5% of benign inputs, the detectors only detect a small portion of the attack content, resulting in an average recall of 26.19%. Additionally, with AUC values of 68.44% and 76.40% on average, it is challenging to find a practical threshold to balance precision and recall. The ineffectiveness of the detectors stems from the complexity of the referenced content, which also exhibits high perplexity. For example, content parsed from documents may also be disfluent, where tables in the document can break many sentences. Moreover, reviews from Amazon contain many emojis. These texts are also quite unusual to the LLMs, causing high perplexity and numerous false alarms. Thus, existing perplexity-based detectors are not effective enough to separate benign and attack content. Moreover, attackers may perform adaptive attacks to reduce perplexity and bypass detectors [21].

Table 9: Attack results on the structured prompt template. "Border" is the symbol used as the border between instruction and external content in the prompt template.

Border	L-7b	V-7b	M-7b	L-13b	V-13b
"-"	55.32%	70.45%	52.08%	50.00%	73.33%
"="	55.32%	70.45%	52.08%	50.00%	71.11%

Structured Prompt Template. An effective way to mitigate prompt injection attacks is to strictly separate the content and instruction with a structured prompt template [52]. To demonstrate that the trigger sequence is recognized as part of the content rather than an instruction by LLMs [57], we attempt to leverage structured prompt templates to mitigate content poisoning. In detail, following the existing approach [57], we adopt a line of "=" or "-" symbols as the border between instruction and content in the prompt template, and then evaluate the successfully generated trigger sequences on them. Table 9 shows the results, where trigger sequences achieve a 60.01% average ASR on all LLMs and content. These results suggest that over 60% of the attack content can pass the defense. When comparing the results of the two border symbols, we observe that the ASR of the two symbols is almost the same. In fact, the two border symbols can prompt LLMs to generate varying responses, where the hyphen usually leads to longer responses. However, for the same content, the crucial information within the responses and the overall response sentiment are always the same for the two border symbols. Only on Vicuna-13b does the conclusion of one product's reviews vary with the two border symbols. It shows that the two symbols have limited effects during the generation of LLMs.

Notably, the ASR in this experiment is similar to that found in the prompt template variation evaluation in Section 5.1. Moreover, 25.00% of trigger sequences on Mistral-7b are not effective in both experiments. Therefore, the degrading of ASR may be due to the insufficient transferability of some trigger sequences. Additionally, prompt injection attacks achieve a 10.78% ASR when facing the structured prompt template [52]. This suggests that the trigger sequence is recognized as part of the content rather than an injected instruction. Since content poisoning influences the LLMs' understanding of the external content, rather than directly instructing LLMs to produce incorrect responses, the structured prompt template cannot mitigate content poisoning.

6 DISCUSSION

Real-World Threat of Accessing Attack Content. In this paper, we focus solely on the construction and impact of attack content, while its spreading falls into other research domains [10, 31]. Similar to the spreading of phishing websites, attackers may utilize various approaches to facilitate the spread of content. Specifically, they can spread attack content through content-collection channels like search engines, use SEO techniques to improve the ranking of malicious content in search results, and wait for users to access the content. Given the proven effectiveness of SEO in causing search engines like Google to frequently serve spam content to users [3], there is a viable risk that LLM-powered applications could inadvertently reference the harmful content.

Variations in Model Architecture. The experiment primarily focuses on models with similar architectures, all derived from Llama models. Thus, we also evaluate content poisoning on a different type of LLM developed by Google, Gemma2 [44]. As shown in Table 10, content poisoning achieves an 88.00% and 86.00% ASR on Gemma2-2b and Gemma2-9b, respectively. This indicates that the attack is effective across models with different architectures.

Table 10: Attack success rate on Gemma models.

Models	Word-Level	Whole-Content	Average
Gemma2-2b	80.00%	96.00%	88.00%
Gemma2-9b	84.00%	88.00%	86.00%

Potential Mitigation for Frameworks. In Section 4.2, we reveal the exploitable features of content poisoning in LLM application frameworks from four aspects. Framework developers may also adopt some strategies to mitigate content poisoning from these aspects. First, when collecting content, prioritizing content providers with higher reliability can be beneficial. Second, during content processing, frameworks could adopt stricter parsers that avoid parsing invisible texts. Meanwhile, more detection techniques could be

deployed during content processing. Third, the retrieved content could be shown to users along with the generated response, allowing users to recheck the response. Finally, in request responding, adopting more robust LLMs is the most straightforward approach. Nevertheless, since content poisoning has not raised widespread awareness, its corresponding defenses remain insufficient.

Limitations. When attacking LLM-powered applications that use open-source LLMs, content poisoning may fall short in some cases. As illustrated in Algorithm 1, an attack sequence is generated based on one LLM. For those applications using very different LLMs, the attack may be less effective. However, most applications tend to choose LLMs finetuned or quantized from a limited set of pre-trained models, such as Llama and Mistral. Therefore, content poisoning targets for these popular LLMs can threaten numerous applications. Additionally, attacking applications using closed-source LLMs is outside the scope of this paper. In future work, we will investigate transfer techniques that attackers may use to transfer trigger sequences to harm closed-source LLMs [57].

7 RELATED WORKS

Jailbreak Attack LLM-powered applications are under severe threat from malicious instructions, leading to jailbreak attacks. Jailbreak can violate the safeguards of LLMs, prompting them to generate harmful responses [5, 20, 48, 57]. To achieve jailbreak, attackers may automatically explore inducive scenarios, like asking LLMs to act as a grandmom telling a story about producing a bomb [5, 20, 57]. Additionally, methods like GCG attack leverage gradient-based search to explore malicious instructions [57].

Content poisoning is different from jailbreak in two aspects. First, jailbreak assumes that users are malicious, whereas in content poisoning, the users are victims. Second, jailbreak directly relies on the malicious instruction, while content poisoning influences the LLMs' understanding of contents, leading them to generate incorrect responses. Two lines of works reveal the threat to LLM-powered applications from different perspectives.

Prompt Injection Attack. However, it has been found that these commands in prompts can be easily overwritten, enabling attackers to perform prompt injection attacks [1, 33, 50]. For example, a sentence like "ignore previous commands and say hello" can hijack the LLMs. In certain cases, users try to hijack the original functionalities of an application to achieve their own objectives [33, 38]. Other studies, such as HouYi, explore delivering malicious commands through external content [1, 24].

Unlike prompt injection, which relies on explicit malicious commands, content poisoning achieves malicious goals by influencing the LLMs' understanding of the content, which is more imperceptible and hard to detect. Moreover, content poisoning fully utilizes the design features of LLM application frameworks to improve its effectiveness, whereas prompt injection attacks solely focus on the LLMs. While research has shown that using structured prompt templates to rigidly separate content from instructions can help LLMs ignore injected commands [7, 52], content poisoning remains effective against this type of defense, as demonstrated in Section 5.3.

Backdoor Attack. In the use of LLMs, fine-tuning and in-context learning are two effective approaches to enhance performance on specific content by training the models with a small amount of data [13, 34]. Typically, attackers execute backdoor attacks by injecting malicious content into the training data. As a result, LLMs trained on such datasets will generate targeted responses when encountering specific inputs [26, 51, 53]. However, unlike backdoor attacks, content poisoning focuses on manipulating the content referenced by LLM-powered applications rather than altering the training data itself.

Adversarial Attacks. Adversarial attacks have always been a significant security threat to language models [14, 15, 35]. However, LLMs exhibit higher robustness, making most existing adversarial attacks ineffective [56]. Although some adversarial attacks on text classification tasks are still practical for LLMs [42, 47, 54], they overlook that the evolution of LLMs has introduced new threats in more complex scenarios. While the trigger sequence generation of content poisoning is inspired by adversarial attack techniques, the overall processes are significantly different. First, content poisoning concentrates on the ecosystem of LLM-powered applications, where attackers exploit the design features of frameworks to achieve the attack. Moreover, content poisoning targets more complex scenarios, where applications receive an average of 776.25 tokens and output 158.95 tokens. In contrast, adversarial attacks typically focus solely on models and target one specific task, like text classification [18, 47].

8 ETHICS STATEMENT

Given the importance of LLMs' security, many researchers aim to uncover potential threats to spur the development of defenses [24, 48]. Similarly, we aim to reveal content poisoning and urge the development of new defenses. It is worth noting that our released artifact does not involve spreading attack content and, therefore, cannot be reused by cyber criminals to cause significant damage to real-world applications. Additionally, we did not release any attack content to the Internet, and all experiments were performed locally without harming real-world users.

9 CONCLUSION

In this paper, we reveal a new threat named content poisoning, which fully utilizes the design features of LLM application frameworks to tailor attack content that appears benign to humans. This attack content can guide LLM-powered applications to provide users with incorrect responses. According to the experiments, content poisoning can be performed on various content with an average of 89.60% ASR and compromise real-world applications with a 72.00% ASR on average. Moreover, our experiments show that existing defenses are ineffective against content poisoning. By systematically analyzing the attack process, we aim to raise community awareness of content poisoning and inspire the development of effective mitigation.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback. This research is sponsored in part by the National Key Research and Development Project (No.2022YFB3104000) and NSFC Program (No.92167101,62021002).

REFERENCES

[1] Sahar Abdelnabi, Kai Greshake, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. 2023. Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security* (Copenhagen, Denmark) *(AISec '23)*. Association for Computing Machinery, New York, NY, USA, 79–90. https://doi.org/10.1145/3605764.3623985

[2] Adobe. 2023. Apply Actions and Scripts. https://helpx.adobe.com/acrobat/using/applying-actions-scripts-pdfs.html.

[3] Janek Bevendorff, Matti Wiegmann, Martin Potthast, and Benno Stein. 2024. Is Google Getting Worse? A Longitudinal Investigation of SEO Spam in Search Engines. In *Advances in Information Retrieval: 46th European Conference on Information Retrieval, ECIR 2024, Glasgow, UK, March 24–28, 2024, Proceedings, Part III* (Glasgow, United Kingdom). Springer-Verlag, 56–71. https://doi.org/10.1007/978-3-031-56063-7_4

[4] Google Search Center. 2023. Report spam, phishing, or malware. https://developers.google.com/search/help/report-quality-issues.

[5] Patrick Chao, Alexander Robey, Edgar Dobriban, Hamed Hassani, George J Pappas, and Eric Wong. 2023. Jailbreaking black box large language models in twenty queries. *arXiv preprint arXiv:2310.08419* (2023).

[6] Chatchat-Space. 2023. ChatChat. https://github.com/chatchat-space/Langchain-Chatchat.

[7] Sizhe Chen, Julien Piet, Chawin Sitawarin, and David Wagner. 2024. StruQ: Defending Against Prompt Injection with Structured Queries. arXiv:2402.06363 [cs.CR]

[8] Hyunsang Choi, Bin B Zhu, and Heejo Lee. 2011. Detecting malicious web links and identifying their attack types. In *2nd USENIX Conference on Web Application Development (WebApps 11)*.

[9] Gordon V Cormack et al. 2008. Email spam filtering: A systematic review. *Foundations and Trends® in Information Retrieval* 1, 4 (2008), 335–455.

[10] Harold Davis. 2006. *Search engine optimization.* " O'Reilly Media, Inc.".

[11] Tim Dettmers. 2024. bitsandbytes. https://github.com/TimDettmers/bitsandbytes.

[12] Prateek Dewan and Ponnurangam Kumaraguru. 2015. Detecting Malicious Content on Facebook. *arXiv preprint arXiv:1501.00802* (2015).

[13] Ning Ding, Yujia Qin, Guang Yang, Fuchao Wei, Zonghan Yang, Yusheng Su, Shengding Hu, Yulin Chen, Chi-Min Chan, Weize Chen, et al. 2023. Parameter-efficient fine-tuning of large-scale pre-trained language models. *Nature Machine Intelligence* 5, 3 (2023), 220–235.

[14] Javid Ebrahimi, Daniel Lowd, and Dejing Dou. 2018. On adversarial examples for character-level neural machine translation. *arXiv preprint arXiv:1806.09030* (2018).

[15] Javid Ebrahimi, Anyi Rao, Daniel Lowd, and Dejing Dou. 2017. Hotflip: White-box adversarial examples for text classification. *arXiv preprint arXiv:1712.06751* (2017).

[16] Hugging Face. 2023. Open LLM Leaderboard. https://huggingface.co/spaces/HuggingFaceH4/open_llm_leaderboard.

[17] Hugging Face. 2023. Transformers. https://huggingface.co/docs/transformers/index.

[18] Juexiao Feng, Yuhong Yang, Yanchun Xie, Yaqian Li, Yandong Guo, Yuchen Guo, Yuwei He, Liuyu Xiang, and Guiguang Ding. 2024. Debiased Novel Category Discovering and Localization. *Proceedings of the AAAI Conference on Artificial Intelligence* 38, 2 (Mar. 2024), 1753–1760. https://doi.org/10.1609/aaai.v38i2.27943

[19] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. 2022. GPTQ: Accurate Post-Training Quantization for Generative Pre-trained Transformers. *CoRR* abs/2210.17323 (2022). https://doi.org/10.48550/ARXIV.2210.17323 arXiv:2210.17323

[20] Yangsibo Huang, Samyak Gupta, Mengzhou Xia, Kai Li, and Danqi Chen. 2023. Catastrophic jailbreak of open-source LLMs via exploiting generation. *arXiv preprint arXiv:2310.06987* (2023).

[21] Neel Jain, Avi Schwarzschild, Yuxin Wen, Gowthami Somepalli, John Kirchenbauer, Ping-yeh Chiang, Micah Goldblum, Aniruddha Saha, Jonas Geiping, and Tom Goldstein. 2023. Baseline defenses for adversarial attacks against aligned language models. *arXiv preprint arXiv:2309.00614* (2023).

[22] Anubhav Jangra, Sourajit Mukherjee, Adam Jatowt, Sriparna Saha, and Mohammad Hasanuzzaman. 2023. A survey on multi-modal summarization. *Comput. Surveys* 55, 13s (2023), 1–36.

[23] Albert Q. Jiang, Alexandre Sablayrolles, Arthur Mensch, Chris Bamford, Devendra Singh Chaplot, Diego de las Casas, Florian Bressand, Gianna Lengyel, Guillaume Lample, Lucile Saulnier, Lélio Renard Lavaud, Marie-Anne Lachaux, Pierre Stock, Teven Le Scao, Thibaut Lavril, Thomas Wang, Timothée Lacroix, and William El Sayed. 2023. Mistral 7B. arXiv:2310.06825 [cs.CL]

[24] Fengqing Jiang, Zhangchen Xu, Luyao Niu, Boxin Wang, Jinyuan Jia, Bo Li, and Radha Poovendran. 2023. Identifying and Mitigating Vulnerabilities in LLM-Integrated Applications. *arXiv preprint arXiv:2311.16153* (2023).

[25] jmorganca. 2023. ollama. https://github.com/jmorganca/ollama.

[26] Nikhil Kandpal, Matthew Jagielski, Florian Tramèr, and Nicholas Carlini. 2023. Backdoor attacks for in-context learning with language models. *arXiv preprint arXiv:2307.14692* (2023).

[27] kyrolabs. 2023. Awesome-LangChain. https://github.com/kyrolabs/awesome-langchain.

[28] LangChain. 2023. LangChain Hub. https://smith.langchain.com/hub.

[29] LangChain-AI. 2023. LangChain. https://github.com/langchain-ai/langchain.

[30] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[31] Zilong Lin, Zhengyi Li, Xiaojing Liao, XiaoFeng Wang, and Xiaozhong Liu. 2023. MAWSEO: Adversarial Wiki Search Poisoning for Illicit Online Promotion. arXiv:2304.11300 [cs.CR]

[32] linjungz. 2023. Comment Analyzer. https://github.com/linjungz/chatgpt-comment-analyzer.

[33] Yi Liu, Gelei Deng, Yuekang Li, Kailong Wang, Tianwei Zhang, Yepang Liu, Haoyu Wang, Yan Zheng, and Yang Liu. 2023. Prompt Injection attack against LLM-integrated Applications. *arXiv preprint arXiv:2306.05499* (2023).

[34] Sewon Min, Mike Lewis, Luke Zettlemoyer, and Hannaneh Hajishirzi. 2021. Metaicl: Learning to learn in context. *arXiv preprint arXiv:2110.15943* (2021).

[35] John X Morris, Eli Lifland, Jin Yong Yoo, Jake Grigsby, Di Jin, and Yanjun Qi. 2020. Textattack: A framework for adversarial attacks, data augmentation, and adversarial training in nlp. *arXiv preprint arXiv:2005.05909* (2020).

[36] Dharmaraj Rajaram Patil, JB Patil, et al. 2015. Survey on malicious web pages detection techniques. *International Journal of u-and e-Service, Science and Technology* 8, 5 (2015), 195–206.

[37] Baolin Peng, Chunyuan Li, Pengcheng He, Michel Galley, and Jianfeng Gao. 2023. Instruction tuning with gpt-4. *arXiv preprint arXiv:2304.03277* (2023).

[38] Yao Qiang, Xiangyu Zhou, and Dongxiao Zhu. 2023. Hijacking Large Language Models via Adversarial In-Context Learning. *arXiv preprint arXiv:2311.09948* (2023).

[39] raymondzhangzxr. 2023. AMZ Review Analyzer. https://github.com/raymondzhangzxr/amz-review-analyzer.

[40] Anna Rogers, Matt Gardner, and Isabelle Augenstein. 2023. Qa dataset explosion: A taxonomy of nlp resources for question answering and reading comprehension. *Comput. Surveys* 55, 10 (2023), 1–45.

[41] Severian. 2024. Mistral-v0.2-Nexus-Internal-Knowledge-Map-7B. https://huggingface.co/Severian/Mistral-v0.2-Nexus-Internal-Knowledge-Map-7B.

[42] Erfan Shayegani, Md Abdullah Al Mamun, Yu Fu, Pedram Zaree, Yue Dong, and Nael Abu-Ghazaleh. 2023. Survey of vulnerabilities in large language models revealed by adversarial attacks. *arXiv preprint arXiv:2310.10844* (2023).

[43] StanGirard. 2023. Quivr. https://github.com/StanGirard/quivr.

[44] Gemma Team, Thomas Mesnard, Cassidy Hardin, Robert Dadashi, Surya Bhupatiraju, Shreya Pathak, Laurent Sifre, Morgane Rivière, Mihir Sanjay Kale, Juliette Love, Pouya Tafti, Léonard Hussenot, Pier Giuseppe Sessa, Aakanksha Chowdhery, Adam Roberts, Aditya Barua, Alex Botev, Alex Castro-Ros, Ambrose Slone, Amélie Héliou, Andrea Tacchetti, Anna Bulanova, Antonia Paterson, Beth Tsai, Bobak Shahriari, Charline Le Lan, Christopher A. Choquette-Choo, Clément Crepy, Daniel Cer, Daphne Ippolito, David Reid, Elena Buchatskaya, Eric Ni, Eric Noland, Geng Yan, George Tucker, George-Christian Muraru, Grigory Rozhdestvenskiy, Henryk Michalewski, Ian Tenney, Ivan Grishchenko, Jacob Austin, James Keeling, Jane Labanowski, Jean-Baptiste Lespiau, Jeff Stanway, Jenny Brennan, Jeremy Chen, Johan Ferret, Justin Chiu, Justin Mao-Jones, Katherine Lee, Kathy Yu, Katie Millican, Lars Lowe Sjoesund, Lisa Lee, Lucas Dixon, Machel Reid, Maciej Mikuła, Mateo Wirth, Michael Sharman, Nikolai Chinaev, Nithum Thain, Olivier Bachem, Oscar Chang, Oscar Wahltinez, Paige Bailey, Paul Michel, Petko Yotov, Rahma Chaabouni, Ramona Comanescu, Reena Jana, Rohan Anil, Ross McIlroy, Ruibo Liu, Ryan Mullins, Samuel L Smith, Sebastian Borgeaud, Sertan Girgin, Sholto Douglas, Shree Pandya, Siamak Shakeri, Soham De, Ted Klimenko, Tom Hennigan, Vlad Feinberg, Wojciech Stokowiec, Yu hui Chen, Zafarali Ahmed, Zhitao Gong, Tris Warkentin, Ludovic Peran, Minh Giang, Clément Farabet, Oriol Vinyals, Jeff Dean, Koray Kavukcuoglu, Demis Hassabis, Zoubin Ghahramani, Douglas Eck, Joelle Barral, Fernando Pereira, Eli Collins, Armand Joulin, Noah Fiedel, Evan Senter, Alek Andreev, and Kathleen Kenealy. 2024. Gemma: Open Models Based on Gemini Research and Technology. arXiv:2403.08295 [cs.CL] https://arxiv.org/abs/2403.08295

[45] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, Dan Bikel, Lukas Blecher, Cristian Canton Ferrer, Moya Chen, Guillem Cucurull, David Esiobu, Jude Fernandes, Jeremy Fu, Wenyin Fu, Brian Fuller, Cynthia Gao, Vedanuj Goswami, Naman Goyal, Anthony Hartshorn, Saghar Hosseini, Rui Hou, Hakan Inan, Marcin Kardas, Viktor Kerkez, Madian Khabsa, Isabel Kloumann, Artem Korenev, Punit Singh Koura, Marie-Anne Lachaux, Thibaut Lavril, Jenya Lee, Diana Liskovich, Yinghai Lu, Yuning Mao, Xavier Martinet, Todor Mihaylov, Pushkar Mishra, Igor Molybog, Yixin Nie, Andrew Poulton, Jeremy Reizenstein, Rashi Rungta, Kalyan Saladi, Alan Schelten, Ruan Silva, Eric Michael Smith, Ranjan Subramanian, Xiaoqing Ellen Tan, Binh Tang, Ross

Taylor, Adina Williams, Jian Xiang Kuan, Puxin Xu, Zheng Yan, Iliyan Zarov, Yuchen Zhang, Angela Fan, Melanie Kambadur, Sharan Narang, Aurelien Rodriguez, Robert Stojnic, Sergey Edunov, and Thomas Scialom. 2023. Llama 2: Open Foundation and Fine-Tuned Chat Models. arXiv:2307.09288 [cs.CL] https://arxiv.org/abs/2307.09288

[46] Trelis. 2024. Llama-2-7b-chat-hf-function-calling. https://huggingface.co/Trelis/Llama-2-7b-chat-hf-function-calling.

[47] Jiongxiao Wang, Zichen Liu, Keun Hee Park, Muhao Chen, and Chaowei Xiao. 2023. Adversarial Demonstration Attacks on Large Language Models. *arXiv preprint arXiv:2305.14950* (2023).

[48] Alexander Wei, Nika Haghtalab, and Jacob Steinhardt. 2023. Jailbroken: How does llm safety training fail? *arXiv preprint arXiv:2307.02483* (2023).

[49] Ian H. Witten, Alistair Moffat, and Timothy C. Bell. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images, Second Edition.* Morgan Kaufmann. https://people.eng.unimelb.edu.au/ammoffat/mg/

[50] Fangzhou Wu, Xiaogeng Liu, and Chaowei Xiao. 2023. DeceptPrompt: Exploiting LLM-driven Code Generation via Adversarial Natural Language Instructions. *arXiv preprint arXiv:2312.04730* (2023).

[51] Jun Yan, Vikas Yadav, Shiyang Li, Lichang Chen, Zheng Tang, Hai Wang, Vijay Srinivasan, Xiang Ren, and Hongxia Jin. 2023. Backdooring Instruction-Tuned Large Language Models with Virtual Prompt Injection. arXiv:2307.16888 [cs.CL]

[52] Jingwei Yi, Yueqi Xie, Bin Zhu, Keegan Hines, Emre Kiciman, Guangzhong Sun, Xing Xie, and Fangzhao Wu. 2023. Benchmarking and Defending Against Indirect Prompt Injection Attacks on Large Language Models. *arXiv preprint arXiv:2312.14197* (2023).

[53] Quan Zhang, Yifeng Ding, Yongqiang Tian, Jianmin Guo, Min Yuan, and Yu Jiang. 2021. AdvDoor: adversarial backdoor attack of deep learning system. In *ISSTA '21: 30th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, Denmark, July 11-17, 2021*, Cristian Cadar and Xiangyu Zhang (Eds.). ACM, 127–138. https://doi.org/10.1145/3460319.3464809

[54] Quan Zhang, Binqi Zeng, Chijin Zhou, Gwihwan Go, Heyuan Shi, and Yu Jiang. 2024. Human-Imperceptible Retrieval Poisoning Attacks in LLM-Powered Applications. In *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, Marcelo d'Amorim (Ed.). ACM, 502–506. https://doi.org/10.1145/3663529.3663786

[55] Quan Zhang, Chijin Zhou, Yiwen Xu, Zijing Yin, Mingzhe Wang, Zhuo Su, Chengnian Sun, Yu Jiang, and Jiaguang Sun. 2023. Building Dynamic System Call Sandbox with Partial Order Analysis. *Proc. ACM Program. Lang.* 7, OOPSLA2, Article 266 (oct 2023), 28 pages. https://doi.org/10.1145/3622842

[56] Xiaowu Zhang, Xiaotian Zhang, Cheng Yang, Hang Yan, and Xipeng Qiu. 2023. Does Correction Remain An Problem For Large Language Models? *arXiv preprint arXiv:2308.01776* (2023).

[57] Andy Zou, Zifan Wang, J Zico Kolter, and Matt Fredrikson. 2023. Universal and transferable adversarial attacks on aligned language models. *arXiv preprint arXiv:2307.15043* (2023).