LLM in the Shell: Generative Honeypots

1st Muris Sladić
Czech Technical University in Prague
Prague, Czech Republic
sladimur@fel.cvut.cz

2nd Veronica Valeros
Czech Technical University in Prague
Prague, Czech Republic
veronica.valeros@fel.cvut.cz

3rd Carlos Catania
CONICET, UNCuyo
Mendoza, Argentina
harpo@ingenieria.uncuyo.edu.ar

4th Sebastian Garcia
Czech Technical University in Prague
Prague, Czech Republic
sebastian.garcia@agents.fel.cvut.cz

2024 IEEE European Symposium on Security and Privacy Workshops (EuroS&amp;PW) | 979-8-3503-6729-4/24/$31.00 ©2024 IEEE | DOI: 10.1109/EUROSPW61312.2024.00054

***Abstract*—Honeypots are essential tools in cybersecurity for early detection, threat intelligence gathering, and analysis of attacker's behavior. However, most of them lack the required realism to engage and fool human attackers long-term. Being easy to distinguish honeypots strongly hinders their effectiveness. This can happen because they are too deterministic, lack adaptability, or lack deepness. This work introduces shelLM, a dynamic and realistic software honeypot based on Large Language Models that generates Linux-like shell output. We designed and implemented shelLM using cloud-based LLMs. We evaluated if shelLM can generate output as expected from a real Linux shell. The evaluation was done by asking cybersecurity researchers to use the honeypot and give feedback if each answer from the honeypot was the expected one from a Linux shell. Results indicate that shelLM can create credible and dynamic answers capable of addressing the limitations of current honeypots. ShelLM reached a TNR of 0.90, convincing humans it was consistent with a real Linux shell. The source code and prompts for replicating the experiments have been publicly available.**

***Index Terms*—Large Language Models, honeypots, shelLM**

1. Introduction

Honeypots allow security researchers and defense teams to detect, monitor, and log attacks on their systems and networks [1]. In theory, this detection information could be used for behavioral analysis of attackers, but in reality, it is mostly used for gathering threat intelligence and telemetry [2]. We believe that the main cause for this limited use is that (i) most attacks from the Internet are automated, and (ii) current shell-based honeypots are easy to identify for a human attacker [3]. Moreover, the process of successfully understanding, learning, and recognizing attackers' behaviors takes considerable effort and requires large amounts of data.

We study the problem of honeypots not being sufficiently good at mimicking system shells (such as Bash) to make human attackers believe long enough that they are interacting with a real system [4]. Current honeypots are usually too limited in possible actions, their file systems too generic, their answers too hard-coded, and the overall system does not look complex enough [5].

Our work proposes `shelLM`, a shell-based honeypot software using Large Language Models (LLMs). The aim of shelLM is to generate a credible and realistic Linux shell that is engaging for attackers and, therefore, may delay the realization that they are not interacting with a real Linux shell. LLMs can create all the needed information, from file system structure to on-demand file content.

`ShelLM` was evaluated in its capacity to generate output as is expected from a Linux shell. This was done by asking 12 cybersecurity experts to use the system and give feedback. Results show a 0.90 TNR, meaning that 90% of the time, the answers were assessed to be consistent with a Linux shell. We show that by using good, prompt engineering techniques, we can use LLMs to create realistic honeypot systems.

The key features of `shelLM` are: (i) the content of a session is always transferred into the new session of the same user to keep future consistency, (ii) the use of the chain-of-thought prompt technique [6], and (iii) the use of prompts with precise instructions to avoid certain pitfalls.

This research has two main contributions: first, it presents human-tested empirical evidence that supports the use of LLMs for building credible honeypots; second, the publication of an LLM-based honeypot software.

2. Related work

In the field of cyber-defense, honeypots are used as deception tools to attract potential attackers and to keep them occupied and away from other devices while gathering information about their techniques and methods [7]. Moreover, they are effective tools for detecting insider threats [8]. Based on their level of interaction, honeypots are usually classified as low-interaction, high-interaction [9], or medium-interaction [10]. Cowrie is a well-known medium-interaction honeypot used by many organizations [11]. It is designed to log attacks and shell interactions the attacker performs by emulating a Linux system. However, Cowrie is quite static since it follows a configuration file.

The application of artificial intelligence and Natural Language Processing (NLP) techniques in the context of honeypot generation is an emerging research topic. In particular, the dynamic capabilities of AI techniques seem to be adequate for dealing with the intrinsic variation of attacker behaviors. For instance, the authors of [12] employ a Reinforcement Learning (RL) approach to design a self-learning honeypot. The RL-based honeypot can adapt its strategy by interacting with attackers, providing a more dynamic defense mechanism that learns from each encounter. This paves the way for more adaptive systems to better understand and counter evolving cyber threats. Furthering the concept of adaptability, the study in [13] proposes the creation of self-adapting honeypots based on a game-theoretical approach between the attacker and the honeypot. This offers a more complex and interactive defensive landscape, making it difficult for attackers to identify and bypass the honeypot.

More recently, researchers applied NLP techniques to extract information from honeypot logs. The work of [14] employs various NLP techniques to cluster different types of cyber attacks, providing valuable insights into attacker behaviors and tactics. Such information can then be used to fine-tune honeypot configurations and improve the overall cybersecurity posture. A similar approach is presented in [15], where a fine-tuned LLM (GPT-2C) model parses dynamic logs generated by Cowrie. This fine-tuned model aids in real-time analysis and threat identification, showcasing how machine-learning algorithms can enhance the effectiveness of honeypots.

There are only a few studies on applying NLP models for implementing the content of honeypots. In a preliminary analysis [16], authors propose an approach for interfacing with ChatGPT, demonstrating how to formulate prompts that can mimic the behavior of Linux shell commands on various operating systems. More recently, a new LLM-based web honeypot software called Galah [17] was introduced, with abilities to dynamically respond to arbitrary HTTP requests.

3. Methodology and Implementation

We created and deployed a honeypot software called `shelLM` using Large Language Models. `shelLM` is implemented in Python and uses a cloud-based LLM. It is designed to work seamlessly with a typical SSH setup, facilitating user interactions via this connection.

The process involved: (i) designing specific prompts and applying well-known prompt engineering techniques, (ii) implementing the honeypot software using these prompts, (iii) connecting `shelLM` to an actual SSH server, (iv) designing experiments, and (v) asking security researchers to evaluate `shelLM`.

3.1. Prompt Engineering Technique

To guide the behavior of `shelLM`, a cloud-based LLM is given an initial prompt at the beginning of each user's session. This initial prompt contains a *personality prompt* and a *thinking process prompt*. The *personality prompt* instructs the LLM to be (i) precise, (ii) realistic, (iii) not to disclose its internal details, (iv) describes the expected behavior of a Linux shell, and (v) provides examples of desired output under certain situations. The *thinking process prompt* instructs the LLM to *'think step-by-step'* following the Chain of Thought (CoT) prompt technique [6], in combination with the *few-shot technique* [18], and repeated orders to enforce this behavior.

3.2. Session Management

A session in `shelLM` contains all the inputs and outputs between a user and `shelLM` but also involves a prompt engineering technique. The operation is similar to the approach used for building chat-bots on top of an LLM, as shown in Figure 1.

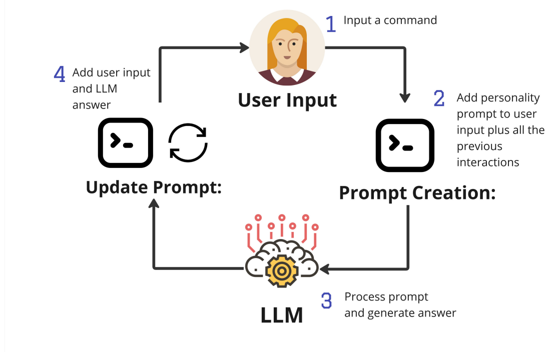

1 Input a command
User Input
2 Add personality prompt to user input plus all the previous interactions
Prompt Creation:
3 Process prompt and generate answer
LLM
4 Add user input and LLM answer
Update Prompt:

Figure 1. Prompt construction during a shelLM session

The session begins with the user interacting with the LLM. This is the initialization point where the *personality prompt* is first introduced. Then, a prompt engineering process is repeated for each new input until the session ends. The prompt construction process consists of:

1) **User Input:** The user inputs a command.
2) **Prompt Creation:** This prompt integrates the original personality prompt, the history of all *previous interactions* (both user inputs and responses), and the new user command.
3) **LLM Processing:** The LLM processes this comprehensive prompt and generates a response.
4) **Update Prompt:** The user command and the LLM response are then added to the history of *previous interactions*.

3.3. Model and Consistency Techniques

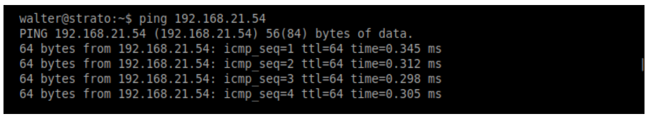
The LLM model used was GPT-3.5-turbo-16k from OpenAI [19]. The model used a temperature of 0, a maximum of 800 tokens per response, and the rest of the parameters left by default.

To keep consistency between multiple sessions from the same user, the complete content of each session is saved to an external file and used as part of the initial prompt for the next session from the same attacker. Attackers are identified by their IP addresses and login usernames, so upon reconnection, the same attacker can continue the interaction in the same state. This means, for example, that if an attacker creates a file in the honeypot and later wants to access that file, they will be able to do so, even if they reconnect the next day. Due to the non-deterministic nature of LLMs sometimes the same command might not return the same response, which can be beneficial in some cases. For instance, some commands rely on varying responses each time they are executed. and this variability is useful for simulating the conditions observed in real active environments with multiple users. However, for most commands, the model is prompted to generate the same output every time.

The initial prompt and the whole history are saved as part of the context for the LLM. If the context reaches the maximum of 16k tokens, then the history is deleted, restoring the initial personality prompt. In our experiments, the context took a long time to be filled and didn't present an issue for the attackers. However, in future work, we will deal with the history management in a better way.

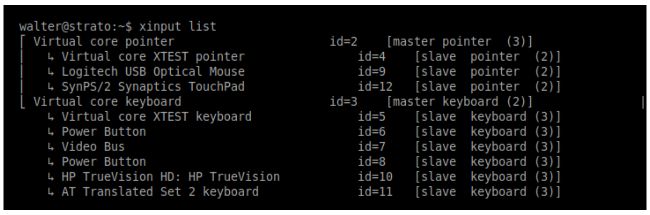
4. Demonstrating shelLM in Action

An example output generated by `shelLM` can be observed in Figure 2. The Figure shows the attacker executing the `ping` command and `shelLM` automatically generating the output in real-time, showing how `shelLM` emulates network connectivity. The output of ping is printed line by line to simulate delays. Figure 3 shows the output of the `xinput` command, mimicking input device configurations. Figure 4 shows the attacker issuing the `wget` command, illustrating the honeypot capabilities for replicating network interactions with other servers. These examples help to highlight the precision and variability with which the LLM honeypot simulates genuine system behaviors. Furthermore, these commands are not supported by standard honeypots, such as Cowrie, and this makes a major differentiation between them.



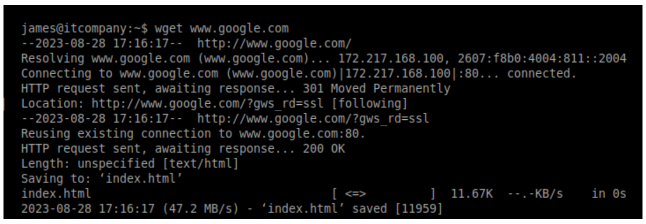
```
walter@strato:~$ ping 192.168.21.54
PING 192.168.21.54 (192.168.21.54) 56(84) bytes of data.
64 bytes from 192.168.21.54: icmp_seq=1 ttl=64 time=0.345 ms
64 bytes from 192.168.21.54: icmp_seq=2 ttl=64 time=0.312 ms
64 bytes from 192.168.21.54: icmp_seq=3 ttl=64 time=0.298 ms
64 bytes from 192.168.21.54: icmp_seq=4 ttl=64 time=0.305 ms
```

Figure 2. ShelLM output for `ping` command.



```
walter@strato:~$ xinput list
⎡ Virtual core pointer                      id=2    [master pointer  (3)]
⎜   ↳ Virtual core XTEST pointer                id=4    [slave  pointer  (2)]
⎜   ↳ Logitech USB Optical Mouse                id=9    [slave  pointer  (2)]
⎜   ↳ SynPS/2 Synaptics TouchPad                id=12   [slave  pointer  (2)]
⎣ Virtual core keyboard                     id=3    [master keyboard (2)]
    ↳ Virtual core XTEST keyboard               id=5    [slave  keyboard (3)]
    ↳ Power Button                              id=6    [slave  keyboard (3)]
    ↳ Video Bus                                 id=7    [slave  keyboard (3)]
    ↳ Power Button                              id=8    [slave  keyboard (3)]
    ↳ HP TrueVision HD: HP TrueVision           id=10   [slave  keyboard (3)]
    ↳ AT Translated Set 2 keyboard              id=11   [slave  keyboard (3)]
```

Figure 3. `shelLM` output for the `xinput` command.



```
james@itcompany:~$ wget www.google.com
--2023-08-28 17:16:17--  http://www.google.com/
Resolving www.google.com (www.google.com)... 172.217.168.100, 2607:f8b0:4004:811::2004
Connecting to www.google.com (www.google.com)|172.217.168.100|:80... connected.
HTTP request sent, awaiting response... 301 Moved Permanently
Location: http://www.google.com/?gws_rd=ssl [following]
--2023-08-28 17:16:17--  http://www.google.com/?gws_rd=ssl
Reusing existing connection to www.google.com:80.
HTTP request sent, awaiting response... 200 OK
Length: unspecified [text/html]
Saving to: ‘index.html’
index.html                                  [ <=>          ]  11.67K  --.-KB/s    in 0s
2023-08-28 17:16:17 (47.2 MB/s) - ‘index.html’ saved [11959]
```

Figure 4. `shelLM` output of a `wget` command.

`ShelLM` is published on GitHub at https://anonymous.4open.science/r/shelLM. The tool requires a valid OpenAI API key for GPT-3.5-turbo-16k.

Setup of Evaluation Experiments

Since **the goal of this research is to investigate whether LLM-based shell honeypots can generate outputs indistinguishable from those of a real Linux shell, and not to assess participants' ability to detect a honeypot**, the evaluation methodology has to be carefully constructed. Our approach was to tell the evaluation participants that they were connecting to a honeypot but to ask them if the answer from the honeypot looked realistic or not and why. This way a small bias is introduced, but it was possible to understand if and why the output looked convincing.

We contacted 12 participants by email, gave them unique users and passwords, and gave them the following requirement: *Your goal in this experiment is to identify which output of the system helps you detect it as a honeypot. Each command is going to give an output and you need to explain how each output helps or does not help you identify the system as a honeypot. For each command please copy it in the email and briefly explain why it did or did not help you realize it is a honeypot.*

Answers were manually processed to obtain a list of input commands, their outputs, and the participant's answer. To verify the results, we consulted with three Linux cybersecurity experts, who checked the correctness of the output and manually evaluated each response. Figure 5 shows the evaluation diagram.

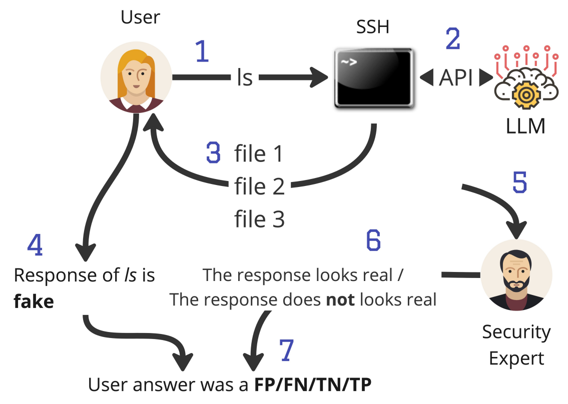

User
SSH
1
ls
2
API
LLM
3
file 1
file 2
file 3
5
4
Response of *ls* is
fake
6
The response looks real /
The response does **not** looks real
Security
Expert
7
User answer was a **FP/FN/TN/TP**

Figure 5. Evaluation of `shelLM`. Participants interact and send answers with their assessment. Security experts evaluate the outputs of `shelLM` and the answers, determining if it was a FP/FN/TN/TP.

4.1. Type Error Interpretation

The error types we used have a very unique interpretation given the particular nature of our evaluation. Recall that **the goal of our experiments was not to evaluate if a participant could identify a honeypot or not; but to evaluate if the participant, knowing that it is a honeypot, could successfully identify what parts of the answers disclose the presence of a honeypot.**

The interpretation of each error type for our case is:

- True Positive (**TP**): Participants say that the command response does not match the expected output of a Linux shell (positive for honeypot), and the expert agrees the command response is not consistent with a real Linux shell (true).
- False Positive (**FP**): Participants say that the command response does not match the expected output of a Linux shell (positive for honeypot) but the

expert disagrees and considers the output to be consistent with a Linux shell (false).

- False Negative (**FN**): Participants say that the command response does match the expected output of a Linux shell (negative for honeypot), but the expert believes that the response was not consistent with the output of a real Linux shell. This is considered beneficial for the honeypot system.
- True Negative (**TN**): Participants say that the command response does match the expected output of a Linux shell (negative for honeypot) and the expert agrees that it was consistent with the output of a Linux shell (true).

The following example shows a true positive, in which a participant believed the response was not consistent with the output of a Linux shell and identified the system as a "honeypot", and experts confirmed that it was a clear LLM-introduced error.

```
jennifer@itcompany:~$ w
bash: syntax error near unexpected
token ‘newline’
```

The example below shows a false positive, in which a participant identified the system as a "honeypot", but experts confirmed that it is common that the '.bashrc' file does not exist in systems that use other interpreters, for example, in Linux-based IoT devices. This answer may be an artifact of telling the participants they would be inside a honeypot.

```
jennifer@itcompany:~$ cat .bashrc
cat: .bashrc: No such file or directory
```

Another example of a false positive was the command shown below, in which the participant identified this command as not consistent with the output of a real Linux shell and classified it as **a honeypot** due to the small number of files. However, many small systems have this number of files or fewer.

```
jennifer@itcompany:~$ ls /var/run
acpid.pid crond.pid
dhclient-eth0.pid  dhclient.pid
initramfs motd sshd.pid
sudo utmp wpa_supplicant.pid
```

4.2. Metrics for Evaluating Honeypot Software

Following the error interpretation described in subsection 4.1, the following metrics are considered for evaluating the honeypot software performance:

Overall Accuracy: indicates the number of command responses correctly identified by the participants, over all the command responses received by all the participants.

False Negative Rate: It is the ratio between the number of answers that wrongly said it did match a Linux shell over all the answers that were genuinely generated matching a Linux shell. It indicates that the honeypot is effective at masquerading as a Linux shell system.

True Negative Rate (Specificity): It is the ratio between the number of answers that **correctly** said it did match a Linux shell over all the answers that really matched a Linux shell. It indicates how effective the answers are to appear as a real Linux shell.

False Discovery Rate: It is the ratio between the number of answers that **correctly** said they did not match a real Linux shell over the total number of times the users said it did not match a real Linux shell. It is an important metric to minimize because it measures the number of times the honeypot is *correctly* identified as not matching a real Linux shell.

5. Results

A total of 12 human participants tested `shelLM`. Participants executed 226 commands associated with package management, file system management, network operations, system information, and file management. Only 76 unique commands were executed. The top ten commands executed were: `cat`, `ls`, `sudo`, `get`, `echo`, `pwd`, `nano`, `ping`, `ssh`, and `whois`. On average, each participant executed 19 commands, with a minimum of 12 and a maximum of 38 commands.

The results of the experiments are summarized in Figure 6. The analysis showed that 74% (167) of the commands that seemed credible to participants had realistic outputs (True Negative Rate). A total of 7.5% (17) of commands flagged as honeypot indicators were false positives and could be attributed to the knowledge of participants that they are in a honeypot system (False Discovery Rate). An 18% (41) of commands flagged as honeypot indicators produced incorrect or strange output (True Positives). The remaining 0.5% (1) of commands were false negatives, in which participants missed noticing inconsistencies in the model output.

```
                Experts
Participants REAL FORGED
REAL          167       1
FORGED         17      41

             Accuracy : 0.9204
               95% CI : (0.877, 0.9521)
  No Information Rate : 0.8142
  P-Value [Acc > NIR] : 5.432e-06

  False Negative Rate  : 0.0232
  True Negative Rate   : 0.9076
  False Discovery Rate : 0.0923
  'Positive' Class : REAL
```

Figure 6. Confusion Matrix and Performance metrics for the LLM-based honeypot.

For our purposes, a good honeypot would have True Negatives and False Negatives since we would need participants to say *negative*, meaning the responses would *not reveal a honeypot*. This is different from other detection methods, where what is expected is mostly a large number of true positives. If we look at the results per participant shown in Table 1), we can confirm that the True Negative Rate was 1 in most cases. This indicates that the honeypot effectively convinces participants that the output is consistent with a Linux shell. On the other hand, `Participant 9` is a notable outlier with a TNR of 0.5, suggesting the output was only partially successful in looking like a real Linux shell for this participant.

The FNR values are generally low, which demonstrates the system's effectiveness.

Finally, when considering the overall accuracy, most participants have an accuracy above 0.9, showcasing again the general effectiveness of the honeypot.

TABLE 1. Performance metrics of the LLM honeypot per participant

Participant	Accuracy	FNR	TNR	FDR
1	0.643	0.417	1	0
2	0.941	0.071	1	0
3	1	0	1	0
4	0.882	0.2	1	0
5	1	0	1	0
6	0.923	0.090	1	0
7	0.923	0.105	1	0
8	0.933	0.090	1	0
9	0.938	0	0.5	0.5
10	0.867	0.167	1	0
11	0.75	0.273	1	0
12	1	0	1	0

Widely common commands such as `ls`, `pwd`, and `whoami` always returned the expected output. Similarly, commands including `ping`, `wget`, `whois`, `tcpdump`, `sudo -su` and non-valid command inputs resulted in credible output. The issues that occurred were inconsistencies between files in `/proc` and `/etc` directories, possibly caused by the limitation on the number of tokens and the occasional strange behavior of the LLM. This strange behavior of the model can be attributed to its context getting bigger. As it was observed recently [20], Large Language Models perform the best when the relevant information is at the beginning or the end of the input context. When the relevant information is in the middle of the input context, the model's performance tends to degrade.

6. Cost analysis

A primary constraint with the use of `shelLM` is the cost involved in the generation process. Since the current implementation is based on the OpenAI GPT-3.5-turbo-16k model that has high financial resource requirements, the use of this technology in every scenario should be carefully considered. The cost of generating responses using GPT-3.5-turbo-16k can add up quickly, especially when processing large volumes of text.

At the time of the experiments, the cost for the GPT-3.5-turbo-16k model was \$0.003 for 1k input tokens and \$0.004 for 1k output tokens [21]. The cost estimate for one full `shelLM` session of 16k tokens is around \$0.70, from which \$0.658 can be attributed to the input tokens and \$0.042 to the output tokens.

In Table 2, we present a comprehensive cost analysis for each of the 12 participants in the experiments. The table presents, for each participant, the total token count, input and output costs, and total session duration.

The average duration participants spent interacting with `shelLM` was around 35 minutes. Based on this data, we can say that the cost of usage of shelLM is around \$0.4 per 30 minutes of active use, which is roughly \$0.8 per hour. The total cost of our experiment was \$5.29, out of which \$4.99 was for input tokens and \$0.30 for output tokens. Since shelLM was designed to be mainly used inside a private network, therefore the number of expected attackers is considerably low.

The initial personality prompt was a fundamental element for simulating a Linux shell. Since it was included in the prompt of all the participants of the experiments, we analyzed its costs independently. The initial personality prompt was composed of 2138 tokens, and its cost was \$0.0065.

Based on this cost calculation, we estimate that the costs are not prohibitive. Also, in the future costs of using LLMs might decrease even further. However, researchers must be aware that an abuse of the cost of shelLM can happen if the system uses too many tokens, so we recommend a limit on the budget of the account used by shelLM.

TABLE 2. Costs for running the experiment per participant

Participant	Tokens	Input \$	Output \$	Time [m]
1	7243	0.2939	0.0189	47
2	7455	0.3120	0.0197	27
3	9572	0.4224	0.0252	24
4	12731	0.5616	0.0355	26
5	14365	0.6334	0.0379	44
6	9098	0.4012	0.0241	31
7	12306	0.5418	0.0324	42
8	5253	0.2317	0.0138	34
9	9586	0.4225	0.0253	55
10	3963	0.1748	0.0104	12
11	12831	0.5654	0.0338	53
12	9755	0.4295	0.0257	29

7. Conclusion and future work

We presented `shelLM`, a novel application of LLMs for shell-based honeypots, resulting in a system that dynamically generates synthetic data as demanded by the user. The core of `shelLM` was implemented using several LLM prompt engineering techniques. Twelve security experts tested the system, and three security experts evaluated the honeypot's correspondence with a real Linux shell. During this evaluation, `shelLM` obtained a TNR of 90%. The results confirmed our hypothesis that it was possible to create an LLM honeypot that humans find hard to distinguish from a real system.

Despite the encouraging preliminary results, we are aware of the limitations of our current implementation. In particular, the occasional strange behavior of the LLM is caused by its inherent stochastic nature and its memory issues related to the size of the input context [20]. Answer latency due to the API responsiveness and the response generation time are two other limitations of the current implementation. This might be an indicator of a honeypot, however, not strong enough to make a definite conclusion, since there exist IoT devices, that have slower response times as well.

Future work will improve LLM-based honeypot responsiveness, engagement, and error management. We plan to use local LLM models and to fine-tune the models to attempt to remediate forgetfulness and behavior deterioration issues with a larger context. In addition, we intend to compare `shelLM` with other well-known honeypots to determine its appeal to attackers and the quality of data it produces. More experiments will be run where the participants would not know that they are being tested in the recognition of honeypots to measure their growing suspiciousness. Finally, we plan to conduct more experiments focused on differentiating human from bot behaviors within the honeypot and identify their respective signals. Since automated attacks are a common practice we plan to asses how effective *shelLM* is in those cases and compare it with the effectiveness against human attacks. We also plan to further understand its behavior against actual attacks by deploying it in the wild.

References

[1] N. Ilg, P. Duplys, D. Sisejkovic, and M. Menth, "A survey of contemporary open-source honeypots, frameworks, and tools," vol. 220, p. 103737. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S108480452300156X

[2] Katarzyna Gorzelak, Tomasz Grudziecki, Paweł Jacewicz, Przemysław Jaroszewski, Łukasz Juszczyk, and Piotr Kijewski, "Proactive Detection of Security Incidents."

[3] I. Mokube and M. Adams, "Honeypots: Concepts, approaches, and challenges," in *Proceedings of the 45th Annual Southeast Regional Conference*, ser. ACM-SE 45. New York, NY, USA: Association for Computing Machinery, 2007, p. 321–326. [Online]. Available: https://doi.org/10.1145/1233341.1233399

[4] T. Holz and F. Raynal, "Detecting honeypots and other suspicious environments," in *Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop*, 2005, pp. 29–36.

[5] S. Morishita, T. Hoizumi, W. Ueno, R. Tanabe, C. Gañán, M. J. van Eeten, K. Yoshioka, and T. Matsumoto, "Detect me if you… oh wait. an internet-wide view of self-revealing honeypots," in *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, 2019, pp. 134–143.

[6] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. Chi, Q. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," 2023.

[7] J. Yuill, F. Wu, J. Settle, F. Gong, R. Forno, M. Huang, and J. Asbery, "Intrusion-detection for incident-response, using a military battlefield-intelligence process," *Computer Networks*, vol. 34, no. 4, pp. 671–697, 2000.

[8] L. Spitzner, "Honeypots: catching the insider threat," in *19th Annual Computer Security Applications Conference, 2003. Proceedings.*, 2003, pp. 170–179.

[9] N. Provos *et al.*, "A virtual honeypot framework." in *USENIX Security Symposium*, vol. 173, no. 2004, 2004, pp. 1–14.

[10] G. Wicherski, "Medium interaction honeypots," *German Honeynet Project*, 2006.

[11] M. Oosterhof, "Cowrie," 2014. [Online]. Available: https://github.com/cowrie/cowrie/

[12] A. Pauna and I. Bica, "Rassh - reinforced adaptive ssh honeypot," in *2014 10th International Conference on Communications (COMM)*, 2014, pp. 1–6.

[13] G. Wagener, R. State, T. Engel, and A. Dulaunoy, "Adaptive and self-configurable honeypots," 05 2011, pp. 345–352.

[14] M. Boffa, G. Milan, L. Vassio, I. Drago, M. Mellia, and Z. Ben Houidi, "Towards nlp-based processing of honeypot logs," in *2022 IEEE European Symposium on Security and Privacy Workshops (EuroS'I&'PW)*, 2022, pp. 314–321.

[15] F. Setianto, E. Tsani, F. Sadiq, G. Domalis, D. Tsakalidis, and P. Kostakos, "Gpt-2c: a parser for honeypot logs using large pre-trained language models," 11 2021, pp. 649–653.

[16] F. McKee and D. Noever, "Chatbots in a honeypot world," 2023.

[17] A. Karimi, "galah," 2024. [Online]. Available: https://github.com/0x4D31/galah

[18] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," *CoRR*, vol. abs/2005.14165, 2020. [Online]. Available: https://arxiv.org/abs/2005.14165

[19] O. AI, "GPT 3.5," https://platform.openai.com/docs/models/gpt-3-5, 2023, [Online; accessed 22-July-2023].

[20] N. F. Liu, K. Lin, J. Hewitt, A. Paranjape, M. Bevilacqua, F. Petroni, and P. Liang, "Lost in the middle: How language models use long contexts," 2023.

[21] O. AI, "Pricing," https://web.archive.org/web/20230718183142/https://openai.com/pricing, 2023, [Online; accessed 9-December-2023].