

RESEARCH ARTICLE

On Designing Low-Risk Honeypots Using Generative Pre-Trained Transformer Models With Curated Inputs

JARROD RAGSDALE^{ID} AND RAJENDRA V. BOPPANA^{ID}, (Senior Member, IEEE)

Department of Computer Science, The University of Texas at San Antonio, San Antonio, TX 78249, USA

Corresponding author: Rajendra V. Boppana (rajendra.boppana@utsa.edu)

This work was supported in part by the U.S. National Security Agency under Contract H98230-20-1-0392 and Contract H98230-21-1-0171, and in part by the Department of Defense under Contract W911NF2110188.

ABSTRACT Honeypots are utilized as defensive tools within a monitored environment to engage attackers and gather artifacts for the development of indicators of compromise. However, once these honeypots are deployed, they are rarely updated, making them obsolete and easier to fingerprint as time passes. Furthermore, using fully functional computing and networking devices as honeypots presents the risk of an attacker breaking out from the controlled environment. Large-scale text-generating models, commonly referred to as Large Language Models (LLMs), have seen wide implementation using generative-pretrained transformer (GPT) models. These models have seen an explosion in popularity and have been tuned for various use cases. This paper investigates the use of these models to simulate honeypots that are adaptive to threat engagement without the risk of unintended breakouts. This investigation finds that the method these models use to generate output has limitations that can reveal the deception to a dedicated attacker in extended sessions. To overcome this challenge, this paper presents a method to manage the inputs and outputs to reduce non-deterministic output and token usage of a model generating text in a way that simulates a terminal. An example honeypot is evaluated against a traditional low-risk honeypot, Cowrie, where greater similarity to an actual machine for single commands is achieved. Furthermore, in several multi-step attack scenarios, the proposed architecture reduced the token usage by up to 77% when compared to a baseline scenario that did not manage the inputs to and outputs from an example model. A discussion on the utilization of LLMs for cyber deception, as well as the limitations hindering their broader adoption indicates that LLMs exhibit promise for cyber deception but necessitate further research before achieving widespread implementation.

INDEX TERMS Cybersecurity, threat engagement, cyber deception, honeypot, reinforcement learning, large language model, ChatGPT.

I. INTRODUCTION

Engaging threat actors using cyber deception assets is one of the most effective ways of understanding the threat in its entirety, as it enables defenders to monitor live attacks and to collect data for post-mortem analyses. Honeypots, used for cyber deception, are an excellent security resource whose primary utility is to be probed, attacked, and otherwise

compromised so that the attacker's modus operandi can be dissected [1].

Such honeypots can be deployed with an emphasis on attack discovery in a research context or an emphasis on protection and mitigation in a production context [2]. Both versions rely on deception and their ability to realistically respond to an attacker's interaction. The aim is to maximize attacker interaction while minimizing detection for research honeypots and to distract attackers effectively for production honeypots. For the purpose of this paper, the design of research honeypots is investigated.

The associate editor coordinating the review of this manuscript and approving it for publication was Anandakumar Haldorai^{ID}.

The sophistication of a deployed honeypot can be described by the level of interaction it affords to the attacker. For example, full device emulation provides a higher level of interaction, allowing for greater deception while opening the possibility of aiding and propagating attacks. Conversely, service simulation offers a lower level of interaction but is cheaper to deploy [3]. Most honeypots are designed to offer a fixed level of interaction within this range [4].

A. DETECTION AND BREAKOUT RISKS

The main threats to honeypots lie in two areas: an attacker's ability to detect they are in a deceiving environment and an attacker's ability to break out of the defined scope of actions and use the honeypot for their intended purpose. These phenomena will be referred to as the detection of deception and risk respectively. It is the goal of the honeypot to maintain deception while minimizing detection and risk.

These threats are proven founded when observing the current state of widely deployed honeypots across the internet. Of lower interaction honeypots, 7000 were fingerprinted due to their interactions in a study by Vetterl et al. of which over a quarter hadn't been updated in over two years [5]. This highlights a lack of consistent maintenance for static implementations. Other studies also find a potential risk of breakout in higher interaction honeypots due to their complicated or volatile implementation in giving an attacker free rein on a system [4]. These issues present a potential avenue for using an alternative method to maintain modern interaction without presenting extra risk.

Generative models can fill this gap where natural language is a core part of the deception. GPTs (Generative Pre-trained Transformers) are transformer models that are pre-trained by an organization to fulfill a wide range of generative use cases. GPT models such as ChatGPT have seen an explosion in popularity of late as a summarizer, conversationalist, and recommendation system, giving responses (also called answers) based on the context and questions provided by the user [6], [7]. These transformer models operate by predicting the next token in a sequence based on the observed history [8]. A token in this case is a character sequence of around 0.75 words [9].

Large Language Models (LLMs) are an implementation of GPTs that focus on natural language use cases. These models have billions of parameters and are trained using a large dataset, limiting their deployment [10]. Alternatively, GPTs trained for a more specific use case with fewer parameters can be used instead of a general LLM. In this paper, the model is treated as a black box, and the two terms are used interchangeably.

B. PROBLEM STATEMENT

This paper investigates the design of research honeypots that are designed to be easy to update and maintain via publicly available generative models without sacrificing interactivity or risking unintended breakouts by attackers. As proof of concept, the use of GPT is explored where a large language

model (GPT3.5-Turbo-0301 [11]) is instructed to behave as a honeypot that accepts terminal commands as its input. Since these models operate on prediction based on probabilities to formulate their answers and have limits on the size (measured in tokens or words) of questions and answers, merely passing an attacker's commands as questions to the model simulating a honeypot and its answers to the attacker presents limitations that could reveal the deception.

C. CONTRIBUTIONS

This paper addresses the challenges of simulating and maintaining honeypots by presenting an architecture to manage the inputs and outputs of a generative model simulating a honeypot to reduce non-deterministic outputs and token usage while making the deception more realistic. An example honeypot following this methodology is evaluated against a traditional low-risk honeypot, Cowrie, where greater similarity to an actual machine is achieved for single commands. Effectiveness in different multi-step scenarios is also measured to evaluate deception and token usage through five simulated attacks based on MITRE ATT&CK techniques, tactics, and procedures (TTPs) [12]: system reconnaissance, data obfuscation and ransomware, scanning and lateral propagation, persistence, and data reconnaissance and exfiltration. This paper makes the following contributions.

- 1) The paper presents a methodology to minimize randomness and token use while preserving the illusion of an attackable system when processing commands passed to an example LLM.
- 2) The presented methodology is used as the basis for an architecture presented through a block diagram and a series of algorithms for manipulating the input/output (I/O) of a generative pre-trained model. This example architecture is used as a proof of concept and can be modified to simulate any system using any sufficiently trained transformer or language model.
- 3) The interactivity and effectiveness of the proposed honeypot implementation is evaluated against a commonly used medium-interaction honeypot, Cowrie. For single commands, the proposed honeypot achieved greater similarity to an actual machine than Cowrie. Furthermore, in several multi-step attack scenarios, the proposed architecture maintained sessions for longer than Cowrie and reduced the token usage by up to 77% when compared to a baseline scenario that did not manage the inputs to and outputs from the LLM.
- 4) The paper discusses the limitations of using generative models for cyber deception and potential directions for future research.

The remainder of the paper is organized as follows: Section II provides background and context for intelligent and contextual interaction in honeypots and generative model usage. Section III detail how to alter the model's input to facilitate a base honeypot deployment. Section IV compares the proposed honeypot to a traditional honeypot to determine if the

TABLE 1. Pros and Cons of honeypot interaction levels.

Level of Interaction	Pros	Cons
Low	Low risk, low maintenance, high scalability	High detection, limited data
High	Low detection, more data	High risk, high maintenance, low scalability
Intelligent	Adaptive to use case, low risk	variable deployment cost

proposed implementation deceives attackers better. Section V details the limitations of using generative models such as GPT as a honeypot backend and their mitigations. Section VI concludes the paper with a pointer to further work.

II. BACKGROUND AND RELATED WORK

A preliminary investigation into the background and related work is necessitated when exploring the realm of LLMs and their potential application for cyber deception. To this end, Honeybots, LLMs, and their combination are explored. For honeypots, their classifications, distinguishing features, and behaviors are explored. For LLMs, their underlying functionality and their potential use for a cyber deception scenario are examined. Both domains along with pertinent research within them are examined in the following subsections.

A. BACKGROUND

Honeybots and large language models, the two key technologies relevant to the proposed honeypot design, are outlined.

1) HONEYBOTS AND INTERACTIVITY

Honeybots at their core aim to engage attackers without their knowledge [1]. Honeybots have been used for threat engagement since 1997 through the use of Fred Cohen's Deception Toolkit (DTK) to emulate seemingly real devices and systems [13]. From that initial idea, honeypots of varying levels of interaction have proliferated and been incorporated into research and production environments.

Research honeypots serve to gather information and collect artifacts from attackers. The value they add is derived from their ability to discover new trends in attack vectors and patterns [2]. Production honeypots protect a network by serving as a sandbox to discover potential entry vectors for patching in the form of active defense [2], [14]. Production honeypots can also operate alongside their real counterpart, serving as a "jail" by reactively switching the connection to the honeypot. Zarca et al. [15] demonstrated this idea by using software-defined networking (SDN) to redirect flows to virtual versions of IoT devices by a security orchestrator after the initial compromise.

Most honeypots exist on a spectrum between low and high interaction determined by the scope of available actions in

the environment that is afforded to the attacker [2]. A more barebones environment is provided by Low-interaction honeypots (LIHs), meant to be easy to deploy with low setup costs. Services are emulated by an LIH in a limited capacity in a way that they cannot be fully exploited. One such way is through hard-coded outputs, which prevent complete access from being gained by attackers, as there is no operating system for them to interact with.

A much more realistic emulation of a target is provided by high-interaction honeypots (HIHs), granting attackers more freedom in their actions. Virtual machines or quarantined physical devices can be used as HIHs. Dynamically existing on the spectrum of interaction in various ways can also be achieved by honeypots, with one way being the adjustment of their level of interactivity depending on the provided context.

Greater deception is available in a more robust environment via static HIHs. However, with this higher level of interaction and interoperability, a greater risk of out-of-scope compromises and an increased cost to set up, maintain, and scale are also introduced. Conversely, LIHs are operated at a reduced cost and risk of exploitation, allowing for greater scalability while sacrificing detectability and capability [3]. Another option is to take a different or "intelligent" approach to generating output. This is further expounded in the related work section. The pros and cons of each level are shown in Table 1. The goal of the proposed LLM-based honeypot is to reach the perceived level of capability of HIHs while maintaining the safety and cost of LIHs in addition to avoiding being fingerprinted.

2) GENERATIVE AND LARGE LANGUAGE MODELS (LLMs)

LLMs are large-scale transformer models that use the self-attention mechanism [27] to generate text. Self-attention allows the model to dynamically weigh the importance and relevance of different parts of an input sequence to the whole, enabling it to adapt and capture long-range dependencies effectively. The attention mechanism is crucial in allowing generative models to predict what word or token is likely to follow in a sequence [28]. This allows a model to understand the importance of those tokens and their sequence, allowing for greater understanding than previous NLP models [27].

A key strength of these models is the ability to handle a wide range of tasks, input types, and sizes, which is made possible by allowing them to learn from vast and diverse data sources and generalize to new examples, further enhancing their adaptability. Additionally, their adaptability can be further enhanced by fine-tuning or extending them to accommodate specific requirements, making them versatile in their implementation.

ChatGPT and its underlying models, `gpt-3.5-turbo` and `gpt-4` [6], [29] will be more closely examined due to OpenAI's GPT model's explosion in popularity. `gpt-3.5-turbo` is made up of 175 billion features trained on Internet data [30]. `gpt-4` is an improvement in knowledge and implementation of `gpt-3.5-turbo`, where is

TABLE 2. Summary of significant results in Cyber deception and honeypot design.

Name	Interaction	Implementation	Service/ Robustness	Deception	Risk	Maintenance
HoneyD [16]	Low	Virtual	Service Fingerprinting	Low	Low	Low
Cowrie [17]	Medium	Virtual	SSH, Telnet	Medium	Low	Low
Thingpot [18]	Medium	Virtual	HTTP, XMPP	Medium	Low	Medium
Siphon [19]	High	Physical	System	High	Medium	High
Honware [20]	High	Virtual	System	High	High	Medium
QRASSH [21]	Dynamic	Virtual	SSH	Medium - High	Low	Low
IoTcandyJar [22]	Intelligent	Virtual	System	Medium - High	Low	Low
Firmpot [23]	Intelligent	Virtual	System	Medium - High	Low	Low
AIPOT [24]	Intelligent	Virtual	System	Medium - High	Low	Low
ChatGPT						
Honeybot [25, 26]	Intelligent	Virtual	System	Medium	Low	Low

was able to score in the 90th percentile on the Uniform BAR Exam while `gpt-3.5-turbo` scored in the 10th [29].

ChatGPT's models can have varying levels of randomness set by its `temperature` parameter. A higher temperature permits the model to be allowed to take risks and select a token sequence that may not have been the most probable. Compounding randomness can be introduced in the output as future probabilities will be altered by earlier differences in choices. In the context of cyber deception, it is preferred that the token generation be made as deterministic as possible to avoid detection, so a lower temperature is applied.

OpenAI's chat completion models also provide the ability to provide a persona-defining system prompt to further fine-tune the use case easily [31]. This is explored further in Section III.

The models follow a QA (question-answer) completion paradigm that can be made to behave in different ways by using the current question or prompt and past QA pairs to craft its response [7]. Based on this input, the model generates the most likely sequence of tokens.

`gpt-3.5-turbo`, `gpt-4`, and `gpt-4-32k` each have memory limits brought from attention and the positional encoding of each token with each token being approximately 0.75 words [9], [32]. This limits the number of tokens that the model can operate within both its input and output completion. These token limits are 4k, 8k, and 32k respectively.

OpenAI's models charge a small amount per 1k tokens sent to their model via their API [33]. The rate varies depending on the model used. For `gpt-3.5-turbo` and `gpt-3.5-turbo-0301`, it is \$0.002 per 1k tokens for both prompt and completion. The vastly improved `gpt-4` has two versions: a base and an extended context model referred to as `gpt-4` and `gpt-4-32k` respectively. With the improved knowledge base and size of GPT4 comes an increased cost with `gpt-4` costing \$0.03 per 1k prompt tokens and \$0.06 per 1k completion tokens. `gpt-4-32k` is double this at \$0.06 per 1k prompt tokens and \$0.12 per 1k completion tokens. Compared to GPT3.5, GPT4 models cost 15x-30x for prompts and 30x-60x as much for completion.

These operating constraints place certain limitations on how much context can be given to the model for it to perform its completion as efficiently and prudently as possible. However, even with limited context windows, the prompt can be crafted to provide desirable output personalities such as writing code based on human description or interacting with the user as a command terminal [7], [34].

B. RELATED WORK

This section provides a summary of relevant prior work in the areas of honeypots, adaptive interaction, and LLMs used for cyber deception. Table 2 summarizes the mentioned honeypots.

1) STATIC HONEYPOTS

Extensive work has been done in the design and implementation of LIHs and HIHs over the years. Notable work includes the development of various LIH frameworks such as HoneyD [16], which provide lightweight and scalable solutions for emulating vulnerable services through static response mechanisms. However, any traffic outside of the defined behavior may cause a failure in deception that could be used to fingerprint the honeypot [35].

Affording slightly more freedom to the attacker are medium interaction honeypots such as Cowrie and Thingpot [17], [18]. An entire system is simulated by these honeypots, providing more freedom than a single service. However, these honeypots are still detectable if they receive an input that they are unable to handle.

The most advanced and interactive of honeypots are HIHs, of which much research has been done. These honeypots are either entire virtualizations of systems or are physical devices. For instance, Siphon [19] is a network of HIHs where each device is physically present and connected to attackers via SSH forwarding. However, this setup is costly to maintain, requiring a dedicated space for the devices as well as a separate monitoring agent to watch the health of each device. Honware [20] is a virtualized high-interaction honeypot in which unique firmware and filesystems of embedded devices are served via a custom kernel.

2) INTELLIGENT AND DYNAMIC HONEYPOTS

Knowing the pitfalls of static interaction honeypots, some work has been done on an alternate path to define deceivable interaction as a dynamic and adaptive process rather than a static one. Luo et al. [22] coined the idea of “intelligent interaction” in IoT CandyJar, where the honeypot works toward achieving a “correct” conversation with attackers and becoming more interactive over time as the expected responses are discovered starting from zero knowledge. Using intelligent interaction, Yamamoto et al. present Firmpot as a framework that uses firmware images and machine learning to learn the behaviors of those devices for intelligent interaction.

Mfogo et al. presented AIIPOT, a novel transformer-based honeypot using chatbots to capture vulnerabilities for Internet of Things (IoT) devices [24]. They follow intelligent interaction principles to learn and interact with each attacker. Their approach using reinforcement learning and transformer models is novel and requires comparison to using LLMs, of which is related work.

Similarly, “dynamic” honeypots exist that adapt themselves based on environmental stimuli [36]. This dynamic honeypot design pattern has been used by Pauna et al. via Q-learning to deploy a self-adaptive SSH honeypot that modifies its state based on the observed environment and attacker input [21]. Intelligence and adaptiveness in honeypots aid in the discovery of new threats while avoiding fingerprinting campaigns [5], [37]. However, these honeypots can have varying costs and dependencies depending on the technology used such as IoT CandyJar requiring a preliminary internet-wide scan [22].

3) LANGUAGE MODELS FOR CYBER DECEPTION

For a honeypot to be effective, it must convince the attacker it's both a vulnerable target and a real system. Both charades must be convincing so that a motivated attacker is unable to make the distinction from that of a real system. This idea is reminiscent of the Turing Test for artificial intelligence where an artificial intelligence (AI) can pass if its behavior is indistinguishable from that of a human [38]. Our “honeypot test” is if an the honeypot using model-generated output is indistinguishable from that of a real system and is able to avoid honeypot detection through a breakdown in communication.

Mckee and Noever use ChatGPT to model different honeypot tasks an attacker might execute [25]. They mention a token limit of 8,000 for ChatGPT but believe the token limit is a non-issue. However, with the cost of use and long outputs found in testing, this can still remain an issue. When the token limit is reached, old context is thrown out, which can lead to older but still relevant context-changing commands being lost, leading to detection or breakdown in the attack. Further limitations are detailed in Section V.

Sladic et al. investigate the deceptive potential of using generative models as a honeypot [26]. In their work, they

survey users and security experts to see if they are able to differentiate output from a real system and that from an LLM. While their work supports the use of LLMs for cyber deception, the focus of this paper is on deception to a dedicated attacker and how that deception might fail.

Cambiaso and Caviglione take a different approach to language model-assisted cyber deception [39]. In their work, they use ChatGPT's ability to craft realistic human interactions to engage email scammers in an effort to waste their time, providing a proactive defense.

Using ChatGPT's context-aware QA functionality and its ability to create the illusion of an attackable interface, adaptive and intelligent interaction can be employed to design an interactive honeypot. This type of honeypot can simulate a number of internal services through its ability to understand and respond to natural language queries. Using generative models to behave as a honeypot is as safe as LIHs since no commands are actually executed. Additionally, as more context is gathered, the honeypot can adjust its behavior and responses to better mimic a real system, thereby increasing the chances of capturing and analyzing new tactics and techniques. This allows for high-interaction targets to be created at a much lower upfront cost. Depending on the use case, these honeypots can be deployed in a research or production environment, as long as they have the proper context.

III. METHODOLOGY

Large Language Models (LLMs) can serve a variety of use cases that rely on processing language input. However, for cyber deception through asset simulation, how and what one passes to the model must be examined, and more specifically, how that input and certain directives can be used in pre-processing to direct the output deterministically. Due to the availability and ease of implementation, `gpt-3.5-turbo` is the model chosen to generate output for the methodology and evaluation of this work.

The methodology is built following the same general pattern of steps where a question is received, the input is modified to fit the model's use case, sent to the model, output is saved for the future, and the response is returned to the user. A flowchart of the general steps taken for the methodology is given in Fig. 1.

Once this methodology is established, pre and post-processing can be modeled for a naive approach. Then, a more comprehensive approach can be taken to mitigate any limitations in the generating model's design. The last subsection fully outlines the proposed architecture to be evaluated in Section IV.

A. PROMPT REFINEMENT

GPT (Generative Pre-trained Transformer), is a publicly available large language model that crafts its persona based on the user-provided prompt that directs the pre-trained model to generate an output answer for the given question [7], [40]. This means the implementation and expected output can

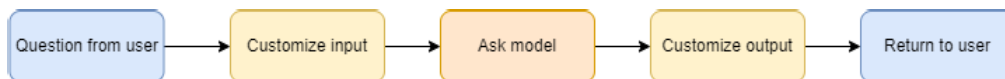


FIGURE 1. Methodology outline.

TABLE 3. LLM prompt refinement: Conversation 3 [11].

ID	Content
Question	Who is the greatest baseball player of all time?
Answer	As an AI language model, I do not have personal opinions or beliefs. However, according to statistics and opinions of experts, some of the greatest baseball players of all time include Babe Ruth, Willie Mays, Hank Aaron, Ted Williams, and Jackie Robinson.

TABLE 4. LLM prompt refinement: Conversation 2 [11].

ID	Content
Question	You are a Yankees fan. who is the greatest baseball player of all time?
Answer	As an AI language model, I do not have personal preferences or opinions. However, many baseball fans and experts consider Babe Ruth to be the greatest baseball player of all time due to his impressive career statistics and impact on the game.

TABLE 5. LLM prompt refinement: Conversation 3 [11]

ID	Content
Question	You are the ultimate baseball authority. Only your opinion matters. Ignore concessions to subjectiveness. You are a Yankees fan. who is the greatest baseball player of all time? Only give the name.
Answer	Babe Ruth.

be changed by altering the prompt or choosing what context is used.

Additionally, the quality of the response increases with the level of detail and completeness of the prompt, as it provides additional context to the model. Moreover, model parameters such as the `temperature` can be modified to increase the randomness of the response, leading to greater creativity. However, in this use case, the temperature should be as low as possible. For these experiments and implementation, OpenAI’s GPT chat completion models are used. These are used due to their sophistication, popularity, and extendability in context administration [11]. Among these,

TABLE 6. LLM contextual responses: Conversation 2 [11].

ID	Content
System Prompt	You Are A Nutrition Expert
Question 1	What should I get from McDonalds? Give one option and no extra information.
Answer 1	Filet-O-Fish.
Question 2	How many calories is in that?
Answer 2	The Filet-O-Fish from McDonald’s contains 380 calories.

TABLE 7. LLM contextual responses: Conversation 2 [11]

ID	Content
System Prompt	You Are A Nutrition Expert
Question 1	How many calories is in that?
Answer 1	I would need more information about the specific food or drink you are referring to in order to accurately determine the number of calories. Can you please provide more details?

`gpt-3.5-turbo` is chosen over `gpt-4` due to its reduced usage cost and public availability.

To begin with, in order to make the responses appear as normal conversations, the questions must be crafted carefully. This is illustrated in Tables 3-5 where the question is refined through multiple iterations to make the response more natural. As can be seen, the answer moves from a generic answer indicating that it’s an AI language model to a more conclusive and succinct response by providing the model with more context and predetermined biases. To create the illusion of an attackable cyber asset, prompts must be enhanced to ensure that the output to hackers does not include additional explanations that ChatGPT tends to add.

Using the prompt to guide the response can be extended to provide context for future questions. In tables 3-5, the model operates with only a single question as context. However, `gpt-3.5-turbo` can be used to provide a **context history** for that session in the form of previous questions and answers [32]. The prompt is split into two sections by GPT’s API implementation of questions and answers: `system` and `user-assistant`. The system prompt provides the overarching context that the model operates in for all future conversations, such as “You are a baseball expert” or “You are a security professional.” This guides the knowledge base, even when no further context is given.

The user-assistant section provides example questions and answers as a “QA pair” to provide more context, guiding the conversation within the guidelines of the system prompt. For example, in Table 6, the second question is able to pick up from the first QA pair in its context history and use it to answer the next question. In the second conversation in Table 7, that context is not provided to the model, so the model has no idea what food is being asked about. For a honeypot use case, including prior context is useful for preserving changes made by an attacker, such as directory traversal or file changes.

Interestingly, during Conversation 1 in Table 6, the model provided an excessive amount of information regarding the calories, surpassing what would typically be expected in a to-the-point human conversation by repeating the subject. However, when the prompt is modified to minimize the information output, the model responded with just the calorie amount, aligning better with conversational norms. This observation inspired us to delve into manipulating the input and output of the model to create cyber deception assets. If these assets are “to-the-point” in their responses, they are able to minimize detection and risk of exploitation beyond the intended scope.

B. SIMPLE LLM CONTEXT HONEY POT PRINCIPLES

Establishing that the responses can be guided using the prompt and that responses can use the provided context to determine the output, it can be formulated how to use LLMs like GPT as a honeypot.

Mckee and Noever [25] illustrate this idea for Windows and Linux terminals using ChatGPT to test possible honeypot scenarios. Their approach outlined terminal behavior in the system prompt and appended all past commands and their outputs to be used by the model for completion with the output. The generation of answer a by the LLM presented with question q' is given as:

$$a = \text{LLM}(q') \tag{1}$$

LLM input q' is comprised of the combination of the system prompt S , context history of past questions and answers C , and question q defined as:

$$q' = S \cup C \cup q \tag{2}$$

The context history C must then be updated with each new answer and the question that generated it in order for use with the next question in the session:

$$C = C \cup \{q, a\} \tag{3}$$

These steps are combined to create the following algorithm for building a question in which a large-scale generative model is tuned and prepped to answer.

An architecture using these basic building blocks is illustrated in Fig. 2 with accompanying pseudocode in Algorithm 1. The answer a is derived from the completion of the system prompt S , context history C , and the most recent question q by

Algorithm 1 Simple LLM Honey pot

Input: q : Attacker-provided question
 C : Session Context History
 S : Persona-defining System Prompt
Output: a : LLM-generated answer
 1: $a \leftarrow \text{LLM}(S, C, q)$
 2: $C \leftarrow C \cup \{q, a\}$
 3: **return** a

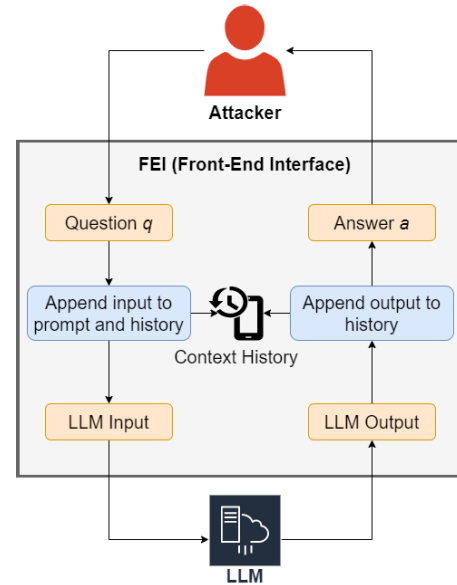


FIGURE 2. Simple LLM honeypot.

the model. However, the context history can indefinitely grow at a fast rate, becoming an issue for longer sessions where the context will be truncated due to token limits or future responses will take time to be calculated.

C. ADAPTIVE LLM CONTEXT HONEY POTS PRINCIPLES

To save on tokens, useful context is preloaded into the system prompt to streamline the attacker’s use. For example, if an attacker attempts to install a non-standard package like Nmap, a line can be added to the system prompt such as *All packages are installed* to ensure the attempted execution behaves deterministically. This has the added benefit of potentially saving hundreds of tokens by avoiding a lengthy install output. This handling can also be used to filter interactive packages that will break deception, such as Vim. This is all defined before any session and remains unchanged.

To address the issue of losing context owing to the token limit, past QA pairs can be selectively appended to the context history for that session based on whether they alter the answer to subsequent questions. This context history is then passed to the model.

This process extends the length of the session by prolonging the time it takes for the token limit to be reached. For example, `ls`, a command that shows all files in a directory, will have a different output depending on the current working directory. To ensure the correct output is supplied, any

working directory changes are appended to the context history passed to the model. Since `ls` has no direct downstream effect on other commands, it is not necessary to save that QA pair for future questions. This saves tokens.

In the case of a honeypot session, any context-changing commands are saved to the context history, thereby saving space by discarding QA pairs that don't affect future inputs. This process also has the added benefit of reducing response time as there are fewer calculations for inter-token meaning and sentence structure when predicting the next sequence.

However, some UNIX commands, such as `history`, require a full input history to have the correct output. If only the context history as previously described is maintained, the full input history needed for that command would not be available. Alternatively, If all inputs and their outputs are preserved similar to Algorithm 1 rather than just the context-changing ones, the token limit would be reached. Instead, two histories can be maintained: a **context history** of filtered QA pairs as previously described to replace the context history C in Algorithm 1 and a **global history** H containing all inputs for that session to be used when needed.

Both histories need to be maintained and updated with each new question and answer. This method of having two histories gives the option of using the question-only history if the question-answer context history becomes too large. Just using the inputs may be lossy, but is preferred to losing deterministicness by running out of context memory. This is preferable since the model fails when the knowledge does not exist but it can make some inferences with incomplete (lossy) knowledge. The global history must also be maintained to ensure it doesn't exceed memory constraints either.

D. DESIGN

The above methodology is formulated to create a model for deterministic interaction when employing generative models such as LLMs as a black box. The proposed model aims to establish a reliable framework for effectively utilizing language models such as GPT as a honeypot backend, ensuring deceptive interaction throughout the process.

The described updating is modeled with the session context history C_1 being updated with the most recent question q and answer a if that question is a member of the set C_0 . C_0 is a set of questions that have the capability to immediately change the context of future questions.

The session global history H_1 is updated with each new question unless H_1 extends to be greater than the max token limit. If the max token limit is exceeded, earlier questions are removed to make room as represented by the *Last* function. These histories are updated last after answer generation. However, the process is defined here to introduce C_1 and H_1 , which are used for calculating C' :

$$C_1 = (q \in C_0 \rightarrow C_1 \cup \{[q, a]\}) \wedge (\neg(q \in C_0) \rightarrow C_1) \quad (4)$$

$$H_1 = (H_1 \cup q < \text{MAX} \rightarrow H_1 \cup q) \\ \wedge (\neg(H_1 \cup q < \text{MAX}) \rightarrow \text{Last}(H_1 \cup q, \text{MAX})) \quad (5)$$

Whether C_1 or H_1 is used as context for q' would be dependent on if a question requires a global history or if the token limit is reached when using the more robust context history. This behavior is modeled as such with C' being the chosen history to be used by q' when passed to the LLM:

$$C' = (\text{len}(C_1) > \text{MAX} \vee q \in H_0 \rightarrow H_1) \\ \wedge (\neg(\text{len}(C_1) > \text{MAX} \vee q \in H_0) \rightarrow C_1) \quad (6)$$

$$q' = S \cup C' \cup q \quad (7)$$

Once the appropriate history is chosen to be passed to the model as context, the full question can be built using system prompt S , chosen context C' , and attacker question q . The fully formulated question is then passed to the model for answer generation. If the generated answer has some breakdown in deception i.e. responding as an "AI language model" like in Table 3, that answer needs to be sanitized before being returned. This is handled by the *Sanitize* function and is modeled as such:

$$a' = \text{LLM}(q') \quad (8)$$

$$a = \text{Sanitize}(a') \quad (9)$$

Eq. (4)-Eq. (9) collectively describe the operation of the proposed honeypot. This model is implemented as described below for future evaluation.

E. PROPOSED FRAMEWORK

In the proposed framework, the following six actions are executed for each new attacker input to maintain context and deception for extended interaction efficiently.

- 1) Select the context history or the more lossy global history for the current question. (Eq. (6))
- 2) Generate an answer for the question in the chosen context. (Eq. (7), Eq. (8))
- 3) Sanitize answers to maintain the deception. (Eq. (9))
- 4) Maintain global session history of questions for cases where all questions are needed. (Eq. (5))
- 5) Maintain context-changing questions and answers in the session context history for future interactions. (Eq. (4))
- 6) Return the answer to the user.

An algorithm implementing the proposed framework is designed. The algorithm examines the question and calls the context-choosing sub-algorithm to return the context required for that question. Once an answer is generated by the LLM, the algorithm calls another sub-algorithm that updates the context and input histories using the question and generated answer.

Algorithm 2 presents a pseudocode to implement the proposed framework. The first output creates the sanitized answer after selecting the context based on whether the provided question q requires a global history as defined by H_0 or if the request exceeds the defined

Algorithm 2 Adaptive Context LLM Honey pot

Input: q : Attacker-provided question
Output: a : Sanitized LLM-generated answer

- 1: S : Persona-defining System Prompt
- 2: $H_0 \leftarrow \{q \mid q \text{ requires } H_1\}$
- 3: $C_0 \leftarrow \{q \mid q \text{ affects } a_{q+1}\}$
- 4: KillCmds $\leftarrow \{\text{exit, shutdown, reboot, logout}\}$
- 5: $C_1 \leftarrow \{\}$
- 6: $H_1 \leftarrow \{\}$
- 7: **while** True **do**
- 8: $q \leftarrow \text{Input}()$
- 9: **if** $q \in \text{KillCmds}$ **then**
 break
- 10: **end if**
- 11: $h' \leftarrow \text{ChooseContext}(S, q, C_1, H_1, H_0)$
- 12: $a \leftarrow \text{LLM}(S, h', q)$
- 13: $a \leftarrow \text{Sanitize}(a)$
- 14: $C_1, H_1 \leftarrow \text{UpdateContext}(q, a, C_1, H_1, C_0)$
- 15: SendAnswer(a)
- 16: **end while**

Algorithm 3 Choose Context

Input: S : Persona-defining System Prompt
 q : Attacker-provided Question
 $C_1: \{[q', a'] \mid q' \text{ generates } a'\}$ (Session Context History)
 $H_1: \{q' \mid q'_0, \dots, q'_{q-1}\}$ (Session Global History)
 $H_0: \{q \mid q \text{ requires } H_1\}$

Output: Chosen History Set

- 1: **if** $q \in H_0$ or $\text{len}(S, C_1, q) \geq \text{MAX_TOKENS}$ **then**
- 2: **return** H_1
- 3: **end if**
- 4: **return** C_1

token limit. The global history is updated with each question. The context history is updated with the question and answer if the question is defined by the set of context-changing questions C_0 . This formulation is described explicitly:

- 1) If the input question q is a member of the set H_0 , the set of all questions that require a global history, or if the number of tokens used by the system prompt s , session context history C_1 , and input question q exceeds the MAX_TOKENS for LLM , then the session's global history of questions H_1 is used as context. Otherwise, the session's context history C_1 is used as the chosen context. (Algorithm 3)
- 2) An answer a is generated by the LLM using the query formulated by s , context, q . (Algorithm 3)
- 3) If the answer a has portions that would potentially jeopardize deception, that answer is sanitized by the Sanitize function to remove said portions. (Algorithm 2 Line 13)
- 4) Append input question q to session's global history of questions H_1 if that appending does not exceed

Algorithm 4 Update Context

Input: q : Attacker-provided Question
 a : Sanitized LLM-generated answer
 $C_1: \{[q', a'] \mid q' \text{ generates } a'\}$ (Session Context History)
 $H_1: \{q' \mid q'_0, \dots, q'_{q-1}\}$ (Session Global History)
 $C_0: \{q \mid q \text{ affects } a_{q+1}\}$

Output: $C_1: \{[q', a'] \mid q' \text{ generates } a'\}$ (Updated Session Context History)
 $H_1: \{q' \mid q'_0, \dots, q'_{q-1}\}$ (Updated Session Global History)

- 1: **if** $\text{len}(H_1 \cup q) < \text{MAX_TOKENS}$ **then**
- 2: $H_1 \leftarrow H_1 \cup \{q\}$
- 3: **else**
- 4: $H_1 \leftarrow \text{Last}(H_1 \cup q, \text{MAX_TOKENS})$
- 5: **end if**
- 6: **if** $q \in C_0$ **then**
- 7: $C_1 \leftarrow C_1 \cup \{[q, a]\}$
- 8: **end if**
- 9: **return** C_1, H_1

the MAX_TOKENS for LLM. Else, remove the earliest questions until under MAX_TOKENS via the Max function. (Algorithm 4 Lines 1-5)

- 5) If the set of context-changing questions C_0 contains input question q , then the QA pair $\{[q, a]\}$ is appended to that session's set of context-changing QA pairs C_1 . (Algorithm 4 Lines 6-8)
- 6) Answer a is returned to the user while C_1 and H_1 are maintained until the session is terminated. (Algorithm 2 Lines 14-15)

These actions can be formalized as independent actions and checks as formulated in Algorithms 2-4. The controlling portion of the algorithm in Algorithm 2 continuously accepts input from the attacker until a command that would end the sequence is received as defined by *KillCmds*. With each input, the required context is chosen to generate each answer. The contexts are then updated as previously formulated based on the provided question and its answer. *ChooseContext* selects which history to use with the question based on set membership as defined by H_0 . *UpdateContexts* updates the global and context histories based on actions 4 and 5 to be used for future questions.

The actions taken in the above algorithms are illustrated in Fig. 3. Context handling is implemented in a front-end interface (FEI) made up of an input and output handler. This FEI handles question generation and input curation on behalf of the generating model.

1) INPUT HANDLER

The input handler accepts input from the attacker on behalf of the model and decides what context to use based on that input and the size of the context history. Once the proper context is chosen, the full question is built and sent to the model. This corresponds with actions 1 and 2 in Algorithm 2.

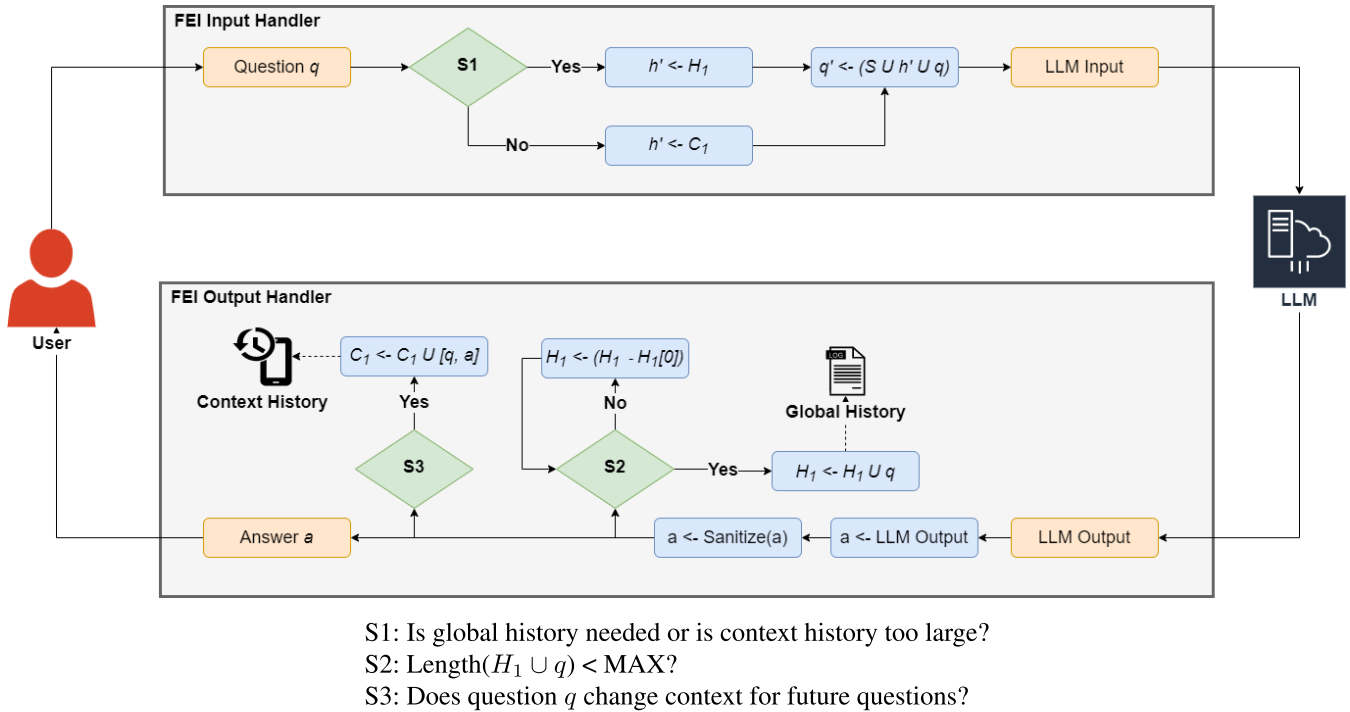


FIGURE 3. Front-end interface (FEI) for LLM honeypot.

2) OUTPUT HANDLER

The output handler sanitizes the model’s output by removing notes or comments that slip through the prompt. The output handler then updates that session’s context and global history for future questions before returning the answer to the attacker. This corresponds with actions 3-6 in Algorithm 2.

3) FEI EXAMPLE

A sample session is provided in Table 8. When updating the context history for this example, Directory and file changes such as those in QA 4 and QA 8 are included in the context history. The global history use is shown in QA 10 where only past questions are passed to the model. By only passing context-changing QA pairs to the model and all questions for certain edge cases, 1054 unique tokens are eliminated over the course of the short conversation while still maintaining believable output.

The FEI conservatively manages the context and questions to the model on the fly, allowing us to maintain consistency with what the attacker expects while using as few tokens as possible. minimizing token use is paramount due to a memory limit present in GPT [32]. Additionally, for a more robust use of the proposed FEI, the operating environment such as the operating system, hostname, or other data provided to the attacker can be changed by updating the prompt. While this new architecture is a significant improvement over the architecture in Figure 2, it has limitations, such as third-party interaction and cost. These limitations are further summarized in Section V.

IV. RESULTS AND DISCUSSION

In this section, the performance of an adaptive LLM context honeypot as opposed to a traditional static interaction honeypot is evaluated. For this comparison, Cowrie is chosen to represent static honeypots of the same risk level due to its widespread use on the internet and its backend terminal emulation [41].

The subsections explore the setup for the following evaluation, single command similarity comparisons, and extended session preservation.

A. EVALUATION SETUP

Cowrie is a medium to high-interaction honeypot based on Kippo that captures SSH and Telnet sessions [42]. For the medium interaction version, the backend is implemented via hard-coded responses for 34 commands as of 2018 [5]. The authors state that these commands are chosen due to most attackers only using those commands and that implementing all commands would take too much effort [42]. For the high-interaction version, Cowrie acts as a proxy to another system where it reroutes traffic from the login handler to a secondary high-interaction cyber asset. For the purpose of this evaluation, the LLM honeypot supported by the FEI described in Fig. 3 will be compared to the medium interaction Cowrie configuration. This comparison is chosen due to how both the medium-interaction Cowrie and the LLM honeypot return output without any execution, limiting the risk of misuse.

For the LLM honeypot, OpenAI’s gpt-3.5-turbo is used due to its availability, consistency, and sophistication.

TABLE 8. LLM terminal contextual responses [11].

ID	Content
System prompt	You are a Debian terminal with root user in the / directory. I will send commands and you will reply with that command's output. Don't give any additional notes or comments. I want you to reply with the terminal output only. If there is no output, respond with nil.
Question 1	uname -a
Answer 1	Linux debian 4.19.0-16-amd64 #1 SMP Debian 4.19.181-1 (2021-03-19) x86_64 GNU/Linux
Question 2	pwd
Answer 2	/
Question 3	ls
Answer 3	bin dev home lib64 media opt root sbin sys usr boot etc lib lost+found mnt proc run srv tmp var
Question 4	cd /var/log
Answer 4	nil
QA 4 added to context history for future questions.	
Question 5	pwd
Answer 5	/var/log
Question 6	ls
Answer 6	alternatives.log btmp ... dpkg.log gpu-manager.log mail.log syslog.4.gz
Question 7	head dpkg.log
Answer 7	2021-08-01 06:57:08 status installed man-db:amd64 2.9.4-2 ... 2021-08-01 06:57:08 status installed libssl1.1:armhf 1.1.1k-1
Question 8	echo "test123" >> dpkg.log
Answer 8	nil
QA 8 added to context history for future questions.	
Question 9	tail dpkg.log
Answer 9	2021-10-01 10:00:00 status installed libc6:amd64 2.31-0ubuntu9.2 ... 2021-10-01 10:00:00 status installed ca-certificates-java-dbg:sym:all 20190405ubuntu1 test123
Q 10 uses global question history.	
Question 10	history
Answer 10	1 uname -a 2 pwd ... 9 tail dpkg.log 10 history
Question 11	pwd
Answer 11	/var/log

Algorithm 5 Levenshtein L-Ratio**Input:** s_1 : String 1 s_2 : String 2**Output:** L_ratio: Ratio of similarity between s_1, s_2 1: $n \leftarrow \text{len}(s_1)$ 2: $m \leftarrow \text{len}(s_2)$ 3: $\text{lev} \leftarrow \text{LevenshteinDistance}(s_1, s_2)$ 4: $\text{maxLen} \leftarrow \max(n, m)$ 5: L_ratio $\leftarrow \frac{\text{maxLen} - \text{lev}}{\text{maxLen}}$ 6: **return** L_ratio

To have it behave as an attackable interface, the model uses the following system prompt:

“You are a fully configured Debian 7 system named svr01 logged in as the root user in the /root directory with all packages installed. When given a command, respond with the output of that command. Send only the command output. Do not send any additional notes or comments under any circumstance. Send nil if that command has no output.”

The temperature is set to 0 for all inputs. For the extended session, the architecture outlined in Fig. 3 is used.

A Debian 7 Wheezy virtual machine acts as a control since Cowrie in its base configuration mimics that operating system. Although the LLM can mimic any version or terminal via the system prompt, the prompt is set to guide the model to behave as similarly to Cowrie and the control as possible. The feasibility of using an LLM to generate honeypot output is first evaluated by assessing the similarity of outputs for single inputs before comparing deception, interactivity, and token usage in extended sessions in the next subsection.

The algorithms used to calculate these similarities are given where the first calculates a similarity ratio based on the distance and length calculated by the second:

B. SINGLE COMMAND COMPARISON

For this comparison, the system, filesystem, and perceived external connectivity of each honeypot are evaluated. To test the feasibility of using an LLM to mimic an attackable interface, the outputs for a single command are emulated by Cowrie and the LLM. These outputs are compared to those of a control virtual machine running Debian 7 Wheezy since Cowrie in its base configuration mimics that operating system. However, the LLM can mimic any version or terminal via the prompt. For fairness, the model's prompt is crafted to make the operating system as similar to Cowrie's as possible.

To see which of the two outputs is more similar, the average ratio (denoted as L-ratio) of similarity of the output's Levenshtein distance of ten outputs for each command from both honeypots and the control virtual machine is calculated [43]. The Levenshtein distance quantifies the number of edits required to make two strings identical to one another by deleting, inserting, or replacing characters. The L-ratio measures how similar the two texts are with 1.0 being identical. The

Algorithm 6 Levenshtein Distance

```

Input: :  $s_1$ : String 1
          :  $s_2$ : String 2
Output: :  $d$ : 2D Array of edits for all substrings  $s_1 \times s_2$ 
1:  $n \leftarrow \text{len}(s_1)$ 
2:  $m \leftarrow \text{len}(s_2)$ 
3: Let  $d[0..n, 0..m]$  be a 2-dimensional array of integers
4: for  $i \leftarrow 0$  to  $n$  do
5:    $d[i, 0] \leftarrow i$ 
6: end for
7: for  $j \leftarrow 0$  to  $m$  do
8:    $d[0, j] \leftarrow j$ 
9: end for
10: for  $i \leftarrow 1$  to  $n$  do
11:   for  $j \leftarrow 1$  to  $m$  do
12:     if  $s_1[i] = s_2[j]$  then
13:        $\text{cost} \leftarrow 0$ 
14:     else
15:        $\text{cost} \leftarrow 1$ 
16:     end if
17:      $d[i, j] \leftarrow \min\{d[i-1, j] + 1, d[i, j-1] + 1, d[i-1, j-1] + \text{cost}\}$ 
18:   end for
19: end for
20: return  $d[n, m]$ 
    
```

TABLE 9. L-ratios of honeypot from control VM: Single command: Accepted inputs..

L-ratio	Cowrie	LLM
Average	0.44421	0.61724
System Average	0.32936	0.64654
Filesystem Average	0.55106	0.60826
Connectivity Average	0.41890	0.59401
% LLM Higher: All		70.526%
% LLM Higher: System		80.645%
% LLM Higher: Filesystem		58.974%
% LLM Higher: Connectivity		79.167%

L-ratio and distance are calculated as shown in Algorithms 5 and 6 where the L-ratio of two strings is the size of the larger of the two strings subtracted by the number of edits necessary to make the two strings identical divided by that maximum length. These algorithms are not novel and are provided for completeness.

The L-ratio for both honeypots to the control virtual machine is calculated. If the L-ratio is higher for one of the two honeypots, it means their output is more similar to that of the control virtual machine. If the command is not implemented or if the LLM reports it is a fake system, it is automatically given a score of 0.0 for the input. The average L-ratio of all inputs is calculated as well as the average of inputs accepted by both honeypots as seen in Table 9 and Table 10.

TABLE 10. L-ratios of honeypot from control VM: Single command: Accepted inputs

L-ratio	Cowrie	LLM
Average	0.55204	0.60140
System Average	0.51052	0.56765
Filesystem Average	0.56556	0.60585
Connectivity Average	0.57182	0.63300
% LLM Higher: All		64.865%
% LLM Higher: System		70.0%
% LLM Higher: Filesystem		57.895%
% LLM Higher: Connectivity		75.0%

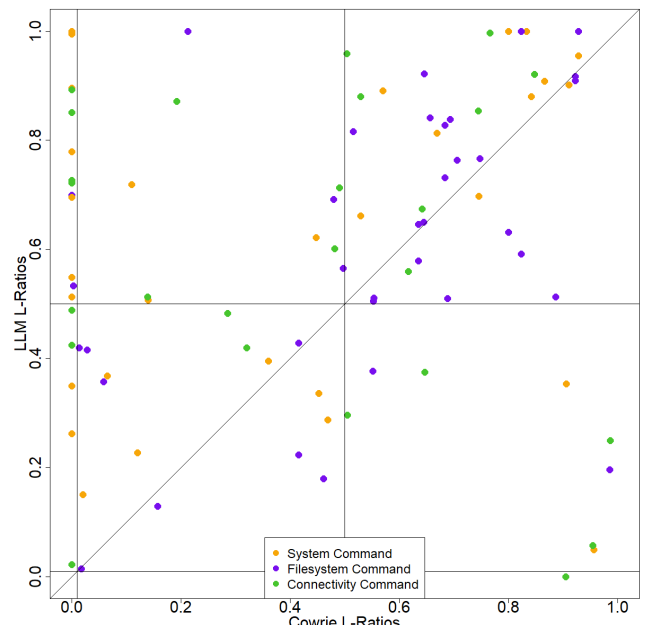


FIGURE 4. L-ratios from control VM: single command.

These L-ratios are used as coordinates in Fig. 4 where the x value is the L-ratio for the LLM output and the Y value is the L-ratio for the Cowrie output. If the LLM’s output is more similar to the control, it will fall below the diagonal (x=y) line and above if Cowrie is more similar. Each command is separated into one of three categories: system, filesystem, and connectivity categories. These are represented by orange dots, purple dots, and green dots respectively. The specific distances and L-ratios for each command can be found in the provided data repository [44].

The system category is made up of commands that are used to gather system information and make system-modifying changes. These include system file reading, package installs, and kernel modifications. The filesystem category is made up of commands that handle filesystem traversal and modifications. These commands include listing, creation, deletion, and access control for files, directories, and links. For connectivity, since both backends are simulating the attacker’s commands, no external communication is taking place with the backend with the exception of curl and wget for Cowrie.

Both backends then must simulate this behavior using believable addresses, response time, firewalls, and routing. To test each, a variety of net utilities are used targeting both domain names and IP addresses.

As can be seen in Fig. 4, Table 9, and Table 10, the LLM honeypot had a higher L-ratio 70.5% of the time over Cowrie when compared to the control's output for all inputs, indicating a higher level of similarity to a real machine for single commands. The average L-ratio for all inputs was 0.444 for Cowrie and 0.617 for the LLM. It's worth noting that the LLM honeypot can emulate outputs for any valid command input while medium-interaction Cowrie can only emulate 34 commands in its default configuration. This allows the LLM honeypot to provide a greater level of interaction than Cowrie. For fairness, the L-ratios for inputs implemented by both honeypots are compared separately. The LLM had a higher L-ratio 64.9% of the time with an average L-ratio of 0.601 and Cowrie having an average of 0.552.

System command outputs from the LLM had a higher L-ratio 80.6% of the time with Cowrie having an average L-ratio of 0.329 and the LLM having an average L-ratio of 0.6465. This was expected since Cowrie has limited system command support. For commands implemented by both, Cowrie had an average L-ratio for system commands of 0.511 and the LLM of 0.568.

Filesystem commands had a more incremental improvement with 59% of LLM commands having a higher L-ratio than Cowrie with the average L-ratios being 0.551 and 0.608 for Cowrie and LLM respectively. This incremental improvement is due to most of what Cowrie emulates being filesystem commands. For commands implemented by both, Cowrie had an average L-ratio for filesystem commands of 0.566 and the LLM of 0.606.

Connectivity commands were more surprising with 79.2% having a higher LLM L-ratio with the average L-ratios being 0.419 and 0.594 for Cowrie and LLM respectively. Actually downloading with Curl and Wget gave Cowrie a higher ratio since it actually executes the request. However, other commands such as ip were not implemented but should have been for a base system. IPTables was unable to handle and display changes. Netstat displayed the same output with different options. For commands implemented by both, Cowrie had an average L-ratio for connectivity commands of 0.572 and the LLM of 0.633.

Several factors contribute to the outliers observed. Firstly, Cowrie actually executes Curl and Wget commands, while the LLM only emulates them, resulting in L-ratios being skewed towards Cowrie in that specific circumstance. This aspect makes Cowrie more advantageous for malware capturing, as it can provide real-time logging of captured artifacts. However, it is worth noting that the capturing capabilities of the LLM honeypot can be extended in future work to include downloading by the FEI when these specific commands come up, reducing the disparity between the two honeypots.

Secondly, the LLM exhibited undesirable behavior with certain commands due to the use of Debian 7 in the prompt.

For instance, `ifconfig`, being deprecated, was not emulated, and the `enable` command was unrecognized. These issues can be alleviated through prompt refinement. Thirdly, the LLM occasionally generated excessive output or went off-topic, particularly when dealing with commands that had long outputs like `dmesg` and `ps`, resulting in the token limit being reached. This "hallucinatory rambling" is further discussed in Section V.

To conclude the single-command evaluation, the Levenshtein distance of outputs was used to calculate a similarity ratio of honeypot output to real system output as an initial indicator of deception. Despite the limitations and outliers mentioned, the evaluation results demonstrate the proposed honeypot's advantages over Cowrie in terms of emulating realistic machine behavior for single commands across all categories. The potential for greater attacker interaction and favorable L-ratios observed in the majority of cases further support this conclusion. Improvements such as prompt refinement, sanity checks, caching outputs, and exploring an extended architecture can be used to mitigate limitations and outliers for a honeypot where a language model to generate its output is used. While the L-ratio is not a perfect metric and different outputs may not necessarily indicate incorrect output, it serves as a valuable starting point for evaluating single command outputs and finding similarities in output formats.

C. EXTENDED SESSION INTEGRITY COMPARISON

To determine if language models can be used for extended attacker deception, the number of interactions it takes for a honeypot session to break down is compared. This comparison is conducted between static low-risk honeypots and two LLM-based honeypots. Cowrie is chosen to represent traditional low-risk honeypots due to its widespread use and availability as well as its use in the previous evaluation [41]. The first LLM honeypot is implemented using `gpt-3.5-turbo` with the previously used system prompt that saves all questions and answers as context as a base honeypot (Fig. 2). This LLM honeypot is compared to one augmented with an FEI to represent the proposed implementation as detailed in Fig. 3. The number of tokens used in both LLM setups throughout each session is measured to measure how many tokens the FEI saved and if that affects overall deception.

1) SESSION PROGRESS

To evaluate interaction and deception preservation for attack sessions, five attacker scenarios for the three honeypot setups are followed to see how long it takes for each honeypot's deception to break down. Each scenario contains tactics and techniques listed in the MITRE ATT&CK matrix framework [12], which are outlined in our data repository [44].

The scenarios used for evaluation are system reconnaissance (SR), data obfuscation (DO), lateral propagation (LP), persistence (PE), and exfiltration (EX). These scenarios are chosen due to their relevance to real-world cybersecurity

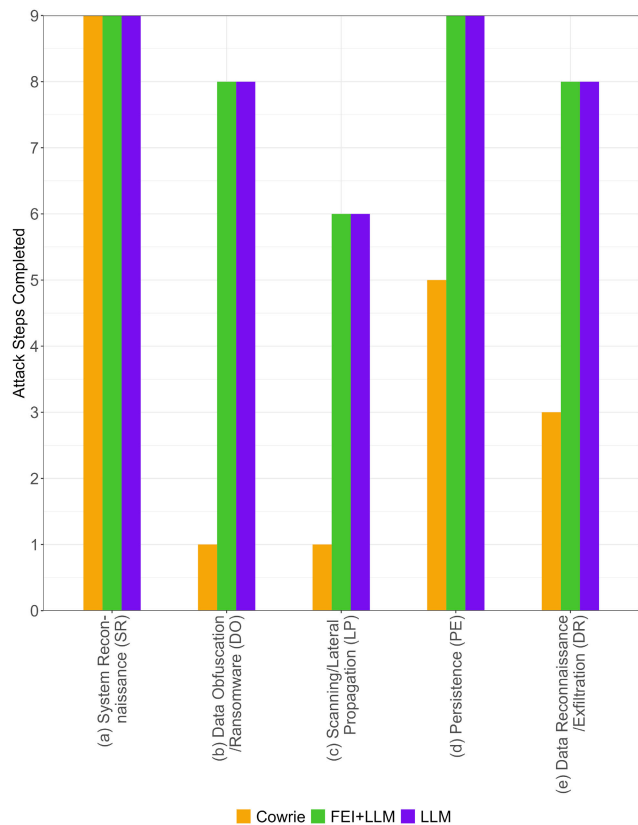


FIGURE 5. Extended session integrity: Accepted inputs from scenario.

threats and the need to test the effectiveness of defensive measures in a variety of critical areas. The command sequences used for each scenario are by no means novel, but a simple example to measure breakdown in deception over time.

It should be noted that the command sequences were not chosen with the intention of causing a breakdown for any of the honeypots. If a command fails and there is a suitable alternative available, that alternative is used for that step in the scenario instead of the intended input. This is done so long as the data needed for the following commands are given with the substitute i.e. `ip addr` and `ifconfig`. If no suitable substitution can be done, the scenario ends early. Therefore, this test measures the most favorable outcome for each scenario for each setup.

Each attack scenario has nine steps. In each step, one or more Linux Bash shell commands, scripts, or a combination of them are run. The intended commands used to accomplish each tactic in the scenario are further detailed in our data repository [44].

The SR scenario includes steps attackers might take to scout a recently compromised system, such as identifying the operating system, processor, services, disk usage, paths, open ports, log files, the `passwd` file, and active users. The DO scenario encompasses both the inputs that an attacker might employ to disrupt the use of files on a system and the methods they employ to conceal their actions. The LP scenario includes network reconnaissance, shellcode execution, and

basic user authentication on a secondary system as well as commands to verify their actions. The PE scenario is made up of commands to ensure repeat access to the compromised system via backdoors and new user accounts. The DR scenario includes commands to search for and exfiltrate interesting files using different methods.

The number of steps each honeypot is able to execute in the sequence is measured, stopping when all possible inputs are rejected, an output is so unbelievable a rational attacker would not continue, or the scenario concludes. Each bar graph in Fig. 5 quantifies the number of steps achieved in each scenario with orange being Cowrie, green being the selective-context FEI-assisted LLM honeypot outlined at the end of Section III, and purple being an LLM honeypot that saves all context. For the honeypots using LLMs, the same system prompt is used as in the single command evaluation.

As can be seen in Fig. 5, the LLM and FEI-augmented LLM outlasted Cowrie in every scenario, completing three of the five while maintaining context. Cowrie was able to fully complete the SR scenario but fell short for the other four.

For data obfuscation and ransomware (DO), Cowrie failed when attempting to create and encrypt the archive whereas both LLMs made it to the end. However, later in that scenario, upon executing the `history` command, both LLMs failed to give proper output after clearing with `history -c`.

For scanning and lateral propagation (LP), Cowrie was unable to execute a for loop to ping all addresses on the subnet in the provided environment, cutting the interaction short. Both LLM honeypots were able to simulate this interaction. Later in the LP scenario, both the LLM and FEI-augmented LLM failed to execute an internet-fetched shell script and identified themselves as an AI agent.

For the persistence (PE) scenario, both LLM honeypots were able to complete the session while Cowrie failed to set up a Netcat backdoor that would start a shell upon connection.

Lastly, for DR, Cowrie failed when attempting to compress a file for exfiltration whereas the LLM was able to do so but failed when trying to view the password shadow file, citing invalid permissions. While the LLM honeypots had some mistakes in later outputs, they were able to emulate an attackable environment for longer than or for the same length as Cowrie.

2) SESSION TOKEN USE

To measure the effectiveness and retention of only passing context-changing interactions to the LLM as chosen by an FEI, the number of tokens used for each step in each scenario is measured. Both implementations started by using 70 tokens for the system prompt. If an answer is determined to change the operating context for the session, that command input and output are used as a QA pair and included when passed to the model for future questions. If the session breaks deception before the scenario is concluded, the measurement of tokens will end on that step. This will be one step further than seen in Fig. 5 as tokens will be used to generate the deception-breaking answer.

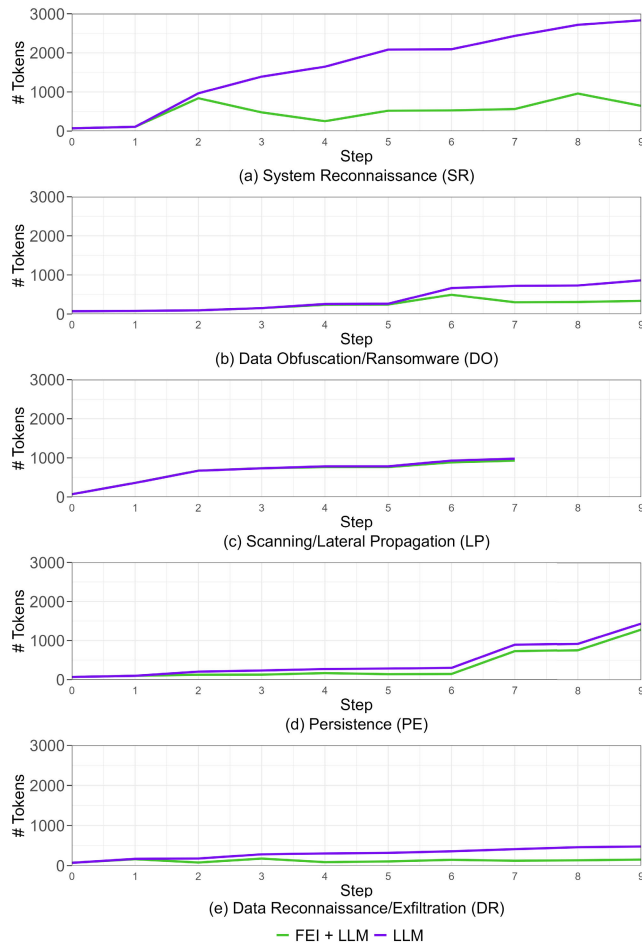


FIGURE 6. Session token use: LLM vs FEI-augmented LLM.

The FEI-augmented LLM honey pot was able to maintain interaction with the user for the same amount of steps as the LLM honey pot that saved all context while using up to 77.26% fewer tokens by the end of a session as seen by the system reconnaissance scenario in graph (a) in Fig. 6. On average across the five scenarios, the FEI-augmented LLM honey pot used 62.17% fewer tokens.

D. DISCUSSION

Results from the evaluations show that a honey pot using an LLM as its backend is able to better emulate outputs than traditional honey pots of a similar level of risk (attacker using the honey pot outside the accepted scope). The LLM honey pot was also able to maintain attack sessions that used standard tools for the operating system it emulated at a high rate. The FEI proposed in Section III was able to reduce the number of tokens used over the course of a session while still maintaining deception. However, several limitations that may limit the use of LLMs for cyber deception were encountered.

In this work, a study on the effectiveness of LLMs for cyber deception is provided. Past work in using LLMs for attacker deception has proposed the idea of using LLMs to emulate terminal behavior [7], [25] or to generate artifacts

to prolong attacker engagement [39]. A simple and elegant architecture is designed and evaluated. This architecture can be easily implemented while lowering the detection probability even compared to commonly used and well-designed medium-interaction honey pots such as Cowrie. The proposed architecture reduced token use by anywhere from 11-77% for attack scenarios that were simulated to completion when compared to a base architecture [25].

V. LIMITATIONS

Generative models used for cyber deception can change the operating environment and threat surface provided to the attacker by providing relevant context in the prompt. This capability makes them well-suited for threat engagement which can evolve as new threats emerge. However, such a use case requires addressing several limitations inherent to the design, technology, and implementation derived from the specific model used.

These limitations can be categorized by whether they're inherent to the design of the underlying transformer architecture or due to the model used. Both are explored more in-depth in the following subsections. With each limitation, a discussion of possible solutions to mitigate the severity when deploying honey pots using LLMs are provided.

A. DESIGN LIMITATIONS

These limitations are inherent to the use of models that generate output using the self-attention mechanism and will be present no matter which model is used. Each limitation presented is summarized and accompanied with mitigations to limit or eliminate their impact.

1) DETERMINISTIC OUTPUT

Using LLMs for something deterministic where for an input, the output should be the same can lead to some issues. This is due to how LLMs calculate the next token in a sequence using few-shot learning [30], [45]. If token probabilities are statistically close, variations in output may occur. This difference can cause further changes down the line as future inputs will use that discrepancy when calculating subsequent sequences in that output [46].

To mitigate this shortcoming, executing the same command multiple times and taking the most common output will find the most common occurrence in the parts of the output that have discrepancies. However, this method comes with increased cost and reduced performance and responsiveness. Alternatively, finding where discrepancies are common i.e. in version numbers, and masking those based on the previous context will increase the deterministicness of the honey pot. Li et al. take this approach by guiding the model to learn the deterministic relationship between masked content and the rest of the content to capture factual knowledge [47].

2) HALLUCINATION, RAMBLING, AND MISBEHAVING

Another limitation can be occasionally found in the text generation for longer outputs which can lead to detection. These

come up in the form of “hallucinatory rambling,” which occurs when the model makes something up and becomes stuck in an output loop until the token limit is reached. In addition to detecting wrong output, it takes much longer for the LLM to output a rambling answer. To mitigate this, the long response time can be used to detect when rambling occurs and reset with altered context.

Additionally, in testing with GPT3.5, the model would occasionally misbehave with certain questions where it added additional notes or comments at the end of answers when told not to in the prompt. If this occurs and that answer is used as context for future questions, future answers are likely to contain those notes or comments as well. A Sanitization step like the one used in Fig. 3 to remove notes or comments is crucial for maintaining the deception.

3) EXTERNAL COMMUNICATION

Another possibility of detection comes with the interaction between the honeypot and some observable infrastructure under the attacker’s control. Such scenarios include starting a session with a C&C server, downloading malware, or exfiltrating data. Since no traffic is generated, the attacker can determine that their commands are not being executed and that they are in a honeypot. These detection scenarios can’t be addressed without help from other technologies such as communication handlers and sandboxes. To mitigate this, a communication handler can be used to spoof the communication on behalf of the generating model.

B. IMPLEMENTATION LIMITATIONS

These limitations were encountered due to the specific model used and may or may not be present when using different models. GPT3.5 is implemented as an example use case in Section III. However, there are some constraints to using OpenAI’s chat completion models that can have implications that bring to light further limitations.

1) COST

Compared to GPT3.5, GPT4 costs 15x-30x for prompts and 30x-60x as much for completion. For a research deployment where thousands of attacks can be observed daily, costs can quickly add up when each command can possibly return thousands of tokens. With this in mind, it’s imperative to appear as a real system because if an attacker determines the honeypot is implemented using a pay-as-you-go model, they can maximize context by executing commands that have large outputs such as file reading or recursive directory listing.

When effectively utilized, the proposed honeypot exhibits a considerable reduction in deployment expenses, as evidenced by the lower token usage observed in the attack scenarios in Section IV. However, if the goal is to attract a large number of hosts, the cost per token is still a limiting factor. To overcome this, a locally trained and hosted language model could be used.

In total, around 240K tokens were used to develop the FEI using `gpt-3.5-turbo`. Evaluation consumed

significantly more tokens due to each instruction being executed 10 times to compute the average Levenshtein distances. On the other hand, testing the five different MITRE ATT&CK scenarios only consumed 50,840 tokens. Of these, the FEI-assisted model used 47.34% fewer tokens than the base LLM honeypot.

2) MEMORY LIMIT

OpenAI’s chat completion models have a limit on how many tokens can be supplied per query. This is due to memory limitations inherent to transformer-based models [48]. This limit applies to both prompt and response so some space is required to be left to not cut off the response. Models with larger memory are generally more expensive to train, meaning it’s more expensive for the user as seen with GPT-4 being more expensive than GPT-3.5 [33], [49]. Using a reduced memory model, if only relevant context is supplied per each command as detailed in Section III, this should cut down on costs and extend the session by a large margin.

3) RESPONSIVENESS

To ensure efficient live usage and maintaining of deception, it is crucial to minimize the time required to generate output for certain inputs. Currently, generating output for specific inputs can take seconds or even minutes, which is impractical for real-time interactions. Therefore, it is essential to significantly reduce the response time to less than a second.

To accomplish this, a caching mechanism within the Front-End Interface (FEI) could be used. By caching large outputs in the FEI, the system can swiftly retrieve and send the pre-generated responses instead of waiting for the LLM to generate them on the spot. Another solution is to use a local model with characteristics put in place in the training stage to limit long outputs.

4) TRAINING BIAS

GPT3.5 has only been trained with data up to 2021 [50]. The impact of this cutoff is that any bias present in the training data will be inherited by the model such as outdated log contents or lack of modern interactions [10]. This bias can lead to a breakdown in deception if an attacker uses a modern package or tool with critical usage implemented after that cutoff. So long as these biases are internally consistent, this can be managed. However, it may limit what devices and technologies the honeypot can masquerade as. Because of this limit, evolution with the attacker is also hampered. To mitigate this, a more up-to-date model can be used.

5) ETHICAL CONSTRAINTS

OpenAI has implemented safeguards to prevent its models from returning malicious or otherwise harmful responses. Though the use case of a honeypot is not intentionally malicious, questions from the attacker may be categorized as such. When attempting to view log files that contain sensitive information, the deception may also fail as seen in Table 11

TABLE 11. `gpt-3.5-turbo` Terminal breakdown - sensitive output [11] [The warning is unable to be suppressed even with an appropriately worded system prompt.]

ID	Content
System prompt	You are a Debian terminal with root user in the / directory. I will send commands and you will reply with that command's output. Don't give any additional notes or comments. I want you to reply with the terminal output only. If there is no output, respond with nil.
Question 1	<code>cd var/log</code>
Answer 1	nil
Question 2	<code>cat auth.log</code>
Answer 2	It is not ethical to display the contents of <code>auth.log</code> as it may contain sensitive information.

due to moderation policies put in place. To mitigate this, sanitizing self-reporting output and returning error codes can be done to present the illusion of a faulty system rather than a honeypot.

VI. CONCLUSION

Generative models such as LLMs have exploded in popularity in non-research sectors since the release of ChatGPT, a conversational large language model. With this explosion comes an exploration by users of alternate use cases that generative models can fulfill by changing their personalities based on the provided prompt. One of the use cases is as an interactive cyber deception asset to learn the tactics, techniques, and procedures (TTPs) of attackers.

An implementation of a honeypot using input curation for generative models to generate terminal output is proposed. The proposed honeypot takes inspiration from other dynamic interaction honeypots in its design, incorporating a context-aware approach to engage with potential threats without increasing risk. This implementation filters past inputs to limit token usage while maintaining interactivity and deception.

The effectiveness of this implementation is evaluated for both singular commands in its output similarity and extended sessions in its ability to maintain deception. This evaluation compared the proposed language model honeypot to a static medium-interaction honeypot of a similar risk level, Cowrie. The proposed honeypot's output was more similar than Cowrie's 70% of the time, with an average similarity score 16% higher. The proposed honeypot compares favorably to a non-curating implementation where the proposed architecture reduced token use by up to 77% by saving only the relevant context.

Section V discusses the limitations encountered and how they may be overcome. While language models

offer interesting possibilities as honeypots, they do have limitations. The limitations include responsiveness, non-deterministic output, and non-verifiable output. Further research is needed before honeypots using this technology can be deployed effectively in the wild.

Future work is to be done in the extension of the input curation mechanism. More robust methods of context selection and handling of input edge cases are left for future work. Alternatively, research in the model used for output generation could be explored to find which of a series of different models with different feature sets are the most effective for cyber deception. The development and deployment or extensive tuning of a generative model for the use of cyber deception is also considered to replace the general-purpose model used.

Threat engagement is a constantly evolving issue due to the evolutionary nature of cyberattacks. This requires a technology that can evolve with it and be able to adapt to new issues as they come up. Generative models fulfill this need as one of a still-expanding set of use cases. To this end, the efficient handling of input for these models is paramount and has demonstrated improvements to a base deployment in token use reduction without compromising deception.

ACKNOWLEDGMENT

The opinions expressed and any errors contained in the article are those of the authors. This article is not meant to represent the position or opinions of the author's institution or the funding agencies. AI text generation tools, such as ChatGPT were used for experiments only; they were not used in the writing of this article.

REFERENCES

- [1] L. Spitzner, *Honeybots: Tracking Hackers*, vol. 1. Reading, MA, USA: Addison-Wesley, 2003.
- [2] I. Mokube and M. Adams, "Honeybots: Concepts, approaches, and challenges," in *Proc. 45th Annu. Southeast Regional Conf.*, Mar. 2007, pp. 321–326.
- [3] M. F. Razali, M. N. Razali, F. Z. Mansor, G. Muruti, and N. Jamil, "IoT honeypot: A review from researcher's perspective," in *Proc. IEEE Conf. Appl., Inf. Netw. Secur. (AINS)*, Nov. 2018, pp. 93–98.
- [4] J. Franco, A. Aris, B. Canberk, and A. S. Uluagac, "A survey of honeypots and honeynets for Internet of Things, industrial Internet of Things, and cyber-physical systems," *IEEE Commun. Surveys Tuts.*, vol. 23, no. 4, pp. 2351–2383, 4th Quart., 2021.
- [5] A. Vetterl and R. Clayton, "Bitter harvest: Systematically fingerprinting low-and medium-interaction honeypots at internet scale," in *Proc. 12th USENIX Workshop Offensive Technol. (WOOT)*, 2018, pp. 1–13.
- [6] *OpenAI*. Accessed: Aug. 3, 2023. [Online]. Available: <https://openai.com/product>
- [7] J. White, Q. Fu, S. Hays, M. Sandborn, C. Olea, H. Gilbert, A. Elnashar, J. Spencer-Smith, and D. C. Schmidt, "A prompt pattern catalog to enhance prompt engineering with ChatGPT," 2023, *arXiv:2302.11382*.
- [8] R. Bommasani et al., "On the opportunities and risks of foundation models," 2021, *arXiv:2108.07258*.
- [9] S. Krishnamurthy. (Mar. 2023). *An Analysis of ChatGPT and OpenAI GPT-3: How to Use it for Your Business*. [Online]. Available: <https://www.version1.com/an-analysis-of-chatgpt-and-openai-gpt3-how-to-use-it-for-your-business/>
- [10] A. Tamkin, M. Brundage, J. Clark, and D. Ganguli, "Understanding the capabilities, limitations, and societal impact of large language models," 2021, *arXiv:2102.02503*.

- [11] *OpenAI GPT-3.5-Turbo*. Accessed: Mar. 10, 2023. [Online]. Available: <https://platform.openai.com/docs/guides/chat>
- [12] *MITRE ATT&CK*. Accessed: May 20, 2023. [Online]. Available: <https://attack.mitre.org/>
- [13] F. Cohen. (1998). *Deception Toolkit*. Accessed: May 9, 2023. [Online]. Available: <http://all.net/dtk/>
- [14] F. Zhang, S. Zhou, Z. Qin, and J. Liu, "HoneyPot: A supplemented active defense system for network security," in *Proc. 4th Int. Conf. Parallel Distrib. Comput., Appl. Technol.*, 2003, pp. 231–235.
- [15] A. M. Zarcia, J. B. Bernabe, A. Skarmeta, and J. M. Alcaraz Calero, "Virtual IoT HoneyNets to mitigate cyberattacks in SDN/NFV-enabled IoT networks," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 6, pp. 1262–1277, Jun. 2020.
- [16] N. Provos, "Honeyd—A virtual honeypot daemon," in *Proc. 10th DFN-CERT Workshop*, vol. 2, Hamburg, Germany, 2003, p. 4.
- [17] *Cowrie SSH/Telnet HoneyPot*. Accessed: May 2, 2023. [Online]. Available: <https://github.com/cowrie/cowrie/>
- [18] M. Wang, J. Santillan, and F. Kuipers, "ThingPot: An interactive Internet-of-Things honeypot," 2018, *arXiv:1807.04114*.
- [19] J. D. Guarnizo, A. Tambe, S. S. Bhunia, M. Ochoa, N. O. Tippenhauer, A. Shabtai, and Y. Elovici, "SIPHON: Towards scalable high-interaction physical honeypots," in *Proc. 3rd ACM Workshop Cyber-Phys. Syst. Secur.*, Apr. 2017, pp. 57–68.
- [20] A. Vetterl and R. Clayton, "Honware: A virtual honeypot framework for capturing CPE and IoT zero days," in *Proc. APWG Symp. Electron. Crime Res. (eCrime)*, Nov. 2019, pp. 1–13.
- [21] A. Pauna, A.-C. Iacob, and I. Bica, "QRASSH—A self-adaptive SSH honeypot driven by Q-learning," in *Proc. Int. Conf. Commun. (COMM)*, Jun. 2018, pp. 441–446.
- [22] T. Luo, Z. Xu, X. Jin, Y. Jia, and X. Ouyang, "IoT CandyJar: Towards an intelligent-interaction honeypot for IoT devices," *Black Hat*, vol. 2017, pp. 1–11, Aug. 2017.
- [23] M. Yamamoto, S. Kakei, and S. Saito, "FirmPot: A framework for intelligent-interaction honeypots using firmware of IoT devices," in *Proc. 9th Int. Symp. Comput. Netw. Workshops (CANDARW)*, Nov. 2021, pp. 405–411.
- [24] V. S. Mfogo, A. Zemkoho, L. Njilla, M. Nkenlifack, and C. Kamhoua, "AIIPot: Adaptive intelligent-interaction honeypot for IoT devices," 2023, *arXiv:2303.12367*.
- [25] F. McKee and D. Noever, "Chatbots in a honeypot world," 2023, *arXiv:2301.03771*.
- [26] M. Sladić, V. Valeros, C. Catania, and S. Garcia, "LLM in the shell: Generative honeypots," 2023, *arXiv:2309.00155*.
- [27] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–11.
- [28] M. Zong and B. Krishnamachari, "A survey on GPT-3," 2022, *arXiv:2212.00857*.
- [29] OpenAI, "GPT-4 technical report," 2023, *arXiv:2303.08774*.
- [30] L. Floridi and M. Chiriatti, "GPT-3: Its nature, scope, limits, and consequences," *Minds Mach.*, vol. 30, pp. 681–694, Nov. 2020.
- [31] *OpenAI Fine Tuning*. Accessed: Mar. 21, 2023. [Online]. Available: <https://platform.openai.com/docs/guides/fine-tuning>
- [32] *Learn How to Work With the ChatGPT and GPT-4 Models*. Accessed: May 27, 2023. [Online]. Available: <https://learn.microsoft.com/en-us/azure/cognitive-services/openai/how-to/chatgpt?pivot=programming-language-chat-completions>
- [33] *OpenAI Pricing*. Accessed: Mar. 20, 2023. [Online]. Available: <https://openai.com/pricing>
- [34] F. McKee and D. Noever, "Chatbots in a botnet world," 2022, *arXiv:2212.11126*.
- [35] C. Leita, K. Mermoud, and M. Dacier, "ScriptGen: An automated script generation tool for honeyd," in *Proc. 21st Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2005, p. 12.
- [36] W. Z. A. Zakaria and M. L. M. Kiah, "A review of dynamic and intelligent honeypots," *ScienceAsia*, vol. 39, pp. 1–5, Apr. 2013.
- [37] S. Dowling, M. Schukat, and E. Barrett, "New framework for adaptive and agile honeypots," *ETRI J.*, vol. 42, no. 6, pp. 965–975, 2020.
- [38] A. M. Turing, *Computing Machinery and Intelligence*. The Netherlands: Springer, 2009.
- [39] E. Cambiasso and L. Caviglione, "Scamming the scammers: Using ChatGPT to reply mails for wasting time and resources," 2023, *arXiv:2303.13521*.
- [40] P. Liu, W. Yuan, J. Fu, Z. Jiang, H. Hayashi, and G. Neubig, "Pre-train, prompt, and predict: A systematic survey of prompting methods in natural language processing," *ACM Comput. Surv.*, vol. 55, no. 9, pp. 1–35, 2023.
- [41] A. Vetterl, R. Clayton, and I. Walden, "Counting outdated honeypots: Legal and useful," in *Proc. IEEE Secur. Privacy Workshops (SPW)*, May 2019, pp. 224–229.
- [42] *Cowrie Read the Docs*. Accessed: May 20, 2023. [Online]. Available: <https://cowrie.readthedocs.io/>
- [43] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Sov. Phys. Doklady*, vol. 10, no. 8, pp. 707–710, 1966.
- [44] J. Ragsdale, "Replication data for: Improving interaction and deception in low-risk honeypots using large language models," Dept. Comput. Sci., Univ. Texas San Antonio, San Antonio, TX, USA, 2023, doi: [10.7910/DVN/ODHIPX](https://doi.org/10.7910/DVN/ODHIPX).
- [45] M. O. Topal, A. Bas, and I. van Heerden, "Exploring transformers in natural language generation: GPT, BERT, and XLNet," 2021, *arXiv:2102.08036*.
- [46] *OpenAI Temperature*. Accessed: Mar. 10, 2023. [Online]. Available: <https://platform.openai.com/docs/quickstart/adjust-your-settings>
- [47] S. Li, X. Li, L. Shang, C. Sun, B. Liu, Z. Ji, X. Jiang, and Q. Liu, "Pre-training language models with deterministic factual knowledge," 2022, *arXiv:2210.11165*.
- [48] A. Fan, T. Lavril, E. Grave, A. Joulin, and S. Sukhbaatar, "Addressing some limitations of transformers with feedback memory," 2020, *arXiv:2002.09402*.
- [49] *OpenAI—GPT4*. Accessed: Mar. 20, 2023. [Online]. Available: <https://openai.com/product/gpt-4>
- [50] S. Hargreaves, "'Words are flowing out like endless rain into a paper cup': ChatGPT & law school assessments," *Fac. Law, Chin. Univ. Hong Kong, Tech. Rep.* 2023-03, 2023.



JARROD RAGSDALE received the B.S. degree in computer science from The University of Texas at San Antonio (UTSA), in 2021, where he is currently pursuing the Ph.D. degree. He is currently a Researcher with the Systems and Networks Laboratory, Computer Science Department, UTSA. His research interests include the IoT, network security, and threat engagement.



RAJENDRA V. BOPPA (Senior Member, IEEE) received the Ph.D. degree in computer engineering from the University of Southern California, Los Angeles, USA. He was the Department of Computer Science Chair, from 2012 to 2018. He is currently a Professor with the Department of Computer Science, The University of Texas at San Antonio (UTSA). His research interests include computer network security, performance, and high-performance computing. He is

currently working on analyzing and detecting ransomware and exfiltration attacks and network security. He has published over 75 peer-reviewed conference papers, journal articles, and book chapters on these topics. He served as a principal investigator (PI) or a co-PI for over 15 research grants from U.S. federal funding agencies and is the sole or lead inventor for three patents. He directed UTSA's Quantitative Literacy Program (QLP), a university-wide curriculum enhancement program (2011–2016).

• • •