HoneyLLM: Enabling Shell Honeypots with Large Language Models

Chongqi Guan, Guohong Cao, and Sencun Zhu
Department of Computer Science and Engineering
The Pennsylvania State University
{cxg5270,gxc27,sxz16}@psu.edu

***Abstract*—Large Language Models (LLMs) have shown significant potential across various domains, including cybersecurity. This paper introduces HoneyLLM, a novel approach to creating high-fidelity shell honeypots using LLMs. We first investigate the potential of different commercial LLMs to emulate shell environments, identifying their characteristics and key challenges in accuracy and consistency. To address these issues, we propose leveraging various prompt engineering techniques, including in-context learning to tackle accuracy-related issues and the chain-of-thought method to maintain response consistency across complex, multi-step attack sessions. Additionally, we design a hybrid architecture for HoneyLLM to handle real-world limitations and improve cost-effectiveness. Through comprehensive offline evaluations, we demonstrate that HoneyLLM can effectively emulate shell environments and handle complex attack scenarios. Our online deployment results show that HoneyLLM, particularly when powered by advanced models like GPT-4, significantly outperforms traditional honeypots in maintaining longer, more effective attack sessions.**

I. Introduction

Honeypots have long been essential tools for cybersecurity practitioners, providing valuable insights into the dynamic threat landscape. These decoy systems are designed to attract, engage and deceive attackers by simulating vulnerable services such as Telnet or SSH in a monitored and controlled environment. Shell honeypots, in particular, closely emulate the shell environment of real devices or operating systems, effectively engaging with attackers and capturing detailed attack behaviors.

Emulating the shell environment presents significant challenges due to its complex nature. Even a simple tool like busybox [1] supports hundreds of shell commands, each with unique input and output patterns. Previous research has addressed this challenge in two ways. The programmatic approach [2, 3] uses scripted methods to construct low-interaction shell honeypots. Although these methods can mimic basic system interactions, they often suffer from limited fidelity due to their simplistic response logic, making them less effective against sophisticated attackers. On the other hand, the real system approach [4, 5, 6] involves leveraging actual operating systems to create high-interaction shell honeypots. These approaches provide a more authentic shell environment but come with substantial deployment costs and potential security risks, particularly if the sandbox environment is compromised.

Recent advancements in *Large Language Models (LLMs)* have opened up new possibilities across various domains,

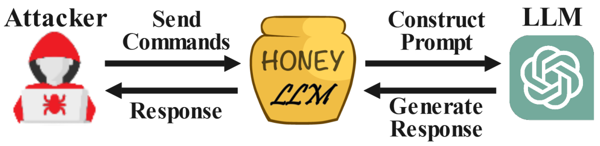


Fig. 1: Leveraging LLMs to emulate shell environment

including cybersecurity. These models, trained on vast amounts of Internet data, have demonstrated remarkable capabilities in understanding context, generating responses, and adapting to various tasks with minimal additional training.

In this paper, we explore the potential of LLMs to emulate shell environments and construct a new generation of shell honeypots, as shown in Figure 1. Our approach aims to maintain the realism of high-interaction honeypots while mitigating the security and privacy risks associated with exposing real systems to attackers. We begin by investigating the performance of state-of-the-art commercial LLMs in emulating shell environments. We collect attack traces to construct real-world attack scenarios in shell environments and then perform extensive experiments against vanilla LLMs using these attack scenarios. We identify the strengths of these LLMs and point out their limitations in terms of accuracy and consistency.

To enhance the capabilities of LLMs and address these limitations, we propose the use of various prompt engineering techniques. Specifically, we employ in-context learning to tackle accuracy-related issues and utilize the chain-of-thought method to maintain response consistency across complex, multi-step attack sessions. We then introduce HoneyLLM, a novel shell honeypot that leverages the power of LLMs to emulate shell environments with high fidelity. To address practical challenges of implementing LLM-based honeypots, such as query rate limitation and cost-effectiveness, we propose a hybrid architecture that combines an LLM-powered high interaction component with a traditional low-interaction shell honeypot.

Through comprehensive offline evaluations, we demonstrate that HoneyLLM can effectively emulate shell environments and handle complex attack scenarios. We then deployed HoneyLLM on the public internet to deal with real attackers. Our online deployment results show that HoneyLLM, particularly when powered by advanced models like GPT-4, significantly outperforms traditional honeypots in maintaining longer, more effective attack sessions.

The main contributions of this paper are as follows:

- We introduce HoneyLLM, a novel approach to create shell honeypots leveraging Large Language Models.
- We provide a comprehensive evaluation of different commercial LLMs' capabilities in emulating shell environments and point out the errors they tend to make.
- Various prompt engineering techniques are employed to construct robust prompts to mitigate these errors.
- We design a hybrid honeypot architecture to address various practical challenges when deploying HoneyLLM honeypots on the public internet.
- We evaluate the effectiveness of HoneyLLM through extensive offline experiments and online deployment.

II. Background

A. Shell Honeypot

Honeypots are decoy systems designed to attract, engage, and deceive potential attackers by mimicking vulnerable services such as Telnet or SSH within a monitored and controlled environment. Shell honeypots, in particular, closely emulate the shell environment of real devices or operating systems, allowing them to effectively engage with attackers and log malicious behavior. They are typically classified as low-interaction or high-interaction shell honeypots based on the fidelity of the system exposed to attackers and the privileges granted.

Low-interaction shell honeypots [2, 3, 4] typically employ conventional programming methodologies to emulate shell environments. These honeypots usually implement functions and services in the shell environment using simple reply logic and fake file systems. Due to their simplicity, low-interaction shell honeypots are relatively easy to deploy and maintain. However, they have limited fidelity and can be detected by human attackers or sophisticated reconnaissance tools.

High-interaction shell honeypots [5, 7, 6], on the other hand, provide a more authentic shell environment by running genuine operating systems at the backend. By exposing real systems to attackers, high-interaction shell honeypots can effectively engage sophisticated adversaries and collect complex attack traces. However, they also present drawbacks. First, they usually incur higher deployment and maintenance costs due to the real system running at the backend. Second, they pose greater security and privacy risks since attackers' behavior is not strictly restricted. If the shell honeypot is completely compromised, it may be used to attack other systems and devices in the real production environment.

In this work, we investigate the potential of leveraging LLMs to build shell honeypots. Our goal is to maintain high fidelity while mitigating the security and privacy risks associated with traditional real system-based honeypots.

B. Large Language Model

Large Language Models (LLMs) are primarily transformer-based models [8] trained on vast amounts of data collected from the internet. They typically leverage attention mechanisms to handle dependencies among words in data, allowing the model to understand context and meaning more effectively than previous architectures. The power of LLMs lies in their ability to generalize from the data they have been trained on, making them adaptable to various tasks with minimal additional training. Their capabilities extend beyond simple text generation to complex conversation, summarization, and reasoning tasks. LLMs have recently become a powerful and versatile technology applied to various fields, including cybersecurity [9, 10, 11, 12, 13, 14]. In this paper, we investigate several well-known, state-of-the-art commercial LLMs listed in Table I and test their ability to emulate shell environments.

TABLE I: LLMs used in this paper

Model	Params	Context Window Limit (tokens per session)	Query Rate Limit (tokens per minute)
GPT-3.5-turbo	unknown	16.4k	60k
GPT-4o	1.76T	128k	30k
Claude-2	130B	100k	25k
Claude-3 Haiku	20B	200k	25k
Claude-3 Opus	2T	200k	10k
Llama 2	70B	8k	N/A

GPT-3.5-turbo [15], first released in 2022, is one of the most prominent LLM. It is an enhanced version of OpenAI's Generative Pre-trained Transformer [16], offering improved performance in speed and efficiency while maintaining high accuracy and coherence in generated responses. GPT-3.5-turbo features a maximum context window of 16,384 tokens, which is sufficiently large for shell emulation (typically less than 10,000 tokens). OpenAI imposes a query rate limit, measured by the number of tokens consumed per minute, which varies based on user tiers. For example, a typical Tier 1 user, can process up to 60,000 tokens per minute with the GPT-3.5-turbo model.

GPT-4o [17] is the most advanced model released by OpenAI as of 2024. Compared to its predecessor, it not only offers improved efficiency but also supports vision input. It features a context window of up to 128k tokens and a typical Tier 1 user can process up to 30,000 tokens per minute.

Llama-2 [18], released in 2023, is a free LLM introduced by Meta. It has multiple versions with different parameter sizes aimed at various levels of accuracy and speed. In this work, we use the Llama2 with 70B parameters. Llama-2 has a context window of up to 8k tokens, and the query rate limit varies depending on the service provider or the hardware of the local machine.

Claude-2 [19], released in July 2023, was the first publicly available model introduced by Anthropic. It was quickly succeeded by a set of three advanced models in ascending order of capability: Claude-3 Haiku, Sonnet, and Opus in early 2024. Claude-3 Haiku is the smallest and fastest model, with a context window limit of 200k tokens. Similar to OpenAI, Anthropic also sets the query rate limit based on user tiers. A Tier 1 user can process up to 50k tokens per minute with Claude-3 Haiku. Claude-3 Opus is the largest model with over 2T parameters. Similarly, it has a context window limit of 200k tokens, and a 10k per minute query rate limit for a tier 1 user. Recently, Anthropic also releases the most state-of-the-art model, Claude 3.5 Sonnet. As the API for Claude 3.5 is not available at the time of writing this paper, we consider it as future work.

III. Vanilla LLM-based Shell Honeypot

LLMs are computational models trained on extensive datasets collected from the internet, enabling them to perform a wide range of tasks even in "zero-shot" settings [20]. Consequently, a straightforward method to create a shell honeypot using an LLM involves providing a basic prompt to the LLM to emulate a shell environment and directing the attacker's commands to the LLM. This approach leverages the LLM's training on comprehensive text data, which includes information about various shell environments.

To evaluate the fidelity of the responses generated by this vanilla LLM approach, we first conduct an offline evaluation. We collect a attack trace dataset consisting of shell commands from attackers and corresponding responses from real systems. These attack traces are used to simulate attack scenarios identical to real attack sessions. We then evaluate the fidelity of the responses generated by different LLMs in these simulated attack scenarios. The subsequent subsections detail the collection of the attack traces and the evaluation of the vanilla LLM-based shell honeypot.

A. Attack Trace Collection

To construct the attack scenarios for offline evaluation targeting shell environments, we begin by collecting attack traces. Our approach involves designing an attack trace collection system, as shown in Figure 2. The system consists of two main components: the frontend which directly interacts with attackers, and the backend which generates responses to attacker commands and saves interaction logs.

The frontend operates on a virtual machine that exposes telnet or SSH ports to attract potential attackers. When an attacker initiates an attack session by logging into the shell environment, the frontend captures their command inputs and forwards them to an input sanitizer. Valid commands are then forwarded to the backend, which transmits them to a running interactive shell environment. To provide more authentic experience, we utilize real operating systems (Ubuntu, Debian, etc.) and IoT device's SSH services to generate responses, which are then returned to the attackers. This interactive process continues until the attacker terminates the session. Regular resets of the operating systems and IoT devices are used to recover from attacks, and all attack session logs are stored in the database for further evaluation.

B. Understanding the Characteristics of Vanilla LLM

After collecting the attack trace dataset consisting of shell commands from attackers and corresponding responses from real systems, we use these traces to construct attack scenarios that are identical to real attack sessions. To perform an offline evaluation for the vanilla LLM-based honeypot, we instruct the LLMs to emulate a shell environment. We then sequentially input the shell commands from the attack session and compare the generated responses to the responses from real systems. We assess the fidelity of the responses from two main aspects.

First, the outputs of the LLM must comply with the general logic of the input shell command. For instance, for commands like "echo Hi | cat -n" that have a fixed output, the response generated by the LLM must be exactly the same as the response in the attack trace dataset. For shell commands like "uname -a" or "ifconfig", although the specific output may vary depending on the system configuration, network environment, and time, the general structure of the output should remain the same.

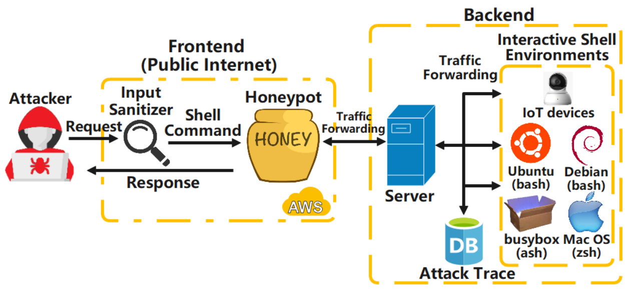


Fig. 2: Attack Trace Collection

Second, the outputs of the LLM must be contextually consistent in an attack session. Attack behaviors typically exhibit temporal dependencies and certain control logic, where the execution of commands relies on the effects of previous commands. Therefore, the same shell commands in different attack sessions may have completely different outputs. For instance, the expected response of an "ls" command will change dramatically when the attacker is in a different directory. Inconsistent responses from the LLM are likely to fail to induce further attacks, thereby reducing the overall fidelity of the honeypot and the deception capability of HoneyLLM.

Based on these two aspects, an accurate response should be both logically correct and contextually consistent. Using these evaluation metrics, we inspect and evaluate the outputs generated by different LLMs in each attack session. Figure 3 shows the accuracy of vanilla LLM-generated responses for different shell commands. On one hand, even with a basic prompt template, nearly all vanilla LLMs can effectively handle individual commands that have fixed outcomes or lack contextual dependencies. For example, commands like "chmod", "rm", "kill", and "cd" have predictable and simple outcomes, and nearly all models achieve 100% accuracy for these commands. Commands like "ifconfig", which have a relatively complex output but usually do not depend on other shell commands in the attack session, can also be effectively handled by state-of-the-art models like GPT-4, with an accuracy rate of 88%. On the other hand, most models struggle to generate accurate responses for commands that rely on the effects of previous commands or require additional information from the system or the internet.

The errors introduced by vanilla LLM can be roughly categorized into four types: Unknown Command, Incorrect Response Format, Clearly Incorrect Content, and Inconsistent Response.

Unknown Command errors typically occur when a LLM fails to identify a valid command, resulting in a "Command not found" response message. Listing 1 presents a representative example from our offline evaluation using GPT-3.5 turbo. In this instance, the model fails to generate the expected response for a "curl" command, potentially hindering an attacker's ability

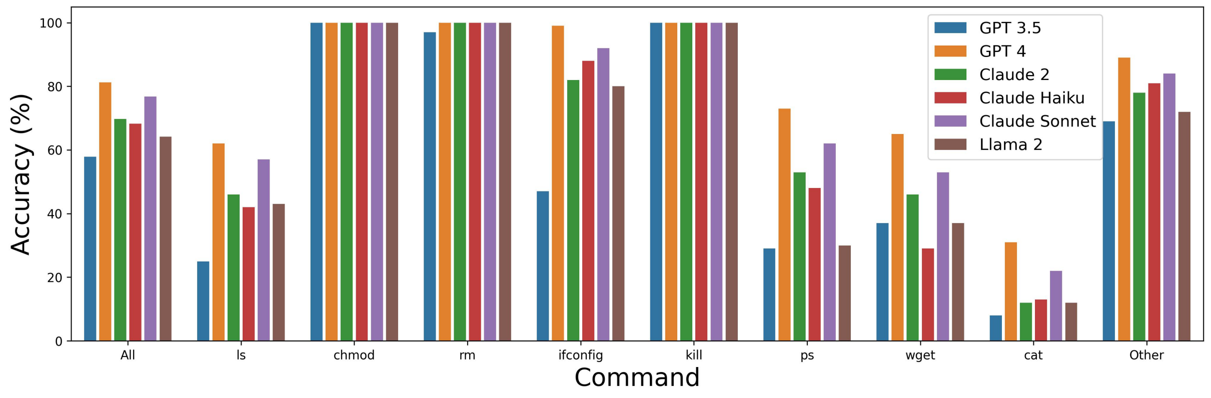


Fig. 3: Accuracy of vanilla LLM-based honeypot for different shell commands

to launch subsequent attacks. These errors are likely due to gaps in the model's knowledge base. They can be effectively mitigated through in-context learning [21], a technique we will explore in depth in Section IV-A.

```
curl -O http://5.181.80.168/bins.sh -O ....
bash: curl: command not found
~/tmp$
```

Listing 1: Unkown Command Example

Incorrect Response Format errors typically occur when specific flags are included with shell commands, and the model omits these flags, resulting in a generated response that deviates from the expected structure. This issue also can be effectively addressed through in-context learning techniques.

Clearly Incorrect Content errors primarily occur when the generated response contains inaccurate or artificial information. As shown in Listing 2, when instructed to emulate a Linux shell and execute an initial "ls" command, GPT-3.5 turbo may occasionally respond with a list of typical Windows directory names (e.g., "Desktop, Documents, Downloads, Music, Pictures, Public, Videos"). This response diverges significantly from the expected root directory structure of a Linux system, which should include entries such as "bin, dev, etc, home, proc, sys, tmp, usr, var". This issue can also be effectively addressed through in-context learning techniques.

```
ls
Desktop documents downloads music pictures public
~/tmp$
```

Listing 2: An example of Clearly Incorrect Content

Inconsistent Response is the most common mistake made by LLMs when emulating an SSH environment. It usually occurs in attack sessions where attackers execute a series of dependent commands. Listing 3 shows a typical attack session where an attacker uses "wget" to upload malware and then tries to execute it. In the LLM response, the model first indicates that the "wget" command and "chmod" commands were executed successfully, but later shows that the "Mozi.7" file does not exist. This example highlights the challenges most models face when handling complex, interdependent commands in extended attack sessions that require memorization across multiple commands. This issue can be effectively mitigated using a chain-of-thought strategy [22, 23], which will be addressed in Section IV-A.

```
cd ./tmp
~/tmp$
wget http://112.249.111.71:46403/Mozi.7
Connecting to 112.249.111.71:46403 .....
chmod 777 Mozi.7
~/tmp$
./Mozi.7
bash: ./Mozi.7: No such file or directory
~/tmp$
```

Listing 3: Inconsistent Response Example

IV. HoneyLLM: Enabling Shell honeypots with LLMs

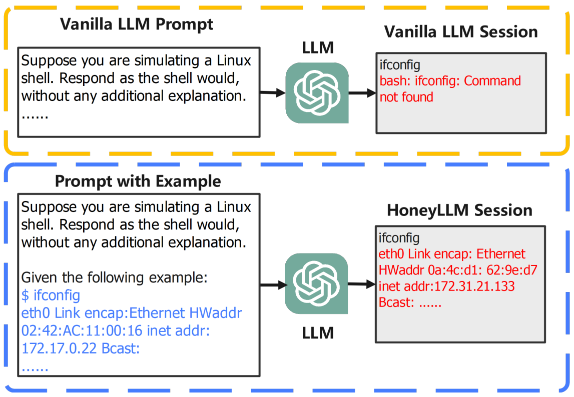
In this section, we first introduce various prompt engineering techniques that can effectively mitigate the errors observed in Vanilla LLM-based shell honeypots. We then present the system design of HoneyLLM and its approach to address real-world challenges.

A. *Prompt Engineering*

To address the errors observed in vanilla LLM-based shell honeypots, we explore two prompt engineering techniques:

1) In-context Learning: It is a powerful capability exhibited by LLMs, enabling them to adapt to new tasks and generate appropriate responses based solely on a few examples provided within the input prompt. Unlike traditional machine learning approaches that require extensive fine-tuning on task-specific datasets, in-context learning offers a quicker and more lightweight alternative. By leveraging the vast knowledge encoded in their pre-trained parameters, these models can effectively learn from just a handful of examples, paving the way for more efficient and versatile LLM-based systems.

HoneyLLM leverages in-context learning by providing the LLM with specific attack trace examples in the input prompt, enabling it to swiftly adapt and emulate a realistic shell environment. By supplying corresponding real system response examples, the model learns to reproduce accurate command outputs. Figure 4 illustrates this concept. When prompted to



```
Suppose you are simulating a Linux
shell. Respond as the shell would,
without any additional explanation.
......
```

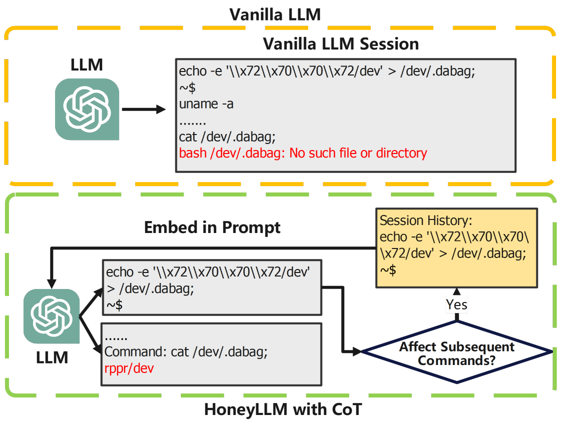
```
ifconfig
bash: ifconfig: Command
not found
```

```
Suppose you are simulating a Linux
shell. Respond as the shell would,
without any additional explanation.

Given the following example:
$ ifconfig
eth0 Link encap:Ethernet HWaddr
02:42:AC:11:00:16 inet addr:
172.17.0.22 Bcast:
......
```

```
ifconfig
eth0 Link encap: Ethernet
HWaddr 0a:4c:d1: 62:9e:d7
inet addr:172.31.21.133
Bcast: ......
```

Fig. 4: In-context learning example



```
echo -e '\\x72\\x70\\x70\\x72/dev' > /dev/.dabag;
~$
uname -a
.......
cat /dev/.dabag;
bash /dev/.dabag: No such file or directory
```

```
Session History:
echo -e '\\x72\\x70\\x70\
\x72/dev' > /dev/.dabag;
~$
```

```
echo -e '\\x72\\x70\\x70\\x72/dev'
> /dev/.dabag;
~$
```

```
......
Command: cat /dev/.dabag;
rppr/dev
```

Fig. 5: Chain of thought example

emulate a shell environment, a vanilla GPT-3.5 Turbo model may return a “command not found" result for the “ifconfig" command, failing to engage potential attackers effectively. This typical Unknown Command error can be mitigated using in-context learning techniques. In the “prompt with example" box in Figure 4, we first provide the model with real system responses in the attack trace dataset for the “ifconfig" command. Subsequently, when asked to generate a response, the model correctly follows the response structure, producing an authentic output as desired.

2) Chain of Thought (CoT): CoT involves prompting language models to generate step-by-step thoughts leading to their final answers, mimicking human problem-solving processes. By encouraging models to articulate intermediate reasoning steps, CoT significantly improves performance on multi-step tasks with complex dependencies.

Upon analyzing the collected attack traces, we observe that received attack command sequences often exhibit complex interdependencies. As discussed in previous section, Vanilla LLMs often struggle to retain the effects introduced by previous attack commands, resulting in inconsistent responses across the attack session. Figure 5 illustrates a typical attack trace that vanilla LLMs handle poorly. The attacker first uses “echo" to generate a “.dabag" file and then attempts to “cat" the file after a few interaction rounds. The vanilla GPT-3.5 Turbo quickly forgets the previous command it processed and returns a “No such

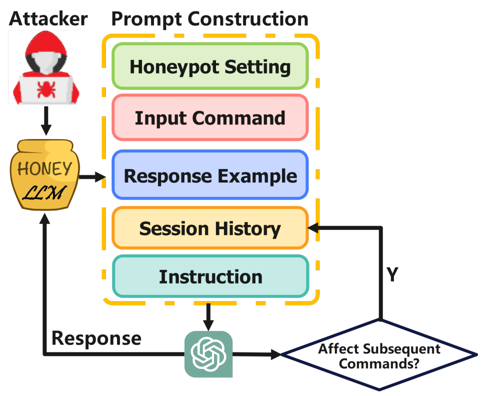


Fig. 6: Prompt construction

file or directory" error. This typical “Inconsistent Response" error can be mitigated using Chain of Thought techniques. In the “HoneyLLM with CoT" box in Figure 5, we instruct the LLM to evaluate the “echo ... > /dev/.dabag" command and determine whether it may influence subsequent commands in this attack session. As the GPT-3.5 Turbo model returns “Yes" we incorporate this command and its corresponding generated response into the session history. This updated history is then integrated into the prompt for follow-up commands, ensuring a more consistent and context-aware response generation process. Subsequently, when asked to generate a response for “cat /dev/.dabag” command, the model correctly prints the file content as desired.

B. Prompt Construction

Figure 6 illustrates the general design for a prompt that integrates the in-context learning and chain-of-thought techniques. Upon receiving a shell command from an attacker, we construct a prompt consisting of five components:

- **Honeypot Setting:** This component defines the configuration used to emulate a shell environment. It includes system information such as kernel version, username, time, and other relevant system details to ensure the emulation appears realistic to the attacker.
- **Input Command:** This is the shell command issued by the attacker to HoneyLLM in the current interaction round.
- **Response Example:** This component incorporates the in-context learning concept. To address cases where the LLM may struggle with certain commands, HoneyLLM searches the collected attack trace and selects appropriate response examples for reference.
- **Session History:** By implementing the Chain of Thought approach, this component ensures that the LLM maintains awareness of system state changes throughout the attack session. It dynamically records all commands that potentially influence subsequent responses. This session history is continuously updated and embedded into each new prompt, allowing the LLM to consider the cumulative effects of previous actions when generating responses.
- **Instruction:** This component defines general strategies and guidelines for emulating the SSH session. For ex-

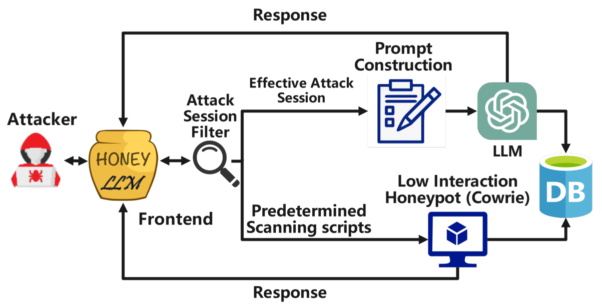


Fig. 7: HoneyLLM system overview

ample, we may add instructions such as “without any additional explanation" to avoid unnecessary explanations, or “Do not auto-correct misspelled commands" to maintain an authentic response to misspelled shell commands.

The constructed prompt is then sent to the commercial LLMs via APIs. In addition, we query the LLM to determine if the current command might potentially change the system state and influence the responses of subsequent commands. If the LLM indicates that a command may affect the system state, we will incorporate this command and its corresponding generated response into the session history for the next interaction round.

C. *System Design*

Designing HoneyLLM for real-world scenarios faces two practical constraints beyond fidelity: query rate limitations for commercial LLMs and cost-effectiveness. First, In real-world online deployment, HoneyLLM may receive a high volume of concurrent attacks that exceed the rate limitations of commercial LLMs. Additionally, previous research [24, 25, 26] indicates that when deploying honeypots to the public internet, most traffic consists of predetermined scanning scripts conducted by reconnaissance tools like Nmap [27]. These scripts typically follow fixed attack sequences and are indifferent to responses, making it inefficient to route them to LLMs.

To address these challenges, we employ a hybrid approach in designing HoneyLLM, as shown in Figure 7. HoneyLLM emulates vulnerable SSH servers by opening the SSH port on the frontend deployed on the public internet. Upon receiving an SSH connection from an attacker, we first filter out predetermined scanning script sessions. Analysis of collected attack traces reveals that these scripts usually have distinct fingerprints, allowing us to employ simple matching techniques for identification. These sessions are then redirected to Cowrie [3], an open-source, low-interaction honeypot that emulates a shell environment using Python scripts. For effective attack sessions, we leverage the aforementioned prompt engineering techniques to construct appropriate prompts and send them to commercial LLMs via APIs. The responses generated by LLMs are then returned to the attacker. This interactive process continues until the attacker terminates the session, with all attack session logs storing in backend database. In addition, to avoid query rate limitations, only one active attack session is processed by the LLM at a time, while all other concurrent attacks are redirected to the low-interaction honeypot.

Model	Effective Attack Session	Average Session Length
GPT-3.5 turbo	2672	4.37
GPT-4o	2763	5.83
Claude-3 Haiku	2797	5.36
Cowrie-Only	2386	2.96

TABLE II: Basic statistics in online deployment

V. Performance Evaluations

In this section, we assess the fidelity and efficacy of HoneyLLM through offline evaluations and online deployment.

A. *Offline Evaluation*

To evaluate the effectiveness of the aforementioned prompt engineering techniques employed in HoneyLLM, we conduct an offline evaluation leveraging the collected attack trace. We construct a total of fifty attack sessions using the attack trace dataset. For each attack session, we sequentially input the shell commands into HoneyLLM and analyze the generated responses. An attack session is considered successful if all generated responses are reasonable, accurate, and consistent. If any error occurs, the session is terminated and marked as a failure. For comparison, we also evaluate the vanilla LLM based HoneyLLM without these prompt engineering enhancements.

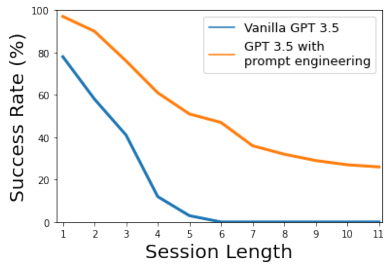
Figure 8 illustrates the success rates of HoneyLLM powered by different commercial LLMs across varying session lengths. The graphs consistently demonstrate that for all six models (GPT-3.5 turbo, GPT-4, Claude-2, Claude-3 Haiku, Claude-3 Opus, and Llama-2), the success rate improves dramatically when employing prompt engineering techniques. In particular, for GPT-3.5 turbo, Claude 2, Claude Haiku and Llama-2, prompt engineering techniques elevate the success rate of short attack session to nearly 100%. This improvement is primarily attributed to in-context learning, which effeively mitigates most “Unknown Command", “Incorrect Response Format" and “Clearly Incorrect Content" errors.

For longer attack sessions, HoneyLLM with all models shows significant improvement, likely due to the Chain-of-Thought technique. By embedding necessary interaction history into the prompt, we enhance the models’ ability to maintain context across multiple commands and thereby produce consistent responses. These evaluation results indicate that prompt engineering can effectively mitigate the errors encountered in vanilla LLM-based honeypot implementations.

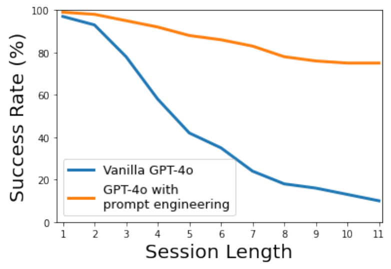
In addition, HoneyLLM with GPT-4o exhibits the highest overall success rate among the six models. With prompt engineering, it achieves an average of 90.2% success rate across all attack sessions, and maintains around 80% success rate even for the rare, complex, and extended attack sessions. This result underscores the enhanced capability of HoneyLLM in accurately emulating shell environments.

B. *Online Deployment*

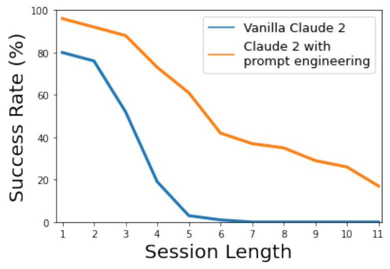
To evaluate the effectiveness of HoneyLLMs in deceiving real-world attackers, we deployed HoneyLLM powered by different commercial LLMs on the public internet. For comparison, we also deployed a separate Cowrie-only honeypot instance. In the evaluation, for HoneyLLM, only attack sessions



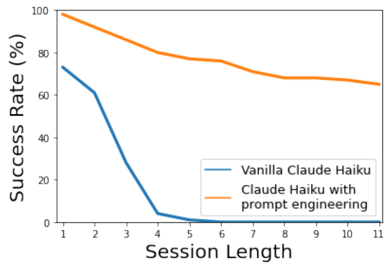
(a) GPT-3.5 turbo



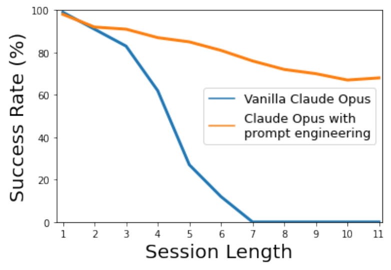
(b) GPT-4o



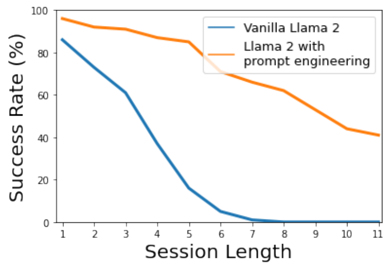
(c) Claude-2



(d) Claude-3 Haiku



(e) Claude-3 Opus



(f) Llama-2

Fig. 8: Success rate with respect to session lengths (i.e., number of interaction rounds in an attack session)

processed by LLMs were considered effective. For the Cowrie-only honeypot, we used the aforementioned filtering mechanism to filter out scripted attacks and counted the rest as effective attack sessions.

1) Basic Statistics: We implemented the hybrid structure, filtering mechanism, and rerouting rules, and then deployed three HoneyLLM instances (powered by GPT-3.5 Turbo, GPT-4, and Claude 3 Haiku) on the public internet for a three-week period in May 2024. Analysis of the collected honeypot logs revealed that the majority of traffic consisted of automatic scanning scripts with obvious fingerprints, which were filtered out by our mechanism. As shown in Table II, using HoneyLLM with GPT-3.5 Turbo as an example, only 2,672 attack sessions (approximately 9.8% of total attack sessions) were unique and considered effective.

For the effective attack sessions, we calculated the average session length by counting the number of interaction rounds. HoneyLLM powered by GPT-4o, Claude 3 Haiku, and GPT-3.5 Turbo had average session lengths of 5.83, 5.36, and 4.37 rounds, respectively, while Cowrie-only had an average session length of 2.96 rounds. GPT-4o clearly outperformed other models and Cowire-only in the real deployment phase. This is likely because attackers tend to perform various pre-attack checks in the reconnaissance stage, and GPT-4 is more effective at bypassing these checks compared to other models.

2) Effectiveness of Our Hybrid Architecture: Since many commercial LLMs have limitations on the query rate, typically measured by the number of tokens consumed, we have designed the hybrid architecture for HoneyLLM. In this subsection, we evaluate the effectiveness of our design and demonstrate the importance of our filtering mechanism in meeting these query rate limitations. Figure 9 illustrates the query rate, measured by the number of tokens consumed, from 17:00 to 19:00 on May 27, 2024. Using HoneyLLM with GPT-3.5 Turbo as an example, the green line represents the query limit for a typical Tier 1 user. The blue line indicates token consumption by effective attack sessions processed by GPT-3.5 Turbo. The red line shows the estimated token consumption for attacks filtered to the low-interaction honeypot, Cowrie. As attack sessions forwarded to Cowrie are not tokenized, we estimate the token consumption based on the number of tokens that would be consumed by LLMs. For all three models, the blue line remains significantly below the green line, demonstrating that HoneyLLM, due to its hybrid architecture, operates well within the query limit. On the other hand, without our hybrid architecture, HoneyLLM would not be able to handle these attacks due to the query rate limitations of the LLMs.

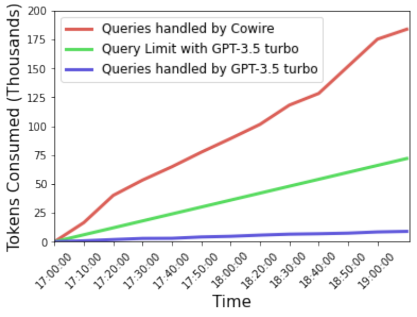
3) Geographic Location Distribution of Attacks: We analyzed the geographic locations of attacks in the effective attack sessions based on their IP addresses. Figure 10 shows the distribution of attackers by their countries of origin. Using HoneyLLM with GPT-3.5 Turbo as an example, 21.4%, 11.6%, and 13% of the attackers had IP addresses from the United States, China, and Russia, respectively. India, Germany, and other countries made up the remaining list. Notably, the largest proportion of attackers were from the United States, likely because HoneyLLM's IP was located in the United States, and some attackers tend to perform area-specific scans first.

VI. Discussions on Limitations and Future Work

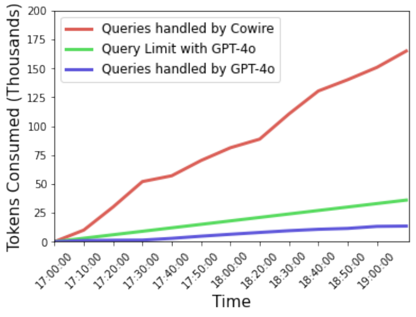
In this section, we discuss the limitations of HoneyLLM observed during offline evaluations and online deployment, and suggest directions for future improvements.

A. *Cost & Query Rate Limitations*

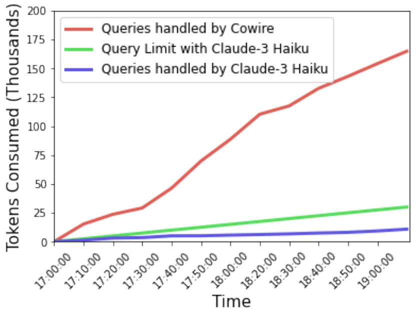
As shown in Table I, the query costs for commercial LLMs are relatively high, especially considering the large volume



(a) GPT-3.5 turbo



(b) GPT-4o



(c) Claude-3 Haiku

Fig. 9: Tokens consumed from 17:00 to 19:00 on May 27, 2024 (green line represents the query limit of the LLM).

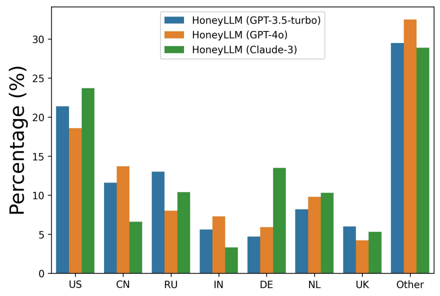


Fig. 10: Attacker distribution based on country of origin

of attacks typically received by honeypots deployed on the public internet. Furthermore, most commercial LLMs impose restrictions on query rates, limiting the honeypots' ability to handle concurrent attacks effectively.

To address these challenges and optimize cost-effectiveness, a hybrid honeypot architecture (such as HoneyLLM) consisting of both low-interaction and high-interaction component is necessary. In the future, we intend to further investigate and refine this hybrid architecture. For instance, we can route simple commands with predictable outcomes to low-interaction honeypots to reduce costs. For complex, context-dependent commands, we can leverage LLMs as the high-interaction component. By embedding session logs from the low-interaction component into the LLM prompts, we can maintain context without synchronization issues. This approach could balance the need for realistic, adaptive responses with the practical constraints of deployment costs and query limitations, resulting in a more efficient and effective honeypot system.

B. *Prompt Injection attacks*

A prompt injection attack [28, 29, 30, 31] is a technique used to manipulate or exploit LLMs by inserting specific instructions or content into the input prompt. In LLM-based honeypots, where an attacker's shell command is embedded into the prompt and forwarded to the LLM, prompt injection attacks may occur when attackers embed malicious instructions into the shell command. This can potentially cause the model to behave in unintended ways, reveal its honeypot nature, or even disclose sensitive information such as the specific model being used.

To mitigate this issue, we plan to design an input sanitizing mechanism to detect shell commands that could potentially mislead the Large Language Model. In addition, we plan to leverage various prompt engineering techniques to further mitigate the risk of prompt injection attacks in the future.

VII. Related Work

Research and open-source projects on honeypots [3, 4, 5, 6, 25, 32, 33] have attracted considerable attention in recent years. Cowrie [3], a widely used and well-maintained medium-interaction Shell honeypot, leverages a programmatic approach. It emulates a Unix system using Python scripts and logs all attack traffic and shell interactions performed by attackers. IoTPot [4] takes a real-system approach, emulating the Telnet and SSH services of IoT devices by forwarding attacker commands to a sandboxed Linux operating system at the backend. Similarly, VPNhoneypot [5] and IoTCMal [6] construct hybrid shell honeypots using a Virtual private network (VPN). These systems employ a low-interaction honeypot at the frontend to handle login attempts and utilize real devices or operating systems at the backend to manage sophisticated SSH sessions. The frontend honeypot and backend operating system are connected through VPN.

Recently, researchers have begun to apply LLM technology to the honeypot domain. For example, GPT-2C [34] uses a fine-tuned LLM model to parse logs generated by the Cowrie honeypot in real time, providing runtime threat analysis to assist intrusion detection systems. Boffa *et al.* [11] employ various models to process honeypot logs offline, clustering different types of cyber attacks against SSH services based on their attack patterns. They aim to speculate on the attack objectives to support various security activities.

VIII. Conclusions

In this paper, we investigated the feasibility of utilizing Large Language Models (LLMs) to enable shell honeypots. We designed HoneyLLM, a hybrid shell honeypot that leverages various commercial LLMs. To address accuracy and consistency issues inherent in vanilla LLMs when emulating shell environments, we designed robust prompts using various prompt engineering techniques. Specifically, for complex commands that LLMs initially struggled to identify or respond to correctly, we employed in-context learning by feeding the models with examples selected from real attack traces. To address inconsistencies in responses generated by LLMs across attack sessions, we utilized the chain-of-thought technique and integrated essential session history into the prompts. Moreover, to improve cost-effectiveness and manage potential query rate limitations during real-world online deployment, we designed a hybrid architecture for HoneyLLM. This system filters pre-determined scanning scripts to low-interaction honeypots while directing genuine attack sessions to LLMs. Through extensive offline experiments and online deployment, we demonstrated that HoneyLLM can effectively handle prolonged, complex attack sessions and significantly extend engagement with online attackers.

IX. Acknowledgments

This research was partially sponsored by the Army Research Laboratory under Cooperative Agreement Number W911NF-24-2-0132. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Army Research Laboratory or the U.S. Government.

References

[1] Busybox: The Swiss Army Knife of Embedded Linux. https://busybox.net/.

[2] N. Provos. Honeyd: A virtual honeypot framework. *USENIX Security Symp.*, 2004.

[3] Cowrie SSH/Telnet Honeypot. https://github.com/cowrie/cowrie.

[4] Y. Pa, S. Suzuki, K. Yoshioka, T. Matsumoto, T. Kasama, and C. Rossow. IoTPOT: Analysing the Rise of IoT Compromises. *USENIX Workshop on Offensive Technol.*, 2015.

[5] A. Tambe, Y. Aung, R. Sridharan, M. Ochoa, N. Tippenhauer, A. Shabtai, and Y. Elovici. Detection of Threats to IoT Devices Using Scalable VPN-Forwarded Honeypots. *ACM Conf. on Data and Application Security and Privacy (CODASPY)*, 2019.

[6] B. Wang, Y. Dou, Y. Sang, Y. Zhang, and J. Huang. IoTCMal: Towards a Hybrid IoT Honeypot for Capturing and Analyzing Malware. In *IEEE Int'l Conf. on Commun. (ICC)*, 2020.

[7] F. Dang, Z. Li, Y. Liu, E. Zhai, Q. Chen, T. Xu, Y. Chen, and J. Yang. Understanding Fileless Attacks on Linux-Based IoT Devices with HoneyCloud. *ACM MobiSys*, 2019.

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. Gomez, L Kaiser, and I. Polosukhin. Attention Is All You Need. *arXiv preprint arXiv:1706.03762*, 2023.

[9] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury. Large Language Model guided Protocol Fuzzing. *NDSS*, 2024.

[10] H. Pearce, B. Tan, B. Ahmad, R. Karri, and B. Dolan-Gavitt. Examining Zero-Shot Vulnerability Repair with Large Language Models. *IEEE Symp. on Security and Privacy*, 2023.

[11] M. Boffa, G. Milan, L. Vassio, I. Drago, M. Mellia, and Z. Ben Houidi. Towards NLP-based Processing of Honeypot Logs. *IEEE European Symposium on Security and Privacy Workshops*, 2022.

[12] S. Singh, F. Abri, and A. Namin. Exploiting Large Language Models through Deception Techniques and Persuasion Principles. *IEEE International Conf. on Big Data (BigData)*, 2023.

[13] N. Vishwamitra, K. Guo, F. Romit, I. Ondracek, L. Cheng, Z. Zhao, and H. Hu. Moderating New Waves of Online Hate with Chain-of-Thought Reasoning in Large Language Models. *IEEE Symp. on Security and Privacy*, 2024.

[14] X. He, S. Zannettou, Y. Shen, and Y. Zhang. You Only Prompt Once: On the Capabilities of Prompt Learning on Large Language Models to Tackle Toxic Content. *IEEE Symp. on Security and Privacy*, 2024.

[15] GPT-3.5 turbo. https://platform.openai.com/docs/models/gpt-3-5-turbo.

[16] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. Wainwright, et al. Training language models to follow instructions with human feedback. *arXiv preprint arXiv:1606.01540*, 2016.

[17] GPT-4o. https://platform.openai.com/docs/models/gpt-4o.

[18] H. Touvron, L. Martin, K. Stone, P. Albert, et al. Llama 2: Open Foundation and Fine-Tuned Chat Models. *arXiv preprint arXiv:2307.09288*, 2016.

[19] Anthropic Claude 2. https://www.anthropic.com/news/claude-2.

[20] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, et al. Language Models are Few-Shot Learners. *Advances in Neural Information Processing Systems (NeurlIPS)*, 2020.

[21] D. Zhou, N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D Schuurmans, C. Cui, O. Bousquet, Q. Le, and E. Chi. Least-to-Most Prompting Enables Complex Reasoning in Large Language Models. *International Conf. on Machine Learning (ICML)*, 2023.

[22] J. Wei, X. Wang, D. Schuurmans, M. Bosma, b. ichter, F. Xia, E. Chi, Q. Le, and D. Zhou. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models. *Advances in Neural Information Processing Systems (NeurlIPS)*, 2024.

[23] B. Wang, X. Deng, and H. Sun. Iteratively Prompt Pre-trained Language Models for Chain of Thought. *Conf. on Empirical Methods in Natural Language Processing (EMNLP)*, 2022.

[24] M. Antonakakis, T. April, M. Bailey, M. Bernhard, et al. Understanding the Mirai Botnet. *USENIX Security Symp.*, 2017.

[25] C. Guan, H. Liu, G. Cao, S. Zhu, and T. La Porta. HoneyIoT:Adaptive High-Interaction Honeypot for IoT Devices Through Reinforcement Learning. *ACM Conf. on Security and Privacy in Wireless and Mobile Networks (WiSec)*, 2023.

[26] O. Alrawi, C. Lever, K. Valakuzhy, R. Court, K. Snow, F. Monrose, and M. Antonakakis. The Circle Of Life: A Large-Scale Study of The IoT Malware Lifecycle. *USENIX Security Symp.*, 2021.

[27] Nmap: Open-source network scanner. https://nmap.org/.

[28] K. Greshake, S. Abdelnabi, S. Mishra, C. Endres, T. Holz, and M. Fritz. Not What You've Signed Up For: Compromising Real-World LLM-Integrated Applications with Indirect Prompt Injection. *ACM Workshop on Artificial Intelligence and Security (AISec)*, 2023.

[29] G. Deng, Y. Liu, Y. Li, K. Wang, Y. Zhang, Z. Li, H. Wang, T. Zhang, and Y. Liu. MasterKey: Automated Jailbreak Across Multiple Large Language Model Chatbots. *NDSS*, 2024.

[30] W. Si, M. Backes, Y Zhang, and A. Salem. Two-in-One: A Model Hijacking Attack Against Text Generation Models. *USENIX Security Symp.*, 2023.

[31] S. Kumar, M. Cummings, and A. Stimpson. Strengthening LLM Trust Boundaries: A Survey of Prompt Injection Attacks. *IEEE International Conf. on Human-Machine Systems (ICHMS)*, 2024.

[32] C. Guan, X. Chen, G. Cao, S. Zhu, and T. La Porta. HoneyCam: Scalable High-Interaction Honeypot for IoT Cameras Based on 360-Degree Video. *IEEE Conf. on Commun. and Network Security (CNS)*, 2022.

[33] T-Pot - The All In One Honeypot Platform. https://github.com/telekom-security/tpotce.

[34] F. Setianto, E. Tsani, F. Sadiq, G. Domalis, D. Tsakalidis, and P. Kostakos. GPT-2C: a parser for honeypot logs using large pre-trained language models. *IEEE/ACM International Conf. on Advances in Social Networks Analysis and Mining*, 2022.