Project 4: Natural Language Processing
Group 2

Chiara Iorio - s343732, Renato Mignone - s336973,
Claudia Sanna - s343470

The goal of this laboratory is to apply Natural Language Processing (NLP) techniques in the context of cybersecurity. Specifically, we will analyze a dataset consisting of SSH sessions, represented as sequences of SSH entities, such as commands, flags, parameters, and separators. We will perform dataset characterization, then we will analyze the impact of different tokenization strategies and experiment with various pre-trained Language Models, comparing their performance. Finally, we will use the fine-tuned model for inference to investigate cybersecurity threats and infer the intentions of potentially malicious users.

1 Dataset Characterization

In this section, we explore the provided training and test datasets.

Labels exploration Fig. 1a illustrates the distribution of tags in both the training and test sets. The datasets comprise seven distinct tags: *Defense Evasion*, *Discovery*, *Execution*, *Impact*, *Not Malicious Yet*, *Other*, and *Persistence*. Generally, the number of Bash words assigned to a tag is higher in the training set than in the test set. In fact, the training set contains 11475 Bash words distributed among 251 sessions, while the test set only 6197 (108 sessions). Both sets exhibit a similar distribution profile: *Discovery* ($\approx 50\,\%$) is the most prevalent tag, followed by *Execution* ($\approx 25\,\%$), and *Persistence* ($\approx 10\,\%$). The remaining four tags are significantly less frequent, each associated with less than $4\,\%$ of Bash words.

The `echo` command The `echo` command has been assigned to 6 different tags: *Persistence* (104 times), *Execution* (39 times), *Discovery* (31 times), *Not Malicious Yet* (8 times), *Impact* (6 times), and *Other* (4 times). We will now analyze two examples: one session in which `echo` is labelled as *Persistence* and one as *Execution*.

(`echo`, Persistence): `[...]; echo "root:HGbB4i9gUXMh" | chpasswd | bash ; [...]`. The `echo` command outputs the string `root:HGbB4i9gUXMh`, which is piped to `chpasswd`, that updates user passwords from standard input. By changing the `root` user's password to a known value, the attacker gains persistent administrative access to the compromised system.

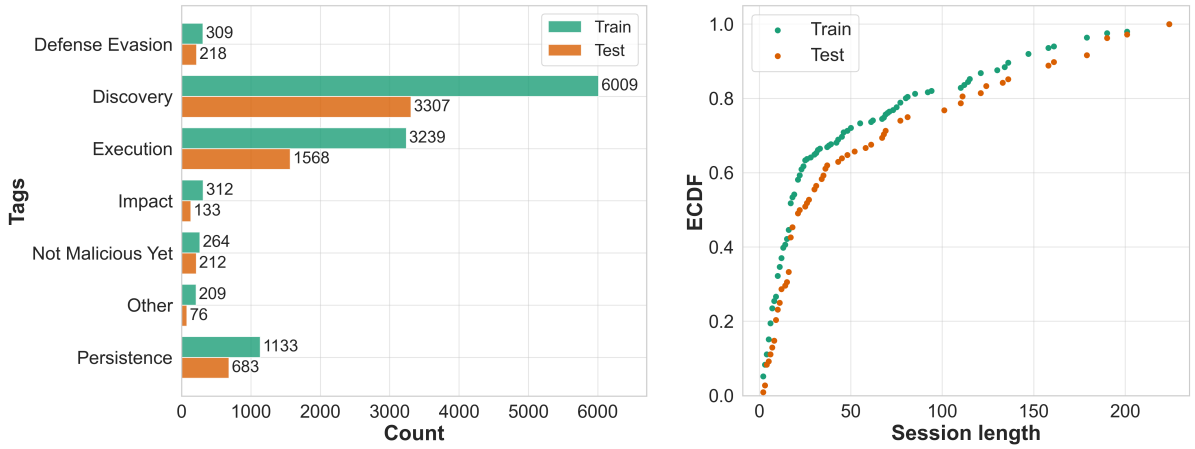
(`echo`, Execution): `[...]; echo "base64_payload" | base64 --decode | bash ;`. The `echo` command outputs a base64 payload, that is then decoded and executed. Hence, this sequence executes arbitrary code, probably a malware binary.

In both cases, the `echo` command is used to output a string. However, the meaning of this string and the goal of the subsequent commands influence the labelling of `echo` itself: *Persistence* in the first case, and *Execution* in the second one.

Bash words exploration Fig. 1b shows the ECDF of Bash words per session in both the training and test sets. The distribution shows that most sessions are relatively short, with $50\,\%$ containing less than 25 words, and $\approx 80\,\%$ less than 100 words. While both datasets share the same minimum (2 words) and maximum (224 words) lengths, the test set is generally characterized by longer sessions. Specifically, the mean session length in the test set is 57.38 words, compared to 45.70 words in the training set. This right-skewed distribution indicates the presence of a few longer sessions that contrast with the high frequency of shorter ones.

2 Tokenization

In this section, we compare the tokenization behavior of two pre-trained tokenizers: BERT (google-bert/bert-base-uncased), a general-purpose model trained on English text, and UniXcoder (microsoft/unixcoder-base), a code-oriented model designed for programming languages.



(a) Tag distribution.

(b) ECDF of Bash words per session.

Figure 1: Plots about Dataset Characterization. Both plots refer to the training and test sets.

Command tokenization We first tokenize a sample list of SSH commands: `cat`, `shell`, `echo`, `top`, `chpasswd`, `crontab`, `wget`, `busybox`, and `grep`. The BERT tokenizer divides words into subword units based on frequency in general English text, while the UniXcoder tokenizer understands common shell commands and structures from its code-centric training. Both tokenizers keep certain words intact (e.g., `cat`, `echo`, `grep`) because they represent high-frequency tokens in their respective training data: UniXcoder has many bash commands in its vocabulary from code training, while BERT retains them only if they appear frequently in general text.

Full corpus tokenization When tokenizing the entire training corpus, BERT produces on average 178.6 tokens per session (max 1889 tokens), while UniXcoder produces 409.3 tokens per session (max 28 920 tokens). Both tokenizers produce high token counts because SSH sessions contain many special characters and non-standard strings. BERT generates fewer tokens because it maps unknown or rare character sequences to `[UNK]` tokens, while UniXcoder, designed for code, attempts to tokenize every character sequence individually. Specifically, in the longest session (224 words), BERT produces 113 `[UNK]` tokens, while UniXcoder produces zero unknown tokens but generates 28920 tokens by splitting special characters. UniXcoder performs code-specific splitting on characters like `{`, `&&`, and handles file paths by tokenizing each path component separately.

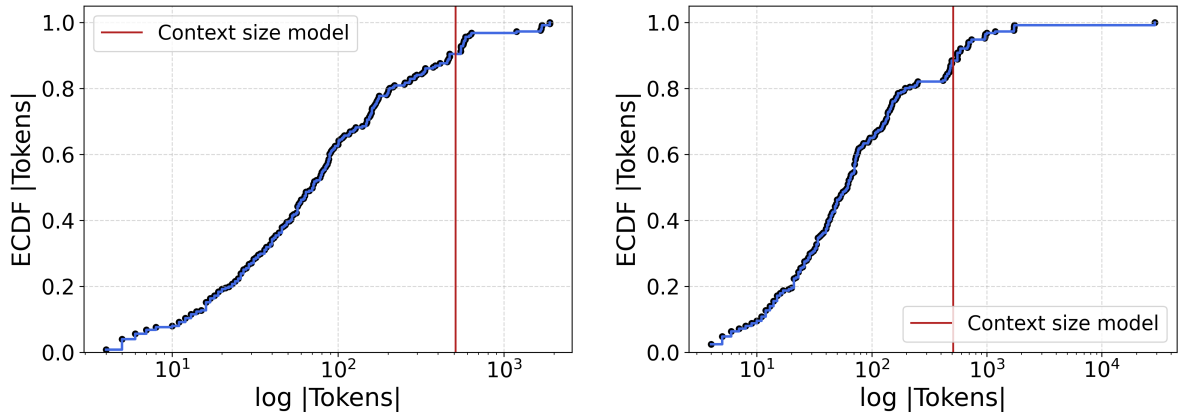
Fig. 2 shows the ECDF distribution of token counts. For BERT (fig. 2a), approximately 90 % of sessions fall below the 512-token limit, with a steep rise indicating that most sessions cluster around 100–300 tokens. For UniXcoder (fig. 2b), only about 87 % of sessions remain below the limit, with a longer tail extending to nearly 30000 tokens due to the verbose tokenization of special characters. With the 512-token limit, 24 sessions would be truncated by the BERT tokenizer and 29 by the UniXcoder tokenizer.

Word truncation preprocessing Since 98 % of words are shorter than 30 characters, we truncate all words exceeding this length before tokenization. After preprocessing, token counts decrease significantly: BERT produces on average 128 tokens per session (max 613), while UniXcoder produces 111 tokens per session (max 540). This reduces truncated sessions to only 6 for both tokenizers.

The token-to-word ratios after preprocessing are 2.79 for BERT and 2.43 for UniXcoder. **UniXcoder achieves the better (lower) ratio**, indicating more efficient tokenization for SSH commands. Fig. 3 visualizes this relationship: both plots show a linear correlation between words and tokens, but UniXcoder's scatter points lie closer to the diagonal (slope ≈2.4) compared to BERT (slope ≈2.8), confirming UniXcoder's more compact representation for this domain.

3 Model training

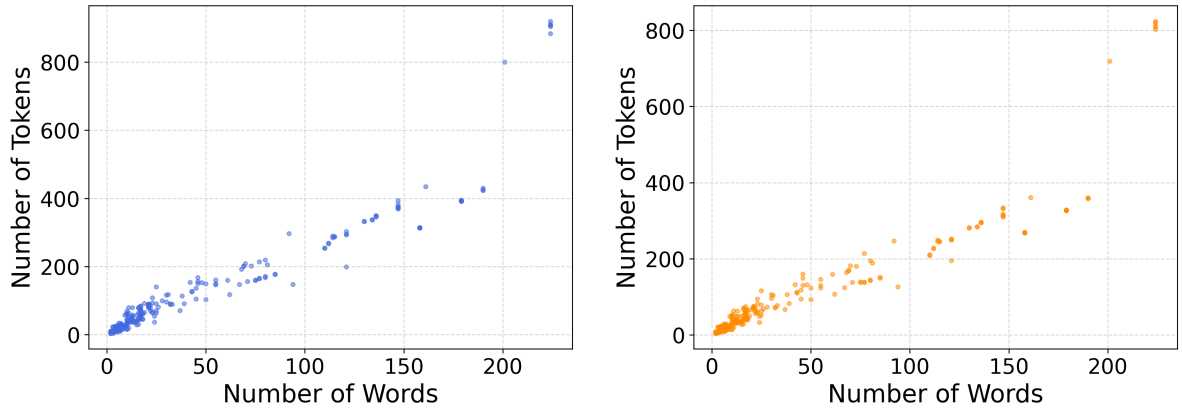
In this section, we train and compare different transformer-based models for token-level classification of SSH sessions into MITRE tactics. We evaluate: (1) pre-trained BERT, (2) randomly initialized (naked) BERT, (3) pre-trained UniXcoder, and (4) alternative fine-tuning strategies with frozen layers. For each



(a) ECDF for BERT.

(b) ECDF for UniXcoder.

Figure 2: ECDF distribution of tokens per session. The red line indicates the 512-token context limit.



(a) BERT.

(b) UniXcoder.

Figure 3: Number of words vs number of tokens per session after word truncation at 30 characters.

configuration, we perform a grid search over learning rates and select the best model based on validation macro F1-score. We split the data into 80% training and 20% validation, using the provided test set for final evaluation.

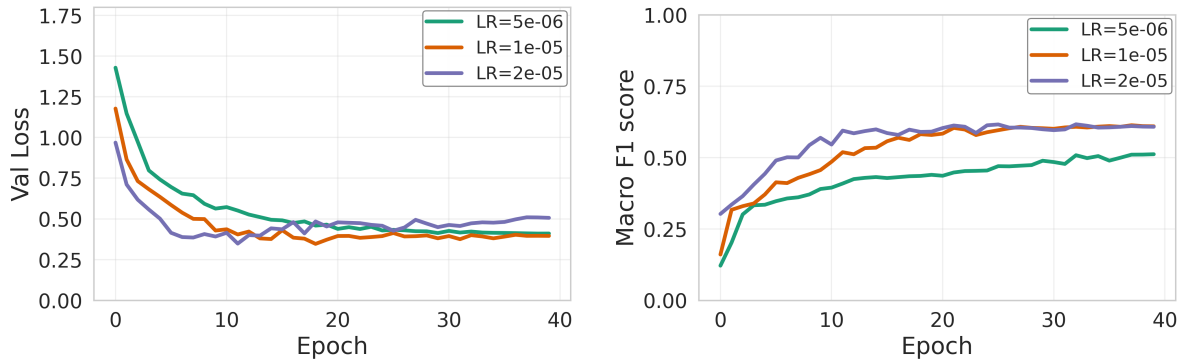
3.1 Fine-tuning Pre-trained BERT

We fine-tune a pre-trained BERT model (google-bert/bert-base-uncased) for token classification. After grid search over learning rates (5×10^{-6}, 10^{-5}, 2×10^{-5}), we select the configuration with 20 epochs based on validation performance. Fig. 4 shows the validation loss and macro F1-score curves for different learning rates: the 10^{-5} learning rate achieves the best balance between convergence speed and final performance, with validation loss stabilizing around epoch 10 and F1-score reaching its peak around 0.68. Lower learning rates (5×10^{-6}) converge more slowly, while higher ones (2×10^{-5}) show faster initial progress but slightly worse final F1.

The model performs well on high-support classes (*Discovery*: 0.94, *Execution*: 0.91) but struggles with minority classes (*Defense Evasion*: 0.40, *Impact*: 0.31, *Other*: 0.25), as shown in fig. 6. The recall-precision imbalance (0.57 vs 0.84) indicates that the model is conservative, preferring precision over recall for rare classes.

3.2 Naked BERT Baseline

To verify that pre-training is essential, we train a naked BERT model—the same architecture initialized with random weights—from scratch. After grid search (5×10^{-6}, 10^{-5}, 5×10^{-5}), we select the configuration with 25 epochs based on validation F1-score.



(a) Validation loss.

(b) Validation macro F1-score.

Figure 4: Validation curves for pre-trained BERT with different learning rates.

Table 1: Comparison between pre-trained BERT and naked BERT on the test set. Precision, Recall, and F1 are computed using macro-averaging.

Model	**Token Acc.**	**Precision**	**Recall**	**F1**
Pre-trained BERT	0.8694	0.8402	0.5746	0.6399
Naked BERT	0.7402	0.5469	0.4775	0.4925

Table 1 compares the results, while fig. 6 visualizes the per-class F1 differences. The naked BERT achieves only 74.02 % token accuracy, a drop of nearly 13 percentage points compared to the pre-trained version. This confirms that SSH command classification is not trivial: the 251 training samples are insufficient for learning from scratch, and pre-trained weights provide essential linguistic knowledge that the naked model cannot discover with such limited data.

3.3 Fine-tuning UniXcoder

Since UniXcoder was pre-trained on a large code corpus, we hypothesize it has more relevant prior knowledge for SSH command analysis. After grid search (5×10^{-6}, 10^{-5}, 4×10^{-5}), we select the configuration with 15 epochs. Fig. 5 shows the validation curves: the 10^{-5} learning rate achieves stable convergence with validation loss decreasing smoothly and F1-score reaching ≈0.72, outperforming all BERT configurations.

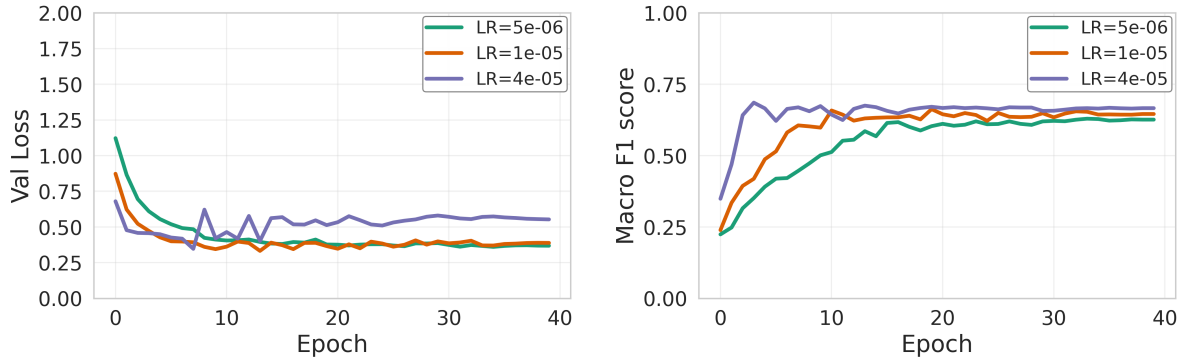
The hypothesis is confirmed: UniXcoder achieves 89.01 % token accuracy, outperforming pre-trained BERT by over 2 percentage points. The macro F1-score improves from 0.6399 to 0.6909 (+5.1 points). UniXcoder's specialized vocabulary and representations learned from code corpora enable better handling of shell syntax, operators, and path structures. As shown in fig. 6, UniXcoder is significantly better at identifying *Defense Evasion* samples (F1-score 0.76 vs 0.40 for BERT), likely because code pre-training helps recognize obfuscation patterns.

3.4 Alternative Fine-tuning Strategies

We explore parameter-efficient fine-tuning by freezing portions of the UniXcoder encoder. We test two configurations: (1) training only the last 2 encoder layers + classification head, and (2) training only the classification head. Full fine-tuning trains all 125.3M parameters, "Last 2 + Head" trains 14.18M (11.31 %), and "Head Only" trains only 5.38k (0.004 %).

Full fine-tuning achieves the highest F1-score (≈0.72) with stable convergence. "Last 2 + Head" reaches F1 ≈0.67, demonstrating that freezing most layers still yields competitive results. "Head Only" converges quickly but plateaus at F1 ≈0.57, indicating that the frozen encoder cannot fully adapt to the SSH domain.

Performance trade-offs Table 2 summarizes the test set results. "Last 2 + Head" loses only 1.87 % accuracy while reducing trainable parameters by 88.69 %, offering a good efficiency-performance trade-off with 2–3× training speedup. However, "Head Only" loses 8.66 % accuracy because frozen representations cannot capture domain-specific patterns.



(a) Validation loss.

(b) Validation macro F1-score.

Figure 5: Validation curves for UniXcoder with different learning rates.

Table 2: Parameter-efficient fine-tuning comparison for UniXcoder on the test set. Metrics are macro-averaged.

Configuration	Trainable Params.	Token Acc.	F1
Full Fine-Tune	100.00%	0.8901	0.6909
Last 2 + Head	11.31%	0.8714	0.6521
Head Only	0.004%	0.8035	0.5628

Fig. 6 provides a direct comparison of per-class F1-scores across all model configurations, including support counts for each class. UniXcoder (Full) consistently outperforms both BERT variants on minority classes: *Defense Evasion* improves from 0.40 (BERT) to 0.76 (+36 points), and *Impact* from 0.31 to 0.55 (+24 points). Naked BERT confirms the importance of pre-training, with substantially lower F1-scores across all classes. Progressive freezing degrades performance proportionally: "Last 2 + Head" maintains competitive scores on majority classes but drops on *Defense Evasion* (0.59) and *Other* (0.18). "Head Only" shows the steepest decline, particularly on *Defense Evasion* (0.24), confirming that adapting encoder representations is essential for recognizing obfuscation patterns.

3.5 Summary and Best Model Selection

Table 3 summarizes all models on the test set. The fully fine-tuned UniXcoder achieves the best performance with 89.01 % token accuracy and 69.09 % macro F1-score, demonstrating that code-specific pre-training provides a significant advantage for SSH command classification. We select this model (trained for 15 epochs) for the inference task.

4 Inference

We now focus on the inference dataset (*cyberlab.csv*). First, we applied the same preprocessing adopted previously, truncating words longer than 30 characters. Then, we used the best model identified in §3 (i.e., UnixCoder-base, $lr = 10^{-5}$, 15 epochs) to predict MITRE tags for all the inference sessions, assigning to each bash word the tag associated with the first corresponding token.

4.1 Predicted tags analysis

Tags frequency for `cat`, `grep`, `echo`, and `rm` Table 4 summarizes the tag frequencies for the required commands (`cat`, `grep`, `echo`, and `rm`), computed as the number of times that tag was assigned to a command, divided by the total number of occurrences of that command. Only `grep` is uniquely associated with a single tag (*Discovery*), while `cat` is assigned to two different tags: *Discovery* (90.9 % of times) and *Execution* (9.1 %). `echo` is mainly associated with three different tags: *Persistence*, *Discovery* and *Execution*. Finally, `rm` is assigned to four tags (*Discovery*, *Execution*, *Defense Evasion* and *Persistence*). Considering these commands, some tags are almost never associated with any session; this is the case of *Impact*, *Not Malicious Yet*, and *Other*.

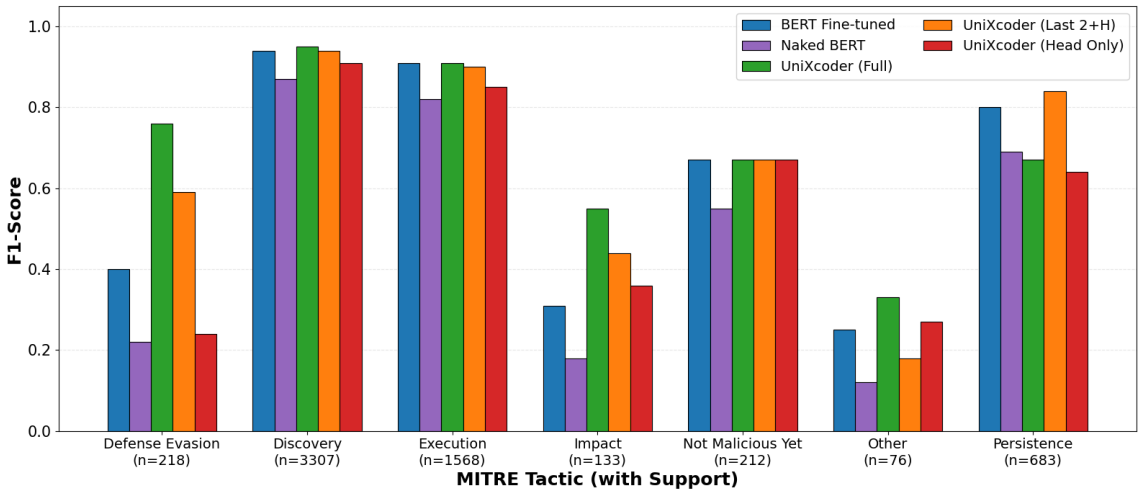


Figure 6: Per-class F1-scores on the test set for all model configurations, with support counts shown below each class. UniXcoder (Full) achieves the best performance on minority classes, while Naked BERT confirms the importance of pre-training.

Table 3: Summary of all models on the test set. Precision, Recall, and F1 are macro-averaged. Best results in bold.

Model	Token Acc.	Precision	Recall	F1
Pre-trained BERT	0.8694	0.8402	0.5746	0.6399
Naked BERT	0.7402	0.5469	0.4775	0.4925
UniXcoder (Full)	**0.8901**	0.8359	**0.6265**	**0.6909**
UniXcoder (Last 2+H)	0.8714	0.8043	0.5972	0.6521
UniXcoder (Head Only)	0.8035	0.7206	0.5093	0.5628

Session examples We will now qualitatively analyze one example of a session for each unique tuple (command, predicted tag), focusing only on those appeared more than 1 % of times.

(`cat`, Discovery): `[...]; cat /proc/mounts; [...]`. This command reads and displays the contents of the file `/proc/mounts`, listing information about storage devices, network mounts and virtual file systems. Attackers can use it to analyze the target's file system structure before attempting further exploitation or lateral movement. Since it is purely information gathering, the tag *Discovery* is appropriate.

(`cat`, Execution): `[...]; echo "1" > /var/tmp/.sysc436621 ; cat /var/tmp/.sysc436621 ; [...]`[1]. In this case, `cat` reads a hidden file from the temporary directory, created by a previous `echo`. Even if the `cat` command itself does not execute any malicious code (it looks like a validation step), it has likely been marked as *Execution* because it precedes the subsequent code execution in the attack chain. Based on that, a more appropriate tag for this instance may have been *Not Malicious Yet*.

(`grep`, Discovery): `cat /proc/cpuinfo | grep name | wc -l ; [...]`. The `grep` command filters the `cat` output to find lines containing the word "name", enabling the attacker to profile the target

[1]The file path has been shortened for visualization purposes. The original one is `/var/tmp/.systemcache436621`.

Table 4: Tags frequency for `cat`, `grep`, `echo`, and `rm`.

Tag	`cat`	`grep`	`echo`	`rm`
Defense Evasion	0.000	0.000	0.000	0.049
Discovery	0.909	1.000	0.334	0.738
Execution	0.091	0.000	0.234	0.196
Impact	0.000	0.000	0.000	0.000
Not Malicious Yet	0.000	0.000	0.001	0.000
Other	0.000	0.000	0.004	0.000
Persistence	0.000	0.000	0.427	0.017

system's CPU specifications. The *Discovery* tag looks accurate, since attackers are gathering hardware details before attempting exploitation or lateral movement.

(echo, Persistence): `[...]; echo "root:XP3IUReH9hhH" | chpasswd | bash ; [...]`. The `echo` command outputs the string `root:XP3IUReH9hhH`, which is piped to `chpasswd`, that updates user passwords from standard input. By changing the `root` user's password to a known value, the attacker gains persistent administrative access to the compromised system. As this session looks very similar to the one from the training set already analyzed in §1, the *Persistence* tag is appropriate.

(echo, Discovery): `[...]; echo "root 1ntr4n3t" > /tmp/up.txt ; [...]`. It looks like the `echo` command is used to save credential credentials to the `/tmp/up.txt` file, potentially for future access. Hence, a *Persistence* tag may be more appropriate.

(echo, Execution): `[...]; echo "base64_payload" | base64 --decode | bash ;`. The `echo` command outputs a base64 payload, that is then decoded and executed. Hence, the command executes arbitrary code, probably a malware binary. As this session looks very similar to the one from the training set already analyzed in §1, the *Execution* tag looks correct.

(rm, Discovery): `[...]; echo "321" > /var/tmp/.v03522123; rm -rf /var/tmp/.v03522123; [...]`[2]. This command sequence writes `321` into a file, which is then immediately deleted. The model has probably marked the `rm` command as *Discovery*, because it is surrounded by file operations that can be associated with a discovery phase, and could be motivated as a permission check. This looks reasonable, given that this sequence does not seem relevant by itself from an attacker's point of view.

(rm, Execution): `[...]; rm -rf /var/tmp/dota* ; [...]`. This command deletes all the files and folders whose names begin with `dota` in the `/var/tmp` directory. This is an active operation, removing files that may contain traces of previous actions. Hence, in that case the *Defense Evasion* tag would have been more appropriated.

(rm, Defense Evasion): `[...]; cp /bin/echo .s ; [...]; rm .s ; exit`. This command sequence copies the `echo` binary in `.s`, that is eventually deleted as last step. This operation is clearly *Defense Evasion* because it is the final cleanup step that removes forensic evidence of the attack.

(rm, Persistence): `[...]; echo "admin Passwd4" > /tmp/up.txt ; rm -rf /var/tmp/dota*`. This command deletes all the files and folders whose names begin with `dota` in the `/var/tmp` directory. Hence, it looks like a *Defense Evasion* technique, removing files that may contain traces of previous actions. However, the model probably considers it as *Persistence* due to the previous steps, such as the `echo` command that saves the credentials to the `/tmp/up.txt` file, making them persistent across restarts.

4.2 Fingerprint Analysis

Subsequently, we focused on the fingerprints extracted from the predictions for the inference dataset. After having identified the unique fingerprints, we sorted them by "date of birth" and we assigned to each an incremental ID. Then, we counted the number of sessions described by a specific fingerprint for each date.

Fig. 7 shows fingerprints IDs appearance over time: the y-axis lists fingerprint IDs (the lower the value, the earlier the fingerprint appeared for the first time), while the x-axis represents the full data-collection period. Each point indicates that at least one session associated with that fingerprint occurred on that date. The size and color of each point reflect how many sessions for that fingerprint were recorded that day: larger and redder points correspond to more sessions.

Fingerprint patterns The plot reveals clear temporal and structural patterns in the inference dataset. During the collection period (lasted four months), 15516 unique fingerprints were observed: horizontal bands indicate fingerprints that reappear over multiple days or weeks, while vertical columns refer to daily batches of observed fingerprints. In the first one month and a half, a relatively small number of unique fingerprints appeared (≈ 1150), while this number increased significantly, reaching 7000, by the 1st of December. In the final month of collection, more than 7000 new fingerprints appeared, and the vertical columns became denser (excluding a day in mid-December, in which no session was received). This likely indicates that attackers moved to a more active phase, both in terms of quantity and diversity of sessions. However, in the last week of collection, many fingerprints disappeared or became rare, as shown by the presence of sparse points in the plot. Still, the last appeared (those with ID ≈ 15000) continued arriving.

Most present fingerprints A subset of fingerprints, primarily with low IDs (< 500), appears persistently throughout the entire observation period. In particular, the fingerprints present on most days are

[2]The file path has been shortened for visualization purposes. The original one is `/var/tmp/.var03522123`.

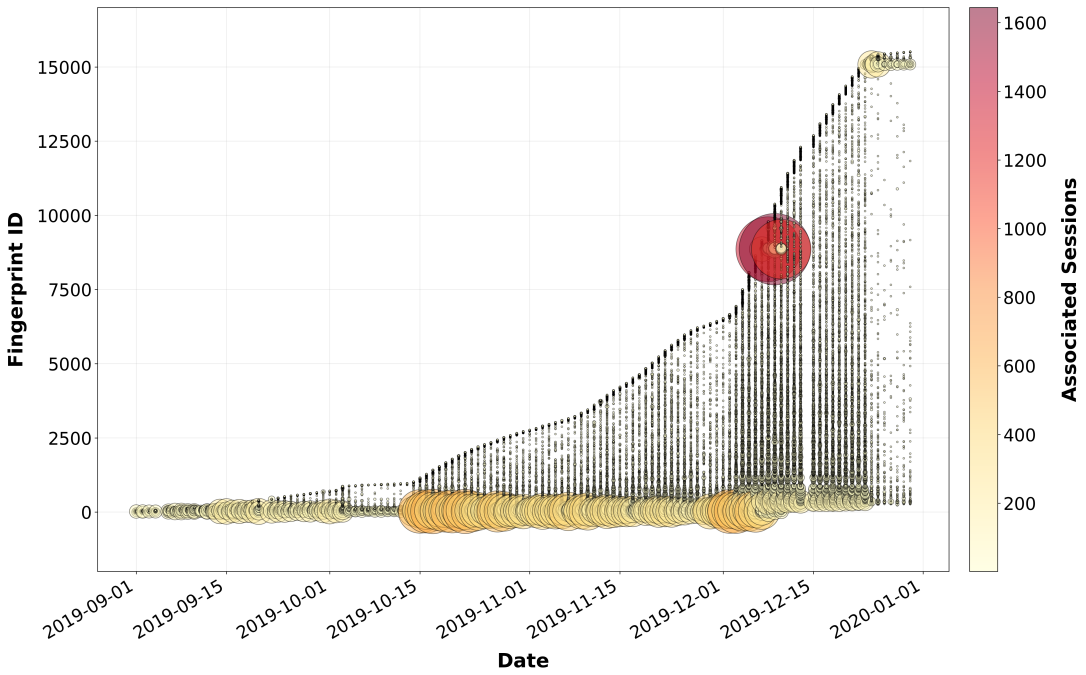


Figure 7: Fingerprint IDs over time. The y-axis lists fingerprint IDs (the lower the value, the earlier the fingerprint appeared for the first time), while the x-axis represents the full data-collection period. Each point indicates that at least one session associated with that fingerprint occurred on that date. The size and color of each point reflect how many sessions for that fingerprint were recorded that day: larger and redder points correspond to more sessions.

those with IDs 0, 17, 19 and 21 (with 99 days of presence out of 122). However, some of them stopped appearing or became rarer in the last period of collection (after the 10^{th} of December).

Fingerprints with many sessions Some fingerprints have numerous associated sessions: examples are fingerprint 21 (26036 sessions), 48 (17639), 25 (7169), 42 (4479) and 8858 (4342). However, these numbers do not always translate in big red circles on the plot, since many sessions span across the whole collection period: the medium-sized yellow/orange clusters at the bottom mostly correspond to "regular" fingerprints, indicating attacks carried out constantly during the entire collection period. Hence, they cannot be considered as sporadic attack campaigns.

Attack campaigns Some fingerprints exhibit burst-like behavior, characterized by sudden appearance and high numbers of associated sessions over short time intervals. Most notably, a dominant fingerprint (8858) emerging in mid-December 2019 ($8^{\text{th}} - 10^{\text{th}}$) reaches the highest session count per day (> 1000) in the dataset, strongly indicating a coordinated attack campaign. This fingerprint is initially characterized by a *Discovery* phase, followed by some *Persistence* actions. Then, there are many *Discovery* commands, some *Execution* actions, and finally *Persistence* steps. Hence, after an exploratory phase, the attacker is probably executing an exploit, finally making its impact persistent. Finally, the late-December period shows a dense concentration of high-ID fingerprints with moderate to high activity, consistent with another possible attack campaign.