Evaluating Few-Shot Learning Generative Honeypots in A Live Deployment

Jarrod Ragsdale
Department of Computer Science
University of Texas at San Antonio
San Antonio, Texas 78249–4871
Email: jarrod.ragsdale@utsa.edu

Rajendra Boppana
Department of Computer Science
University of Texas at San Antonio
San Antonio, Texas 78249–4871
Email: rajendra.boppana@utsa.edu

2024 IEEE International Conference on Cyber Security and Resilience (CSR) | 979-8-3503-7536-7/24/$31.00 ©2024 IEEE | DOI: 10.1109/CSR61664.2024.10679421

***Abstract*—Generative language models have seen an explosion in use for downstream tasks due to their effectiveness in zero or few-shot learning scenarios. One such use case is the ability to emulate command terminals such as a Bash shell in Linux systems. These models' generative and non-evaluative nature makes them prospective candidates for use in threat engagement via output generation for honeypots. Studies have proposed using generative honeypots but have had limited evaluations in live settings. We deploy and evaluate generative honeypots with and without a context selection mechanism alongside a control honeypot. We found that generative model deployments significantly increased session length without risking compromise and that limited context selection can substantially reduce token usage. To illustrate the TTP-capturing potential of generative honeypots, we dissect some sessions observed during this deployment and discuss how the traffic observed can further refine generative model use and few-shot performance.**

I. Introduction

Honeypots are used to deceive attackers into believing they're targeting vulnerable systems to facilitate the discovery of previously unknown tactics, techniques, and procedures (TTPs) for analysis. Honeypots vary in implementation by how much interaction and freedom to operate they offer the attacker. Using these metrics, the most effective honeypots operate such that attacker inputs are executed on an actual system to generate the response to be returned. This method comes with increased maintenance costs from patching and updates [1]. Additionally, given the unknown nature of attacks observed by a honeypot, the safety of a honeypot's operation cannot be guaranteed when operating in this manner, and the operator assumes some risk of the attacker compromising the underlying system. We refer to them as high-interaction and high-risk honeypots. Alternatively, simulated honeypots can be deployed with less interaction but are safer to operate [1]; we refer to them as low-interaction and low-risk honeypots.

A newer approach is to use generative models to simulate honeypot responses. Natural language generation (NLG) models have seen significant advancements in recent years, with the release of the transformer architecture in 2017 by Vaswani et al. [2]. On top of this architecture, pre-trained models with billions of parameters, commonly called large language models (LLMs), have been developed and made readily available to the general public.These models have shown emergent use cases in zero-shot and few-shot learning environments. Zero-shot and few-shot learning refer to a set of techniques where a model operates on limited data using semantic information and descriptions to perform the intended task [3].

One such use case of these techniques is simulating a Bash terminal. Attackers can remotely interact with these terminals via commonly exploited protocols such as SSH and Telnet [4]. Thanks to terminal inputs and outputs in their training data scraped from websites; generative text models can feasibly emulate the output of real systems [5, 6]. The generative nature of these models means no command is evaluated; only an illusion of execution is presented through output generation, thus limiting risk. Additionally, the generating models have the potential to pick up on system-specific dynamics in the input and use them to generate unique output, allowing for the simulation of multiple types of systems. These simulations can be routinely updated by including these dynamics via model parameter updates. These features make such models potential candidates for use as output generators in low-risk honeypot deployments to engage attackers in new and unpredictable ways.

Initial explorations employing text generation models to mimic systems for use as honeypots have focused on assessing their capacity for deception and generation [7]. To improve these models' base output generation potential, the selection of context from past inputs and outputs has also been explored [8]. However, limited studies have been conducted on these implementations in a live deployment to evaluate their ability to interact with attackers and none for protocols such as SSH [9, 10]. Such observations can aid researchers in determining which limitations most affect deployment and should be addressed via model adjustments or input preprocessing. Additionally, a live deployment evaluation will help determine whether using these models for output generation will elicit the discovery of new TTPs. These TTPs can then be classified using existing frameworks, such as the MITRE ATT&CK framework, for dissemination into the appropriate security apparatuses [11].

To this end, we deployed two generative honeypots [7, 8] and a low-risk honeypot, Cowrie [12], on the internet for two weeks. We measured the average session length using inputs and generation efficiency by token usage. The data we collected shows that honeypots that generate their output using language models have a much longer average session length,

and managing the context fed to these models reduces token use by an average of 45%. We examine the sessions observed to determine if generative models elicit new TTPs.

This paper makes the following contributions:

1) We perform a live deployment study of Bash terminal-based honeypots that generate their output on the fly using generative language models in a few-shot learning method. To the best of our knowledge, this was the first such deployment of Bash honeypots using generative models.
2) We cluster the observed sessions to find unique interactions captured by the generative honeypots and provide a case study of a unique session, exploring its TTPs and how model-based output generation performed.
3) The data collected is made available for future work [13].

The rest of the paper is organized as follows: Section II introduces the core concepts and related work of honeypots and generative models for attacker interaction. Section III outlines the collection setup, evaluated honeypots, and observed metrics. Section IV the studies findings and case studies on the most common sessions. Section V summarizes our findings and how the observed data can be used to guide future work.

II. Background and Related Work

This section reviews different honeypot approaches and how generative text models have been used to simulate them. We also explore different evaluation methods and criteria used to determine the efficacy of a honeypot. Prior results and gaps in research are also discussed.

A. Traditional Honeypots

Honeypots are a security resource whose utility is found in their ability to be probed or compromised [14]. One of the earliest such resources is the Deception Toolkit, released in 1998 to engage attackers so that other attacks would be prevented via opportunity cost [15]. Since then, honeypot systems and architectures have been proposed for various monitoring objectives [1]. These honeypots' sophistication and operational effectiveness have been traditionally defined using interaction as the key metric [16].

The interaction level of a honeypot is determined by the amount of freedom it affords the attacker. Freedom is measured by the range of actions an attacker can take when interacting with the system. Additional characteristics such as cost, risk, and deception are derived from this interaction level. Low and medium-interaction honeypots (LIH and MIH) typically only emulate services and protocols in a limited capacity with little to no input validation or evaluation. For example, Cowrie, a medium-interaction SSH honeypot, provides a facade of a fully-fledged system and utilities. However, these utilities come with extremely limited functionality due to its surface-level evaluation of input [12]. This allows for limited data collection and deception, usually limited to the initial stages of compromise. The lightweight nature of these honeypots makes them much more scalable and easy to maintain, but an attacker may easily see the deception [17].

On the other hand, high-interaction honeypots (HIHs) aim to allow an attacker as much freedom as possible in their interaction. These honeypots are typically full-fledged operating systems that minimally restrict the attacker [16]. This allows the honeypot to collect a large amount of data with a low chance of detection by the attacker, requiring more system resources for each deployment. However, with this freedom comes an additional risk of unintended use, such as an attacker breaking out of the controlled environment. This risk necessitates regular updates of the underlying system and monitoring to detect exploitation [18].

B. Generative Models for Honeypot Simulation

Honeypots using generative models to create or manage output dynamically are a relatively recent development, made possible by transformers and their ability to parallelize the semantic capturing mechanism, referred to as attention, and calculate dependencies of the complete sequence rather than sequentially [2]. Attention refers to techniques in which a vector representation of each token in a sequence is adjusted with values corresponding to how relevant each token is to every other token in the sequence, given positioning and semantic relationships. A token, in this context, is a sequence of characters that is passed to an embedding model to calculate the semantic values to be stored as a vector for attention to operate on. Several works have explored the integration of these models and their predecessors into attacker engagement through honeypot deployment. These works use models with some attention mechanisms to facilitate output selection or generation.

C. Related Work

The Firmpot honeypot simulates a web server using a Seq2Seq model with an attention mechanism to select pre-collected responses from a database with the highest probability of extending the session [19]. This model's attention mechanism modifies the importance of each field of an attacker's HTTP request, namely, the path, headers, and method. Yamamoto et al. [19] employ a gated recurrent unit (GRU) based recurrent neural network (RNN) Seq2Seq decoder model to choose suitable responses from a table of outputs. These outputs are populated by virtual device responses and chosen using attention-modified request embeddings. This method introduces the ability to dynamically choose which device will respond based on the contents of each request, allowing a singular deployment to effectively interact with attackers targeting unique systems simultaneously.

AIIPot, proposed by Mfogo et al., follows a similar approach to Firmpot in using a Seq2Seq model to select outputs using an HTTP request as input [20]. AIIPot replaces Firmpot's output selection model with a fine-tuned BERT (Bidirectional Encoder Representations from Transformers). This model selects the responses most likely to continue the session from a database from which a Markov Decision Process (MDP) selects the honeypot's response using a reward function based on the final session length.

ML-based honeypots like AIIPot and Firmpot use generative models to select outputs from pre-generated options, limiting the output to static and pre-known request and response sessions. Thus, if new outputs are added to the database, the model must either estimate, be updated, or retrained, limiting the scalability of these implementations. This output table can be replaced with on-the-fly output generation from more capable generative models such as GPT3.5 [5]. McKee and Noever use GPT3.5 and provide examples of simulating Bash shells without fine-tuning in a zero-shot environment [7]. They focus on the potential of interaction instead of the potential of deception. Sladic et al. survey participants of various skill levels to see how deceptive in practice these models are with regularly updating prompts [21]. The survey had a sample size of 12 participants where 76 unique commands were executed over an average session length of 19 in which the honeypot fooled most participants.

The use of these large-scale generative models can be costly for extended use and local deployability. Therefore, any way to preprocess inputs to prune unnecessary portions of the input sequence will have beneficial downstream effects. Ragsdale and Boppana [8] perform an initial evaluation on a context-selection mechanism specifically for a Bash terminal honeypot use case. They examine how past session interaction affects the generation of the most recent input.

Wang et al. [22] present a framework for context handling in which past interactions are pruned based on their impact on the operating environment to reduce token usage. However, they do not process the model's response to sanitize outputs that may cause detection. Additionally, their evaluation is performed using session data from a traditional honeypot deployment in a counterfactual analysis.

Cambiaso and Cavligione [9] examine how language models can be used as a conversational honeypot sending emails [9]. They believe that using such models to generate email responses to suspicious solicitations reduces the attacker's capacity to carry out attacks on other victims. While this use case and interaction differ from those previously discussed, the underlying motivation and generation method are the same. In their study, they interact with 11 scammers, of which four have extended exchanges.

Vasilatos et al. [10] propose using data generated from interacting with industrial control system devices to fine-tune existing models. They evaluate whether LLMs can generalize the observed behaviors from the generated data to generate output that is different from those observed but in line with a real system. This approach would be more effective than few-shot learning using general-purpose LLMs. However, the amount of data required for general shell interaction using a similar method would be immense.

We have not seen any fine-tuned generative models for emulating bash terminal output. Additionally, only bash to natural language datasets could be found, limiting tuning capabilities [23]. Therefore, preparing a fine-tuned model is infeasible until such a model or data is available, making large foundation models used in a few-shot environment the best option. To this end, we found `gpt-3.5-turbo-0301` to be suitable for bash terminal simulation [5, 8].

Due to the closed implementation of the chosen model, managing input and output using model parameters is not possible. Without the ability to modify model parameters, the only way to modify output is to either include or exclude parts of the input by manipulating the session history to be used as context for sequence generation. There have been limited studies regarding using generative language models in a live deployment and how that deployment facilitates new data collection.

III. Experimental Setup

In this section. the setup for the proposed live deployment evaluation is described. The mechanism of each honeypot is discussed, along with the metrics used to evaluate each deployment. For this study, we measure session length to evaluate deception and capability and token use to evaluate efficiency. We evaluate two generative models with different context selection mechanisms, using Cowrie as a control for session length. A preliminary evaluation of input preprocessing for token use reduction was also conducted.

A. Cowrie: a Traditional Rule-Based Honeypot

Cowrie is a medium-interaction honeypot capable of emulating a filesystem for SSH/Telnet sessions for multiple users simultaneously [12]. Cowrie's front-end handles glob authentication while the back-end simulates responses for 71 standard Bash utilities. These utilities are emulated with predefined output and behaviors, altered by other session data, such as the current working directory and active user passed by the session handler and the emulated filesystem defined by the configuration. We chose Cowrie (ver 2.5.0) as our control for session length because of the similar risk found in its method of output generation, in which no commands are executed. This will provide an excellent baseline for evaluating generative honeypots based on their ability to generate reasonably believable output at a similar risk level.

B. Naive Generative Honeypot

Given a session's input history, a naive generative honeypot deployment must generate output consistent with attacker expectations. To achieve this, previous interactions are included in the model's input to enable their utilization for output generation. This requires saving the input and output after each interaction to be reused as context for the following input. Specific outputs, such as reading the contents of a file, can cause thousands of tokens to be generated, which can be highly inefficient for use as context when token use is a constraint.

C. Dual Context-Selection Generative Honeypot

Not all of a session's history is relevant to the output of the next command. For example, given sequence "`whoami; cd x; ls`," `whoami` has no impact on the output of `ls`. Thus, including that segment of the session history as input to the model for output generation is unnecessary.

We conducted a preliminary study on using such a method to maintain a filtered context updated only with commands

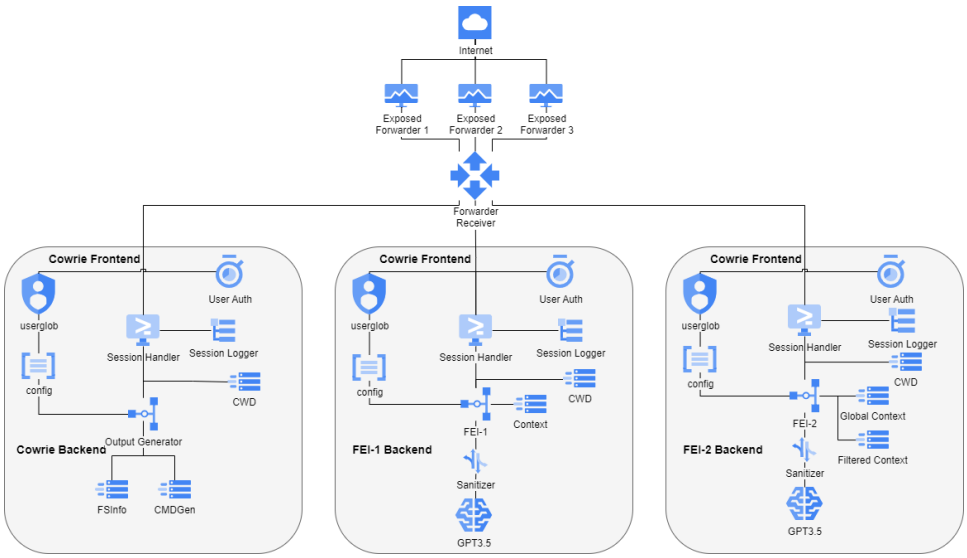

Fig. 1. Generative Model-Based Output Generation Live Deployment Setup

that drastically change future input's outputs alongside a global input history for commands requiring it [8].

This particular method of context preprocessing only defines context-changing on a global scale, not considering how an input affects the outputs of an individual input but rather if an input changes the operating state of the overall system. Input and output redirection are not considered during preprocessing and are left for future work on how these behaviors affect context selection. However, even with these limitations, token usage was drastically reduced in a controlled environment. By processing input and output this way during a live deployment, fewer tokens can be used without sacrificing consistency. This allows for use with more hardware-constrained model-based generation when using local models.

D. Deployment Setup

Cowrie can be split into two parts: the front end, which handles authentication and communication, and the back end, which handles output generation. The two context management schemes discussed in Section III-B-Section III-C are integrated into a front-end interface (FEI) for the chosen text generation model, labeled FEI-1 and FEI-2 respectively [7, 8]. These FEIs are integrated with Cowrie's front end and are evaluated against Cowrie in a default configuration. Each FEI passes the input through and receives output from a sanitization process between the generating model, `gpt-3.5-turbo-0301`.

To ethically handle passing user data to an external entity, any personally identifiable information (PII) within the inputs, such as IP addresses for downloading malware, is sanitized and replaced with an anonymized version. These inputs are then sequentially sent to their corresponding output generators for processing and any necessary session data. These replacements are reverted before being returned to the user. In our experience, generative models sometimes misbehave and return output not conducive to threat engagement, such as refusal to generate output due to moderation safeguards [8]. Any such outputs are sanitized before being returned to the user to avoid detection. The sanitizer handles both duties.

Cowrie's authentication is relaxed to allow almost all username and password combinations and maximize successful logins for each deployment. Using Cowrie's front end for all three deployments minimizes external factors such as transport or session layer detection that may introduce variability into the study's results. This ensures that any observed differences in honeypot performance are likely due to the effectiveness of the generated output in extending sessions rather than other factors outside the study's scope. All three output generators are deployed on Ubuntu 22.04 virtual machines with Internet connections. These connections are occasionally rotated to reduce variability related to a specific address. The described deployment is illustrated in Fig. 1.

IV. Evaluation

In this section, we analyze the data collected by the honeypots using the session length, token use, and attack types as the metrics. We use clustering on the observed sessions to find outliers that may indicate more advanced or targeted attacks. We study a unique session to show how generative honeypots can be used to find new TTPs.

A. Live Deployment Results

The three honeypots were deployed concurrently for 14 days from 9/19/2023 - 10/2/2023. During that time, 12,191 sessions with one or more inputs were captured: 6,559 were handled by Cowrie, 2,794 were handled by FEI-1, and 2,838 were handled by FEI-2. This uneven distribution in samples can be caused by several factors, including the time difference in detection by a scanning agent such as Shodan, chance encounter by a dedicated attacker, or repeated activity targeting that address [24]. However, since all three honeypots present the same

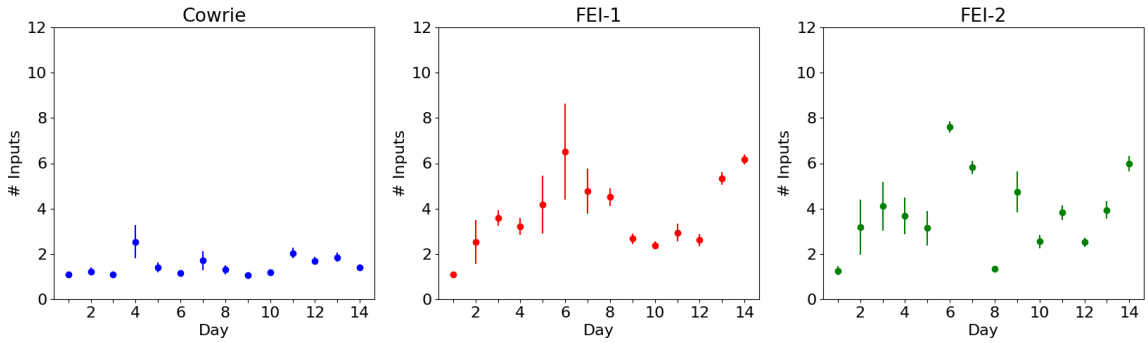

Fig. 2. Average Session Length by Day (The vertical bars represent 95% confidence intervals)

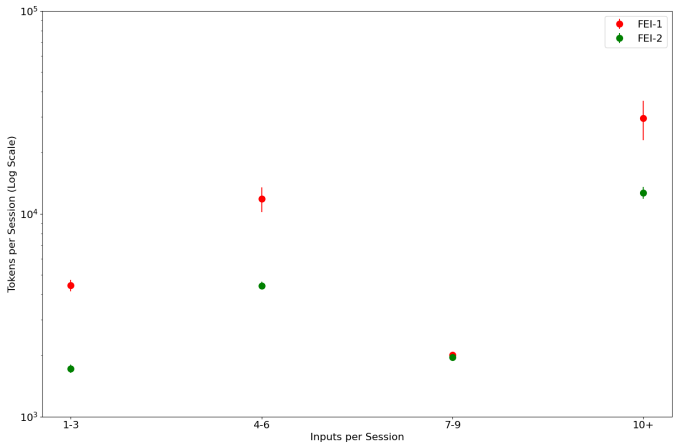

Fig. 3. Token Use by Session Length (The vertical bars represent 95% confidence intervals)

interface, an attacker cannot know which output generation method is used until an input is sent post-authentication. Therefore, we believe the captured sessions provide statistical relevance to the conclusions reached. Forwarder-honeypot assignments are changed five times in this period, ensuring the data captured from each deployment is not tied to the public address associated with a certain forwarder.

For base capability and deception, session length is measured in the number of inputs received. This is a standard metric for the evaluation of honeypots, as it provides insight into the effectiveness of the honeypot in luring and holding the attacker's attention [25, 20, 19]. Although attackers achieving their objectives may impact session length, this still offers some insight into the honeypot's effectiveness in engaging attackers.

To measure output generation efficiency and the effect of context preprocessing on model alignment, token usage by generative honeypots is measured. Since the model used for output generation, GPT3.5, is closed and remotely accessed via an API for this evaluation [5]. This means that the model's parameters and generation pipeline used for inference are unknown, limiting the evaluation of this study to input and output layers. The hidden implementation, API load, and network latency will all affect the generation time. Therefore, generation time is not a reliable metric for this evaluation using closed-source models and is left for when these metrics can be observed in a controlled environment with a local model. However, generation time can profoundly impact deception if not managed.

1) Capability and Deception: FEI-2 maintained sessions the longest with an average input length of 4.473 while FEI-1 had an average session length of 4.280 and Cowrie 1.348 over the two-week deployment, a 331% increase in average session length from Cowrie to FEI-2 and 4.5% increase from FEI-1 to FEI-2. In addition to having a longer average session length, FEI-1 and FEI-2 also observed a more significant variance in sessions. This is shown in their higher standard deviations, with a study-wide deviation of 3.254 and 3.687, respectively, as opposed to Cowrie's standard deviation of 1.535.

2) Generation Efficiency: We calculate the average token use per input of 1192.62 and 595.06 for FEI-1 and FEI-2, respectively. This is found in the average session token usage divided by the average session length. Therefore, with FEI-2 having a similar session length to FEI-1, we conclude that FEI-2's preprocessing method did not negatively affect session length.

While FEI-1 and FEI-2 had similar trends in token use based on the number of inputs per session, FEI-2 grew at a slower and more consistent rate, as evidenced by its narrow confidence intervals. For sessions with more than 10 inputs, the average token use was 29558 with a 95% confidence interval width of 6515.14 for FEI-1 and 12710 with a 95% confidence interval width of 871.51 for FEI-2. This is shown in Fig. 3 where FEI-1 has higher token use than FEI-2 for sessions of most input lengths. The outlier in the 7-9 input range is due to most sessions in that range having a similar attack pattern where the attempted inputs produced little output, thus having minimal effect on token usage. Sessions in this range are mostly found in cluster 1 in Fig. 4.

3) Session Clustering: There were 683 unique sessions in the 12,191 collected. Of these, 57 were seen by more than one honeypot, while the remaining 626 were seen by at most one of the three deployments. We include sessions seen by more than one deployment in clustering to fairly split the observed

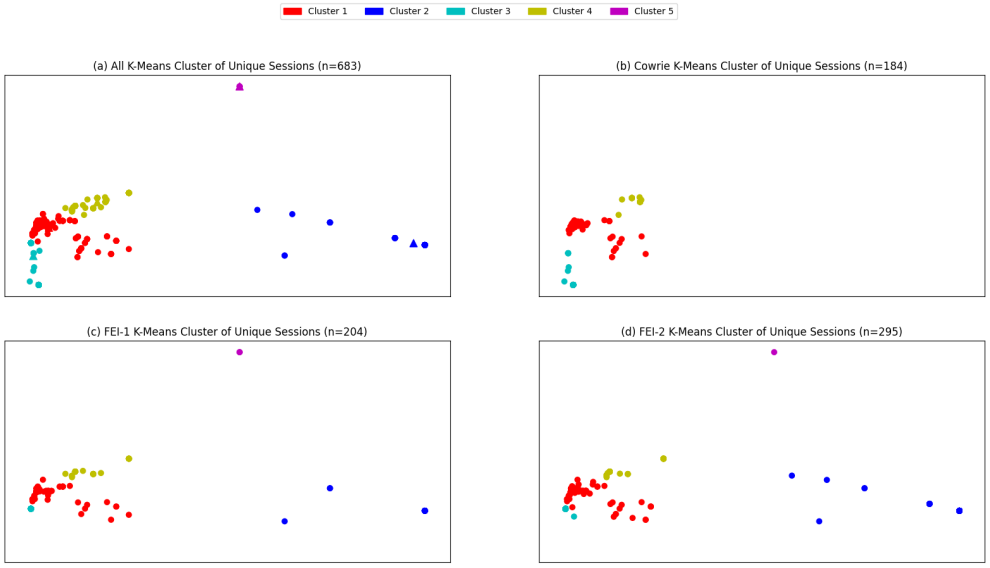

Fig. 4. K-Means Clustering of Unique Sessions (k = 5, v = 59)

sessions into subsets seen by each deployment. This makes the sample size used for clustering 683. Clustering of these sessions can be used to find outliers that could potentially be more advanced or targeted attacks.

K-Means clustering is performed using a 59-dimension vector, where each dimension represents the count of 58 uniquely observed commands from the session's input and a catch-all non-valid command. Fig. 4(a) shows the Principle Component Analysis (PCA) projection of the performed k-means clustering on a plane based on command presence and frequency in a session. Triangles represent the centroids of clusters.

Subsets of the clustered sessions are shown in Fig. 4(b)-(d), where each subset represents the sessions observed by that deployment. As shown in Fig. 4(a), most sessions were clustered to the lower left. This is more due to the stark difference in session content and session lengths of clusters 2 and 5 than to the similarities of clusters 1, 3, and 5.

Cluster 1 consists of mostly short sessions with unique commands executed only once or twice. Common commands include `uname`, `uptime`, `wget`, `lspci`, and `lspcu`. There is no discernable pattern of attack sequences in this cluster. The shortest command sequence in this cluster is 1. All three deployments had many sessions in this cluster, signaling minimal performance gain and TTP discovery potential from generative text model command output generation.

Sessions in cluster 2 all follow an attack sequence of system recon > process termination > process checking. This sequence checks for system information to be used for artifact retrieval and persistent processes from previous compromises, killing those processes if they exist. Some sessions in this cluster continue their attack while others end here. Those who stopped could have done so for a number of speculative reasons, such as the attacker achieving their desired outcome, dissatisfaction with returned output, or potential disruptions in the attacker's capabilities. The continuing sessions follow a repeating sequence of artifact retrieval > artifact execution > artifact removal that repeats one or more times. The filenames of the retrieved artifacts using a utility such as `wget` match the process names checked for in the first sequence. Cluster 2 was only observed by FEI-1 and FEI-2.

Cluster 3 also has a 2-part sequence structure common in all cluster members. The first part of sequences in this cluster comprises commands following a sequence of system check > shell change > check available utilities. This sequence checks for enabled built-in commands before changing shell environments. One reason to change shell environments may be to avoid command loggers or history files.

For sessions that continue after the reconnaissance sequence, the second part of the sequence is some variation of check web utility availability > check busybox utility availability > check binary contents of standard utility. The checking of binary contents of a standard utility tells the attacker what type of executable to download if they continue. Interestingly, most arguments sent with these commands in the input are minimal and primarily test system capability, not for their actual functionality in a malicious fashion. Cowrie had more sessions in this cluster than FEI-1 and FEI-2. The primary cause we could determine was a breakdown in the FEI when encountering multi-line inputs.

Cluster 4 comprises a couple of unique session patterns with similar goals. The first of these sessions uses commands to change attributes of existing files, most commonly key files, to enable persistence by adding their own SSH key for future sessions. A detailed reconnaissance is performed in no particular order, checking CPU info, memory usage, corn jobs, and filesystem usage. However, besides changing the compromised user's password, no further damage is caused. The

second unique pattern in this cluster also has a similar footprint, though reconnaissance is infrequently performed. This cluster had a relatively even number of sessions for each deployment, signaling minimal performance gain from generative text model command output generation.

Cluster 5 is unique, with only one member seen three times by FEI-1 or FEI-2 with unique arguments. Because of this, the following section discusses this cluster as a case study.

B. Case Study: SSH Persistence and Targeted Reconnaisance

In this session, persistence is achieved by installing an SSH authentication key. After which, an initial reconnaissance is conducted to determine system info, usage, network connections, and firewall status. From there, a more in-depth reconnaissance is conducted on specific system-related artifacts related to crypto-jacking, such as configuration files. Cryptojacking is an attack in which an attacker uses a victim's system to mine cryptocurrency without the victim's knowledge or consent [26].

Interestingly, all these commands are surrounded by echoed command tags, possibly a way for the attacker to verify the output of each command on the attacker's end before the next input is sent. This technique was not seen in other observed sessions. The attacker exited after their reconnaissance, indicating they were possibly checking the system for a prior compromise or were unsatisfied with the output. This session is unique due to the specific nature of the requested artifacts and the output verification. These sessions were observed only by the honeypots with model-based output generators. The input sequence is provided in our data repository [13].

For cases where the attacker is a returning user continuing from a previous compromise, using interactions from prior sessions to provide context for the current session may elicit new TTPs. Given how the model generates output based on an input sequence, a preprocessing mechanism to load interactions from previous sessions into the input sequence for a returning attacker can extend information gathering to multiple sessions. This mechanism would necessitate a way to fingerprint individual users so that the correct context is loaded. This method of context loading from prior sessions is easier using generative models than traditional honeypots due to the time needed to re-execute and the possibility of observation, i.e., malware loading, to reach the same context. Data for fingerprinting can include authentication, i.e., key usage, same username and password, source address, repeated unique command sequences, or timing of received input.

C. Discussion

Our live deployment shows notable improvements in session length when deploying a honeypot with a generative model handling output creation compared to a traditional rule-based output generation deployment. Additionally, a significant reduction in token usage is observed when preprocessing inputs to the generating model based on relevance to the most recent command without a dropoff in session length. This reduction saw token limit exceptions drastically decrease along with significant savings when using a pay-as-you-go model. Reducing token use allows models with smaller context windows to handle this use case, resulting in a lower-cost deployment.

Unique sessions were observed by the generative model honeypots that were not captured by the control deployment. Given the nature of a live deployment evaluation, the length of the study, and the singular observation point for each deployment, it's possible that the control honeypot was not given the opportunity to interact with those same sessions. However, the fact that FEI-1 and FEI-2 both observed these unique sessions and that the control had many more samples observed, we believe this is not the case. Therefore, we attribute the capturing of these outliers to the model's ability to generate believable output dynamically instead of using rule-based or pre-defined sequences.

V. Conclusions

The use of foundation generative models for prolonged conversations has allowed them to be used in more niche use cases. One such use case is the ability to emulate a Bash terminal. Their ability to generate output without evaluation makes them ideal candidates for use in enhancing low-risk honeypot interaction and threat engagement. Preliminary studies have examined the ability of these models for honeypot use, though none have been deployed for live interaction as a general-purpose shell terminal [7, 21]. To this end, the feasibility of using generative models to produce terminal output as a honeypot in a live deployment is examined. Two generative honeypot backends are deployed alongside Cowrie, a popular medium-interaction SSH honeypot [12]. These generative honeypot backends represent currently proposed works [7, 8].

Over a two-week concurrent deployment, we found that the average session length was extended by 331.8% when using generative models compared to traditional honeypot implementations. Additionally, we found that the evaluated context selection preprocessing slightly extended session length while greatly reducing token usage by 45.6% per request.

A unique session is examined to see how honeypots utilizing model-based output generation can be used to find unique TTPs and to discuss ways to improve preprocessing for threat engagement using generative text models. Such improvements include communication handling, inter-input system presentation modification, input authentication, and user fingerprinting. In these studies, we analyze the attacker's motivation and how generative model output can be used to steer the session so that the interaction time with an attacker and the amount of information learned can be maximized.

Current obstacles to deploying generative models for threat engagement include a lack of available training data for fine-tuning making larger foundation models the primary use. When such data becomes available, a model can be deployed locally and be modified per the deployment's needs [27].

The use of these models to improve honeypots has still only been preliminarily explored. The improvements mentioned will facilitate further studies into the feasibility of these models for

improving honeypots through threat engagement. Initial results from this live deployment evaluation show promise for using these models in production and live environments.

ACKNOWLEDGMENT

This research is partially supported by the National Security Agency through the awards H98230-21-1-0171 and H98230-20-1-0392 and the Army Research Office contract W911NF2110188. The opinions expressed, and any errors in the paper are those of the authors.

REFERENCES

[1] Javier Franco et al. "A survey of honeypots and honeynets for internet of things, industrial internet of things, and cyber-physical systems". In: *IEEE Communications Surveys & Tutorials* 23.4 (2021), pp. 2351–2383.

[2] Ashish Vaswani et al. "Attention is all you need". In: *Advances in neural information processing systems* 30 (2017).

[3] Bernardino Romera-Paredes and Philip Torr. "An embarrassingly simple approach to zero-shot learning". In: *International conference on machine learning*. PMLR. 2015, pp. 2152–2161.

[4] Hwanjo Heo and Seungwon Shin. "Who is knocking on the telnet port: A large-scale empirical study of network scanning". In: *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*. 2018, pp. 625–636.

[5] Long Ouyang et al. "Training language models to follow instructions with human feedback". In: *Advances in neural information processing systems* 35 (2022), pp. 27730–27744.

[6] Colin Raffel et al. "Exploring the limits of transfer learning with a unified text-to-text transformer". In: *Journal of machine learning research* 21.140 (2020), pp. 1–67.

[7] Forrest McKee and David Noever. "Chatbots in a honeypot world". In: *arXiv preprint arXiv:2301.03771* (2023).

[8] Jarrod Ragsdale and Rajendra V Boppana. "On Designing Low-Risk Honeypots Using Generative Pre-Trained Transformer Models With Curated Inputs". In: *IEEE Access* 11 (2023), pp. 117528–117545.

[9] Enrico Cambiaso and Luca Caviglione. "Scamming the Scammers: Using ChatGPT to Reply Mails for Wasting Time and Resources". In: *arXiv preprint arXiv:2303.13521* (2023).

[10] Christoforos Vasilatos et al. "LLMPot: Automated LLM-based Industrial Protocol and Physical Process Emulation for ICS Honeypots". In: *arXiv preprint arXiv:2405.05999* (2024).

[11] *Mitre ATT&CK*. URL: https://attack.mitre.org/ (visited on 05/03/2024).

[12] *Cowrie SSH/Telnet Honeypot*. URL: https://github.com/cowrie/cowrie/ (visited on 03/12/2024).

[13] Jarrod Ragsdale. *Datasets Used for Research on Evaluating Few-Shot Learning Generative Honeypots in a Live Deployment*. Version V1. 2024. DOI: 10.7910/DVN/UE8IXF. URL: https://doi.org/10.7910/DVN/UE8IXF.

[14] Lance Spitzner. *Honeypots: tracking hackers*. Vol. 1. Addison-Wesley Reading, 2003.

[15] Fred Cohen. *Deception ToolKit*. 1998. URL: http://all.net/dtk/ (visited on 04/01/2024).

[16] Iyatiti Mokube and Michele Adams. "Honeypots: concepts, approaches, and challenges". In: *Proceedings of the 45th annual southeast regional conference*. 2007, pp. 321–326.

[17] Ashish Girdhar and Sanmeet Kaur. "Comparative study of different honeypots system". In: *International Journal of Engineering Research and Development e-ISSN* (2012), pp. 23–27.

[18] Wenjun Fan, Zhihui Du, and David Fernández. "Taxonomy of honeynet solutions". In: *2015 SAI Intelligent Systems Conference (IntelliSys)*. IEEE. 2015, pp. 1002–1009.

[19] Moeka Yamamoto, Shohei Kakei, and Shoichi Saito. "Firmpot: A framework for intelligent-interaction honeypots using firmware of iot devices". In: *2021 Ninth International Symposium on Computing and Networking Workshops (CANDARW)*. IEEE. 2021, pp. 405–411.

[20] Volviane Saphir Mfogo et al. "AIIPot: Adaptive Intelligent-Interaction Honeypot for IoT Devices". In: *arXiv preprint arXiv:2303.12367* (2023).

[21] Muris Sladić et al. "LLM in the shell: Generative honeypots". In: *arXiv preprint arXiv:2309.00155* (2023).

[22] Ziyang Wang et al. "HoneyGPT: Breaking the Trilemma in Terminal Honeypots with Large Language Model". In: *arXiv preprint arXiv:2406.01882* (2024).

[23] Quchen Fu et al. "NL2CMD: An Updated Workflow for Natural Language to Bash Commands Translation". In: *arXiv preprint arXiv:2302.07845* (2023).

[24] *Search Engine for the Internet of Everything*. URL: https://www.shodan.io/ (visited on 05/01/2024).

[25] Tongbo Luo et al. "Iotcandyjar: Towards an intelligent-interaction honeypot for iot devices". In: *Black Hat* 1 (2017), pp. 1–11.

[26] Hugo LJ Bijmans, Tim M Booij, and Christian Doerr. "Inadvertently making cyber criminals rich: A comprehensive study of cryptojacking campaigns at internet scale". In: *28th USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 1627–1644.

[27] Lei Li et al. "Cascadebert: Accelerating inference of pre-trained language models via calibrated complete models cascade". In: *arXiv preprint arXiv:2012.14682* (2020).