Natural Language Processing (NLP)

Alessandro Rosario Torrisi - s330732, Simone Sambataro - s331812,
Floralucy Cusano - s334142

Before starting, we produced 2 notebook, in, we kept the session as they were, while in , for each session we truncated every word longer then 30 char. The consideration about the not truncated session are in the extra section.

1 Task 1: Dataset Characterization

- **Explore the labels: How many different tags do you have? How are they distributed (i.e., how many bash words are assigned per tag)? Plot a barplot to show the distribution of tags** $\rightarrow$ Our analysis of the dataset, as presented in Figure 1, reveals the distribution of tags. The most prevalent tag is "Discovery," which aligns with expectations given that the data originates from a honeypot. Honeypots are strategically deployed within IT environments to observe the actions of potential attackers. Their visibility to attackers is intentionally high, allowing us to capture and understand their patterns and tactics. More precise number are present in the code.

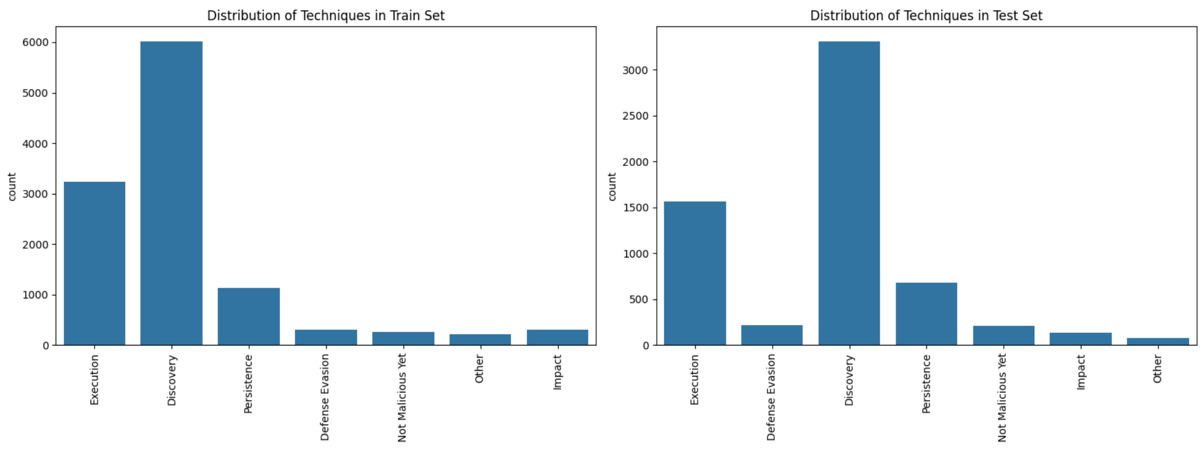

Figure 1: Bar Plot of all the Mitre Tags

We even went a little further and tried to analyze the p-value and ξ distribution.

  - The proportions of the tags in the train vs. the test are compatible (ξ^2 p $= 0.435 < 0.05$, which means there are no significant differences).
  - The duration of the sessions in the train versus in the test follows the same distribution (KS D $= 0.124$, p $= 0.176 > 0.05$, which means there are no significant differences).

- **Explore a single bash command– 'echo'**: How many different tags are assigned? How many times per tag? Can you show 1 example of a session where 'echo' is assigned to each of these tactics: 'Persistence', 'Execution'. Can you guess why such examples were labeled differently? $\rightarrow$ Our analysis of the dataset revealed the distribution of observed attack tags and their corresponding counts related to the echo command. The tag **Persistence** emerged as the most frequently observed category, with 104 instances, highlighting its relevance in scenarios where `echo` is used to write into files or set variables, leaving traces in the system. The **Execution** tag was assigned 39 times, while **Discovery** appeared 31 times. Less frequent tags include **Not Malicious Yet** (8 instances), **Impact** (6 instances), and **Other** (4 instances).

In the Bash session, we pinpoint an example in the notebook where echo is labeled as "Execution" and twice as "Persistence.

- Regarding the First instance (Execution), it looks like we are trying to print the current working directory, and that could be the reason why the label is execution.
- While for Persistence it seems like we are trying to put attacker's public key into 'authorized_keys' ensuring long term access to the machine.

- **Explore the Bash words: How many Bash words per session do you have? Plot the Estimated Cumulative Distribution Function (ECDF) →**

Looking at the left hand (linear-scale) ECDF in Figure 2, the curve climbs steeply: half of the

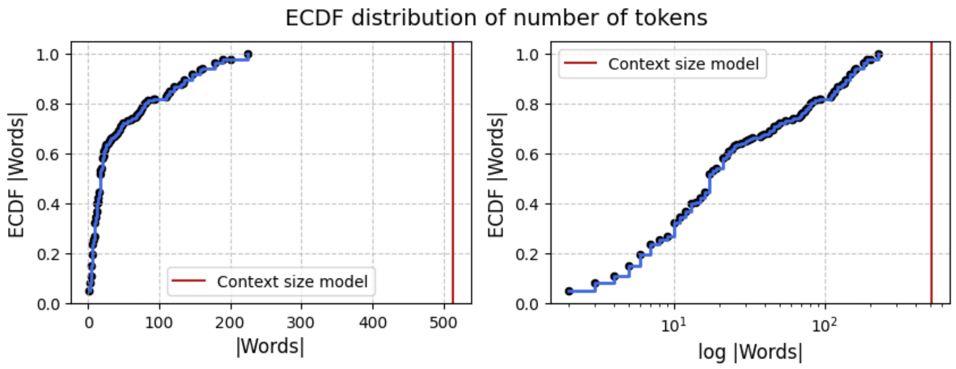

Figure 2: ECDF of words

SSH sessions end before the 30-word mark, three-quarters finish by ≈ 70 tokens, and 90% are done by ≈ 140 word. The curve flattens out and reaches 1.0 just above 225 words, meaning the very longest session in the whole set is still only about 230 words.

The right hand plot, shown on a $\log_{10}$ x-axis, portrays the same events from an alternative angle. Because the x-axis is logarithmic, a more or less straight diagonal indicates a long-tail process: a few sessions do stretch past 150-180 tokens, but the increase in the ECDF remains gradual and there is no sudden spike of extreme outliers. Even when the scale is expanded all the way to 300 words ($\log_{10} \approx 2.5$), the curve has already plateaued, confirming that 230 words is effectively the maximum.

However, it's important to note that the ECDF of bash words per session is irrelevant to completing the model's training and is included solely for illustrative purposes.

2 Task 2: Tokenization

- **Tokenize the following list of SSH commands: [cat, shell, echo, top, chpasswd, crontab, wget, busybox and grep]. Q: How do tokenizers divide the commands into tokens? Does one of them have a better (lower) ratio between tokens and words? Why are some of the words held together by both tokenizers?**

For a more detailed understanding of how the two tokenizers segment different words, we encourage you to examine the accompanying notebook.

When processing the example `"session"` command, BERT's tokenizer generated 16 sub-tokens, while UniXcoder's produced 14. This discrepancy stems from BERT's WordPiece model being trained exclusively on general English text. As a result, BERT tends to split rare or compound words into familiar English components. For instance:

- `chpasswd` is broken down into `ch`, `##pass`, and `##wd` (and occasionally even `cr`).
- `busybox` becomes `busy` and `##box`.

These splits reflect common English morphemes rather than shell syntax. This can also lead to misinterpretations, such as BERT recognizing `cat` as the animal instead of the Unix utility.

In contrast, UniXcoder, having been trained on extensive code datasets, produces fewer, more cohesive tokens for these terms, reflecting a better understanding of code specific terminology.

- **Q: How many tokens does the BERT tokenizer generate on average for the whole corpus? How many with the Unixcoder? Why do you think it is the case? What is the maximum number of tokens per bash session for both tokenizers?**

  Our analysis yielded the following token statistics for each model:

  - **BERT-base average tokens**: 176.59
  - **UniXcoder-base average tokens**: 407.27
  - **BERT-base maximum tokens produced**: 1887
  - **UniXcoder-base maximum tokens produces**: 28918

  From these results, it is clear that UniXcoder-base, on average, produces significantly more tokens per input compared to BERT-base, and also exhibits a considerably higher maximum token count. The observed disparity in token counts can be attributed primarily to the distinct pretraining methodologies of the two models:

  - **UniXcoder's Multimodal Pre-training**: UniXcoder was pretrained on a diverse corpus encompassing both natural language and various types of source code, including Bash scripts. This exposure to programming related patterns allows its tokenizer to better capture the granularity and structure inherent in code. Consequently, it often breaks down complex commands, long strings, or intricate scripting logic into a greater number of fine-grained sub-tokens that are semantically meaningful within a coding context.
  - **BERT's Natural Language Focus**: In contrast, BERT was trained exclusively on general English text. Its WordPiece tokenizer is optimized for natural language and tends to combine sequences of characters into larger chunks that are common in human language. When encountering programming constructs or unusual character sequences found in code, BERT's tokenizer may not split them as finely, often treating them as less common "words" or combinations thereof, resulting in fewer overall tokens for the same input.

  Furthermore, our dataset contains exceptionally long shell script snippets, particularly those involved in decoding and executing Base64-encoded payloads (which are often indicative of malicious activity). In these cases:

- **How many sessions would currently be truncated for each tokenizer?** → with Bert there would be 24 truncated session, while for Unix there would be 29 (Note: we matched the UnixCoder tokenizer's maximum length to BERT's 512 tokens to ensure a fair comparison in subsequent tasks).

- **Select the bash session that corresponds to the maximum number of tokens. Q: How many bash words does it contain? Why do both tokenizers produce such a high number of tokens? Why does BERT produce fewer tokens than Unixcoder**

  - **How many Bash words?**
    There are 134 tokens, but if we split that one line snippet on whitespace and then discard the purely shell operator tokens (such as `||`, `&&`, `|`, `;`, `>`, etc.), we end up with **97 actual "words"** (i.e., command names, arguments, paths, literals, etc.).
  - **Why so many tokens in both tokenizers?**
    Both BERT and UnixCoder use subword tokenization, so every little piece of that giant Base64 blob (and every path, flag, punctuation mark, etc.) gets chopped into its own tokens or subtokens. A few hundred shell words can easily become **thousands** of subword tokens once you start splitting on slashes, dashes, letter digit runs, punctuation, and so on.
  - **Why does BERT produce fewer tokens than UnixCoder?**
    BERT's WordPiece tokenizer emits the special `[UNK]` token for sequences it really can't decompose into known subwords—so it often collapses an entire unknown blob (like that Base64 string) into **one** `[UNK]`, whereas UnixCoder's tokenizer doesn't fall back to a single unknown

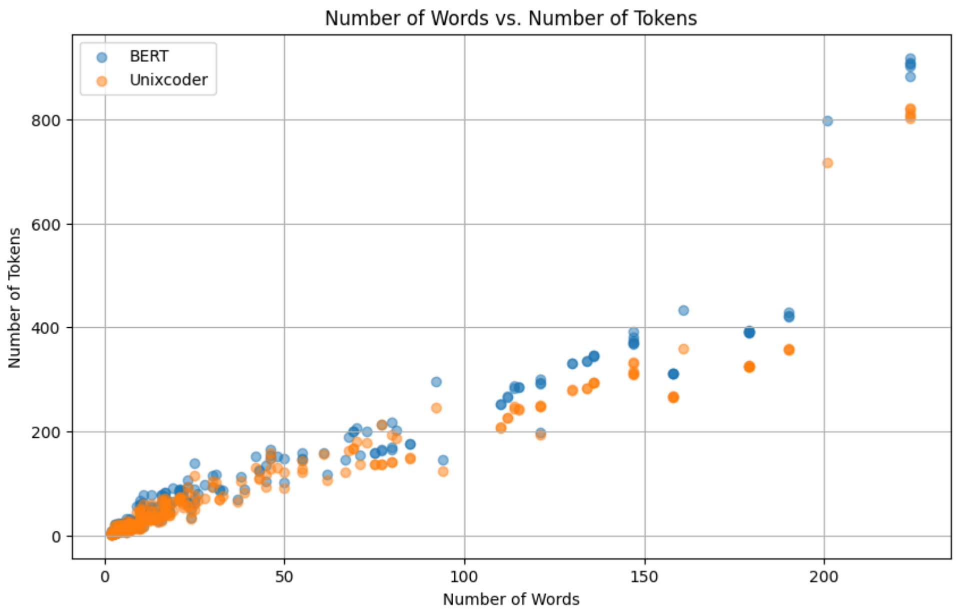

Figure 3: Token Ratio

token but instead drills down and breaks everything into tiny pieces (it is based on the BPE). That fallback to [UNK] is what keeps BERT's count ($\sim 1.9k$ tokens) much lower than UnixCoder's ($\sim 29k$ tokens).

- **Re-tokenize the processed sessions after truncating long words. Q: How many tokens per session do you have with the two tokenizers? Plot the number of words vs number of tokens for each tokenizer. Which of the two tokenizers has the best ratio of tokens to words?** $\rightarrow$ After truncating words longer than 30 characters, BERT generates on average **126.4 tokens** while Unixcoder generates **108.5 tokens** . Between the two, Unixcoder is more "token-efficient" — it produces fewer tokens per word on average ($\approx$ 108 tokens/session) than BERT ($\approx$126 tokens/session), giving it the better tokens-to-words ratio.
- **Q: How many sessions now get truncated?** Now, both tokenizers truncated six sessions each. The updated average token counts per session are 107.78 for BERT and 91.62 for Unixcoder.

NB: For Task 3, all models were fine-tuned without truncating words longer than 30 characters, as doing so caused a marked performance collapse across every model tested. A possible explanation we have given is that, although truncating very long words might seem to simplify inputs, it actually removes semantic and structural cues crucial for model performance. Truncation can strip off meaningful morphological elements—such as technical prefixes or suffixes—thus confusing the model, and it can disrupt subword tokenization pipelines, leading to unnatural token sequences or [UNK] tokens. Moreover, by eliminating rare but highly informative long tokens (e.g., URLs, hashes, or lengthy identifiers), the model could lose discriminative power on instances where these tokens appear. However, in Task 4 we reintroduced truncation of words exceeding 30 characters, since omitting it during inference resulted in abnormally inflated fingerprint values.

3 Task 3: Model Training

- Fine-tuning of the Bert Model : here we report the result after fine-tuning the pre trained Bert
  - **token_accuracy: 85.06 %**
  - **token_f1: 64.67 %** , **token_precision: 81.41 %**, **token_recall: 59.23%**
  - F1 score per class are reported in figure 4. We want to highlight that the labels that have a high F1 are also some of the ones that are more present in the dataset (discovery, execution), while it is quite strange that the F1 of Persistence is so high with respect to the labels present

in the dataset. This might mean that this technique is quite "easy" to detect or is made always by certain tokens or particular patterns.

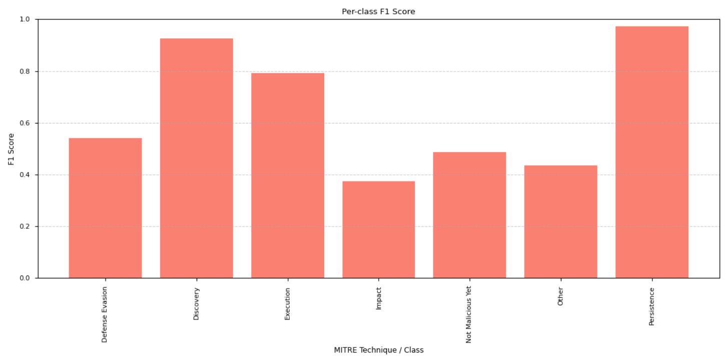

Figure 4: F1 per class

- **average fidelity: 82.3 %**
  On average 80% of each session's tokens match the black-box labels, but the long tail of low fidels sessions (seen in the histogram) highlights occasional catastrophic failures, more on the notebook.

- **Now train naked Bert without loading the thew weights of bert. Can you achieve the same performance?** →
  - **token accuracy: 77.05%.**
    Significantly lower than the fine-tuned variants, showing it never reaches the same generalization without pre trained representations. It's also worth noting that the validation curve is less stable than it was previously.
  - **token_f1: 50.04%**.
    Reflects that under represented classes suffer even more: the model has no prior knowledge of subword semantics, so it struggles to detect rarer techniques.
  - **token_precision: 60.23%** vs. **token_recall = 49.17%**
    Both drop sharply compared to fine-tuning. Medium precision still indicates it's conservative (when it predicts a class it's fairly often right), but recall plummets, meaning it misses over half of true labels.
  - On average only **75% of tokens** per session match the black-box labels, with a heavy tail of sessions at near zero fidelity, revealing catastrophic failures where entire sequences go unrecognized. More plots on the notebook about the shift in the fidelity in comparison with the previous model.

  The significant drop in all those metrics shows that 251 session are not enough to train a model from scratch.

- **Now train Unixcoder**. Since Unixcoder was pretrained with a coding corpus, the hypothesis is that it has more prior knowledge even on SSH (and therefore, it can obtain better results). **Can you confirm this hypothesis? How do the metrics change compared to the previous models?** →

Metric	**Value**
Token Accuracy	88.64%
Token F1	72.83%
Token Precision	78.10%
Token Recall	71.66%
Fidelity	86.10%

UnixCoder outperforms both BERT variants by a wide margin—+3.0 pp token accuracy, +8 pp macro-F1, and +11 pp average fidelity over the naked BERT. Its strong pretraining on code gives it noticeably better coverage of shell-like commands, at the expense of the rare “Other” and ”Impact” classes which remain challenging.

- **Load ‘SmartDataPolito/SecureShellBert’ from Huggingface and fine-tune it again. How will it perform against a newer, code specific, model such as Unixcoder? Has the performance changed? Which is your best model? →**

  The “Polito” model (SecureShellBERT fine-tuned) follows the same training pipeline like previous models but starts from the `SmartDataPolito/SecureShellBert` weights. Here’s a brief summary:

  Table 1: **Training Dynamics**

Loss	**Value**
Training Loss	steadily decreases to ∼0.10 by epoch 28 (best)
Validation Loss	plateaus around 0.23–0.25, with best at epoch 28

  Table 2: **Token-level Metrics**

Metric	**Value**
Token Accuracy	90.75%
Token F1	75.52%
Token Precision	85.79%
Token Recall	71.42%

  Compared to UnixCoder, this is a further improvement, showing that SecureShellBERT’s domain specific pretraining boosts both overall correctness and rare class coverage.

  Table 3: **Confusion Matrix Highlights**

Metric	**Value**
Persistence	1.00
Discovery	0.97
Execution	0.81
Defense Evasion	0.61
Not Malicious Yet	0.87
Other	0.38
Impact	0.35

  Table 4: **Fidelity Analysis**

Metric	**Value**
Average session fidelity	0.857

  Performance is lowest for the ”Other” and ”Impact” classes, as shown in both the confusion matrix and per class fidelity, highlighting the model’s difficulty in handling these miscellaneous actions.

  Since our priority is maximize per-token correctness and balanced performance across infrequent tactics, **SecureShellBERT fine-tuned** is the clear winner.

- **Finally, perform an alternative fine-tuning of your best model. In particular, try fine-tuning: Only the last 2 encoding layers + classification head, Only the classification head**

  How many parameters did you fine-tune in the scenario where everything was frozen? How many do you fine-tune now? Is the training faster? Did you have to change the LR to improve convergence when freezing the layers?
  How much do you lose in performance?

Aspect	**Frozen base (”head only”)**	**Last-2-layers + head**
Trainable parameters	≈ 5,383 (out of 124 M total)	≈ 14.18 M
Training speed	390-batches in ∼3 m 56s (∼12 s per epoch)	390 - batches in ∼8 m 52s (∼26 s per epoch)
Learning rate	Had to bump from 5×10^{-5} to 5×10^{-4} to get stable training of so few weights	5×10^{-5} already worked, but with 1×10^{-4} performances increased a bit
Performance Loss	Token accuracy ↓ from ∼90.75% to ∼66.6% (−21.8 pp) Token F_1 ↓ from ∼75.0% to ∼35.27% (−40 pp)	Token accuracy ≈ 87.43% Token F_1 ≈ 70.06%

Freezing almost the entire model gives a **2× speedup** but at the cost of **20 to 40 points** of token-level accuracy and F_1. This can be attributed to the fact that only the classification head was fine-tuned on a model that had not been explicitly trained on Bash session data but merely domain-adapted. Freezing almost all layers in this scenario inevitably leads to suboptimal performance.

As mentioned above, in both cases a grid search revealed that We needed to increase the learning rate at least by an order of magnitude to train both models effectively.

If resources allow, unfreezing just the final two transformer layers recovers most of the lost performance with only a slight slowdown compared to tuning only the classification head. Moreover, unfreezing those two layers together with the classification head strikes the optimal balance between training speed (still ≈ 35% faster than end-to-end fine-tuning), model accuracy and the number of trainable parameters.

4 Task 4: Inference

4.1 sessions

Each Session is present in the notebook `Task4.ipynb` and is not reported here to avoid overload. Focus on the commands `cat`, `grep`, `echo`, and `rm`.

Q: For each command, report the frequency of the predicted tags. Report the results in a table. Are all commands uniquely associated with a single tag? For each command and each predicted tag, qualitatively analyze an example of a session. Do the predictions make sense? Give 1 example of a session for each unique tuple (command, predicted tag).

Breakdown by Tactic (per Command)

Tactic	**cat**	**grep**	**echo**	**rm**
Discovery	860,146 (99.9%)	994,594 (100.0%)	204,576 (27.0%)	257,772 (75.2%)
Execution	580 (0.1%)	–	336,437 (44.4%)	58,020 (16.9%)
Persistence	–	7 (0.0%)	214,989 (28.4%)	7,387 (2.2%)
Not Malicious Yet	–	–	1,694 (0.2%)	–
Other	–	–	190 (0.0%)	–
Impact	–	–	5 (0.0%)	–
Defense Evasion	–	–	1 (0.0%)	19,496 (5.7%)

- `cat` and `grep` are overwhelmingly used for **Discovery**, which aligns with their roles in reading system information and logs.
- `echo` shows a more diverse profile, with high percentages in **Execution**, **Persistence**, and a notable portion still used for **Discovery**. Its role in writing configuration or initiating scripts makes it useful for multiple phases of attack.
- `rm` is also heavily used in **Discovery**, possibly due to cleanup tasks post-discovery, but its high use in **Execution** and **Defense Evasion** suggests it may be leveraged for covering tracks or disrupting normal operations.

All the Sessions are present in the lab4.ipynb notebook, here we highlight just the Conclusion

- Cat:
  - **Discovery**: we can notice that in this example the cat that are identified as Discovery all try to access /proc/cpuinfo that contains info about the CPU (processor, ids, chache size and so on, or trying yo access temporary file
  - **Execution**: cat command that are associated to Execution might depend on the content of the temporary file, depending if the file is an executable or not
- Grep: This session is quite strange because in this case grep has 2 predictions in which the neighbors of words are similar but the predictions are similar.

  - **Discovery**: The attacker is querying the system state — in this case, whether the SSH key is already installed. This qualifies as Discovery under the MITRE ATT&CK framework because it's gathering information about authentication configuration before making changes.
  - **Persitence**: At this point in the script, the attack is already appending the SSH key to authorized_keys, enabling persistence via SSH access. Even though grep is reading, it's part of an idempotent step to ensure the key isn't appended multiple times. It might be considered still as a discovery step done in support of a perstistence step

- Echo: looking at the `echo` command sequence, it seems we are dealing with a long, concatenated command (with improperly closed quotation marks or ticks), likely intended for execution in parts elsewhere. Here's a breakdown of the predictions:
  - **Execution**: The first `echo` appears to output a sequence that could later be used for execution. Given that it might be piped or passed to another command in a larger context, classifying it as **Execution** is reasonable.
  - **Not Malicious Yet**: This `echo` command outputs some data, but its destination or usage is not clear. Since it doesn't directly indicate malicious behavior, the classification as **Not Malicious Yet** is acceptable, though ambiguous.
  - **Persistence**: The third `echo` writes a value (`"321"`) to a temporary file. This could be part of an attempt to stage a payload or maintain access. Since it involves writing to the filesystem, the label **Persistence** is appropriate.
  - **Discovery**: The final `echo` writes what looks like credentials (`"admin ascend"`) to a file. This is not about gathering system information but rather establishing persistence. Therefore, labeling it as **Discovery** is incorrect; **Persistence** would be more accurate.

    ```
    echo "admin printemps" > /tmp/up.tx
    ```

  Those are Consideration of three different ssh session
  - **Defense Evasion**: Rather than letting the script spit out dozens of file-removal messages (e.g. removed '/var/log/secure', etc.), the attacker prints a single, innocuous "done..." message. This is a way to reduce noise output and making the victim that everything is ok.
  - **Impact** : This could be a way to insert a marker file that misleads administrators or automation systems, possibly to cause confusion or manipulate logs.

    ```
    echo "admin logout" > /tmp/up.txt ; rm -rf /var/tmp/dota*
    ```

  - **Other**: looking at the echo it seems like we are passing a base 64 encoded string trough a pipe, that will be used by perl, so it could be considered as an execution, So this could be considered as a wrong influenced that is probably influenced by the surrounding words

- Rm
  - **Persistence**: looking at the first rm commands, we can see that the attacker is trying to delete hidden files such as .ssh. They then recreate /.ssh and inject their own public key into authorized_keys. This guarantees that even if the victim changes their password or rotates other credentials—the attacker retains password-less SSH access indefinitely.

    ```
            rm -rf .ssh

            mkdir .ssh
            echo "ssh-rsa AAAAB3NzaC1yc2EAAAABJQAAAIBbHn rsa-key-20131125"
            >>~/.ssh/authorized_keys
            chmod 700 ~/.ssh
            chmod 600 ~/.ssh/authorized_keys
    ```

  - **Defense Evasion**: attacker systematically deletes every major log file and shell history they can find: login records (wtmp, lastlog), authentication logs (secure), file-transfer logs (xferlog), general syslogs (messages), mail logs, plus the user's own Bash history

```
rm -rf /var/log/wtmp \
       /var/log/lastlog \
       /var/log/secure \
       /var/log/xferlog \
       /var/log/messages \
       /var/run/utmp \
       /var/log/maillog \
       /root/.bash_history \
       *.log
```

  - **Discovery** : by stripping out user-level startup scripts and DHCP client configs, the attacker forces the shell and network stack into a "vanilla" state—no custom aliases, functions, or stale lease files to obscure what's really happening. With all that clutter gone, subsequent enumeration commands (ls, cat /proc/cpuinfo, ps, lscpu, etc.) produce clean, predictable output, making it far easier to discover exact OS version, CPU model, running processes, open services, and network configuration.
  - **Execution** : After downloading and unpacking their custom payload archive, the attacker immediately removes the .tgz file. This "fetch–unpack–cleanup" pattern is a hallmark of scripted execution: the payload is installed into memory or a private service directory, then the on-disk installer is deleted so that only the running code remains. Removing the installer artifact both finalizes the execution step and reduces the forensic footprint.

4.2 Finger prints

Each sequence of word predictions creates a 'fingerprint'. A fingerprint can be used to describe different sessions, since by construction these sessions have the same sequences of MITRE Tactics. Plot a scatter plot showing the time on the x-axis and the fingerprint ID on the y-axis.

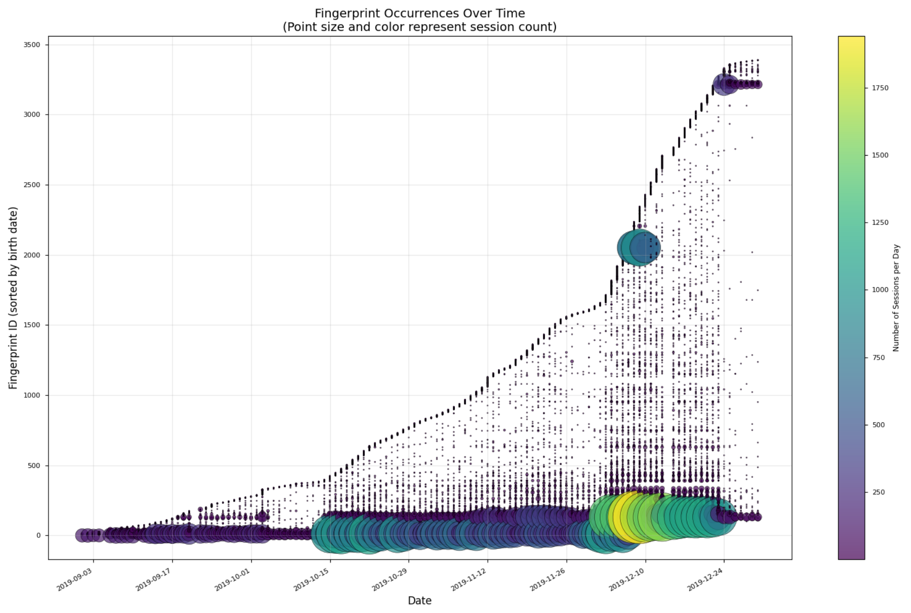

Figure 5: Finger print patterns

- Does the plot provide some information about the fingerprint patterns? Are there fingerprints that are always present in the dataset?
  - From the plot in Figure 5 highlights how the pool of SSH fingerprints grows at varying rates over time. Some fingerprints keep showing up for months , like the ones with ID 11 that started around 01/09/19 and ended around 10/12/19, while a few others appear thousands of times in a single day (like finger print 131 that had a spike of over 1941 session in one day and Finger print 122 that had 781 session in one day), evidence that automated attacks are repeatedly hitting the honeypots with the same keys.

- Are there fingerprints with a large number of associated sessions?

Funnel Pattern	**Total sessions**
FP 11	42 752
FP 131	35 377
FP 13	22 784
FP 122	15 030
FP 18	9 408
FP 153	6 810
FP 0	6 222
FP 14	4 452
FP 2052	2 478
FP 2051	2 237

Table 5: Top funnel patterns

  – Yes, the top 3 has over 20k total sessions over the span of around 3 months of data, showing that there has been an effort that lasted for a long window of attack. In fact, the dataset has data for 118 days, and in the top five of most common fingerprints, we have attacks that lasted for 100 days out of the 118. In particular, fingerprint 11 was present for 99 days with a total number of sessions 42752, and fingerprint 0 was present for 100 days with a total number of sessions of 6222. It is also important to notice that all those finger prints have a first initial part that is similar, which is a long sequence of discovery, implying there have been used different techniques to scrape the network information of the target. All those finger prints have the initial part similar, and then have some differences in the middle and final part, some focus on execution, while other try to get persistence over other systems.

- Can you detect suspicious attack campaigns during the collection?

  – One thing that we can notice from Figure 5 is that we have the we have an increase of finger prints over time, this could imply that there has been a growing effort in trying to attack the honeypot. One interesting thing to notice is how the types of attack have a major spike during December and also the volume of certain finger prints increases during that period (see also figure 6), in particular in the period of Christmas, which is a normal think for attacks that generally occur during holidays where employees tent to let their guard down due to the beginning of winter break.

All the complete data is present in the notebook.

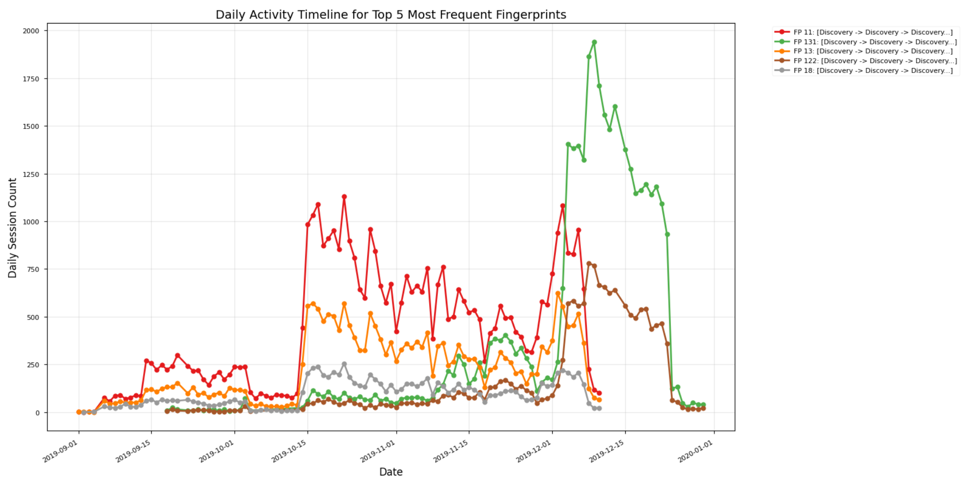

Figure 6: Top 5 most common fingerprints