PASTRAL: Privacy-aware AST and TRansformer-based Anomalous command-Line detection

Xiayan Ji **Ecenaz Erdemir** **Kyuhong Park** **Bhavna Soman** **Yi Fan**
Amazon Web Services, New York
{xjiae, ecenaz, kyuhongp, bhavna, fnyi}@amazon.com

Abstract

Command-lines are a common attack surface in cybersecurity. Yet they often contain sensitive user information, creating a *dual challenge*: systems must detect suspicious commands accurately while protecting user privacy. Existing approaches typically tackle one challenge without the other. To address this gap, we present PASTRAL, a practical framework for privacy-preserving detection of suspicious command-lines. Our main insight is that suspicious activities are typically rare and highly diverse in large-scale multi-user environments, which makes them naturally well suited to anomaly detection. PASTRAL represents command-lines using language-model and Abstract Syntax Tree (AST)-based embeddings, applies differential privacy (DP) noise injection at the embedding layer, and performs detection with a conditional variational autoencoder. By design, only differentially private embeddings are shared, which provide sufficient signal for accurate detection while abstracting away unnecessary details. Empirical evaluation demonstrates that PASTRAL achieves strong anomaly detection performance and sustains a favorable privacy-utility trade-off. Our real-world case study outlines practical considerations for deploying secure LLM detection systems in production.

1 Introduction

Rapid threat detection is critical in cybersecurity, as many attacks exploit misconfigurations and vulnerabilities [1, 2] and leave traces in command-line activity [3]. However, command-lines often contain sensitive information such as API tokens, file paths, and credentials. This creates a dual challenge: systems must detect suspicious activity reliably while also protecting user privacy.

This challenge is especially pronounced in multi-user settings, where a service provider (SP) must protect many customers simultaneously. Customers are concerned about what SPs can access [4, 5] and often do not want their raw data to be used for model training [6] or even post-detection investigation. At the same time, suspicious activity is both rare and highly diverse, making it difficult to rely on rule-based signatures or supervised classifiers. Moreover, what appears suspicious often depends on context: commands that are routine for one organization may look unusual for another. These realities naturally motivate an anomaly detection (AD) approach, where the system learns a broad distribution of command-line behaviors and flags deviations as potential threats, all while protecting user privacy.

Existing methods typically achieve either strong detection or privacy preservation, but not both. Language model-based approaches have shown promise for command-line detection [7, 8, 9, 10, 11], delivering strong semantic modeling but paying little attention to privacy. Conversely, privacy-preserving systems aim to safeguard sensitive data [12, 13], but they do not leverage pretrained language models and often sacrifice detection utility. To our knowledge, no prior work jointly achieves language model-based anomalous command-line detection with integrated privacy guarantees.

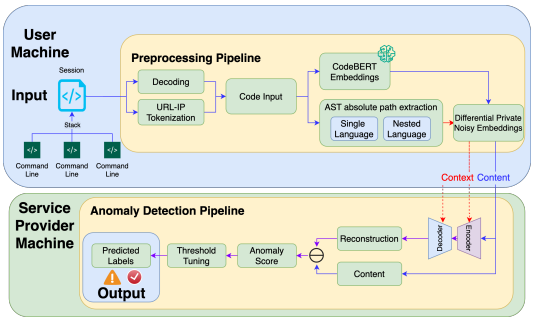


Figure 1: Overview of PASTRAL. Command-line representation is extracted on the user side, where language-model embeddings and AST features are combined and perturbed with DP noise injection. The resulting private embeddings are then sent to the SP for anomaly detection via a CVAE.

In this paper, we present PASTRAL, a privacy-preserving framework for anomalous command-line detection. Figure 1 shows the workflow: on the user side, command-lines are represented with language-model and Abstract Syntax Tree (AST)-based embeddings, and differential privacy (DP) noise injection [14] is applied at the embedding layer. The private embeddings are then sent to the SPs, where a conditional variational autoencoder (CVAE)-based anomaly detector [15] performs detection. By design, PASTRAL ensures raw commands never leave the user's machine; only differentially private embeddings reach threat detection SPs, thereby mitigating sensitive attribute inference without sacrificing detection quality. Our contributions are:

- A representation of command-lines that combines language-model with AST structure.
- An embedding pipeline with DP noise injection to limit sensitive content inference.
- A CVAE-based anomaly detector tailored for command-line behavior.
- An empirical study showing strong detection performance and a favorable privacy-utility trade-off.

In Section 2, we describe the user-side representation extraction consisting of language model and AST embeddings, with an DP privacy guarantee. Section 3 presents the SP-side conditional anomaly detection with a CVAE. Section 4 reports detection results and ablations that quantify the benefits of AST conditioning and the quantify the privacy-utility trade-off. Finally, Section 5 discusses lessons learned and directions for secure LLM deployment. This work focuses on static analysis of command-line payloads; robustness to dynamically generated or staged runtime payloads is beyond our current scope and left for future work.

2 Privacy-preserving Command-line Embedding (User Machine)

In this section, we describe the user-side command-line representations extracted using language model and AST. While AST provide us with syntactic structure of the program flow, language model embeddings capture semantic representations that bridge natural language and programming language constructs. In addition, we inject DP noise to the extracted representations for privacy guarantees.

2.1 Embedding Models

Our approach to command-line embedding extraction aims to derive meaningful representations by focusing on two key aspects: language model-based and syntactic structure-based embeddings. We leverage pre-trained model specialized in diverse programming languages, to capture the semantic meaning of each command-line. Some preprocessing steps for the command-lines like de-obfuscation, IP and URL tokenization are described in Appendix A.

Complementing this, we extract syntactic structure-based embeddings from absolute AST paths to capture their inherent structural relationships. The combination of these embeddings creates a semantic and structural representation of command-lines, serving as input for our AD model.

Language Model-based. Our methodology employs multilingual pre-trained language models like [8, 11, 10, 9], which process code as natural language using transformer-based architectures, to generate d-dimensional vectors that encapsulate sequence representations. A maximum of L lines is selected from each command-line session, adhering to tokenization limits. Followed by that, the

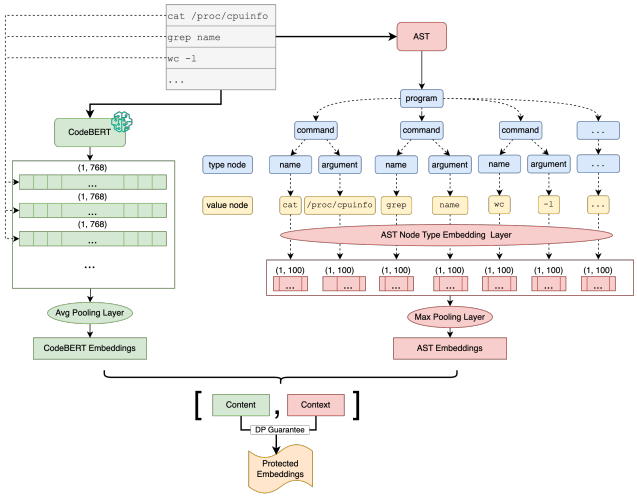


Figure 2: Session-level representations are obtained by pairing per-line CodeBERT embeddings (content) with AST-path embeddings (context), followed by average and max pooling across lines respectively, before injecting differential privacy noise to provide a formal privacy guarantee.

language models process each line through their transformer layers, producing hidden states from the last layer. To standardize these representations, we apply mean pooling, which converts variable-length token sequences into fixed-size vectors. As CodeBERT embeddings already encode diverse command semantics; averaging stabilizes these contextual differences and captures overall intent rather than token-level noise. Empirically, we find that CodeBERT [8] attains the best performance for our task, but the framework itself is model-agnostic and readily supports alternative language model choices. Detailed experimental comparisons are provided in Section 4.

AST-based ASTs represent code structure and are widely used in compilers and analysis tools [16]. We use the Tree-sitter library [17], which applies context-free grammars to convert plain text into structured node hierarchies, to parse command-lines into ASTs. For each command-line, we extract absolute paths from the root to each token (avoiding quadratic scaling from relative paths). This design maintains linear scaling and robustness to line reordering or token renaming, with results and discussion provided in Appendix F. Each AST node is assigned an embedding based on its node type from a finite set of bash grammar tokens, using simple numeric tokenization rather than a pre-trained model. Token embeddings are computed by summing node embeddings along these paths (Appendix B, Fig. 6), then aggregated across tokens into an m-dimensional vector for downstream tasks. We use max pooling to highlight the most anomalous nodes along similar syntactic paths. This mitigates the information loss with CodeBERT's average pooling over longer sequences and better preserves distinctive structural deviations.

2.2 Differential Privacy (DP) Guarantees

To limit sensitive content inference, we ensure SP's access to only privacy-protected data while maintaining effective AD. DP provides formal privacy guarantees for data analysis [14]. A randomized mechanism $\mathcal{M} : \mathcal{D} \to \mathcal{R}$ satisfies (ϵ, δ)-DP if for any adjacent inputs D, D' differing in one entry and any output subset $\mathcal{S} \subseteq \mathcal{R}$:

$$Pr[\mathcal{M}(D) = \mathcal{S}] \leq e^{\epsilon} \cdot Pr[\mathcal{M}(D') = \mathcal{S}] + \delta \tag{1}$$

For vector-based embeddings, we employ the Gaussian mechanism by adding calibrated noise to outputs:

$$\mathcal{M}(D) = f(D) + \sigma \Delta_2 f \cdot \mathcal{N}(0, \mathbb{I}d) \tag{2}$$

where $\Delta_2 f = \max_{D \sim D', D, D' \in \mathcal{D}} |f(D) - f(D')|_2$ represents sensitivity. $\mathcal{M}(D)$ is (ϵ, δ)-DP if $\sigma \geq \epsilon^{-1}\sqrt{2\ln(1.25/\delta)}$ for $\epsilon \in (0, 1)$.

Our framework applies DP to both embedding mechanisms ($f_{\text{CodeBERT}}(\cdot)$ and $f_{\text{AST}}(\cdot)$). By normalizing embedding vectors to unit length, we ensure sensitivity $\Delta_2 f \leq 2$, allowing proper noise

calibration. Since querying our detector makes two queries on the same command-line, we leverage composition theory to assess cumulative privacy loss:

Proposition 2.1 (Advanced composition [18])**.** *The composition of n (ϵ, δ)-DP mechanisms is $(\epsilon', n\delta + \delta')$-DP for any $\delta' > 0$, where $\epsilon' = n\epsilon(e^{\epsilon} - 1) + \epsilon\sqrt{2n\ln(1/\delta')}$.*

Furthermore, post-processing preserves DP guarantees:

Proposition 2.2 (Post-processing [14])**.** *If $\mathcal{M}(D)$ is (ϵ, δ)-DP, then for any function g, $g \circ \mathcal{M}(D)$ also satisfies (ϵ, δ)-DP.*

Using these properties, we establish our main privacy guarantee:

Theorem 2.1 (Privacy guarantee for PASTRAL)**.** *Let $\mathcal{M} = (\mathcal{M}_1, \mathcal{M}_2)$ be the composition of two (ϵ, δ)-DP mechanisms. The CVAE-based AD model of PASTRAL preserves $(2\epsilon(e^{\epsilon} - 1) + \epsilon\sqrt{4\ln(1/\delta')}, 2\delta + \delta')$-DP for its input.*

Proof. From advanced composition with $n = 2$, the CVAE input is $(2\epsilon(e^{\epsilon} - 1) + \epsilon\sqrt{4\ln(1/\delta')}, 2\delta + \delta')$-DP. Since the CVAE applies $g = g_{dec} \circ g_{enc}$ with randomization independent of raw data, post-processing ensures the output maintains the same DP guarantees. □

Notably, training the CVAE on the private data does not require re-querying them from the raw input, thereby not causing a degradation and preserving the initial privacy levels.

In sum, our feature extraction pipeline combines language-model embeddings with AST-derived structural features and DP noise injection to balance detection utility with privacy guarantees; an overview of this user-side embedding extraction process is shown in Figure 2.

3 Conditional Anomaly Detection (Service Provider Machine)

In this section, we propose a novel AD model for command-lines addressing the challenge of rare and highly diverse suspicious activities in production. We approach this as an unsupervised learning problem, using an encoder-decoder architecture to learn representations of *normal* behavior from historical command-line data of the customers. Our model, shown in Figure 3, leverages a Conditional Variational Autoencoder (CVAE) [15] that takes differentially private CodeBERT and AST embeddings as input and produces reconstructions whose errors serve as anomaly indicators triggering AD based on a predefined threshold.

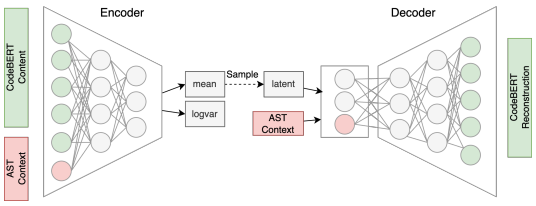


Figure 3: The anomaly detector is a Conditional Variational AutoEncoder (CVAE) that uses CodeBERT embeddings as content and AST embeddings as context. The AST embedding is incorporated into both the input and the latent space of the CVAE. The output is the reconstruction of content.

3.1 Model Architecture

Our model processes differentially private CodeBERT embeddings $x \in \mathbb{R}^d$ and AST embeddings $a \in \mathbb{R}^m$ to reconstruct $\hat{x} \in \mathbb{R}^d$. The encoder $q_\phi(z \mid x_c)$ generates a latent variable z conditioned on the concatenated input $x_c = [x; a] \in \mathbb{R}^{d+m}$, where x is the content and a is the context embedding.

This concatenated input passes through a downsampling ResNet block [19] with three layers (f_1, f_2, f_3), incorporating residual connections and attention mechanisms. The architecture's bottleneck condenses the representation using residual connections and attention, applying group normalization and SiLU activation to produce the mean μ and standard deviation σ of the latent variable $z \in \mathbb{R}^l$. The latent z is sampled using the reparameterization trick [20, 21].

The decoder $p_\theta(x \mid z_c)$ takes the concatenation $z_c = [z; a] \in \mathbb{R}^{l+m}$ and reconstructs the original representation through an upsampling ResNet block. The anomaly score is calculated as $s = \|x - \hat{x}\|_2^2/d$, with higher scores indicating potential anomalies.

3.2 Training Objectives

We train the model by maximizing the conditional log-likelihood, using the Stochastic Gradient Variational Bayes (SGVB) [22] estimator for approximation. The lower bound of the model loss is:

$$\log p_\theta(x \mid a) \geq -KL\left[q_\phi(z \mid x_c) \parallel p_\theta(z \mid a)\right] + \mathbb{E}q\phi(z \mid x_c)\left[\log p_\theta(x \mid z_c)\right]$$

The conditional prior $p_\theta(z \mid a)$ models the latent distribution conditioned on the AST context, with the empirical lower bound:

$$L_{\text{CVAE}}(a, x; \theta, \phi) = -KL\left[q_\phi(z \mid x_c) \parallel p_\theta(z \mid a)\right] + s$$

To train the CVAE, we minimize the negative empirical lower bound, balancing the KL divergence, ensuring the approximate posterior remains close to the prior, with the reconstruction term to enable accurate input reconstruction from the latent representation and context.

4 Experiments

We first introduce the evaluation datasets and present evaluation results.
Datasets. We curate datasets from two in-house and two public sources. The in-house datasets are used for both training and testing, while the public datasets are testing-only, allowing evaluation of out-of-distribution (OOD) performance.

- **In-house commands:** Collected from a decade-long honeypot simulating SSH, web, and IoT. Malicious samples were validated via VirusTotal or MITRE ATT&CK; benign samples were drawn from attack-free sessions.
- **In-house scripts:** Longer command-line sequences reflecting real-world usage. Malicious scripts are collected from VirusTotal and the honeypot; benign scripts from clean virtual machine and GitHub repositories with $\geq$1000 stars.
- **zenodo (public):** Malicious shell-commands covering seven attack types [23].
- **atomic (public):** Malicious powershell-commands simulating penetration threats [24].

Detailed statistics for the datasets is in Appendix C, with implementation details and experiment setup in Appendix D.

Evaluation. Each experiment was run five times using different random seeds, and we report the 95% confidence interval around the mean.

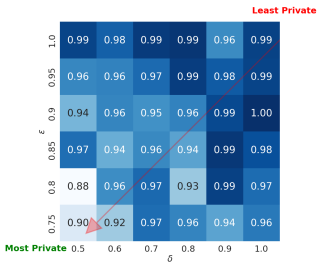
Language Model Choices. For practical deployment, CodeBERT [8] achieves accuracy comparable to billion-parameter LLMs [11, 10] on most benchmarks, while being far more efficient. On three of the four subsets: *scripts*, *cmd-lines*, and *zenodo*, CodeBERT reaches an AUROC of 1.0, essentially identical to StarCoder and Phi-3 (0.9996–1.0000; Appendix Table 4). On the more challenging *atomic* set, CodeBERT performs better, with AUROC 0.9990 compared to 0.9664 for StarCoder (+3.26%) and 0.9812 for Phi-3 (+1.78%). These accuracy results come at much lower cost: relative to StarCoder (3B) and Phi-3 (3.8B), CodeBERT uses 24–30$\times$ fewer parameters and delivers 3.25–3.89$\times$ higher throughput (3.31 vs. 1.02/0.85 inputs/s; Appendix Table 5).

Overall, scaling to large parameters yields negligible utility gains for this task but increase compute overheads, making CodeBERT a strong, practical choice for user-side embedding extraction.

Anomaly Detection Performance. Adding program structure (AST) to a CodeBERT embedding yields clear gains over a language model-only detector (IDS-LLM [25]). On in-house data, AUROC improves from about 0.96 to 0.99 (+3%), while FPR drops from 2.7% to 0.3%. On OOD sets, the effect is even stronger: AUROC approaches 1.0 and FPR falls to zero, compared to 4–5% for the baseline. The ablation in Fig. 5 confirms this: *CodeBERT-only* and *AST-only* each have complementary weaknesses, while their *combination* consistently achieves the best performance with narrow confidence intervals.

Subset	Detection Model	FPR (↓)	Recall (↑)	Precision (↑)	F1-Score (↑)	AUROC (↑)
scripts	PASTRAL	**0.0049 ± 0.0002**	**0.9146 ± 0.1100**	0.9946 ± 0.0008	0.9511 ± 0.0616	0.9885 ± 0.0132
	IDS-LLM	0.0420 ± 0.0101	0.7255 ± 0.0211	0.9380 ± 0.0098	0.8215 ± 0.0065	0.9442 ± 0.0018
cmd-lines	PASTRAL	**0.0001 ± 0.0004**	**0.9995 ± 0.0005**	0.9999 ± 0.0004	0.9997 ± 0.0003	1.0000 ± 0.0000
	IDS-LLM	0.0128 ± 0.0053	0.8462 ± 0.0148	0.9672 ± 0.0013	0.8490 ± 0.0110	0.9824 ± 0.0006
zenodo	PASTRAL	**0.0000 ± 0.0000**	**0.9378 ± 0.1674**	1.0000 ± 0.0000	0.9635 ± 0.0988	1.0000 ± 0.0000
	IDS-LLM	0.0450 ± 0.0054	0.7679 ± 0.0231	0.7334 ± 0.0176	0.7710 ± 0.0074	0.9881 ± 0.0019
atomic	PASTRAL	**0.0000 ± 0.0000**	**0.9099 ± 0.1414**	1.0000 ± 0.0000	0.9497 ± 0.0827	0.9948 ± 0.0145
	IDS-LLM	0.0472 ± 0.0016	0.7661 ± 0.0442	0.9475 ± 0.0065	0.8286 ± 0.0214	0.9589 ± 0.0082

Table 1: PASTRAL model performance compared with related detection models. PASTRAL is robust across in-house (scripts and cmd lines) and out-of-distribution data (zenodo and atomic).



(ε, δ)	σ	$\sigma/\sigma_{\text{base}}$	AUROC	$\Delta\%$
(0.95, 1.00)	0.703	1.05×	0.99	0.00
(0.95, 0.90)	0.853	1.28×	0.98	−1.01
(0.80, 0.70)	**1.346**	**2.01×**	**0.97**	−2.02
(0.75, 0.60)	1.615	2.42×	0.92	−7.07
(0.75, 0.50)	**1.805**	**2.70×**	**0.90**	−9.09

Figure 4: Privacy-utility trade-off across (ε, δ). *AUROC heatmap (left):* moving from least private to most private, noise σ increases and AUROC decreases. *Selected operating points (right):* at $2.01\times$ noise, AUROC merely drop 2%; and even at $2.70\times$ noise, AUROC only drops 9%.

In sum, incorporating AST structure enhances AD performance. A detailed threat-coverage analysis highlights that PASTRAL identifies 99.3% of suspicious samples missed by 65 VirusTotal vendors, after manually labeled by security researchers. (Appendix F)

Privacy-Utility Tradeoff. Our detector sustains strong performance under DP noise injection, reflecting robustness to increasingly stringent privacy constraints. We add Gaussian DP at the embedding layer with L_2-sensitivity $= 1$, $\sigma(\varepsilon, \delta) = \sqrt{2\ln(1.25/\delta)}/\varepsilon$. Figure 4 visualizes the privacy-utility trade-off: moving from the *upper-right* (least private) to the *lower-left* (most private) increases the injected noise σ and decreases AUROC. The table shows representative operating points taking baseline at $(\varepsilon, \delta) = (1.0, 1.0), \sigma_{\text{base}} = 0.668$, AUROC$= 0.99$. In particular, at *about* $2\times$ the baseline noise standard deviation $(\varepsilon, \delta) = (0.8, 0.7), \sigma = 1.35$, AUROC drops only $\mathbf{2}\%$ $(0.99 \rightarrow 0.97)$. Even at *about* $2.7\times$ the baseline noise $(\varepsilon, \delta) = (0.75, 0.5), \sigma = 1.81$, AUROC decreases around $\mathbf{9}\%$ $(0.99 \rightarrow 0.90)$. These preliminary results are promising for deployments that require stricter privacy budgets.

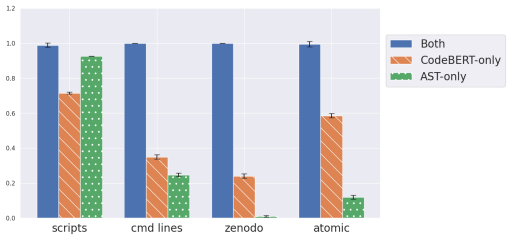


Figure 5: AUCROC with 95% CI for ablation of embeddings (CodeBERT, AST, both).

5 Conclusions and Lessons Learned

PASTRAL provides a practical framework for combining language model-based anomaly detection with formal privacy guarantees in command-line data, addressing key challenges in secure LLM deployment. Unlike prior approaches that focus on either accuracy or privacy, PASTRAL achieves both by performing local embedding extraction and applying DP noise injection before transmission, ensuring raw text never leaves the user machine. Our results show that scaling to larger language models does not yield additional benefits for this task, while augmenting a compact CodeBERT encoder with AST structure markedly improves detection, even under distribution shifts. The privacy-utility trade-off is also favorable: even with two to three times more DP noise, AUROC decreases by only 2–9%, indicating that stricter privacy budgets remain feasible. Together, these findings provide practical lessons learned towards deploying secure LLM systems in production environments.

References

[1] Hassan Takabi, James BD Joshi, and Gail-Joon Ahn. Security and privacy challenges in cloud computing environments. *IEEE Security & Privacy*, 8(6):24–31, 2010.

[2] Ege Tekiner, Abbas Acar, A Selcuk Uluagac, Engin Kirda, and Ali Aydin Selcuk. Sok: cryptojacking malware. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 120–139. IEEE, 2021.

[3] Rafael Uetz, Marco Herzog, Louis Hackländer, Simon Schwarz, and Martin Henze. You cannot escape me: Detecting evasions of {SIEM} rules in enterprise networks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 5179–5196, 2024.

[4] Mohammad Taha Khan, Christopher Tran, Shubham Singh, Dimitri Vasilkov, Chris Kanich, Blase Ur, and Elena Zheleva. Helping users automatically find and manage sensitive, expendable files in cloud storage. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1145–1162, 2021.

[5] Jiayun Zhang, Junshen Xu, Bugra Can, and Yi Fan. React: Residual-adaptive contextual tuning for fast model adaptation in threat detection. In *Proceedings of the ACM on Web Conference 2025*, pages 2488–2499, 2025.

[6] Nicholas Carlini, Florian Tramer, Eric Wallace, Matthew Jagielski, Ariel Herbert-Voss, Katherine Lee, Adam Roberts, Tom Brown, Dawn Song, Ulfar Erlingsson, Alina Oprea, and Colin Raffel. Extracting training data from large language models, 2021.

[7] Sian-Yao Huang, Cheng-Lin Yang, Che-Yu Lin, and Chun-Ying Huang. 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, November 2024.

[8] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[9] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. 2022.

[10] Anton Lozhkov et al. Starcoder 2 and the stack v2: The next generation, 2024.

[11] Marah Abdin et al. Phi-3 technical report: A highly capable language model locally on your phone, 2024.

[12] Soumia Zohra El Mestari, Gabriele Lenzini, and Huseyin Demirci. Preserving data privacy in machine learning systems. *Computers & Security*, 137:103605, 2024.

[13] Zhaoyu Wang, Pingchuan Ma, Huaijin Wang, and Shuai Wang. Pp-csa: Practical privacy-preserving software call stack analysis. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA1):1264–1293, 2024.

[14] Cynthia Dwork. Differential privacy. In *International colloquium on automata, languages, and programming*, pages 1–12. Springer, 2006.

[15] Kihyuk Sohn, Honglak Lee, and Xinchen Yan. Learning structured output representation using deep conditional generative models. In C. Cortes, N. Lawrence, D. Lee, M. Sugiyama, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 28. Curran Associates, Inc., 2015.

[16] Ecenaz Erdemir, Kyuhong Park, Michael J. Morais, Vianne R. Gao, Marion Marschalek, and Yi Fan. Score: Syntactic code representations for static script malware detection, 2024.

[17] Max Brunsfeld. Tree-sitter: An incremental parsing system for programming tools. `https://github.com/tree-sitter/tree-sitter`, 2014. Accessed: 2024-08-09.

[18] Cynthia Dwork, Guy N. Rothblum, and Salil Vadhan. Boosting and differential privacy. In *2010 IEEE 51st Annual Symposium on Foundations of Computer Science*, pages 51–60, 2010.

[19] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[20] Diederik P. Kingma and Max Welling. Auto-encoding variational bayes. *CoRR*, abs/1312.6114, 2013.

[21] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *International Conference on Machine Learning*, 2014.

[22] David A. Knowles. Stochastic gradient variational bayes for gamma approximating distributions, 2015.

[23] Valdemar Švábenský, Jan Vykopal, Pavel Seda, and Pavel Čeleda. Dataset of shell commands used by participants of hands-on cybersecurity training. *Data in Brief*, 38:107398, 2021.

[24] Red Canary Co. Atomic red team. `https://github.com/redcanaryco/atomic-red-team/tree/master`, 2024. Accessed: 2024-07-25.

[25] Jiongliang Lin, Yiwen Guo, and Hao Chen. Intrusion detection at scale with the assistance of a command-line language model, 2024.

[26] Hugging Face. Autoencoderkl - diffusers 0.29.2 documentation. `https://huggingface.co/docs/diffusers/v0.29.2/en/api/models/autoencoderkl#diffusers.AutoencoderKL`, 2023. Accessed: 2024-08-30.

[27] Shuo Li, Xiayan Ji, Edgar Dobriban, Oleg Sokolsky, and Insup Lee. Pac-wrap: Semi-supervised pac anomaly detection. KDD '22, New York, NY, USA, 2022. Association for Computing Machinery.

[28] Yinghao Li. Uncertainty-aware and data-efficient fine-tuning and application of foundation models. 2025.

[29] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *3rd International Conference for Learning Representations*, 2015.

[30] Jing Su, Chufeng Jiang, Xin Jin, Yuxin Qiao, Tingsong Xiao, Hongda Ma, Rong Wei, Zhi Jing, Jiajun Xu, and Junhong Lin. Large language models for forecasting and anomaly detection: A systematic literature review, 2024.

[31] Mohamed Amine Ferrag, Mthandazo Ndhlovu, Norbert Tihanyi, Lucas C. Cordeiro, Merouane Debbah, Thierry Lestable, and Narinderjit Singh Thandi. Revolutionizing cyber threat detection with large language models: A privacy-preserving bert-based lightweight model for iot/iiot devices. *IEEE Access*, 12:23733–23750, 2024.

[32] Michael Guastalla, Yiyi Li, Arvin Hekmati, and Bhaskar Krishnamachari. Application of large language models to ddos attack detection. In Yu Chen, Chung-Wei Lin, Bo Chen, and Qi Zhu, editors, *Security and Privacy in Cyber-Physical Systems and Smart Vehicles*, pages 83–99, Cham, 2024. Springer Nature Switzerland.

[33] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, August 2016.

[34] Yara documentation. `https://virustotal.github.io/yara/`, 2024. Accessed: 2024-08-31.

A Basic Preprocessing

We preprocess the data before embedding it into vector representation by decoding Base64 content and tokenizing URLs and IP addresses to counter evasion techniques and improve command-line quality.

Base64 Decoding: Base64 encoding and code nesting are common evasion techniques. For example, in the potentially malicious script below, we observed that Base64-obfuscated Python code is nested within a shell command:

```
nohup python -c "import base64;exec(base64.b64decode(
'I2NvZGluZzogdXRmLTgKI3NpbXBsZSBodHRwX2JvdAppbXBvcn'
'QgdXJsbGliCmltcG9ydCBiYXNlNjQKaW1wb3J0IG9zCgpkZWYg'
'c29zKCk6CiAgICB1cmwgPSAnaHR0cHM6Ly9wYXN0ZWJpbi5jb2'
'0vcmF3LzA1cDBmVFlmJwogICAgdHJ5OgogICAgICAgIHBhZ2U9'
'YmFzZTY0LmI2NGRlY29kZSh1cmxsaWIudXJsb3Blbih1cmwpLn'
'JlYWQoKSkKICAgICAgICBmID0gb3MucG9wZW4oc3RyKHBhZ2U'
...
'OgogICAgcHJpbnQoJ1NvcnJ5IGJvc3MgSSBjYW5cJ3QgZ2V0I'
'Gluc3RydWN0aW9ucycpCiAgICBwYXNzCg=='
))" >/dev/null 2>&1 &
touch /tmp/.tmpk
```

Without base64 decoding and preprocessing of nested code, malicious content in the command-line may go undetected. We employ regex to identify and decode Base64-encoded content within command-lines, enabling analysis of otherwise obfuscated commands.

Nested Code Recognition: In the previous example, we also observe nested code patterns. These embedded snippets (e.g., Python within Shell) often appear as generic `word` or `string` AST nodes, obscuring intent. To recover structure, we detect such patterns and re-parse them with language-specific Tree-sitter parsers, yielding richer ASTs and improving detection in multi-language contexts.

URL and IP Tokenization: We detect URLs and IPs with regular expressions and prepend special tokens (`URL_TOKEN`, `IP_TOKEN`) before them during encoding. For AST embeddings, corresponding `URL` or `IP` nodes are attached to leaf nodes like `word` and `string` to add contextual information missing from the native AST.

B AST Embedding Example

In Figure 6, we show the AST for the code `setenforce 0 2>/dev/null`. The `setenforce` token generates the word node, signifying its role as a keyword. The subsequent number node for `0` and `file_redirect` node for `2>/dev/null` become leaves branching from the `word` node. Similarly, `file_descriptor` for `2` and `word` for `/dev/null` are leaves of the `file_redirect` node. In this manner, ASTs nest functionally-related chunks of code into subtrees. We begin by parsing each script, which consists of multiple command-lines, into an AST. For each code token within the script, we extract the absolute AST paths from the root node to the token, with the token itself serving as the final leaf of the path. We do not use relative path encodings because their number scales quadratically with the number of leaves. The AST embeddings for each corresponding token are then obtained by summing the embeddings of the AST nodes along these absolute paths, as shown in red paths of Figure 6.

C Dataset Statistics

The detailed training and testing subset composition can be found in Table 2. We train on benign samples from both in-house datasets, as this combination significantly improves performance. For testing, we use set-aside in-house data alongside public datasets (as detailed in Table 2). To ensure balanced evaluation with public malicious datasets, we pair them with equal amounts of randomly sampled in-house benign command-lines. This approach enables assessment of both in-distribution performance and OOD generalization capabilities.

For our in-house scripts and command-lines dataset, we also give an overview of the file length, base64 encoding, and nested code in Table 3. We found that script files can be lengthy, necessitating

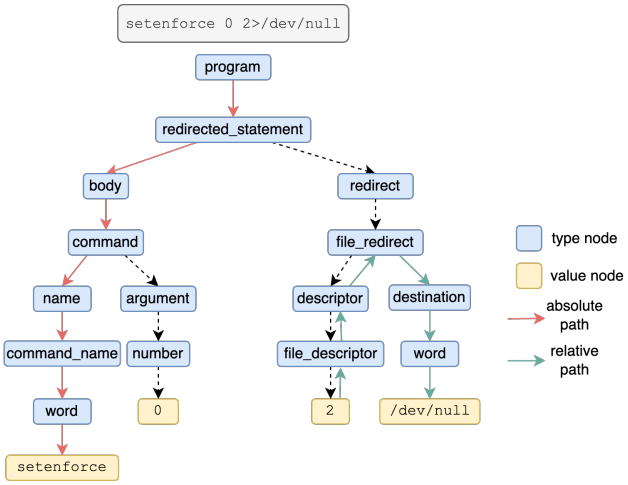


Figure 6: An AST example for the command `setenforce 0 2>/dev/null`. Red paths are absolute (root-to-leaf) and green paths are relative paths (leaf-to-leaf).

Subset	Composition	Count
Training Set	Benign scripts	72,986
	Benign commands	3,112
Testing Subset: scripts	Benign scripts	18,246
	Malicious scripts	18,246
Testing Subset: commands	Benign commands	3,112
	Malicious commands	3,112
Testing Subset: zenodo	Benign commands	267
	Malicious zenodo	267
Testing Subset: atomic	Benign commands	322
	Malicious atomic	322

Table 2: Training and test set compositions.

the limitation of maximum file length. Base64 encoding is prevalent in both malicious and benign files, underscoring the importance of decoding during preprocessing steps.

Dataset	Benign scripts	Malicious scripts	Benign cmd lines	Malicious cmd lines
count	89348	45926	6224	104658
mean	391	72	26	33
std	23528	237	14	15
min	1	1	4	1
max	6293733	11592	145	1401
base64	18072	13672	141	1522
nested	107	12	0	0

Table 3: Summary of file counts and file length statistics. The last two rows show the counts of base64-encoded and nested code files.

D Implementation and Experiment Setup

We use CodeBERT ($d = 768$) and AST ($m = 100$) embeddings with command sequences up to $L = 512$ lines. The model consists of a convolutional layer (kernel=3, 100 channels) and MaxPool (kernel=2, stride=2), followed by encoders/decoders built from three `ResnetBlockCondNorm2D` layers [26]. The latent space has $l = 32$ channels with output matching the CodeBERT dimension. Pretrained language model baselines were used, as fine-tuning degraded OOD performance due to dataset size. We use uncertainty quantification technique to compute thresholds for determine

Subset	Embedding Model	FPR (↓)	Recall (↑)	Precision (↑)	F1-Score (↑)	AUROC (↑)
scripts	CodeBERT	0.0046 ± 0.0026	$\mathbf{0.9997 \pm 0.0007}$	0.9954 ± 0.0026	0.9976 ± 0.0010	1.0000 ± 0.0000
	StarCoder	0.0043 ± 0.0018	0.9867 ± 0.0063	0.9957 ± 0.0018	0.9912 ± 0.0037	0.9996 ± 0.0007
	Phi-3	$\mathbf{0.0038 \pm 0.0033}$	0.7785 ± 0.5406	0.7961 ± 0.5526	0.7872 ± 0.5464	0.9991 ± 0.0004
	UnixCoder	0.0048 ± 0.0010	0.9628 ± 0.0129	0.9950 ± 0.0010	0.9786 ± 0.0065	0.9968 ± 0.0018
	EPVD	0.0050 ± 0.0015	0.9690 ± 0.0026	0.9948 ± 0.0012	0.9805 ± 0.0055	0.9940 ± 0.0013
cmd-lines	CodeBERT	$\mathbf{0.0045 \pm 0.0026}$	$\mathbf{1.0000 \pm 0.0000}$	0.9956 ± 0.0026	0.9978 ± 0.0013	1.0000 ± 0.0000
	StarCoder	0.0052 ± 0.0009	0.9976 ± 0.0006	0.9949 ± 0.0009	0.9962 ± 0.0005	0.9999 ± 0.0000
	Phi-3	0.0051 ± 0.0004	0.9981 ± 0.0013	0.9949 ± 0.0004	0.9965 ± 0.0005	1.0000 ± 0.0000
	UnixCoder	0.0049 ± 0.0007	0.9927 ± 0.0021	0.9950 ± 0.0007	0.9939 ± 0.0013	0.9996 ± 0.0001
	EPVD	0.0083 ± 0.0009	0.9921 ± 0.0009	0.9954 ± 0.0006	0.9950 ± 0.0012	0.9981 ± 0.0004
zenodo	CodeBERT	$\mathbf{0.0337 \pm 0.0235}$	$\mathbf{1.0000 \pm 0.0000}$	0.9677 ± 0.0223	0.9835 ± 0.0114	1.0000 ± 0.0000
	StarCoder	0.0570 ± 0.0192	$\mathbf{1.0000 \pm 0.0000}$	0.9462 ± 0.0170	0.9723 ± 0.0090	1.0000 ± 0.0000
	Phi-3	0.0364 ± 0.0395	0.7976 ± 0.5536	0.7656 ± 0.5321	0.7811 ± 0.5424	0.9994 ± 0.0007
	UnixCoder	0.0494 ± 0.0039	0.9356 ± 0.0519	0.9497 ± 0.0046	0.9423 ± 0.0279	0.9832 ± 0.0094
	EPVD	0.0521 ± 0.0044	0.9226 ± 0.0114	0.9390 ± 0.0032	0.9370 ± 0.0130	0.9824 ± 0.0064
atomic	CodeBERT	$\mathbf{0.0230 \pm 0.0224}$	$\mathbf{0.9975 \pm 0.0042}$	0.9777 ± 0.0214	0.9875 ± 0.0112	0.9990 ± 0.0016
	StarCoder	0.0503 ± 0.0018	0.8389 ± 0.1397	0.9425 ± 0.0111	0.8847 ± 0.0871	0.9664 ± 0.0264
	Phi-3	0.0498 ± 0.0018	0.9102 ± 0.0654	0.9479 ± 0.0049	0.9281 ± 0.0365	0.9812 ± 0.0113
	UnixCoder	0.0497 ± 0.0000	0.7118 ± 0.0996	0.9342 ± 0.0082	0.8062 ± 0.0661	0.9329 ± 0.0242
	EPVD	0.0477 ± 0.0014	0.9077 ± 0.0204	0.9321 ± 0.0042	0.9166 ± 0.0204	0.9712 ± 0.0203

Table 4: Performance metrics for our datasets and LLMs; CodeBERT has a robust performance across all the datasets.

anomaly [27, 28]. Training employed Adam [29] (lr=0.001) for 50 epochs on an Intel Xeon CPU with 4 Tesla V100-SXM2 GPUs.

E Ablation on Language Models

To evaluate, we benchmarked our model against SotA techniques increasingly used in AD for cybersecurity [30, 31, 32]. Specifically, we compared it with SotA LLMs such as StarCoder [10], UnixCoder [9], and Phi-3 [11].

Our analysis aims to: (1) demonstrate that our approach performs comparably to SotA LLMs in command-line threat detection, and (2) highlight the scalability benefits of our model for real-world scenarios with high volume of command-lines.

We extract hidden states from the last layer of each LLM and train an XGBoost classifier [33] on a mix of malicious and benign scripts and cmd-lines from in-house datasets. For a fair comparison, we use the encoder part of the CVAE and implement XGBoost similarly for all models. To threshold the anomaly score for FPR comparison, we apply a statistical learning tool [27].

Table 4 shows that PASTRAL consistently outperforms other models across most datasets and metrics, particularly excelling in AUCROC. In the scripts subset, PASTRAL achieves a perfect AUCROC and the highest TPR, improving on StarCoder by 1.31%. While Phi-3 achieves the lowest FPR, it lags in other metrics. In the cmd-lines subset, PASTRAL matches Phi-3 and StarCoder in AUCROC but surpasses StarCoder with a 13.46% lower FPR. In the zenodo subset, PASTRAL achieves the lowest FPR, 40.88% better than UnixCoder, and a higher F1-Score than StarCoder. In the atomic subset, PASTRAL leads in AUCROC and reduces FPR by 54.27% compared to StarCoder.

In Table 5, we present the latency, throughput, and memory consumption metrics for embedding extraction from LLMs and our proposed model, based on the mean of 100 randomly sampled scripts. Notice that we have listed all the parameters in PASTRAL, including those for CodeBERT. Our CVAE component, with only 2 million parameters, performs much faster during inference. We provided the entire pipeline for a fair comparison.

The comparison demonstrates the efficiency and performance advantages of the PASTRAL. With a combined CPU latency of 368.71 ms, it is 39.0% faster than StarCoder and 55.3% faster than Phi-3, though slower than UnixCoder. For CUDA latency, the combined value is 283.96 ms, making it 55.2% faster than StarCoder and 63.9% faster than Phi-3. The effective throughput of the combined model is 3.31 inputs/s, which is 224.5% higher than StarCoder and 101.2% higher than UnixCoder. The total parameter count for PASTRAL is 127 million, which is 95.77% fewer than StarCoder. The combined memory usage is 567.87 MB, which is 94.37% lower than StarCoder. Overall, PASTRAL

Model	Throughput	Parameters	Memory (MB)
StarCoder	1.02	3B	10090.03
Phi-3	0.85	3.8B	14802.67
UnixCoder	1.64	∼125M	571.56
EPVD	2.10	∼125M	∼500
IDS-LLM	2.50	∼110M	∼420-500
CodeBERT	3.31	∼125M	543.81

Table 5: Computation overhead of the embedding models. Throughput is measured in inputs/s.

Malware Type	Success	Miss	Total	Success %	Miss %
adware	3812	1	3813	99.97%	0.03%
downloader	5137	0	5137	100.00%	0.00%
hacktool	2	0	2	100.00%	0.00%
miner	241	1	242	99.59%	0.41%
pua	92	0	92	100.00%	0.00%
trojan	1593	28	1621	98.27%	1.73%
virus	1	0	1	100.00%	0.00%
vt_miss	148	1	149	99.33%	0.67%
Total	**11026**	**31**	**11057**	**99.72%**	**0.28%**

Table 6: Threat coverage of our approach for threat types identified by VirusTotal. 'vt_miss' row represents samples missed by VirusTotal yet identified malicious by manual inspection. Our approach outperforms all VirusTotal vendors by 79.19% in terms of miss ratio.

strikes a balance between latency, throughput, and resource usage, making it a highly efficient choice for large-scale threat detection.

F Threat Coverage Details and Robustness

In this section, we provide details on the threat coverage and robustness results. Table 6) shows PASTRAL's high detection accuracy across malware families. The model detected 3,812/3,813 adware samples and achieved perfect accuracy on 5,137 downloader samples. Notably, PASTRAL identified 148/149 threats missed by VirusTotal, reducing the miss ratio by 79.19% (from 1.35% to 0.28%). The model's main challenge was with trojans (28 missed out of 1,621), which often use obfuscation and dynamic behavior. For example, some trojans dynamically generate command lists using system files and fetch payloads during execution, making static analysis difficult.

While Table 7 shows the malware family distribution of the 11,057 VirusTotal reports for the malicious scripts, it offers a clear understanding of the distribution within our in-house dataset.

Robustness Against Obfuscation Techniques Given the widespread use of obfuscation techniques, it is critical to consider that adversaries may employ such methods to compromise the detection capabilities of PASTRAL. To evaluate the robustness of PASTRAL against obfuscated command-lines, we applied YARA rules [34] targeting four common obfuscation techniques: XOR encryption, ROT13 substitution cipher, Base64 encoding, and RC4 cipher. Our analysis identified 106 samples in the test set as potentially obfuscated, particularly using XOR encryption and Base64 encoding. These samples were manually labeled as malicious by our security expert, and their distribution is detailed in Table 8. PASTRAL effectively identifies all obfuscated samples as anomalies, assigning them high anomaly scores that are 248.5% above the anomaly threshold, flagging them as highly anomalous.

Family	Count	Ratio (%)	Family	Count	Ratio (%)
bundlore	3814	35.28%	mackeeper	222	2.05%
singleton	3592	33.23%	shlayer	105	0.97%
bash	1939	17.94%	morila	67	0.62%
medusalocker	436	4.03%	surfbuyer	52	0.48%
mirai	421	3.89%	kinsing	50	0.46%

Table 7: Top 10 families with their counts and ratios (in percentages). Singleton means AVClass cannot identify a family name for a sample.

Obfuscation	Count	Score	Coverage
Base64 Encoded	64	0.7787	100%
XOR Encryption	42	0.7893	100%

Table 8: Prediction score for obfuscated samples in test set, threshold=0.22; PASTRAL predicted all samples to be anomaly, providing 100% coverage.