Project 3: Natural Language Processing (NLP)

Giovanni Bloise , Alessandro Mulassano , Manuel Firrera

This project focuses on the application of Natural Language Processing (NLP) methods in the context of cybersecurity, with a particular emphasis on analyzing SSH session data. Each SSH session is represented as a sequence of entities—such as commands, flags, parameters, and separators—collectively referred to as SSH words. For example, in the session ls -la `.\` ;, the command (ls), flag (-la), parameter (`.\`), and separator (;) can be distinguished as meaningful units of analysis. The central objective of the project is to fine-tune a pre-trained language model to perform a Named Entity Recognition (NER) task, where each SSH word is classified according to its corresponding MITRE tactic. This allows for the interpretation of user actions in terms of adversarial techniques, thus contributing to the detection and understanding of potential threats. Throughout the project, different tokenization strategies and pre-trained language models are explored and compared to assess their effectiveness in handling this domain-specific task. Finally, the fine-tuned model is applied to inference, enabling the identification of suspicious behaviors and the intentions of potentially malicious users.

1 Task 1: Dataset Characterization

This task aims to explore and understand the structure of the SSH sessions dataset prior to the application of advanced NLP techniques. The initial step involves verifying the dataset to identify potential issues such as duplicates, missing values, or inconsistencies that could mislead the analysis. The inspection revealed none of these issues, indicating that the dataset is ready for further exploration.

1.1 Exploring the Labels

The first part of the analysis focuses on the dataset labels.

Q: How many different tags are present?
In the training dataset, there are **7** distinct tags:
Defense Evasion, Discovery, Execution, Impact, Not Malicious Yet, Other, Persistence.
The test dataset contains the same set of labels.

Q: How are they distributed? Provide a bar plot showing the distribution of tags (both Train and Test—two bars).
The distribution is as follows:

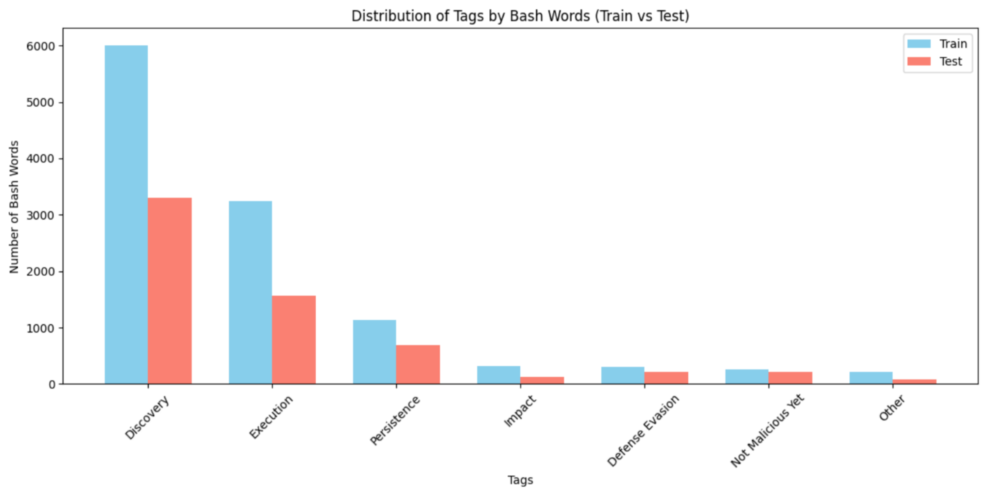
Train Set:

Tag	Count
Discovery	6009
Execution	3239
Persistence	1133
Impact	312
Defense Evasion	309
Not Malicious Yet	264
Other	209

Test Set:

Tag	Count
Discovery	3307
Execution	1568
Persistence	683
Impact	133
Defense Evasion	218
Not Malicious Yet	212
Other	76

As the tables indicate, the dataset suffers from class imbalance. To better illustrate the distribution of the tags, the bar plot below provides a visual representation:



1.2 Exploring a Single Bash Command

This subsection focuses on the analysis of a specific bash command: `echo`.

Q: For `echo`, how many different tags are assigned, and how frequently does each occur?
The command `echo` is associated with **6** different tags:

Tag	Count
Persistence	104
Execution	39
Discovery	31
Not Malicious Yet	8
Impact	6
Other	4

Q: 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?
An example of such a session can be found at index 28 of the training dataset:

```
cat /proc/cpuinfo ... echo root:JrBOFLr9oFxB | chpasswd | bash ...
    echo 1 > /var/tmp/.systemcache436621 ; cat /var/tmp/.
    systemcache436621 ; sleep 15s && cd /var/tmp ; echo
    IyEvYmluL2Jhc2gKY2QgL3RtcAkKcm0gLXJmIC5zc2gKcm0gLXJmIC5tb3 | base64
     --decode | bash ;
```

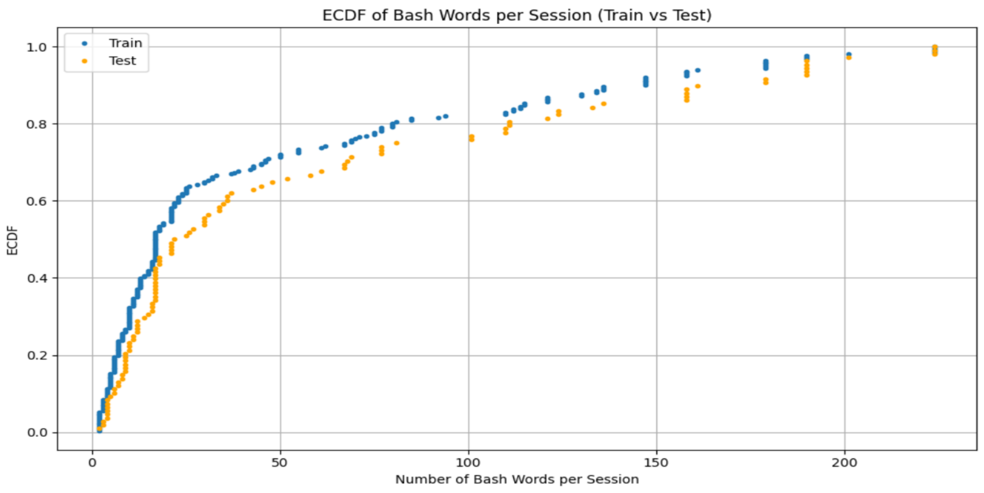
Note: the final `echo` *in this session was truncated.*

The assignment of different tags to `echo` depends on the context of its usage. For example, in the command `echo root:JrBOFLr9oFxB | chpasswd | bash`, echo is used to provide input to another process in order to execute an action (changing the root password). In this case it is categorized as Execution because it directly triggers a malicious activity. On the other hand, in the command `echo 1 > /var/tmp/.systemcache436621`, echo is used to write data into a file under /var/tmp. This behavior is categorized as Persistence, since it leaves an artifact on disk that can later be used to maintain or resume malicious operations.

1.3 Exploring Bash Words

The final part of Task 1 involves extracting statistics on the number of bash words per session.

Q: How many bash words per session are present? Provide the Estimated Cumulative Distribution Function (ECDF).



As shown in the figure, most sessions contain fewer than 50 bash words, while sessions with more than 100 words are relatively rare. The training set is slightly more concentrated on shorter sessions, whereas the test set exhibits a more uniform distribution in the lower range.

2 Task 2: Tokenization

This task investigates the impact of different tokenization strategies. In particular, we compare the tokenizers of two models: BERT-base and Unixcoder-base.

2.1 Tokenization of specific shell commands

We begin by analyzing the tokenization of a set 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?
The BERT tokenizer produces the following tokens: [['cat'], ['shell'], ['echo'], ['top'], ['ch', '`##pass`', '`##wd`'], ['cr', '`##ont`', '`##ab`'], ['w', '`##get`'], ['busy', '`##box`'], ['gr', '`##ep`']]
The Unixcoder tokenizer produces: [['cat'], ['shell'], ['echo'], ['top'], ['ch', 'passwd'], ['cr', 'ont', 'ab'], ['w', 'get'], ['busybox'], ['grep']]
By comparing the average token-to-word ratio, Unixcoder demonstrates a lower value, as it is optimized for code-related text and tends to preserve entire words. This also explains why commands such as `grep` remain intact under Unixcoder, whereas BERT splits them into multiple subtokens.

Average BERT token/word ratio: **1.78** *Average UNIXCODER token/word ratio:* **1.44**

2.2 Tokenization of the entire training corpus

Next, we apply both tokenizers to the entire training corpus.

Q: How many tokens does the BERT tokenizer generate on average? 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?
On average, the BERT tokenizer generates **189.79** tokens per session, while Unixcoder produces **481.45** tokens. The maximum number of tokens generated per session is **1889** for BERT and **28,920** for Unixcoder.
This significant difference is explained by the handling of long or unusual sequences (e.g., repeated `echo AAA...` or Base64 strings). BERT employs the special token `[UNK]` to replace sequences it cannot process, thus compressing long strings into a single token. In contrast, Unixcoder does not use `[UNK]` and instead splits such sequences into numerous subtokens, resulting in a dramatic increase in token count.

Q: How many sessions would currently be truncated for each tokenizer?
BERT: 21 out of 200   Unixcoder: 27 out of 200
This means that only a small fraction of the sessions exceed the maximum sequence length allowed by the model, with Unixcoder showing slightly more truncations compared to BERT.

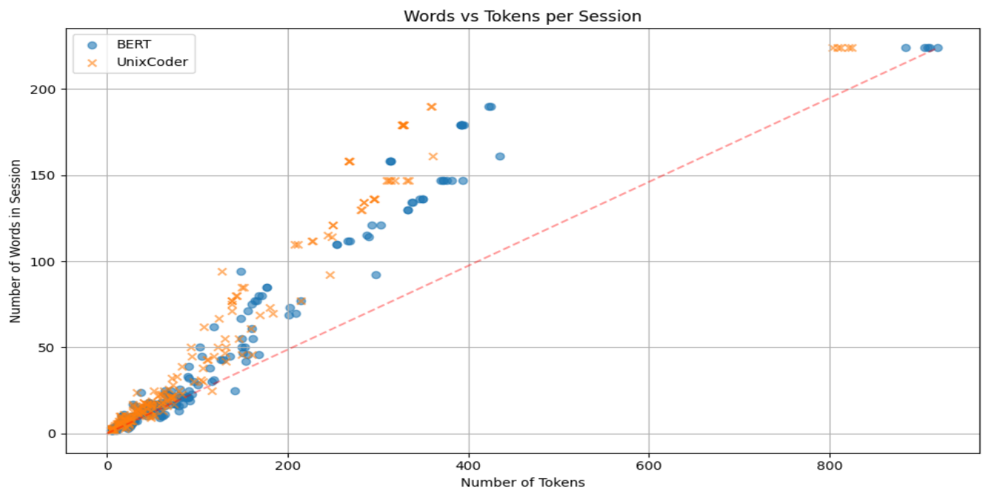
Q: What is the size of the session with the maximum number of tokens, and why do the tokenizers differ?
The session contains **134** words. The extremely high token count arises from the presence of a large Base64 string. As previously noted, BERT replaces such strings with `[UNK]`, reducing the token count, while Unixcoder splits the string into thousands of subtokens.

2.3 Truncating Long Words

We now truncate words longer than 30 characters and repeat the analysis.

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?
After truncation, the BERT tokenizer generates an average of **134.15** tokens per session, whereas Unixcoder produces **115.39**.

Average Token-to-Word Ratio:
BERT: **3.17**
Unixcoder: **2.73**
Thus, Unixcoder maintains a more efficient token-to-word ratio.



Q: How many sessions are truncated after truncating long words?
Both tokenizers end up truncating 5 out of 200 sessions. Once excessively long words are handled, the number of truncated sessions drops considerably and becomes the same for both models.

3 Task 3: Model Training

In this phase of the project, we focus on training and evaluating various pre-trained language models with the goal of identifying which provides the best performance for our use case.

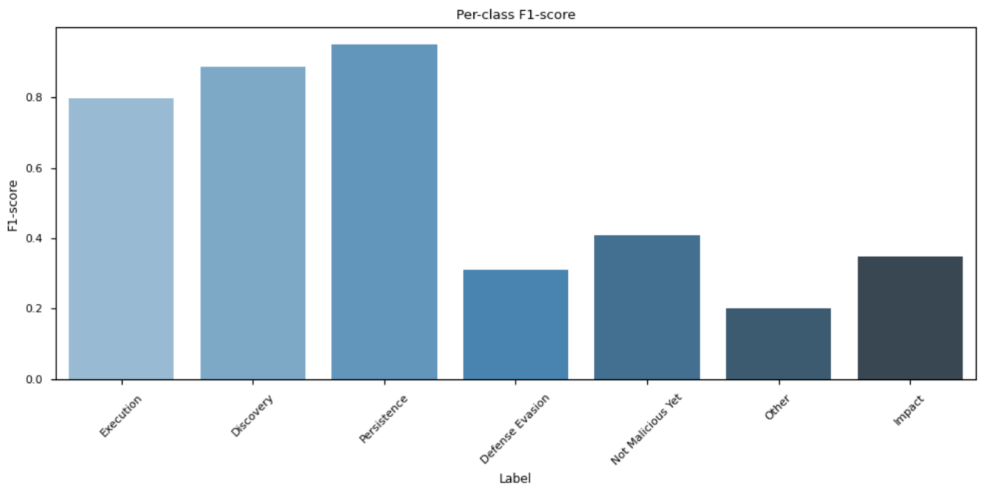
3.1 BERT

The first model tested is BERT. We load the pre-trained model with its corresponding weights from Huggingface. The model classifies each token into one of the MITRE Tactics. However, the results are poor, as BERT was not pre-trained on logs or code, even though it benefits from extensive linguistic pre-training.

3.1.1 Results

Token classification accuracy: 82.98%
Token precision: 81.43%
Token recall: 50.23%
Token F1-score: 55.85%
Given the class imbalance,the most informative metric in this setting is the F1-score, which is relatively low as expected.

Per-class F1-scores: As shown in the bar plot, performance is acceptable for the three largest classes, but significantly worse for the smaller ones.



Average fidelity: 80.10%

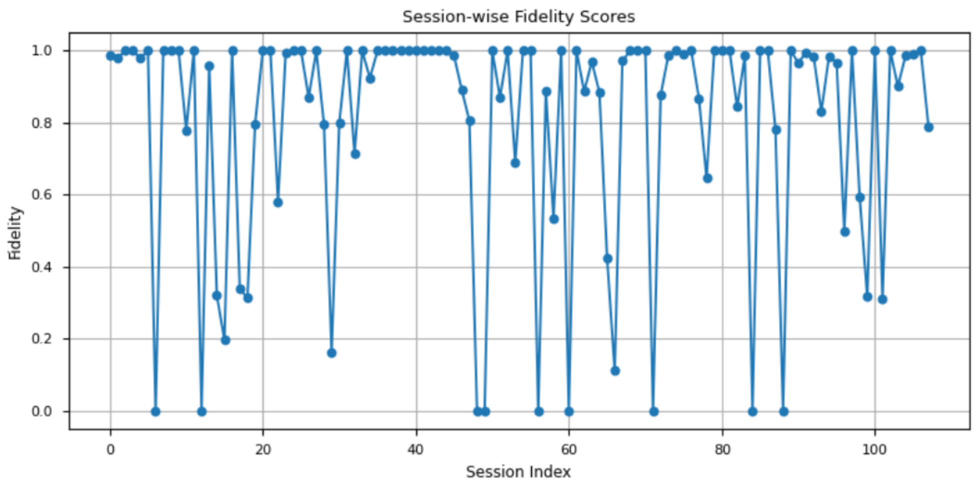


Figure 1: BERT average fidelity

As illustrated in Figure 1, most sessions achieve a fidelity score close to 1, confirming that the model is generally consistent in reproducing the input. However, several sessions drop to very low values, even to 0, highlighting specific cases where the model struggles. Overall, the average fidelity remains high, but the presence of these outliers indicates variability in performance across different sessions.

Q: Can the model achieve "good" results with only 251 training labeled samples? Where

do it have the most difficulties?
The results confirm that BERT does not achieve satisfactory performance. While it classifies the largest classes reasonably well, it performs very poorly on the smaller ones, as shown in the confusion matrix. This is expected, given that BERT is not pre-trained on code and the number of training samples is limited.

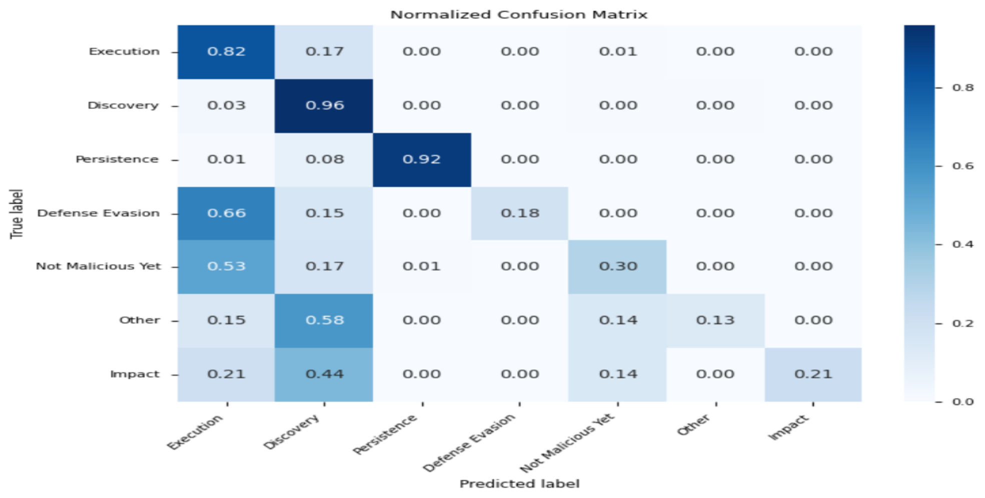


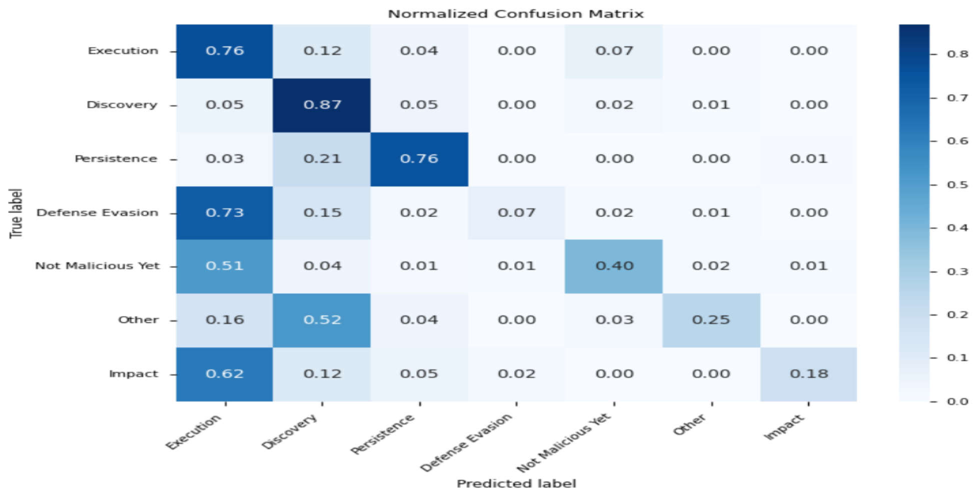
Figure 2: BERT confusion matrix

3.2 Naked BERT

In this experiment, instead of using a pre-trained BERT model, we load only the BERT architecture, without pre-trained weights. As expected, this leads to even weaker results.

3.2.1 Results

Q: Can the same performance be achieved? Report the results.
The performance is indeed worse, as anticipated. Without pre-training, the model starts from scratch, lacking both linguistic and statistical prior knowledge. This requires a much larger dataset to learn meaningful patterns. The results are as follows:
Accuracy: **75.36%** Precision: **57.68%** Recall: **46.97%** F1-score: **49.21%**



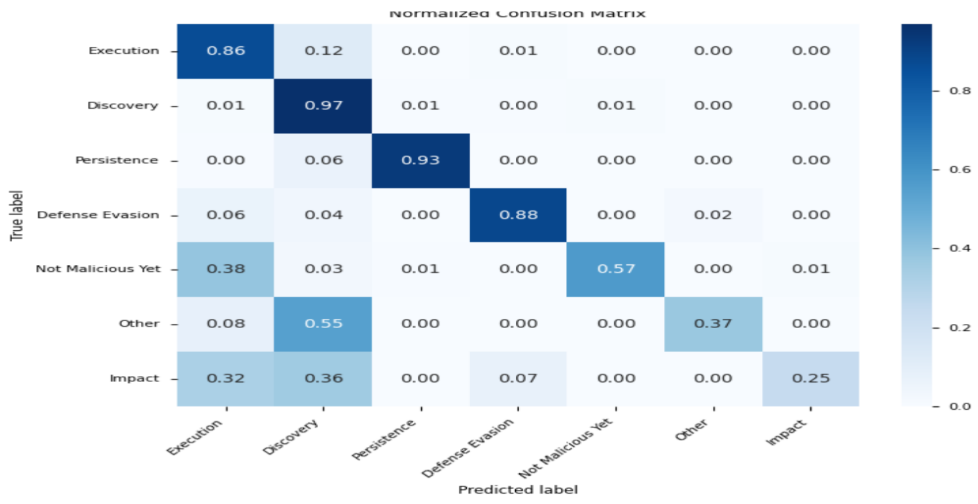
Once again, the model handles the largest classes moderately well but fails considerably on the smaller ones.

3.3 UNIXCODER

We then test the Unixcoder model. Since it was specifically optimized for code and programming languages (including SSH), we expect it to perform better than BERT.

3.3.1 Results

Q: After fine-tuning Unixcoder, can the hypothesis be confirmed that it performs better due to its pre-training? How do the metrics compare to previous models?
The results clearly support the hypothesis:
Accuracy: **88.37%** Precision: **86.84%** Recall: **69.00%** F1-score: **73.72%**



True label \ Predicted label	Execution	Discovery	Persistence	Defense Evasion	Not Malicious Yet	Other	Impact
Execution	0.86	0.12	0.00	0.01	0.00	0.00	0.00
Discovery	0.01	0.97	0.01	0.00	0.01	0.00	0.00
Persistence	0.00	0.06	0.93	0.00	0.00	0.00	0.00
Defense Evasion	0.06	0.04	0.00	0.88	0.00	0.02	0.00
Not Malicious Yet	0.38	0.03	0.01	0.00	0.57	0.00	0.01
Other	0.08	0.55	0.00	0.00	0.00	0.37	0.00
Impact	0.32	0.36	0.00	0.07	0.00	0.00	0.25

Figure 3: Unixcoder confusion matrix

Unixcoder substantially outperforms both versions of BERT. Its recall and precision are significantly higher (with the F1-score roughly 20% higher than BERT). The improvement is particularly evident in the *Defense Evasion* class: while BERT models perform very poorly (0.18 for BERT and 0.07 for Naked BERT), Unixcoder achieves 0.88. Furthermore, Unixcoder also performs better on the smallest classes.

3.4 SecureShellBert

In this section, we evaluate SecureShellBERT, a CodeBERT-based model pre-trained on code. For this reason, we expect its performance to be close to Unixcoder, and significantly better than standard BERT.

3.4.1 Results

How will it perform against a newer, code-specific, model such as Unixcoder? Has the performance changed? Which is your best model?

Metric	Bert	SecureShellBERT	Unixcoder
Accuracy	82.98%	86.21%	88.37%
Precision	81.43%	82.84%	86.84%
Recall	50.23%	62.21%	69.00%
F1-score	55.85%	68.28%	73.72%

Table 1: Performance comparison between SecureShellBERT and Unixcoder.

As expected, SecureShellBERT performs considerably better than BERT, though slightly worse than Unixcoder. In particular, the gain over the vanilla BERT baseline is consistent across all metrics, with a notable improvement in recall (+12%) and F1-score (+12%), which suggests that SecureShellBERT is better at identifying positive instances while maintaining a good balance with precision. Unixcoder, being a more recent code-specific model, still achieves the overall best performance, especially in terms of recall and F1-score, confirming its robustness on this task. Nevertheless, the relatively close gap indicates that SecureShellBERT provides a competitive and more lightweight alternative in scenarios where Unixcoder may be too resource-intensive.

3.5 Freezing the best model

Finally, we take the best model (Unixcoder) and fine-tune it under two different settings:

1) Only the classification head plus the last two layers
2) Only the classification head

3.5.1 Results

Q: 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?
When fine-tuning the last two layers plus the classification head, the full model has 125,344,519 parameters, of which only **14,181,127** are trainable.
When fine-tuning only the classification head, just **5,383** parameters are trained.

As expected, training time decreases significantly: 03:21 for fine-tuning the last two layers plus the head, versus only 02:44 when training just the head. In the second case, the classification head starts from random weights and has very few parameters to update. As expected, performance deteriorates; With only the last two layers and the classification head unfrozen, we obtained an *f1_score of 53.59%*. In this case, the model still performs reasonably well on the major classes, though its performance drops on the minor ones. However, when unfreezing just the classification head, performance worsens further, with an *f1_score of just 17.59%*, and the model struggles across all classes. So, we needed to increase the learning rate to get better performances. In particular, with the first setup, a learning rate of 5e-4 gave better results; with the second, a learning rate of 3e-3 proved most effective.

4 Task4: Inference

In the final part of the lab, we employ the best-performing model from Task 3 to automatically predict MITRE tags for unseen inference sessions. This step demonstrates how NLP-based tools can support security analysts by mapping raw SSH logs to actionable threat intelligence, thereby reducing manual workload and streamlining the investigation process.

Q: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).

	Defense Evasion	Discovery	Execution	Impact	NMY	Other	Persistence
cat	2	860,979	583	0	1	0	0
echo	109	421,927	119,104	5	26,178	232	192,125
grep	0	995,805	0	0	0	0	569
rm	34,058	293,489	12,046	0	4	0	5,432

Table 2: Predicted frequency of tags per command.

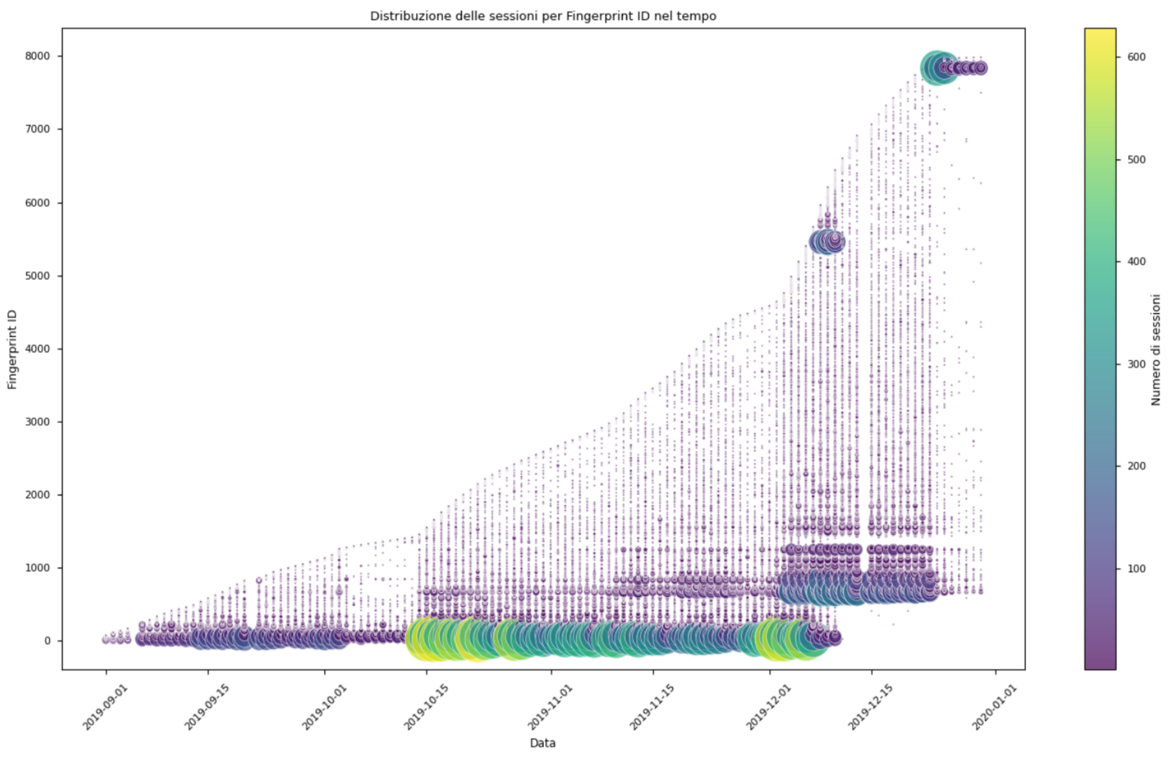
Command	Tag	Example
`cat`	Defense Evasion	`cat /dev/null > /root/.bash_history`
`cat`	Discovery	`cat /proc/mounts`
`cat`	Execution	`cat /proc/cpuinfo | grep name`
`cat`	Not Malicious Yet	`cat /etc/issue | nproc`
`echo`	Defense Evasion	`cp /bin/echo .s`
`echo`	Discovery	`cp /bin/echo .s`
`echo`	Execution	`do echo $i`
`echo`	Impact	`echo "next\nLIkLMn\n" | passwd`
`echo`	Not Malicious Yet	`echo "321" > /var/tmp/.var03522123`
`echo`	Other	`echo "ZXZhbCB1..."`
`echo`	Persistence	`echo "321" > /var/tmp/.var03522123`
`grep`	Discovery	`cat /proc/cpuinfo | grep name`
`grep`	Persistence	`LC_ALL=C grep "ssh-rsa AAAAB3" /.ssh/authorize`
`rm`	Defense Evasion	`rm .s`
`rm`	Discovery	`rm -rf /var/tmp/.var035`
`rm`	Execution	`rm -rf *`
`rm`	Persistence	`rm -rf /var/log/wtmp`

Note: the table reports the ***first occurrence*** *of each (command, tag) pair.*

For the command **cat**, predictions generally make sense: for example, emptying the bash history is a clear evasion technique. However, in some cases (e.g., *Execution*), a different tag such as *Discovery* would be more appropriate. For **echo**, the tags *Execution*, *Impact*, *Not Malicious Yet*, *Other*, and *Persistence* are well justified (e.g., printing variable values or writing to a file), whereas others could be refined. Predictions for **grep** are consistent with the expected use cases, while for **rm**, the assignments to *Defense Evasion* and *Execution* make sense, but other tags appear less accurate.

Q: Extract the unique fingerprints from the dataset, order them by their first appearance date, assign each a progressive ID, count the sessions per fingerprint for each day, and finally visualize everything in a scatter plot (time on the x-axis, fingerprint ID on the y-axis, with point size/color showing the number of sessions)
We identified **7,984** unique fingerprints.

Date	Session Count
2019-09-01	148
2019-09-02	158
2019-09-03	146
2019-09-04	145
2019-09-06	410
2019-09-07	368
2019-09-08	401
2019-09-09	425
2019-09-10	456
2019-09-11	275



Q: Does the plot provide some information about the fingerprint patterns? Are there fingerprints that are always present in the dataset? Are there fingerprints with a large number of associated sessions? Can you detect suspicious attack campaigns during the collection? Give a few examples of the most important fingerprints.

- A low-ID band (approximately 0–200 and 300–800) persists from early September to late December with many bubbles. This likely represents recurring system processes such as cron jobs, monitoring, log rotation, or cleanup activities.
- Several bursts of large yellow/green bubbles (hundreds of sessions) indicate high activity:
  - Mid-October to early December: IDs in the range 0–200.
  - Early to mid-December: IDs around 800–1,000 and 5600-5800.
  - Late December: high IDs (7,600–7,900).
- These bursts correspond to fingerprints generating hundreds of short sessions per day, consistent with automated scripts, botnets, or credential-stuffing campaigns.
- The dataset also shows "waves" of activity where many new IDs appear simultaneously with high session counts:
  - Wave A (Oct 12–20): sudden rise of large bubbles in the low-ID range.
  - Wave B (early to mid-December): widespread activity across IDs 500–1,500, plus new fingerprints above 5,000.
  - Wave C (Dec 27–31): late-December surge of new IDs (7,700–7,900) with very high activity.

Overall, these patterns suggest both benign recurring system activity and suspicious large-scale campaigns. The most relevant fingerprints are concentrated in the following ID ranges: **0–200**, **600–800**, **5,400–5,800**, and **7,700–7,900**.