Semantic Clustering of Obfuscated Command-Line Detections for Alert Reduction

Barbora Štěpánková and **Vojtěch Outrata** and **Martin Kopp**
Cisco Systems, Inc.
Prague, Czechia
{bstepank, voutrata, markopp}@cisco.com

Abstract

We propose a post-processing method for grouping large volumes of command-line detections into semantically coherent cluster-level alerts. The approach combines embedding-based clustering with LLM-based cluster-level filtering: command-lines are first encoded using Sentence-BERT embeddings and grouped via hierarchical agglomerative clustering, after which a large language model evaluates cluster representatives to reduce the number of false positive alerts. We evaluate the method on real-world telemetry from a commercial endpoint protection system, applying it to detections produced by an obfuscation detection model. On one week of data, the pipeline reduces alert volume by approximately 98% while maintaining high cluster purity and semantic coherence, demonstrating its effectiveness as a scalable post-processing step in high-volume detection settings.

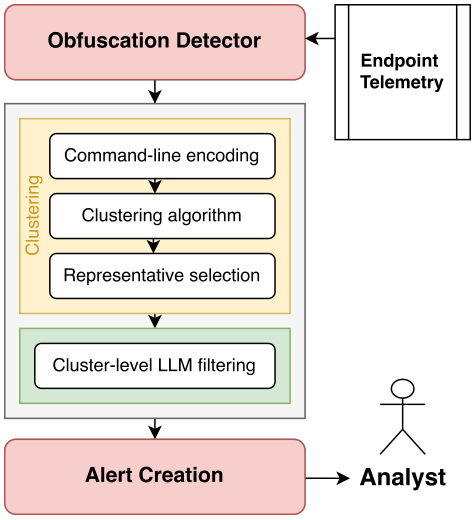


Figure 1: Post-processing block in the alert-creation pipeline. After the lightweight detector, command-lines with high probability are clustered and filtered by LLM, before creating cluster-level alerts.

1 Introduction

Detecting malicious behavior in command-line telemetry is a central challenge in endpoint security. Command-lines collected from real-world environments exhibit extreme class imbalance, with the vast majority being benign, and even high-precision classifiers produce a substantial volume of detections that must be reviewed by analysts (Outrata et al., 2026).

A key difficulty is that many of these detections are redundant. Multiple flagged command-lines may be near-duplicates, minor syntactic variants of one another, or reflections of the same underlying behavior. In addition, despite high classifier precision, a non-negligible number of false positives persist. Together, these factors result in an alert volume that exceeds what human analysts can reasonably review, contributing to alert fatigue and delayed response.

In this work, we address this problem in the context of command-line obfuscation detection. Command-line obfuscation refers to modifying the syntax of a command without changing its underlying functionality, typically to evade detection. We operate on detections produced by an obfuscation classifier deployed in a commercial endpoint protection system, where each detection corresponds to a single command-line instance flagged as potentially obfuscated. Our method serves as a post-processing step: it groups these detections into semantically coherent clusters before alert creation, reducing redundancy while preserving semantic meaning.

To this end, we propose a two-step post-processing pipeline, depicted in Figure 1. In the first step, detections are grouped into semantically coherent clusters, where each cluster is intended to correspond to a single alert. This reduces redundancy by collapsing near-duplicate and behaviorally similar command-lines. In the second step, an LLM-based semantic filter evaluates each cluster to distinguish genuinely obfuscated command-

Proceedings of the Second International Conference on Natural Language Processing and Artificial Intelligence for Cyber Security (NLPAICS'2026)
11–12 June, 2026.

lines from benign but syntactically complex ones, removing additional false positives at the cluster level.

To further analyze the quality of the resulting clusters, we introduce an LLM-based intra-cluster coherence metric that measures how well a single representative command-line captures the behavior of its cluster.

2 Related Work

A large body of prior work has focused on reducing alert fatigue in security monitoring systems by aggregating or correlating single alerts into higher-level structures. For example, Lanigan et al. (2025) focus on constructing meta-alerts, reducing the workload of the human-in-the-loop by reconstructing alert graph and using it as a representant. Also at the alert-grouping level, Kuang et al. (2024) use a two-step pipeline that first identifies temporal and spatial relationships between alerts and then applies LLM-based reasoning to refine uncertain cases. In a related direction, Zha et al. (2024) combine embedding-based similarity with DBSCAN clustering and LLM-based aggregation to merge alerts that share a common root cause. In another line of work, Maosa et al. (2024) perform correlation of security events to reduce alert volume, operating at the level of system events rather than individual low-level artifacts.

Clustering-based approaches have also been widely explored in log and sequence analysis. Egersdoerfer et al. (2022) cluster log sequences based on semantic similarity to obtain compact representations of log behavior, while Turgeman et al. (2022) apply incremental clustering to reduce alert volume in real-time security settings.

Clustering command-line data specifically was explored for example by Švábenský et al. (2022), however they apply clustering to command-line data for behavioral analysis rather than alert reduction. More recent work has explored semantic representations of command-line data. Huang et al. (2024) propose embedding-based representations that aim to capture functional similarity rather than surface syntax. However, these approaches typically focus on grouping commands by intended behavior, whereas our goal is to additionally preserve obfuscation-related structure so that clusters remain interpretable at the level of attack patterns.

In contrast to these approaches, we focus on clustering individual command-line detections produced by an obfuscation detection model, rather than higher-level events or log representations. Compared to prior work that integrates LLMs into the clustering process, we use the LLM only as a second-stage filtering step at the cluster level.

3 Methodology

Our approach consists of four steps, each described in a dedicated subsection below:

1. Normalizing and encoding command-lines into a semantic representation.
2. Clustering semantically similar command-lines.
3. Selecting a representative command-line for each cluster.
4. Filtering clusters using a large language model (LLM).

Steps 1-3 are lightweight and handle the bulk of alert reduction, while the LLM-based filtering in step 4 is applied only to cluster representatives, keeping the number of LLM evaluations low and enabling efficient processing of large-scale telemetry data.

3.1 Command-Line Representation

We first apply basic normalization to command-line strings by replacing dates with `[DATE]`, URLs with `[URL]`, GUIDs with `[GUID]`, and IP addresses with `[IP]`. This reduces superficial syntactic variability while preserving semantic structure. The normalized command-lines are then encoded using Sentence-BERT embeddings (Reimers and Gurevych, 2019), specifically the `all-MiniLM-L6-v2` model, which provides a lightweight but effective semantic representation for sentence-level similarity.

3.2 Clustering Algorithm

The choice of clustering algorithm is guided by several requirements that arise from the nature of command-line detection data. The number of distinct behaviors is not known in advance and varies from day to day, ruling out methods that require a predefined number of clusters. Additionally, some command-lines appear many times with minor variations, forming large clusters, while others occur only once, requiring the algorithm to handle both dense and singleton clusters gracefully.

Given these requirements, we use hierarchical agglomerative clustering (HAC) with average linkage, computed over pairwise cosine distances between Sentence-BERT embeddings. We set a fixed minimum cluster size of 1 to accommodate unique command-lines and use a distance threshold of 0.3 to control cluster granularity. This configuration was selected based on preliminary experiments comparing HAC with alternative clustering methods and embedding models. Detailed results are provided in Appendix A.

3.3 Representative Selection

Each cluster is represented by a single command-line that serves as its proxy in downstream tasks, including LLM-based filtering and alert generation. We choose this representative to capture the shared syntactic structure of the cluster, such as common flags and arguments, while remaining an actual command-line rather than a synthetic summary.

To this end, we first compute the longest common subsequence (LCS) (Hirschberg, 1977) across all command-lines in the cluster, which captures their common structural pattern. We then select as representative the command-line with the smallest Levenshtein distance to this sequence, ensuring that it closely reflects the shared structure while avoiding unnecessary additional elements. Compared to embedding-based selection, this approach preserves interpretable command-line syntax, which is important for both LLM-based analysis and human review of the resulting alerts.

3.4 Cluster-Level Filtering

As a final step, we apply an LLM to each cluster representative to reduce the number of false positive clusters. The LLM serves as a semantic judge, evaluating whether the representative command-line is genuinely obfuscated or merely syntactically complex, a distinction that the upstream detection model does not always capture reliably. Since its decision applies to the entire cluster, this design keeps the number of LLM evaluations proportional to the number of clusters rather than the number of raw detections, making the approach scalable to high-volume telemetry settings.

The model is prompted in a zero-shot setting with a prompt designed to reflect analyst requirements (see Appendix B), and produces a structured JSON output containing the obfuscation label, a short explanation, and a confidence score.

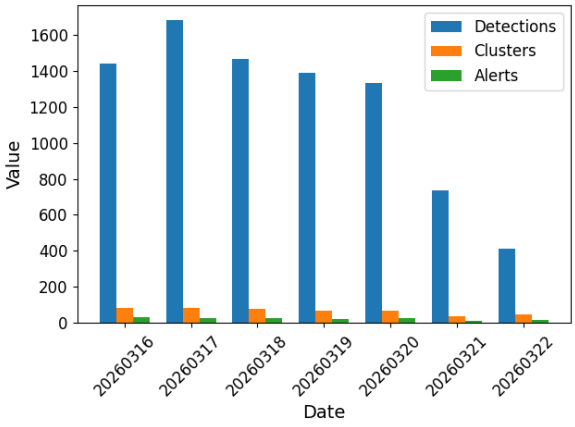


Figure 2: Number of raw detections, clusters, and final alerts over the full week evaluation period.

4 Evaluation

4.1 Dataset

Our dataset consists of command-line detections flagged as potentially obfuscated by an obfuscation detection model deployed in a commercial endpoint protection system. The detection model produces approximately 1400 detections per day in real-world telemetry.

We manually labeled three days of detections, comprising 4432 command-line instances. Two days were used for development and parameter selection, while one day was held out for evaluation.

In addition, we evaluate the proposed pipeline on a full week of unlabeled telemetry data using metrics that do not require ground-truth labels.

4.2 Experimental Setup

We evaluate the proposed pipeline by comparing the number and quality of alerts before and after clustering and LLM-based filtering. As a baseline, we consider the raw detection stream, where each detection corresponds to a separate alert.

For the labeled subset, we evaluate clustering quality using ground-truth annotations. For the full week of data, we evaluate the system using metrics that do not require labels.

4.3 Alert Reduction

On a full week of real-world telemetry data, the input stream contains on average 1207 detections per day. The two stages of the pipeline serve complementary roles: clustering removes the majority of redundancy by collapsing near-duplicate and behaviorally similar command-lines, while the LLM filtering removes clusters corresponding to false

positives rather than collapsing additional meaningful behaviors. On the labeled subset, we confirm that this filtering preserves clusters containing true positives, including borderline cases, while removing clear false positives (see Appendix C).

Overall, the 1207 daily detections are first grouped into approximately 67 clusters and further reduced to about 23 final alerts, corresponding to an approximate 98% reduction in alert volume. This reduction is stable across the evaluation period, as shown in Figure 2, reflecting the consistently high redundancy in the input data. A comparison with a simple deduplication baseline is provided in Appendix C.

4.4 Cluster Quality

In this section, we evaluate the quality of the produced clusters from multiple perspectives. We consider both label consistency on the annotated subset and structural and semantic consistency on the full dataset. Overall, the results indicate that clusters are coherent and suitable for cluster-level alerting without significant loss of behavioral information.

Label Purity We measure label purity of the clustering as the fraction of clusters where all command-lines share the same ground-truth obfuscation label. We observe a perfect label purity (100%) on the labeled subset, suggesting that clusters are consistent with respect to the available annotations. As there are only two available labels for each command-line (obfuscated vs. non-obfuscated) and the goal of the clustering is to produce more fine-grained clusters, this result serves primarily as a basic consistency check, indicating that clusters do not mix the two coarse classes.

Silhouette Score We compute the silhouette score (Rousseeuw, 1987) on the embedding representations of command-lines. The mean score of 0.82 suggests well-separated clusters in the embedding space. However, this metric does not capture semantic equivalence of command-lines.

Semantic Cluster Coherence To evaluate cluster coherence beyond embedding similarity, we introduce an LLM-based coherence metric. For each cluster, an LLM assesses how well the selected representative command-line (Section 3.3) captures each of the remaining members on a 1–5 scale, where lower scores indicate higher agreement. The final coherence score for a cluster is the average over all its members, serving as a proxy for cluster interpretability rather than a strict correctness measure.

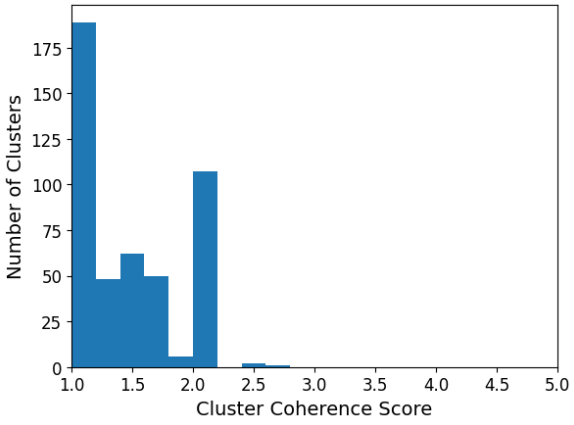


Figure 3: Distribution of LLM-based cluster coherence scores over a full week of telemetry data.

The distribution of scores over the full week of data is shown in Figure 3. Scores generally fall in the low-to-mid range, suggesting that the representatives capture the key structural elements of their respective clusters. The prompt used for this evaluation is provided in Appendix D.

Qualitative Analysis On average, fewer than 8 clusters per day contain more than 10 command-lines, while approximately 31 clusters consist of a single command-line. The largest cluster observed during the evaluation period contains 1061 command-lines and has an average cluster coherence score of 2. We provide examples of clusters and their representatives in Appendix E.

5 Discussion and Conclusion

Our results show that combining embedding-based clustering with LLM-based filtering on our data reduces redundant command-line alerts by 98% while preserving semantic consistency. The approach enables a more scalable alerting process and is effective as a post-processing step after an obfuscation detection model deployed in high-volume telemetry settings where near-duplicate detections are frequent.

We evaluate the resulting clusters using both structural and semantic consistency metrics, and apply LLM-based filtering to further identify clusters corresponding to obfuscated command-lines. Overall, the pipeline provides a practical and scalable way to reduce alert volume and analyst workload in operational endpoint detection systems.

Limitations

The dataset used in this study cannot be publicly released due to sensitive customer information, which limits reproducibility of the experiments. However, we provide detailed descriptions of the pipeline and evaluation setup to allow for replication in similar environments.

In addition, the proposed approach relies on embedding-based clustering and LLM-based filtering, both of which may be sensitive to model choice and parameter settings. Finally, the evaluation of semantic cluster quality relies partially on LLM-based judgments, which may introduce variability in the assessment.

References

Chris Egersdoerfer, Di Zhang, and Dong Dai. 2022. Clusterlog: Clustering logs for effective log-based anomaly detection. In *2022 IEEE/ACM 12th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*, pages 1–10.

Daniel S. Hirschberg. 1977. Algorithms for the longest common subsequence problem. *J. ACM*, 24(4):664–675.

Sian-Yao Huang, Cheng-Lin Yang, Che-Yu Lin, and Chun-Ying Huang. 2024. CmdCaliper: A semantic-aware command-line embedding model and dataset for security research. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 20188–20206, Miami, Florida, USA. Association for Computational Linguistics.

Jinxi Kuang, Jinyang Liu, Junjie Huang, Renyi Zhong, Jiazhen Gu, Lan Yu, Rui Tan, Zengyin Yang, and Michael R. Lyu. 2024. Knowledge-aware alert aggregation in large-scale cloud systems: a hybrid approach. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '24, page 369–380, New York, NY, USA. Association for Computing Machinery.

Bronagh Lanigan, Zeinab Rezaeifar, Federico Cruciani, Michael Milliken, Jordan Vincent, Samuel Moore, Muhammad Aaqib, Alan Mills, Pushpinder K. Chouhan, Alfie Beard, Chris D. Nugent, Luke Chen, and Alex Healing. 2025. Alert correlation for intelligent threat detection and response. *Intelligent Systems with Applications*, 28:200606.

Herbert Maosa, Karim Ouazzane, and Mohamed Chahine Ghanem. 2024. A hierarchical security event correlation model for real-time threat detection and response. *Network*, 4(1):68–90.

Vojtěch Outrata, Barbora Štěpánková, Michael Adam Polák, and Martin Kopp. 2026. Command-line obfuscation detection in real-world telemetry under extreme class imbalance. In *Proceedings of the Second International Conference on Natural Language Processing and Artificial Intelligence for Cyber Security*.

Nils Reimers and Iryna Gurevych. 2019. Sentence-BERT: Sentence embeddings using Siamese BERT-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, pages 3982–3992, Hong Kong, China. Association for Computational Linguistics.

Peter Rousseeuw. 1987. Rousseeuw, p.j.: Silhouettes: A graphical aid to the interpretation and validation of cluster analysis. comput. appl. math. 20, 53-65. *Journal of Computational and Applied Mathematics*, 20:53–65.

Lior Turgeman, Yaniv Avrashi, Gabriella Vagner, Nadeem Azaizah, and Someshwar Katkar. 2022. Context-aware incremental clustering of alerts in monitoring systems. *Expert Systems with Applications*, 210:118489.

Junjie Zha, Xinwen Shan, Jiaxin Lu, Jiajia Zhu, and Zihan Liu. 2024. Leveraging large language models for efficient alert aggregation in aiops. *Electronics*, 13(22).

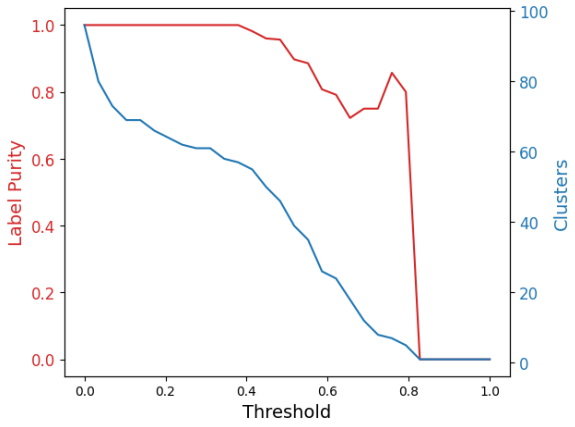
Valdemar Švábenský, Jan Vykopal, Pavel Čeleda, Kristián Tkáčik, and Daniel Popovič. 2022. Student assessment in cybersecurity training automated by pattern mining and clustering. *Education and Information Technologies*.

A Clustering Algorithm Choice

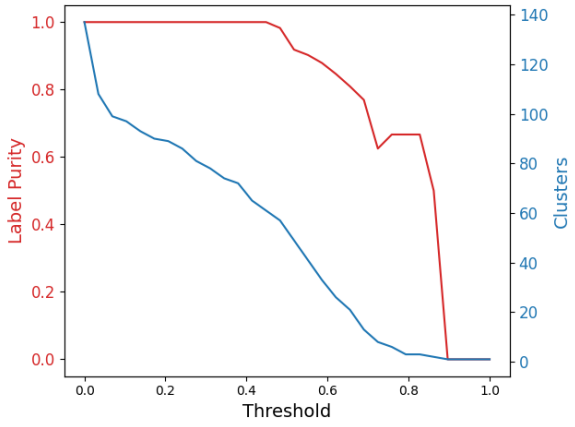
Algorithm We compare DBSCAN, HDBSCAN, Mean Shift, and hierarchical agglomerative clustering (HAC) as none of these algorithms require predefined number of clusters. We evaluate the performance of these algorithms on three subsets of the labeled data: full day of data (1302 command-lines, 411 positive and 891 negative) and 675 and 104 command-lines sampled from the day. The results can be seen in Table 1.

HAC shows both excellent results on the labeled test set as well as consistent creation of coherent clusters, discovered during manual evaluation.

Hyperparameters The average linkage was chosen based on manual inspection of the resulting clusters. The threshold was chosen as a tradeoff between keeping high label purity and cluster semantic coherence, while having the minimal feasible amount of clusters. Figure 4 shows the label purity and number of clusters across different threshold on two of the labeled subsets.



(a) Subset of 675 samples



(b) Full-day dataset (1302 samples)

Figure 4: Label purity and number of clusters as a function of the clustering threshold for two labeled datasets (675-sample subset and full-day dataset with 1302 samples).

Samples	Algorithm	Clusters	Label Purity
104	DBSCAN	21	1.00
104	HDBSCAN	6	0.67
104	Mean Shift	19	1.00
104	HAC	22	1.00
675	DBSCAN	61	1.00
675	HDBSCAN	33	0.94
675	Mean Shift	49	0.98
675	HAC	63	1.00
1302	DBSCAN	76	1.00
1302	HDBSCAN	45	0.97
1302	Mean Shift	74	0.98
1302	HAC	79	1.00

Table 1: Clustering performance across dataset sizes.

B LLM Filtering

We use Llama 4 Maverick with a temperature of 0.3 and top-p of 0.9, and limit the generation to at most 512 tokens. The prompt is as follows:

```
"messages": [
  {
    "role": "system",
    "content": "You are a cybersecurity analyst
AI. You analyze Bash commands and determine if
they are obfuscated. Always return ONLY a valid
JSON object. Never include explanation, markdown,
or any extra text. Do not use single quotes.
Only output valid, compact JSON.

A command is considered obfuscated if it uses
intentional techniques to hide or disguise its
true behavior, especially when the final
executed action is not directly readable
without decoding or reconstructing the command.
Strong indicators include:
- Multi-stage encoding or decoding (e.g.,
base64, hex, URL encoding) used to conceal
execution
- Dynamic command construction intended to
hide intent (e.g., eval, exec, bash -c with
constructed strings)
- String manipulation to reconstruct commands
(e.g. concatenation, reversal, substitution)
- Nested subshells or chained command
substitution used to obscure final execution
- Indirect execution patterns used to conceal
what command is actually run
- Environment variables used in a way that
obscures execution flow

However, DO NOT consider the line obfuscated
because of these alone:
- Normal escaping characters (e.g., ^, \\,
backticks) when used in standard shell syntax
- Standard quoting patterns (single, double, or
nested quotes) when used conventionally
- Placeholder values like ???, XXX, [NUM], or
$VAR used for templating or configuration
- Pattern matching or text processing tools
(e.g., awk, grep, egrep, sed, find, glob
patterns, regex) when not used for concealment
- Printing, formatting, or display commands
(e.g., echo, printf, less)
- Legitimate scripting or operational workflows,
including container management, PATH
configuration, or system administration tasks
that are readable in intent

Only classify as obfuscated when there is clear
evidence of intent to conceal execution behavior,
not merely syntactic complexity.

Be strict about returning ONLY the following
JSON format:
{
  \"obfuscated\": true or false,
  \"confidence\": float between 0.0 and 1.0,
  \"reasons\": [\"short explanation\"]
}"
  },
  {
    "role": "user",
    "content": "Analyze the following Bash
command and determine if it is obfuscated:
```

```
[COMMAND_LINE]

Respond ONLY with a valid JSON object in this
format:
{
  \"obfuscated\": true or false,
  \"confidence\": float between 0.0 and 1.0,
  \"reasons\": [\"short explanation\"]
}

No markdown, no commentary. JSON only."
  }
]
```

C Alert Reduction Analysis and Baseline

Alert Reduction Analysis On the labeled subset, the LLM filtering preserves all positive clusters while removing the majority of negative clusters.

	Pos	Neg	All
All clusters	25	59	84
Filtered clusters	25	6	31

Table 2: Number of positive and negative clusters before and after LLM filtering described in Section 3.4 on the labeled subset of the data.

Deduplication baseline We evaluate a simple deduplication baseline on command-lines after the normalization described in Section 3.1. This baseline produces on average 1028 unique command-lines per day, corresponding to a 15% reduction in alert volume. In contrast, semantic clustering reduces the same input to approximately 67 clusters per day, corresponding to a 94.5% reduction before the LLM filtering stage. This indicates that the reduction is not achieved by normalization alone, but also by grouping semantically similar command-lines.

D Semantic Cluster Coherence Prompt

```
messages = [
  {
    "role": "system",
    "content": "You are a cybersecurity analyst
AI. You compare two shell commands and score how
well they belong to the same behavioral cluster.

Clustering is based primarily on execution
technique and structure, not just final outcome.
Commands that achieve the same result using
different techniques should NOT be considered a
strong match.

Use this scale:

1 = Same intent AND same execution technique
(nearly identical structure or method)
2 = Same intent with minor variations in technique
(small changes in flags or structure)
3 = Same general goal but different execution
technique (different tools or fundamentally
different approach)
4 = Related only at a high level; different
technique and partial overlap in components
5 = Different intent and different technique

Rules:
- Prioritize execution technique over outcome
similarity
- Different methods/tools for achieving the same
result should reduce similarity
- Obfuscation is part of technique comparison
(encoding, eval, reconstruction methods count as
different techniques)
- Ignore superficial syntax differences if
technique is identical

Return ONLY a single number (1-5). No explanation."
  },
  {
    "role": "user",
    "content": "Compare these command-lines:

Command 1:
[REPRESENTATIVE_COMMAND_LINE]

Command 2:
[COMMAND_LINE_FROM_CLUSTER]

Return only a number from 1 to 5."
  }
]
```

E Cluster Examples

Very simple example is the cluster with following longest common subsequence (LCS):

```
/bin/bash -c printf '%s' "$HOME"
```

And the following command-lines as example members of the cluster:

```
/bin/bash -c printf '%s\\n' \"$HOME\"
/bin/bash -lc printf '%s\n' \"$HOME\"
/bin/bash -lc printf '%s\\n' \"$HOME\"
```

A good example of a more complex cluster is for example a cluster with the folowing longest common subsequence (LCS):

```
/bin/bash -c curl --location --request GET '[URL]
--header 'Authorization: Bearer [BASE64].[BASE64]
.U' --data ''
```

This cluster contains 112 command-lines differing in url and the Base64 encoded text. Following are three randomly sampled examples (with the base64 text shortened for privacy reasons).

```
/bin/bash -c curl --location --request GET '[URL]
--header 'Authorization: Bearer [BASE64].[BASE64]
.O-8pqtMvNfMZkwmHxZYNTqt93nBZDHdWUHqRqbYGx_c4FBx
                    ...
G1HPizniFjcg6c1GKaQLOSwuNFkm5MTkZ10r5XSqrjmBiyfl
' --data ''

/bin/bash -c curl --location --request GET '[URL]
```

```
--header 'Authorization: Bearer [BASE64].[BASE64]
fQ.aQMV8ui4Euqj5vkw_qCDiDceu40MtWoMH_TewJd_KNroI
                        ...
kXRPCwReC5aRzoi5G6O_fJCJamOvYldZotPNSXgodpvVpSDz
' --data ''

/bin/bash -c curl --location --request GET '[URL]
--header 'Authorization: Bearer [BASE64].[BASE64]
fQ.fpvAavEeYsyukAAMibTd0eeUlEEiOJ6HV_2LZl4fFgpiI
                        ...
lhlXEGGv3Nj6cyc9ncKKCvNKBpLuMJbq0OZbMp2D8xWa6Jno
' --data ''
```

An example of a cluster which is problematic but correctly handled by our setup are for example command-lines which contain a certificate. Due to the majority of the command being a random string, it is difficult to cluster them together. This cluster was classified as non-obfuscated by the LLM-filtering phase. The commands of the cluster have on average 3564 characters. The LCS of this cluster has 566 characters. Shortened version of the LCS for privacy reason is as follows.

```
/bin/sh -c echo "-----BEGIN CERTIFICATE-----
MIIGCCBgAwIBAgIQQA+ANBgkqhkiG9w0BAQsFADBy
MQswCQYDVQQGEwJVUzESMBAGA1UEChMJSWRlblRydXN0MS4w
IENBIE8xMBINAMIAMjM
                        ...
Ni0s7tKqCqzlihUJ+PcpNU/tOOOoR0glWyIBQnB
-----END CERTIFICATE-----" > /root/[SERVER]
```