Politecnico di Torino

AI and Cybersecurity

Project 3: Natural Language Processing (NLP)

Use of pre-trained Language Models for the analysis of malicious SSH based logs

Angelo Barbera - s326432, Giorgia Moscato - s324647, Alessandro Genova - s330893

Table Of Contents

1 Task 1: Dataset Characterization

1.1 Explore the labels

Q: How many different tags do you have? How are they distributed? Plot a barplot to show the distribution of tags.
There are 7 unique tags: `Defense Evasion`, `Discovery`, `Execution`, `Impact`, `Not Malicious Yet`, `Other`, `Persistence`. The tag distribution is shown in Tab. 1.

Tactic	Train Count	Train %	Test Count	Test %
Defense Evasion	309	52.37	218	53.36
Discovery	6,009	28.23	3,307	25.30
Execution	3,239	9.87	1,568	11.02
Impact	312	2.72	133	3.52
Not Malicious Yet	264	2.69	212	3.42
Other	209	2.30	76	2.15
Persistence	1,133	1.82	683	1.23

Table 1: Distribution of MITRE tactics across training and test sets

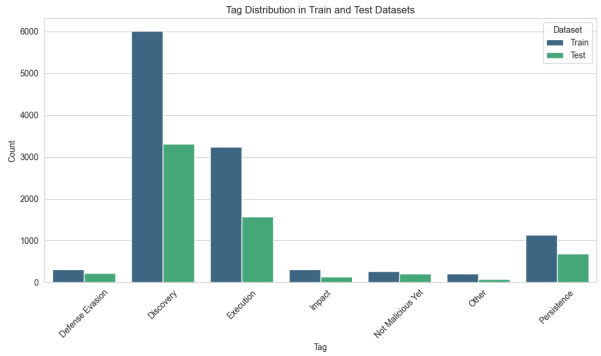

Figure 1: MITRE tactics barplot

1.2 Explore a single bash command – 'echo'

Q: 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 labelled differently?
The count of tags of `echo` bash command for the training and test set is shown in Tab. 2.

Tactic	Train Count	Test Count
Defense Evasion	0	1
Discovery	31	26
Execution	39	32
Impact	6	0
Not Malicious Yet	8	6
Other	4	0
Persistence	104	59

Table 2: Counts of SSH session tags in training and test sets

An example of a session where `echo` is assigned to `Persistence` is:

```
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 ;
```

An example of a session where `echo` is assigned to `Execution` is:

```
cat /var/tmp/.systemcache436621 ; echo 1 > /var/tmp/.systemcache436621 ; cat /var/tmp/.
    ↪ systemcache436621 ; sleep 15s && cd /var/tmp ; echo
    ↪ IyEvYmluL2Jhc2gKY2QgL3RtcAkKcm0gLXJmIC5zc2gKcm0gLXJmIC5tb3VudGZzCnJtIC1yZiAuWDEzLX
VuaXgKcm0gLXJmIC5YMTctdW5peApta2RpciAuWDE3LXVuaXgKY2QgLlgxNy11bml4Cm12IC92YXIvdG1wL2RvdGE
udGFyLmd6IGRvdGEudGFyLmd6CnRhciB4ZiBkb3RhLnRhci5negpzbGVlcCAzcyAmJiBjZCAvdG1wLy5YMTctdW5p
eC8ucnN5bmMvYwpub2h1cCAvdG1wLy5YMTctdW5peC8ucnN5bmMvYy90c20gLXQgMTUwIC1TIDYgLXMgNiAtcCAyM
iAtUCAwIC1mIDAgLWsgMSAtbCAxIC1pIDAgL3RtcC91cC50eHQgMTkyLjE2OCA+
    ↪ PiAvZGV2L251bGwgMj4xJgpzbGVlcA4bSAmJiBub2h1cCAvdG1wLy5YMTctdW5peC8ucnN5bmMvYy90c2
0gLXQgMTUwIC1TIDYgLXMgNiAtcCAyMiAtUCAwIC1mIDAgLWsgMSAtbCAxIC1pIDAgL3RtcC91cC50eHQgMTcyLjE
2ID4+IC9kZXYvbnVsbCAyPjEmCnNsZWVwIDIwbSAmJiBjZCAuLjsgL3RtcC8uWDE3LXVuaXgvLnJzeW5jL2luaX
RhbGwgMj4xJgpleGl0IDA= | base64 --decode | bash ;
```

These samples are labelled differently because in the first case, `echo` is used to change the root password, aligning with `Persistence`, while in the second, it triggers command execution, fitting the `Execution` tactic.

1.3 Explore the Bash words

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

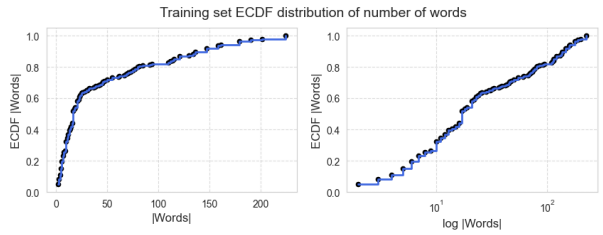

Figure 2: Training set ECDF

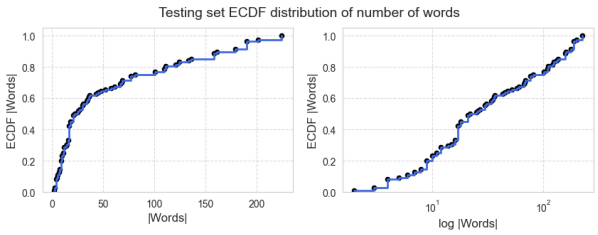

Figure 3: Test set ECDF

The ECDFs shown in Fig. 2 and Fig. 3 is used to plot the proportion of samples that have less than or equal to a given word number.

2 Task 2: Tokenization

2.1 Tokenize the following list of SSH commands: [`cat`, `shell`, `echo`, `top`, `chpasswd`, `crontab`, `wget`, `busybox`, `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 original sentence is `"cat shell echo top chpasswd crontab wget busybox grep"`.

The BERT tokenized sentence is: `['[CLS]', 'cat', 'shell', 'echo', 'top', 'ch', '##pass', '##wd',`

'cr', '##ont', '##ab', 'w', '##get', 'busy', '##box', 'gr', '##ep', '[SEP]'].
It contains **18** tokens, and the ratio between tokens and words is **2**. [CLS] and [SEP] are respectively the start and end of sequence tokens. The prefix ## indicates that a token is a continuation of the previous token.

The Unixcoder tokenized sentence is: ['<s>', 'cat', 'Ġshell', 'Ġecho', 'Ġtop', 'Ġch', 'passwd', 'Ġc', 'ront', 'ab', 'Ġw', 'get', 'Ġbusy', 'box', 'Ġgrep', '</s>'].
It contains **16** tokens, and the ratio between tokens and words is **1.78**. <s> and </s> are respectively the start and end of sequence tokens. The prefix Ġ indicates that the token has a space immediately before it.

Unixcoder has a lower token-to-word ratio than BERT because it was pre-trained on both natural language and source code, enabling it to better recognize bash commands and split them into more meaningful tokens (e.g., passwd is not split by Unixcoder, unlike BERT).

2.2 Now tokenize the entire training corpus with both tokenizers

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?
BERT generates on average **178** tokens with a maximum of **1889**. Unixcoder generates on average **409** tokens with a maximum of **28920**. This difference arises from their pre-training: BERT, trained on English text, often maps unknown sequences into [UNK] tokens, which keeps the token count relatively low. Unixcoder, trained also on code, avoids [UNK] and splits into many more meaningful sub-tokens, which increases the token count.

Q: How many sessions would currently be truncated for each tokenizer?
BERT's context size is **512**, with **24** truncated sessions. Unixcoder's context size is **1024**, with **8** truncated sessions.

2.3 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 with the maximum number of tokens contains **134** bash words. BERT generates **1889** tokens, while Unixcoder generates **28920**. The high token count is due to a long character sequence in the session. BERT uses [UNK] for unrecognized sequences, producing fewer tokens, while Unixcoder split them into multiple tokens, reflecting its code-specific pre-training.

2.4 Truncate words longer than 30 characters. Re-tokenize the processed sessions

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?

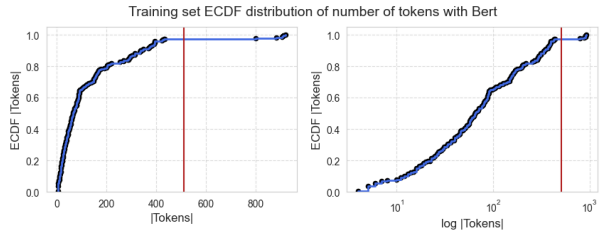

Figure 4: ECDF token distribution for BERT

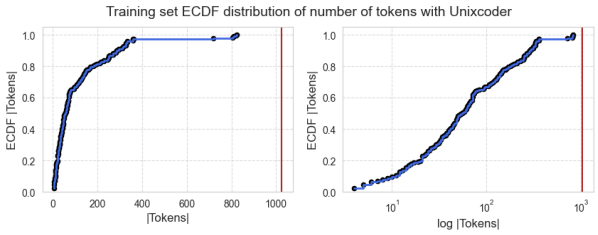

Figure 5: ECDF token distribution for Unixcoder

The distribution of tokens for BERT and Unixcoder is shown in Fig. 4 and Fig. 5. The vertical red line indicates the context size of the tokenizers.

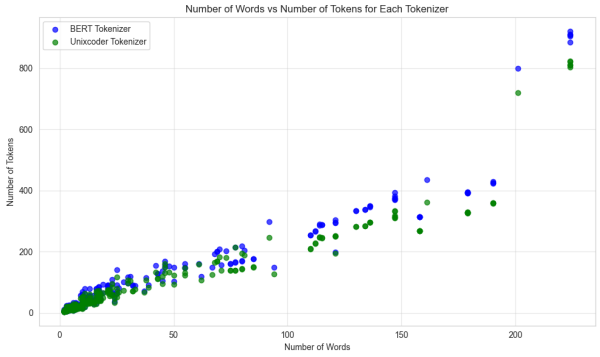

Figure 6: Number of words vs number of tokens

The number of words versus tokens is shown in Fig. 6. For sessions with fewer than 90 words, BERT generates slightly more tokens than Unixcoder; for more than 90 words, BERT generates significantly more.

Q: How many sessions now get truncated?
BERT has **6** truncated sessions, while Unixcoder has none due to its larger context size.

3 Task 3: Model Training

3.1 E2E fine-tune a BERT model

Q: Can the model achieve "good" results with only 251 training labeled samples? Where does it have the most difficulties?
Fine-tuning the `google-bert/bert-base-uncased` model for Named Entity Recognition leverages transfer learning from its pre-training on large datasets (e.g., 800M words from BookCorpus, 2500M from Wikipedia), enabling decent performance. The test set metrics are shown in Tab. 3.

Metric	Value
Average Session Fidelity	80.00%
Token Accuracy	82.25%
Token F1 Score	51.53%
Token Precision	81.01%
Token Recall	47.74%

Table 3: Test set metrics for E2E fine-tuned BERT

The model struggles with rare classes, as the F1 score (51.53%) is substantially lower than the accuracy (82.25%), indicating difficulties in correctly identifying minority tags. Classes with fewer than 2000 tokens exhibit poor performance, while more common classes (e.g., Execution, Discovery, Persistence) perform adequately.

3.2 Train 'naked' BERT in an end-to-end manner

Q: Can you achieve the same performance? Report your results
Training a "naked" BERT model (without pre-trained weights) from scratch for 30 epochs with a learning rate of 1e-5 yields significantly worse performance compared to fine-tuning. The experimental results on the test set are shown in Tab. 4.

Metric	Value
Average Session Fidelity	71.00%
Token Accuracy	72.49%
Token F1 Score	39.40%
Token Precision	54.01%
Token Recall	38.05%

Table 4: Test set metrics for naked BERT

Compared to the fine-tuned model, this results in a drop of approximately 10 percentage points in accuracy (82.25% - 72.49% = **9.76 pp**) and 27 points in precision (81.01% - 54.01% = **27.00 pp**), due to the lack of pre-trained knowledge and the small dataset (251 samples), which is insufficient for effective learning from scratch.

3.3 E2E fine-tune Unixcoder

Q: Since Unixcoder was pre-trained on code, it should have stronger priors on shell commands. Can you confirm this hypothesis? How do its metrics compare to the previous models?
Unixcoder, pre-trained on both natural language and source code, was expected to better parse and represent bash tokens. After fine-tuning on our SSH logs, it achieves higher scores than the fine-tuned and 'naked' BERT, as can be seen in Tab. 5.

Metric	Value
Average Session Fidelity	82.00%
Token Accuracy	85.35%
Token F1 Score	62.48%
Token Precision	75.84%
Token Recall	57.81%

Table 5: Test set metrics for E2E fine-tuned Unixcoder

Compared to fine-tuned BERT's 80% fidelity and 82.25% accuracy, Unixcoder shows a **2 percentage point** gain in fidelity (82% - 80%) and a **3 percentage point** gain in accuracy (85.35% - 82.25% = **3.10 pp**), confirming that its code-aware pre-training enhances SSH log understanding. The improvements in F1 (62.48% - 51.53% = **10.95 pp**) and recall (57.81% - 47.74% = **10.07 pp**) further highlight its superior ability to detect minority (malicious) tactics.

3.4 E2E fine-tune Secure-Shell-BERT

Q: How will Secure-Shell-BERT perform against a newer, code-specific model such as Unixcoder? Has the performance changed? Which is your best model?
We fine-tuned the entire Secure-Shell-BERT model end-to-end with a learning rate of 1e-5. The confusion matrices is shown in Fig. 7 and the metrics in Tab. 6.

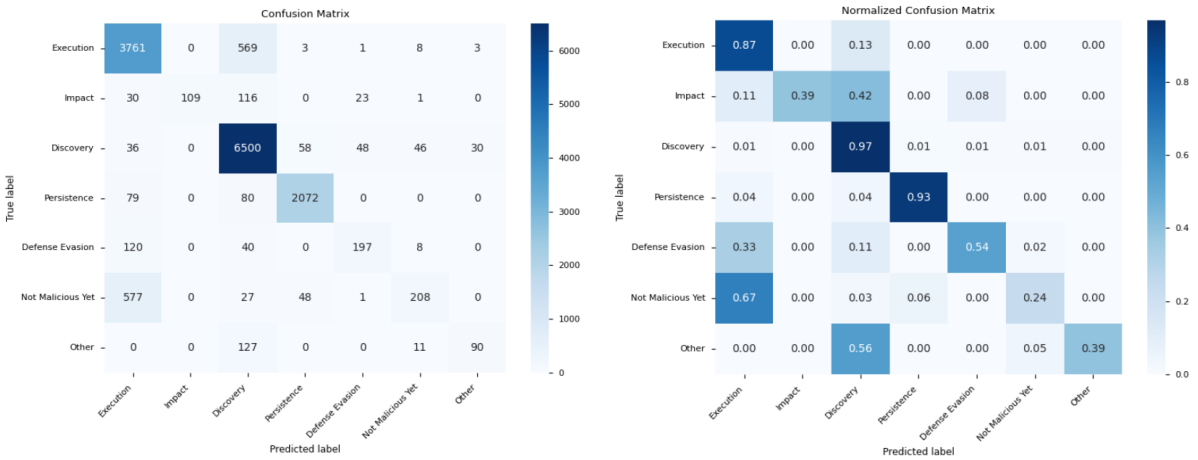

Figure 7: Confusion matrices (absolute and normalized) for e2e fine-tuned Secure-Shell-BERT.

Metric	Value
Average Session Fidelity	85.00%
Token Accuracy	86.09%
Token F1 Score	67.94%
Token Precision	83.39%
Token Recall	61.84%

Table 6: Test set metrics for E2E fine-tuned Secure-Shell-BERT

For reference, the other models yield:

- **Unixcoder**: fidelity 82%, accuracy 85.35%, F1 62.48%, precision 75.84%, recall 57.81%.
- **Fine-tuned BERT**: fidelity 80%, accuracy 82.25%, F1 51.53%, precision 81.01%, recall 47.74%.

Secure-Shell-BERT outperforms both Unixcoder and fine-tuned BERT, achieving the highest fidelity (85%), accuracy (86.09%), and F1 score (67.94%). We selected Secure-Shell-BERT as our best model due to its superior overall performance, particularly in recall (61.84%), which is critical for minimizing malicious false negatives in cybersecurity applications.

3.5 Lightweight fine-tune on the best model Secure-Shell-BERT

Q: How many parameters did you fine-tune when most layers were 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?
We experimented with two lightweight fine-tuning strategies on Secure-Shell-BERT, both using a learning rate of 1e-4, the metrics and the trainable parameters are shown in Tab. 7. The confusion matrices with the last two layers and classifier trained are shown in Fig. 8.

Scenario	Trainable	Frozen	Fidelity	Accuracy	F1	Precision	Recall
Last 2 layers + classifier	14M	109M	84.00%	85.03%	66.86%	82.14%	60.73%
Classifier only	5k	124M	52.00%	57.25%	21.15%	25.84%	22.54%

Table 7: Trainable and frozen weights + Test set metrics for lightweight fine-tuned Secure-Shell-BERT

Figure 8: Confusion matrices (absolute and normalized) for lightweight fine-tuned Secure-Shell-BERT with the last two layers and classifier trained.

- **Training Efficiency**: As expected, freezing more layers significantly speeds up training. Fine-tuning only the last 2 layers and classifier stopped early at epoch 15 with a total training time of 116.90 seconds (0.12 epochs/s). Fine-tuning only the classifier completed in 180.46 seconds up to epoch 29 (0.16 epochs/s). In contrast, full end-to-end fine-tuning stopped at epoch 24 with a total training time of 370.07 seconds (0.06 epochs/s).
- **Learning Rate**: We increased the learning rate to 1e-4 (from 1e-5) when freezing layers to improve convergence and achieve competitive results.
- **Performance Trade-off**: Tuning only the last two encoder layers yields competitive performance, basically the same of full fine-tuning, while being far more computationally efficient. Fine-tuning only the classifier results in a substantial performance loss (F1 21.15% vs. 67.94%, a drop of **46.79 pp**).

4 Task 4: Inference

4.1 Focus on the commands `cat`, `grep`, `echo`, `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).
Analysing the Tab. 8 containing the label frequencies per command we can observe that:

- `grep` is almost exclusively tagged as `Discovery` (over 99.9% of occurrences).
- `cat` is mostly classified as `Discovery`, only a few examples are marked `Execution`, and there is just a single `Impact` case.
- `echo` is split across three major tactics, `Persistence`, `Execution` and `Discovery`, showing its context sensitivity.
- `rm` spans `Execution`, `Defense Evasion` and `Persistence`, reflecting its dual use in cleanup and active operations.

Comm.	Discovery	Persistence	Execution	Other	Not Malicious Yet	Defense Evasion	Impact
`grep`	995987	387	0	0	0	0	0
`cat`	860906	0	621	0	0	0	1
`echo`	200770	406683	150260	186	1750	1	24
`rm`	259004	5469	74698	0	0	5468	1

Table 8: Tag frequencies per command

The Tab. 9 shows a session example for each unique tuple (command, predicted tag). The model's contextual understanding, enabled by transformer-based embeddings, assigns tags based on surrounding tokens, making predictions meaningful.

Command	Predicted Tag	Session Example
grep	Discovery	`grep -R "root" /etc/passwd`
grep	Persistence	`some_script.sh | grep "cron" | tee /var/spool/cron/root`
cat	Discovery	`cat /proc/mounts | grep ext4`
cat	Execution	`cat payload.bin | base64 -d | bash`
cat	Impact	`cat /etc/shadow > /tmp/shadow_backup`
echo	Persistence	`echo "root:NewPass123" | chpasswd`
echo	Execution	`echo "YmFzaCAtaSA+JiAvZGV2L3RjcDIvMTIzNCY=" | base64 -d | bash`
echo	Discovery	`echo "Checking disk"; df -h`
rm	Defense Evasion	`rm -f /var/log/auth.log`
rm	Execution	`rm -rf /tmp/install_dir && ./install.sh`
rm	Impact	`rm -rf /home/user/documents`

Table 9: SSH session examples for command-tag tuples

4.2 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

The plot in Fig. 9 shows sessions per fingerprint ID over time: the color represents the session count; y is the jittered ID. We can observe that:

- **Suspicious campaigns**:
  - **Dec 8–12**: A compact yellow cluster at mid/high IDs suggests one or a few fingerprints suddenly generated many sessions (typical of scripted scans).
  - **Dec 23–30**: Activity extends to the very top of the ID range with bright yellow points; this looks like an intensive, short-lived burst from very high-ID fingerprints.
- **Different zones**:
  - **Low-ID group (about the first 50 IDs)**: is present essentially across the whole timeline and with high counts. This looks like background/infrastructure traffic that never disappears.
  - **From December**: The fingerprint unique id curve rises sharply in December. This sharp increase could be due to a coordinated attack campaign exploiting new vulnerabilities, leading to the rapid generation of new fingerprint IDs.
  - **Dec 14**: A narrow vertical gap around seems to be a collection gap rather than attacker behavior.
  - **Dec 24**: After December 24, the plot becomes noticeably sparser, with activity concentrated in the upper range (at 4800) where bright yellow density persists. This suggests that while overall fingerprint diversity decreases, high-id fingerprints remain active. Possible explanations include a shift to targeted attacks on high-activity fingerprints, or exhaustion/resolution of earlier campaigns on lower-range fingerprints, or maybe just limitations/problems in data collection.

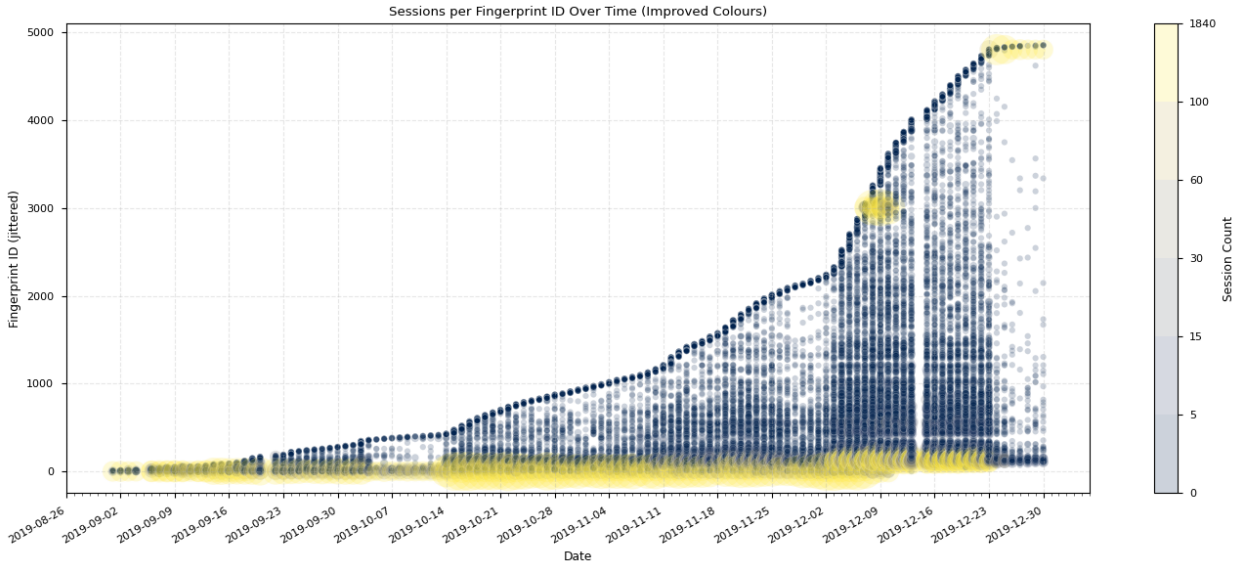

Figure 9: Sessions per Fingerprint ID over time