Project 3: Natural Language Processing (NLP)

Scursatone Matteo - s332153, Lorenzato Fabio - s332186, Milani Alessandro - s332136

August 29, 2025

1 Task 1: Dataset Characterization

During the first phase, we had to explore the dataset and understand which tags do we have and how they are distributed. The training dataset contains 251 labelled sessions with 11.475 total words, while the test dataset contains 108 labelled sesssione with 6.197 total words.

Q). 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 (both for Train and Test — 2 bars).

Every words is labelled with one of the following MITRE tags: **Discovery**, **Execution**, **Persistence**, **Impact**, **Defense Evasion**, **Not Malicious Yet**, **Other**. The dataset is unbalanced and the distribution of the tags between the training and test set is not the same, as shown in Figure 1. The most frequent tag is **Discovery**, while the least frequent one is **Other**.

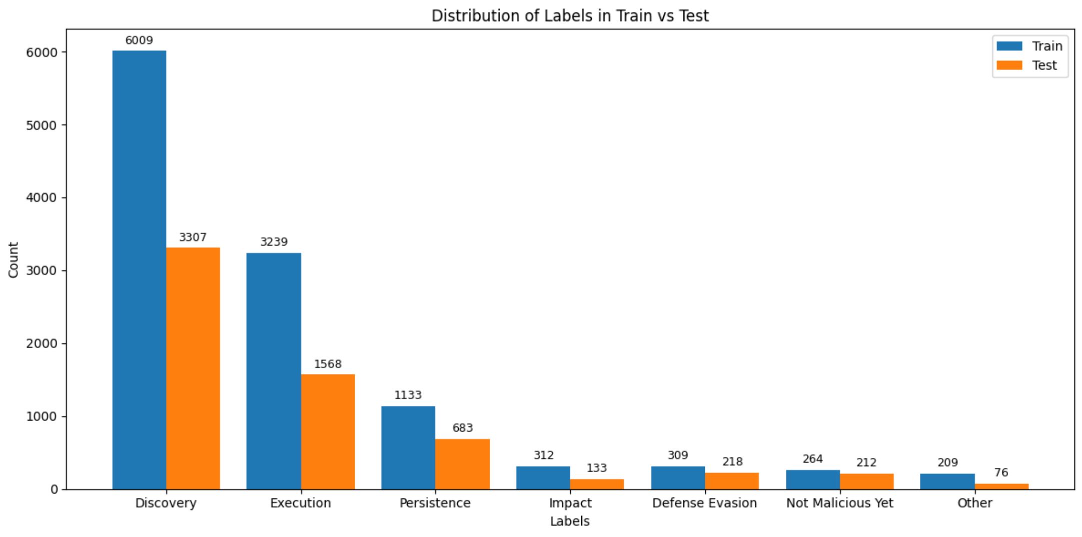


Figure 1: Label distribution in the dataset for train and test sets.

Q.) 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? We now focused on the analysis of the the word inside the sessions. Especially, we wanted to understand how many different tags are assigned on the word 'echo'. In the training set the word echo is assigned to 6 different tags, as shown in figure 2. This means that the same word can be assigned to different tags depending on the context. This makes the task of classifying the words more difficult, since we cannot simply assign a tag to a word based on its frequency in the training set.

For example we can confront the following two sentences:

- "echo root:JrBOFLr9oFxB | chpasswd | bash ;"
- "echo 1 > /var/tmp/.systemcache436621 ; cat /var/tmp/.systemcache436621 ; sleep 15s"

That refers to two different actions, the first one is used to change the password of the root user in order to mantain control and access to the server, while the second one is used to create or modify a temporary file. The first "echo" is tagged as **Persistence**, while the second one is tagged as **Execution**.

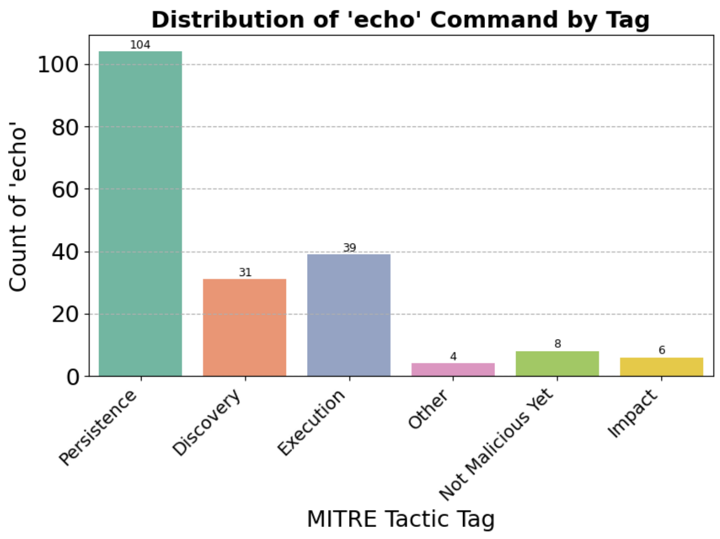


Figure 2: Tags assigned to the word 'echo' in the training set.

Q.) How many Bash words per session do you have? Plot the Estimated Cumulative Distribution Function (ECDF)5 — example in the sample notebooks.

For the last point of this task we are going to explore the length of the sessions in terms of number of words 1 and with ECDF distribution 3. We can see that the sessions are quite short, with a median of 17 words in the training set and 23.5 in the test set. The maximum length is 224 words for both sets. This means that we have to be careful when choosing the model, since some models may not be able to handle long sequences. From the ECDF we can also see that there are some outliers that are much longer than the average.

	Min	Median	Mean	Max
Train	2	17	45.72	224
test	2	23.5	57.38	224

Table 1: Statistics of the session length in terms of number of words.

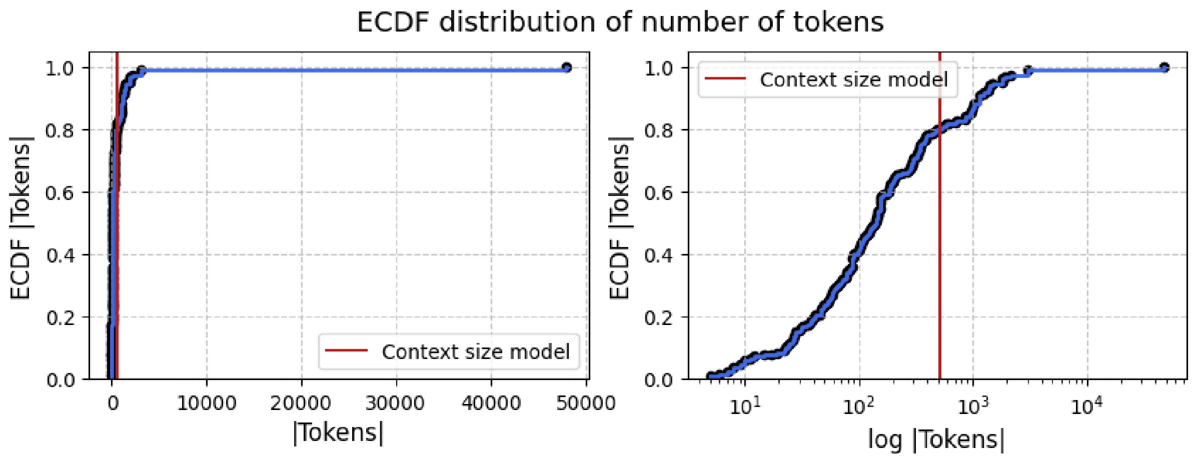


Figure 3: ECDF of the session length in terms of number of words.

2 Task 2: Tokenization

In this second task we have to confront different tokenization strategies. The first one is related to "bert-base-uncased" tokenizer, that is the one used by the BERT model, while the second one is the "unixcoder-base" tokenizer, that is the one used by the UnixCoder model.

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?

Bert model is using BertTokenizerFast that is a fast implementation of the BERT tokenizer. It is based on the WordPiece algorithm that is a subword tokenization algorithm. It is able to handle out-of-vocabulary words by breaking them into smaller subwords. This is useful for handling rare words or misspellings. The tokenizer is also case insensitive, meaning that it converts all the text to lowercase before tokenizing it. This can be useful for handling text with different cases, but it can also lead to loss of information. The vocabulary size of the BERT tokenizer is 30522 tokens.

UnixCoder model is using RobertaTokenizerFast that is a fast implementation of the RoBERTa tokenizer. It is based on the Byte-Pair Encoding (BPE) algorithm that is another subword tokenization algorithm. It is also able to handle out-of-vocabulary words by breaking them into smaller subwords. The tokenizer is case sensitive, meaning that it preserves the case of the text. This can be useful for handling text with different cases, but it can also lead to an increase in the vocabulary size. The vocabulary size of the UnixCoder tokenizer is 51416 tokens.

They have also differences on how they were pretained, BERT was pre-trained on a large corpus of English text, while UnixCoder was pre-trained on a large corpus of code. This means that the UnixCoder tokenizer may be better suited for handling technical terms and code snippets that are present in our dataset, moreover Unixcoder tokenizer better divide into token words: [cat, shell, echo, top, chpasswd, crontab, wget, busybox, grep] with a lower ratio between number of tokens and words. For example chpasswd is tokenized as [ch, ##pass, ##wd] by BERT tokenizer, while it is tokenized as [Ġch, passwd] by UnixCoder tokenizer while other are handled similarly like echo that is tokenized as [echo] by BERT tokenizer and [Ġecho] by UnixCoder tokenizer.

Q.) 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?

Now we can compare the two tokenizer on the whole training dataset.

	Min	Mean	Max	Std
Bert	2	176.58	1887	314.79
Unix	2	407.32	28918	2579.13

Table 2: Statistics of the session length after tokenization.

	Min	Mean	Max	Std
Bert	2	126.38	918	164.67
Unix	2	108.53	822	144.62

Table 3: Statistics of the session length after tokenization + **truncation of words**.

Q.) How many sessions would currently be truncated for each tokenizer (remember: you can determine the maximum context of a model with 'tokenizer. model_max_length')?

The maximum length of the sessions after tokenization is 512 for BERT tokenizer and 1000000000000000019884624838656 for UnixCoder tokenizer. This means that with BERT tokenizer 24 sessions will be truncated to 512 tokens while none will be truncated with UnixCoder tokenizer.

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

The maximum number of token generated by the two tokenizers is 1887 for BERT and 28918 for UnixCoder. In 9th word of the longest sessione (index 14) there is a long base64 encoded string that unixcoder have difficulties to handle.

The massive difference with the two tokenizer is due to the fact that UnixCoder tokenizer is able to break down the words into smaller subwords, leading to a higher number of tokens while BERT tokenizer use the token '—UNK—' to represent out-of-vocabulary words, leading to a lower number of tokens. This can be a problem for our task, since we may lose important information about the context of the 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

In order to handle the long words we will truncate words longer than 30 characters, this choice is motivated by the fact that there are long words encoded in base64 that are malicious scripts and to handle them better. We truncate 178 words in the training set and 146 in the test set and this allow us to reduce the number of tokens used especially with UnixCoder tokenizer, as shown in table 3.

The ratio between number of token and number of words after truncation is 3.04 for BERT tokenizer and 2.58 for UnixCoder tokenizer, this means that UnixCoder tokenizer is able to better divide the words into tokens in our dataset and this is probably due to the fact that it was pre-trained on a large corpus of code even if it does not use —UNK— token for out-of-vocabulary words.

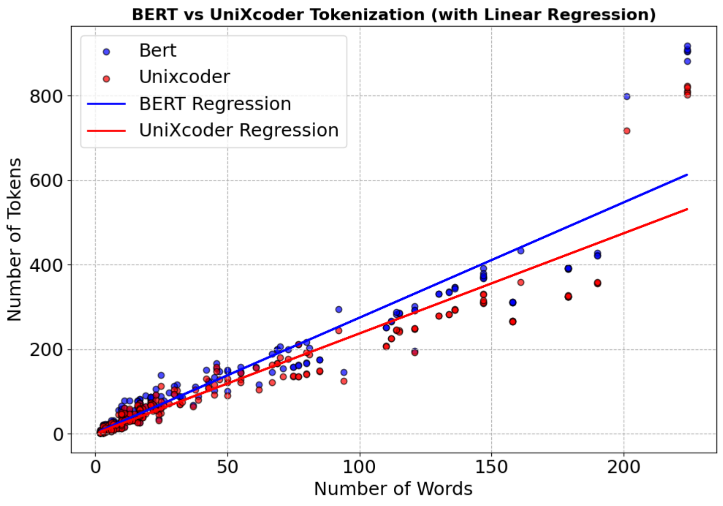


Figure 4: Bert vs UnixCoder tokenization length regression.

Q.) How many sessions now get truncated? After truncation of words just 6 sessions will exceed the maximum length of 512 tokens with BERT tokenizer, while none will exceed it with UnixCoder tokenizer.

3 Task 3: Model Training

We fine-tuned pre-trained language models for token-level classification on a proprietary dataset and evaluated performance on a held-out test set. The following configurations were considered: BERT (`google-bert/bert-base-uncased`), randomly initialized BERT (to assess the effect of pre-training), UniXcoder (`microsoft/unixcoder-base`), and SecureShellBert (`SmartDataPolito/SecureShellBert`). Given that SecureShellBert achieved the best performance, we further examined partial fine-tuning strategies to reduce training cost: pdating only the last two encoder layers and the classifier, and updating only the classifier. Models were evaluated using token-level accuracy, macro-averaged precision, recall, and F1 score, per-class F1 scores, and average session fidelity (fraction between the number of correct predictions and the total number of tokens).

3.1 BERT

Q.) Can the model achieve "good" results with only 251 training labeled samples? Where do it have the most difficulties?

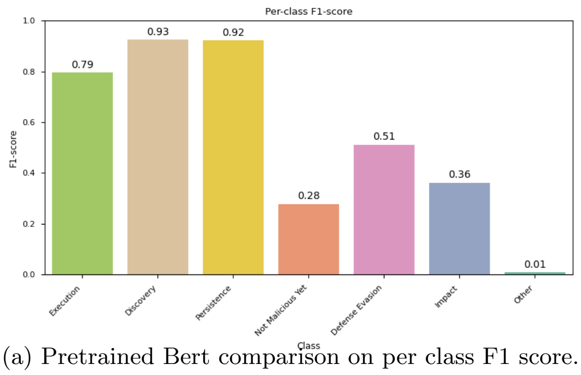
Table 4 summarizes the performance of the four evaluated models. SecureShellBert outperformed the other models across all metrics, achieving an accuracy of 89%, a macro-averaged F1 score of 71%, and an average session fidelity of 89%. UnixCoder also performed well, with an accuracy of 85% and a macro-averaged F1 score of 67%. The randomly initialized BERT model had the lowest performance, with a difference of 5% in all metrics compared to the pre-trained BERT model, highlighting the importance of pre-training.

Despite the limited training set (251 labeled samples), pre-trained models achieve "good" performance in just 3/7 of the classes (Discovery, Execution, Persistence), while struggling with the remaining classes, as shown in Figure 7. This is likely due to the class imbalance in the dataset, with some classes having very few samples.

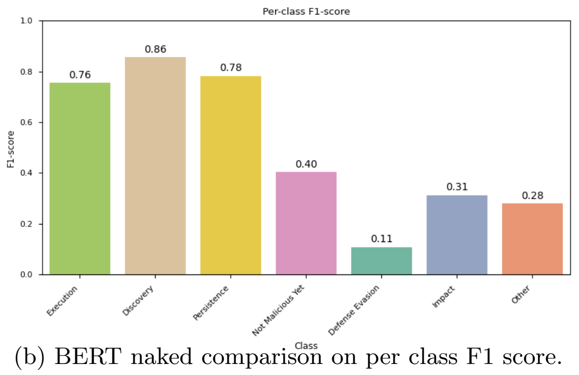
Q.) Can you achieve the same performance? Report your results.

BERT's model performance is boosted by pre-training, with a difference of 5% in all metrics compared to the randomly initialized BERT model. This highlights the importance of pre-training, especially when labeled data is scarce. Biggest differences are observed in the label Defense Evasion where the F1 score in pre-trained BERT is 0.51 while in naked BERT is 0.11. Surprisingly, in class **Other** and **Not Malicious Yet** perfomed better the naked BERT with a F1 score of 0.28 and 0.40 respectively, while pre-trained BERT achieved a F1 score of 0.01 and 0.28 in both classes.

Figure 5: BERT pretrained and naked comparison on per class F1 score.



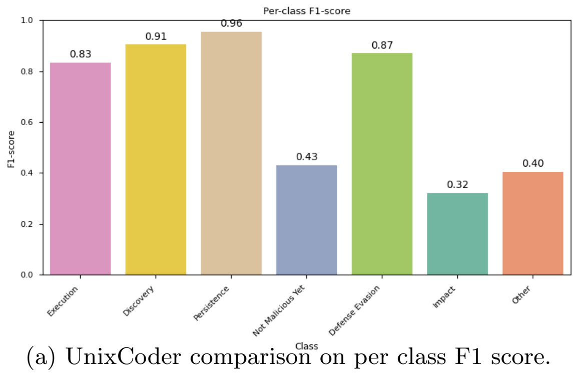
(a) Pretrained Bert comparison on per class F1 score.



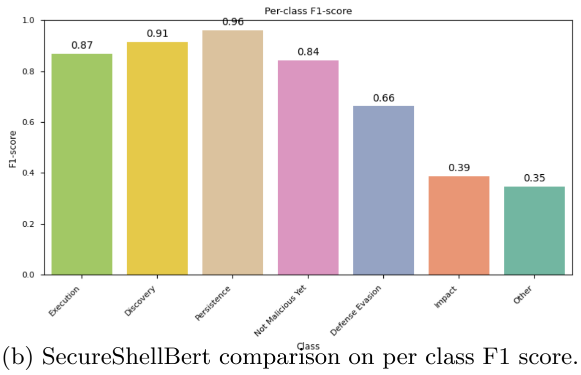
(b) BERT naked comparison on per class F1 score.

3.2 UnixCoder and SecureShellBert

Figure 6: UnixCoder and SecureShellBert comparison on per class F1 score.



(a) UnixCoder comparison on per class F1 score.



(b) SecureShellBert comparison on per class F1 score.

Q.) Unixcoder Can you confirm this hypothesis? How do the metrics change compared to the previous models? – Q.) How will it perform against a newer, code-specific, model such as Unixcoder? Has the performance changed? Which is your best model?

Figures 6a and 6b show the per-class F1 scores for UnixCoder and SecureShellBert, respectively. Both models perform well on the classes with more samples (Discovery, Execution, Persistence), but Unixcoder outperform SecureShellBert in the class **Defense Evasion** with a f1score of 0.87 vs 0.66. However, SecureShellBert outperforms UnixCoder in the classes **Not Malicious Yet** with a f1score of 0.84 vs 0.43. **Impact** and **Other** instead have similar results on both models.

In the end SecureShellBert outperformed the other models across all metrics resulting in the best model for our task, achieving an accuracy of 89%, a macro-averaged F1 score of 71%, and an average session fidelity of 89%. We will use this model for the final task of inference on the test set and also for the partial fine-tuning strategies.

Our hyppothesis that a model pre-trained on code would perform better than a model pre-trained on natural language is confirmed by the results, with UnixCoder outperforming BERT in all metrics. However, SecureShellBert, with a domain adapatation improved the performance even more, achieving the best results overall.

Model	**Accuracy**	**Precision**	**Recall**	**F1**	**Fidelity**
BERT pretrained	0.83	0.51	0.67	0.54	0.83
BERT naked	0.77	0.47	0.62	0.50	0.78
UnixCoder	0. 85	0.64	0.79	0.67	0.87
Secure Shell BERT	0.89	0.68	0.84	0. 71	0.89

Table 4: Performance metrics for the four evaluated models.

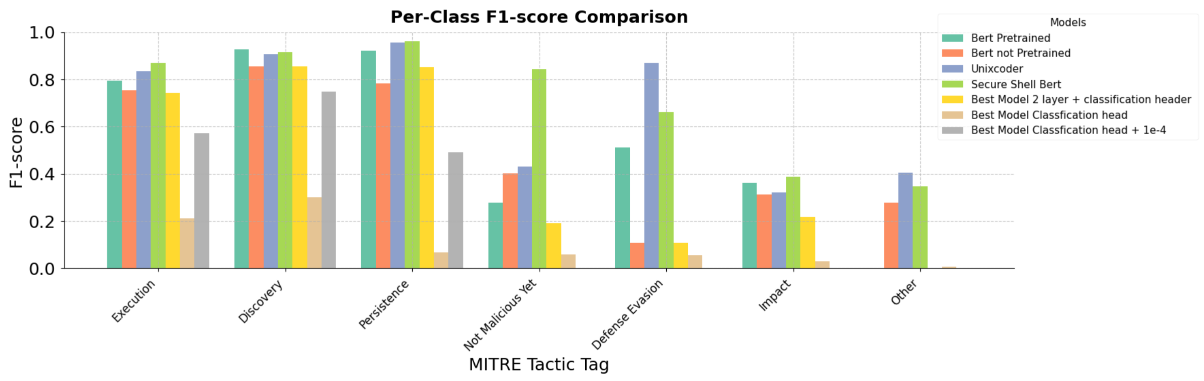


Figure 7: Model comparison on F1 score.

3.3 Partial Fine-tuning Strategies

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?

Model	Train Params	Train Time	F1-Score	Score Diff	Time Diff
Full Fine-tuning	124M	16:45	0.71	0.0%	0.0%
2 Layers + Classifier	14M	10:27	0.42	−40.8%	+61.3%
Classifier Only 1e-5	5K	4:44	0.11	−84.5%	+271.1%
Classifier Only 1e-4	5K	4:38	0.25	−64.8%	+279.6%

Table 5: Comparison of partial fine-tuning strategies for SecureShellBert.

Freezing the lower layers of the model during fine-tuning significantly reduces the number of trainable parameters and training time, but at the cost of a substantial drop in performance. Updating only the classifier results in a drastic decrease in F1 score to 0.11, while fine-tuning the last two encoder layers and the classifier yields a moderate F1 score of 0.42. This suggests that while partial fine-tuning can be more efficient, it may not capture the necessary task-specific features as effectively as full fine-tuning. Moreover increasing the learning rate for the classifier only from 1e-5 to 1e-4 seems to improve the performance of the model, but it is still far from the full fine-tuning performance.

4 Task 4 Inference

For the final task we had to use the best model obtained in the previous task to predict the labels of the test set and submit the results. The best model is SecureShellBert with full fine-tuning. As asked only the prediction for the first token of each Bash word are considered, and for truncated sessions, only the Bash words that received a prediction are included in the submission.

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 analyse an example of a session. Do the predictions make sense? Give 1 example of a session for each unique tuple (command, predicted tag).

Command	Discovery	Exec	Impact	Not Mali	Other	Persistence	Def Evasion
cat	861231	292	1	0	0	0	0
echo	229235	183795	0	723	242	345676	0
grep	995967	0	0	0	0	407	0
rm	295149	1725	0	0	0	5445	42315

Table 6: Example mapping of 4 Bash commands to the 7 MITRE labels used in the dataset.

As shown in table 6, some commands are strongly associated with specific MITRE labels. For instance, the command `cat` is predominantly linked to the **Discovery** label, while `echo` is frequently associated with **Persistence**. This indicates that certain commands have a higher likelihood of being used in specific contexts or for particular purposes within the dataset.

- **cat** is mainly used for **Discovery**, as it is often employed to read and display file contents, which is a common activity during the reconnaissance phase of an attack.
- **echo** is frequently associated with **Persistence**, as it can be used to create or modify files that help maintain access to a compromised system, but it is also used for **Execution**, **Discovery**.
- **grep** is primarily linked to **Discovery**, as it is used to search for specific patterns within files, aiding in information gathering.
- **rm** is associated with both **Discovery** and **Defense Evasion**, as it can be used to delete files, which may disrupt system operations or help evade detection by removing evidence of malicious activity.

For each command and each predicted tag, we will qualitatively analyze an example session in order to understand why the model predicted that tag and if the prediction is correct or not.

Cat command:

- **Execution**: cat /var/tmp/.systemcache436621 → This seams like a misclassification, the command is used to read a file and display its contents, which is more related to **Discovery** rather than **Execution**.
- **Discovery**: cat /proc/mounts → This is a correct classification, the command is used to read the file that contains information about the currently mounted filesystems, which is useful for gathering information about the system.
- **Impact**:cat passwd → This is a misclassification, the command is used to read the file that contains user account information, which is more related to **Discovery** rather than **Impact**.

Echo command:

- **Execution**: echo "IyEvYmluL2Jhc2gKY2QgL3RtcAkKc | base64 –decode | bash" → This is a correct classification, the command is used to decode a base64 encoded string and execute it as a bash script, which is an execution of a potentially malicious payload.
- **Discovery**: dd bs=52 count=1 if=.s ||cat .s || while read i; do echo $i; done < .s ;→ This is a correct classification, the command is used to read a file and display its contents, which is useful for gathering information about the system.
- **Persistence**: echo "root:XP3IUReH9hhH"|chpasswd|b ; → This is a correct classification, the command is used to change the password of the root user, which is a common technique used by attackers to maintain access to a compromised system.
- **Not Malicious Yet**: echo "321" ¿ /var/tmp/.var03522123 → This is a correct classification, the command is used to create a file and write a string to it, which is not necessarily malicious but could be used for malicious purposes in the future.
- **Other**: echo "ZXZhbCB1bnBhY2sgdT0+cXtfIkZVW |base64 –decode — perl → This is a misclassification, the command is used to decode a base64 encoded string and execute it as a perl script, which is more related to **Execution** rather than **Other**.

Grep command:

- **Discovery**: cat /proc/cpuinfo | grep name | wc -l ; → This is a correct classification, the command is used to read the file that contains information about the CPU and search for the name of the CPU, which is useful for gathering information about the system.
- **Persistence**: `LC_ALL=C grep "ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAABAQ"  /.ssh/authorized_keys` → This is a correct classification, the command is used to search for a specific string in a file and add it to the authorized keys file, which is a common technique used by attackers to maintain access to a compromised system.

Rm command:

- **Execution**: rm -rf /var/tmp/dota* ; → This is a misclassification, the command is used to delete files and directories, which is more related to **Defense Evasion** rather than **Execution**.

- **Discovery**: rm .s → This is a misclassification, the command is used to delete a file, which is more related to **Defense Evasion** rather than **Discovery**.
- **Persistence**: rm -rf .ssh → This is a correct classification, the command is used to delete the .ssh directory, which is a common technique used by attackers to maintain access to a compromised system.
- **Defense Evasion**: rm .s; → This is a correct classification, the command is used to delete a file, which is a common technique used by attackers to evade detection by removing evidence of their activity.

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.

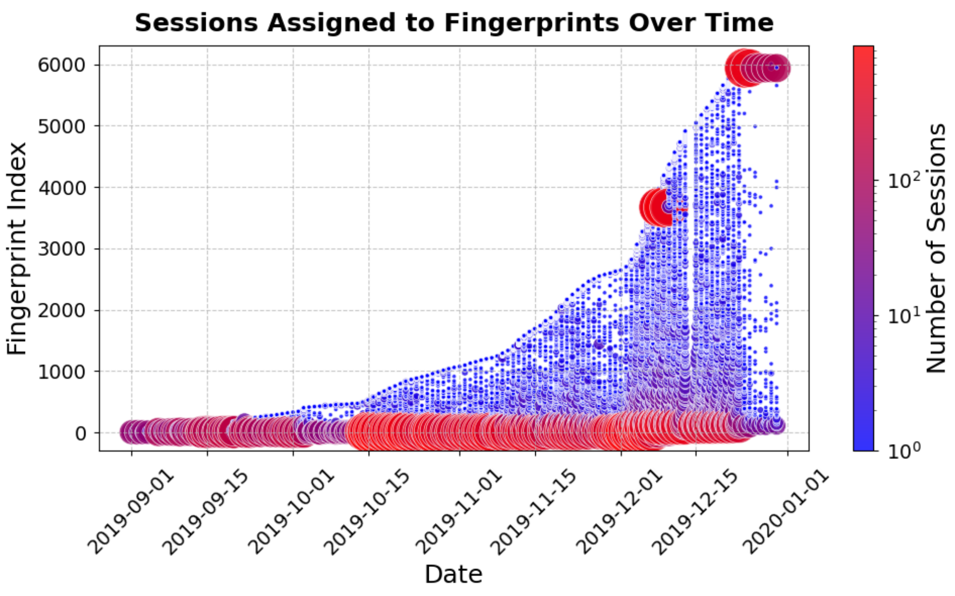


Figure 8: Sessions assigned to fingerprint over time

Figure 8 shows the sessions assigned to each fingerprint over time. We can see that some fingerprints are more active than others, with some fingerprints having multiple sessions in a short period of time. This could indicate that these fingerprints are associated with more active attackers or automated scripts.

There are 174262 different session and 6003 unique fingerprints identified. None of the fingerprints are present every day in our dataset, but some are present more frequently than others. There are 7 fingerprints that are present more than 80% of the days in the dataset and 3 of them are the same as the 3 most common fingerprints in the dataset 9.

This 3 fingerprints are really similar:

- ‘Discovery*9’, ‘Persistence*8’, ‘Discovery*102’, ‘Execution*11’, total session : 38202
- ‘Discovery*9’, ‘Persistence*8’, ‘Discovery*104’, ‘Execution*9’, total session : 25479
- ‘Discovery*9’, ‘Persistence*8’, ‘Discovery*103’, ‘Execution*10’, total session : 18471

Here is an example of a session assigned to the most common fingerprint:

```
cat /proc/cpuinfo | grep name | wc -l ; echo ‘‘root:
y7OH57fD4hew’‘|chpasswd|b ; echo ‘‘321’‘ > /var/tmp/.var03522123 ; rm -rf
/var/tmp/.var03522123 ; cat /var/tmp/.var03522123 | head -n 1 ; 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 ; echo ‘‘root pwlamea’‘ > /tmp/up.txt ; rm -rf /var/tmp/dota* ; cat
/var/tmp/.systemcache436621 ; echo ‘‘1’’ > /var/tmp/.systemcache436621 ; cat
/var/tmp/.systemcache436621 ; sleep 15s \&\& cd /var/tmp; echo
"IyEvYmluL2Jhc2gKY2QgL3RtcAkKc | base64 --decode | bash
```

The 3 most common fingerprints are really similar, a first part about Discovery and then a part about Persistence, with some variations in the middle part. After other Discovery commands, the last part is about Execution. This could indicate that these fingerprints are associated with a specific type of attack or attacker. Moreover the other fingerprints are similar to these 3, with some variations in the middle or on the last part.

Other example of fingerprints thet is present more than 80% of the days in the dataset and is the 8th most common fingerprint in the dataset is: ‘Discovery*9’, ‘Persistence*12’, ‘Discovery*77’, ‘Persistence*4’, ‘Execution*2’, ‘Defense Evasion*3’.

```
cat /proc/cpuinfo | grep name | wc -l ; echo -e
"S150Y25176460\nJHc29G7AFT8e\n ; echo "S150Y25176460\nJHc29G7AFT8e\n ; echo
"321" > /var/tmp/.var03522123 ; rm -rf /var/tmp/.var03522123 ; cat /var/tmp/.
var03522123 | head -n 1 ; 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) ; crontab -l ; w ; uname -m ; cat
/proc/cpuinfo | grep model | grep name | wc -l ; top ; uname ; uname -a ;
lscpu | grep Model ; echo "admin S150Y25176460" > /tmp/up.txt ; rm -rf
/var/tmp/dota*
```

This sort of fingerprint is similar to the previous ones, but with some variations, all this attacks are similar in the sequence of commands used and they are probably automated scripts used by attackers to perform reconnaissance, maintain access, and execute payloads on compromised systems.

Figure 9 shows the most common fingerprint in the dataset, with each color representing a different tag. We can see that this fingerprint is associated with multiple tags, indicating that the same fingerprint can be used for different purposes or by different attackers. This figure demonstrates that some sort of attacks are similar in the sequence of commands used, even if they are used for different purposes.

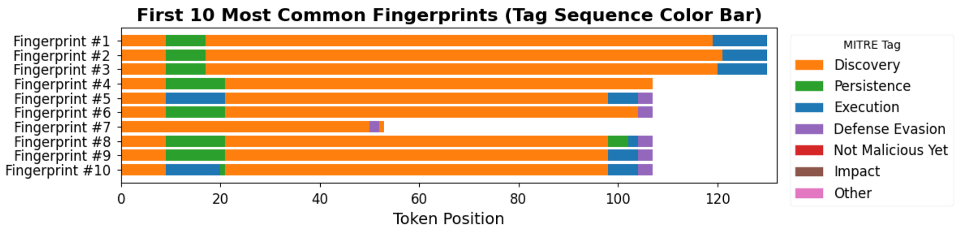


Figure 9: Most common fingerprint with sequence color