# DNApp: Desafios de Manutenção da Integridade em Hosts via Caracterização de Aplicações baseada em Instruções

**Felipe Duarte Silva**[1], **Marco Zanata Alves**[1], **Paulo Lisboa de Almeida**[1], **André Grégio**[1]

[1] Departamento de Informática - Universidade Federal do Paraná (UFPR)
Curitiba - PR - Brasil

{felipeduarte, mazalves, paulorla, gregio}@ufpr.br

***Abstract.*** *In this paper, we introduce DNApp, a method for identifying privileged Linux programs based on syntactically analyzing their assembly instructions. To do so, we apply TF-IDF to vectorize opcode $n$-grams (bigrams, trigrams, and 4-grams) extracted from five Ubuntu versions' executable files. Then, we evaluate the average vectors using k-means and Silhouette coefficient to show that smaller samples can be better distinguished with 128-256 dimensions 4-grams, whereas 512 dimension bigrams works better for greater sets. Although we found clustering consistency, the method presents limitations (e.g., overlapping, undersized vectors). Overall, DNApp is promising to identify malicious changes in privileged binaries without relying on static signatures.*

***Resumo.*** *Este artigo propõe o DNApp, método para identificar executáveis privilegiados em Linux por meio da análise sintática de instruções assembly, utilizando n-gramas de opcode (bi, tri e 4-gramas) vetorizados com TF-IDF em binários de cinco versões do Ubuntu. Os vetores médios são avaliados com k-means e coeficiente de Silhouette, mostrando que 4-gramas com 128–256 dimensões separam melhor amostras pequenas, enquanto bigramas com 512 dimensões funcionam melhor em conjuntos maiores. Agrupamentos funcionais são formados de modo consistente, embora existam limitações como sobreposição e vetores restritos. O método pode ajudar a identificar modificações maliciosas em binários privilegiados, independentemente de assinaturas estáticas.*

## 1. Introdução

Mais de 50% dos sites ativos utilizam distribuições Linux como sistema operacional [W3Techs 2025], fazendo com que compreender a forma como programas são distribuídos e executados nesses ambientes seja essencial. No Linux, programas são, em geral, arquivos binários no formato ELF, um padrão que define a estrutura dos executáveis para que o sistema possa interpretá-los corretamente [Alan Lacerda 2021].Em sistemas Linux, executáveis presentes em `/usr/bin`, como `ls`, `grep` e `curl`, são de uso geral e podem ser iniciados por qualquer usuário [Linux Foundation 2015]. Já `/usr/sbin` armazena programas de administração do sistema que exigem privilégios de superusuário para serem executados. A substituição de um desses binários por código malicioso pode conceder ao invasor privilégios de *root* e, com isso, controle total da máquina. Por exemplo, o *malware* Ebury, ativo desde pelo menos 2011, compromete servidores Linux por meio da substituição de binários legítimos do OpenSSH, como `sshd`, por versões Trojanizadas capazes de roubar credenciais e manter acesso persistente [Bureau et al. 2014]. Em março

de 2024, um *backdoor* introduzido no pacote `xz-utils` foi detectado durante testes de rotina da distribuição Debian [Edge 2024]. Ambos os casos mostram que modificações sutis em componentes nativos podem escapar de revisões tradicionais de código-fonte.

Atualmente, a detecção de infecções em binários é comumente realizada pela verificação de integridade, como o *hash* SHA-512 distribuído com cada pacote [Debian Project 2024]. Apesar de efetiva contra alterações não-autorizadas conhecidas, essa abordagem falha em identificar binários maliciosos que já chegam acompanhados de um hash ou assinatura válidos. Esse é o caso, por exemplo, de ataques à cadeia de suprimentos, como o incidente envolvendo a empresa 3CX, em que invasores comprometeram o processo de construção de *software* e distribuíram um aplicativo adulterado com uma assinatura digital válida [Greenberg 2023]. Assim, o desenvolvimento de métodos capazes de detectar modificações em executáveis privilegiados sem depender exclusivamente de assinaturas se faz relevante, e pode levar a sistemas mais seguros e confiáveis. Com isso, o objetivo deste artigo é estabelecer um método de identificação de executáveis privilegiados em sistemas Linux por meio de seu padrão sintático de instruções.

## 2. Estado da Arte

***Fingerprinting* sintático com vetores de $n$-gramas.** O *disassembly* do executável tem sua sequência de instruções convertida em $n$-gramas de opcode e representada de forma vetorial, com ou sem o uso de técnicas de ponderação como o Term Frequency-Inverse Document Frequency (TF-IDF) [Salton and Yang 1973]. O vetor resultante é submetido a classificadores tradicionais como Árvores de Decisão e Máquina de Vetor de Suportes (SVMs). [Gray et al. 2024] aplicam essa estratégia a processos Windows/Unix, extraindo bigramas e trigramas para o treinamento de Florestas Aleatórias, obtendo cerca de 90% de acurácia na distinção entre código legítimo e malicioso. [Saini et al. 2025] utilizam unigramas e bigramas de opcode extraídos de binários que passaram por *disassembly* para treinar classificadores baseados em aprendizado supervisionado. Os vetores de frequência são normalizados e servem de entrada para modelos como SVMs, KNNs (*K-nearest neighbors*) e Árvores de Decisão. Os resultados do SVM com bigramas apresentou F1-score de 64%, superando as demais abordagens. Apesar de não empregar TF–IDF, o estudo mostra que combinações simples de instruções são suficientes para capturar padrões discriminativos entre famílias de malware.

**Assinaturas estáticas de malware.** Nessa abordagem, o binário é descrito por hashes fixos ou por histogramas de $n$-gramas sem ponderação. [Jalilian et al. 2020] comparam a eficácia de assinaturas SHA-256 (*bytewise*) com a de distribuições de $n$-gramas de opcode ($n \in [1..4]$), utilizando classificadores como SVMs e Florestas Aleatórias. Os resultados mostram que a verificação por hash é eficaz para identificar alterações não-autorizadas, mas se torna insuficiente frente a cenários em que o binário malicioso já é distribuído com uma assinatura ou hash aparentemente legítimos, como em ataques à cadeia de suprimentos. Por outro lado, as distribuições de $n$-gramas apresentaram desempenho robusto, mantendo AUC (área sob a curva ROC)—métrica que quantifica a capacidade do classificador em distinguir entre classes—sendo 1,0 o desempenho perfeito, acima de 0,90. A melhor performance foi alcançada com a combinação de 1, 2 e 3-gramas, resultando em acurácia de 77,32%. [Chen et al. 2021] utilizam frequências de opcode e características estruturais extraídas de executáveis Windows no formato PE para avaliar diferentes classificadores, incluindo $K$ Vizinhos Mais Próximos (KNN), e relatam acurácia de até 99% em seus ex-

perimentos. No entanto, destacam que a eficácia do método diminui frente a técnicas de ofuscação, que impactam negativamente a acurácia e elevam a taxa de falsos-positivos.

**Modelos baseados em atenção.** Tais modelos usam mecanismos similares aos descritos em [Vaswani et al. 2017], onde redes neurais profundas são treinadas para aprender as correlações entre tokens (e.g., bigramas), levando assim à decisões que consideram não só informações simples dos vetores de características (*embeddings*) como contagens de bigramas, mas também as relações que esses tokens possuem. Apesar de poderosos, tais sistemas comumente requerem vastas quantidades de dados para treinamento. [Romanov et al. 2023] fazem o *disassembly* do código, geram trigramas de opcode e treinam perceptron multicamadas (MLP) e uma rede DiSAN; alcançam 87 % de precisão na atribuição de autoria de cinco programadores. [Zhang et al. 2020] particionam $n$-gramas em *patches* fixos, aplicam TF–IDF e utilizam uma Rede Neural Convolucional (CNN) que também conta com mecanismos de atenção. Os autores reportam ganho de 4 p.p. da rede quando comparada com um SVM na identificação de famílias de *ransomware*, à custa de maior tempo de treinamento.

A Tabela 1 resume os principais trabalhos que aplicam técnicas de assinatura sintática à análise de binários maliciosos, incluindo o uso de $n$-gramas, vetorização por frequência ou TF–IDF e diferentes modelos de classificação supervisionada.

**Tabela 1. Assinatura sintática de binários. RF: Random Forest, GBM: Gradient Boosting Machine, SVM: Support Vector Machine, MLP: Multi-Layer Perceptron, DiSAN: Directional Self-Attention Network, CNN: Convolutional Neural Network.**

| Referência | $n$-gramas | Vetorização | Modelo |
|---|---|---|---|
| [Gray et al. 2024] | 2–3 | TF–IDF | RF, GBM |
| [Saini et al. 2025] | 3 | Frequência | SVM |
| [Jalilian et al. 2020] | 1–4 | Frequência | SVM, RF |
| [Chen et al. 2021] | 1 | Frequência | $k$-NN |
| [Romanov et al. 2023] | 3 | Embeddings | MLP, DiSAN |
| [Zhang et al. 2020] | Blocos fixos de n-gramas | TF–IDF | CNN com atenção |

Apesar de mostrar o valor dessas representações, nenhum dos estudos citados avalia a *estabilidade sintática* de um mesmo executável ao longo das diferentes versões de uma distribuição, investiga cortes de vocabulário por dimensão fixa (64–512) e seu impacto em agrupamento não supervisionado, ou propõe a média de perfis entre versões como uma assinatura sintática para detecção de trojans em binários privilegiados, lacunas estas abordadas pelo presente trabalho.

## 3. Metodologia

O fluxo metodológico empregado neste trabalho é composto por oito etapas: (i) coleta de dados; (ii) extração de $n$-gramas; (iii) contagem e seleção de $n$-gramas; (iv) vetorização TF–IDF; (v) normalização Min–Max; (vi) amostragem, agregação e clusterização; (vii) avaliação de agrupamentos; e (viii) projeção bidimensional por *t*-SNE.

**Coleta de dados.** Ao todo, foram considerados 75 executáveis disponíveis em `/usr/sbin` presentes em todas as versões da distribuição utilizada para os testes. A lista completa dos nomes dos executáveis está disponível no link:

https://github.com/secretrdlab/DNApp. Os binários foram extraídos de `/usr/sbin` nas versões do Ubuntu 22.10, 23.04, 23.10, 24.04 e 24.10, todas disponibilizadas como imagens Docker oficiais (`ubuntu:<versão>`) [Canonical Ltd. 2025]. Optou-se por versões intermediárias (não LTS) para captar variações semestrais em pacotes e bibliotecas. Os contêineres foram instanciados com Docker Engine 26.1.3 [Docker Documentation 2025], e os pacotes `binutils` e `file` foram instalados. Cada binário ELF passou por *disassembly* via `objdump -d`, com sintaxe AT&T, restringindo a decodificação às seções executáveis [Free Software Foundation 2024].

**Extração de $n$-gramas.** Após o *disassembly*, foram extraídos apenas os mnemônicos de instrução, descartando registradores, operandos imediatos e endereços. Aplicou-se uma janela deslizante de tamanho $n$ e passo 1 para gerar $n$-gramas sobrepostos, com $n \in \{2, 3, 4\}$; essa faixa captura dependências locais sem criar problemas de dimensionalidade muito grande [Santos et al. 2013]. Dessa forma, considerando as 75 aplicações, as cinco versões do Ubuntu e os três valores de $n$, foram gerados 39 arquivos distintos de $n$-gramas.

**Contagem e Seleção de $n$-gramas.** Os $n$-gramas extraídos foram contados individualmente por binário e foram removidos os tokens com ocorrência em menos de dois binários ou em mais de 80% deles, para eliminar elementos pouco discriminativos e ruídos.

**Vetorização TF-IDF.** Após a filtragem por frequência, os $n$-gramas que restaram de cada binário foram vetorizados por TF-IDF, atribuindo pesos conforme a frequência relativa de cada token no conjunto de aplicações. Cada vetor resultante codifica a distribuição dos $n$-gramas de uma aplicação em um espaço esparso de características.

**Normalização Min-Max.** Os vetores gerados por TF-IDF foram normalizados para o intervalo $[0, 1]$ usando a técnica Min-Max, aplicada por dimensão. Essa transformação preserva a distribuição relativa entre amostras e evita que alguma dimensão influencie desproporcionalmente as etapas seguintes, como média vetorial e clusterização.

**Amostragem, agregação e clusterização.** Dos 92 binários identificados inicialmente, 75 estavam presentes nas cinco versões do Ubuntu analisadas. A partir deles, foram definidos três tamanhos de amostra: 25, 50 e 75 executáveis.Para cada aplicação, os vetores TF–IDF normalizados de cada versão foram reunidos, e sua média aritmética foi calculada. O vetor resultante representa a assinatura sintática média da aplicação, utilizada como base para a clusterização. A clusterização foi realizada com o algoritmo *k*-means++, que define os centróides iniciais com base em uma distribuição ponderada [Arthur and Vassilvitskii 2006]. O número de *clusters* $K$ foi definido pelo método do cotovelo, que identifica o ponto em que aumentos em $K$ deixam de melhorar a qualidade do agrupamento [Han et al. 2011]. Foram considerados $K \in \{4, 6, 8\}$. O algoritmo foi configurado com limite de 300 iterações por execução, definido manualmente para garantir a estabilização dos centróides.

**Avaliação de agrupamentos.** A qualidade dos grupos foi mensurada pelo coeficiente de *Silhouette* [Kaufman and Rousseeuw 2009]. Para cada ponto $i$, define-se $a_i$ como a distância média entre $i$ e os pontos do mesmo *cluster*, e $b_i$ como a menor distância média entre $i$ e os pontos de outro *cluster*. O coeficiente de Silhouette é dado por:

$$s_i = \frac{b_i - a_i}{\max(a_i, b_i)}, \qquad s_i \in [-1, 1].$$

Valores próximos de 1 indicam boa separação entre os grupos; valores próximos de 0 sugerem sobreposição entre *clusters*; e valores negativos indicam alocação incorreta.

**Projeção bidimensional por *t*-SNE.** Para análise visual dos agrupamentos, os vetores médios foram projetados em duas dimensões via *t*-SNE [van der Maaten and Hinton 2008], com PCA prévio para redução de ruído e custo. Foram usados perplexidade 20, taxa de aprendizado 200 e 1000 iterações.

## 4. Experimentos e Resultados

Os testes foram executados nos contêineres descritos na Seção 3, preservando o mesmo ambiente para as cinco versões do Ubuntu. As implementações foram feitas na linguagem de programação Python na versão 3.8.10, com os pacotes Pandas 2.0.1, numpy 1.24.3, e scikit-learn 1.3.2. Foram avaliadas combinações com diferentes tamanhos de amostra ($|A| = 25, 50, 75$), $n$-gramas ($n = 2, 3, 4$) e dimensionalidades TF–IDF ($d = 64, 128, 256, 512$). As amostras de 25 e 50 aplicações foram extraídas aleatoriamente do conjunto de 75 binários, para avaliar o comportamento do método em diferentes escalas. Para cada configuração, aplicou-se o fluxo da Seção 3, com *k-means* e número de *clusters* proporcional ao tamanho da amostra ($k = 4$, 6 e 8). A avaliação foi qualitativa, via projeções *t-SNE*, e quantitativa, pelo coeficiente de *Silhouette*. A Tabela 2 mostra as médias e desvios padrão dos coeficientes de *Silhouette*, calculados a partir de 30 execuções do *k-means* por configuração. Nota-se uma queda dos coeficientes conforme $|A|$ aumenta de 25 para 75, indicando menor separação entre grupos. Para $|A| = 25$ e 50, os maiores valores foram obtidos com bigramas em 256 e 512 dimensões — padrão que se manteve com 75 aplicações. Isso sugere que bigramas aliados a vetores mais longos favorecem a separabilidade em diferentes escalas. Os desvios padrão revelam a estabilidade dos agrupamentos. O maior desvio foi registrado com 4-gramas em 128 dimensões e 50 aplicações (0,0232), indicando maior variabilidade nas partições geradas, possivelmente pela dificuldade de separar grupos de forma consistente nesse cenário.

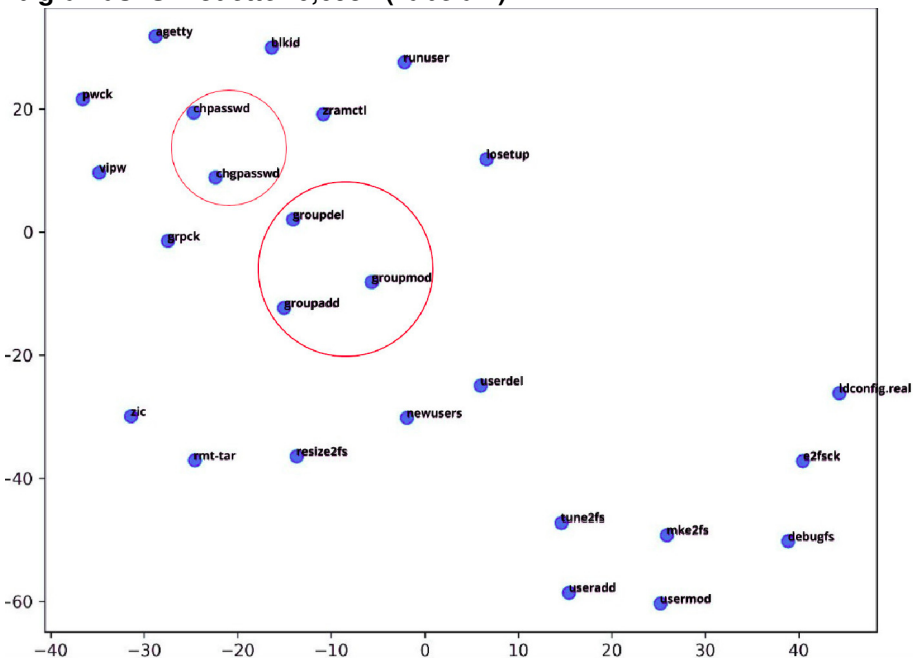
**Tabela 2. Média e desvio padrão do coeficiente de *Silhouette* (30 execuções) para diferentes dimensões, n-gramas e tamanhos de amostra**

| Dimensão | N-grama | 25 apps (K=4) | 50 apps (K=6) | 75 apps (K=8) |
|---|---|---|---|---|
| 64 | Bigrams | 0.6937 ± 0.0000 | 0.6083 ± 0.0000 | 0.5135 ± 0.0001 |
| | Trigrams | 0.6903 ± 0.0000 | 0.5339 ± 0.0000 | 0.4801 ± 0.0005 |
| | 4-grams | 0.6771 ± 0.0000 | 0.5678 ± 0.0006 | 0.4502 ± 0.0048 |
| 128 | Bigrams | 0.6943 ± 0.0000 | 0.6100 ± 0.0000 | 0.5129 ± 0.0070 |
| | Trigrams | 0.6905 ± 0.0000 | 0.5382 ± 0.0050 | 0.4830 ± 0.0017 |
| | 4-grams | 0.6798 ± 0.0000 | 0.5640 ± 0.0232 | 0.4613 ± 0.0007 |
| 256 | Bigrams | 0.6953 ± 0.0000 | 0.6118 ± 0.0000 | 0.5170 ± 0.0004 |
| | Trigrams | 0.6905 ± 0.0000 | 0.5398 ± 0.0000 | 0.4850 ± 0.0000 |
| | 4-grams | 0.6814 ± 0.0000 | 0.5693 ± 0.0170 | 0.4655 ± 0.0012 |
| 512 | Bigrams | 0.6957 ± 0.0000 | 0.6125 ± 0.0000 | 0.5164 ± 0.0062 |
| | Trigrams | 0.6917 ± 0.0000 | 0.5426 ± 0.0000 | 0.4875 ± 0.0010 |
| | 4-grams | 0.6832 ± 0.0000 | 0.5746 ± 0.0000 | 0.4700 ± 0.0009 |

A Figura 1 apresenta a projeção *t-SNE* obtida para bigramas com 512 dimensões no cenário de 25 aplicações, configuração que teve o maior coeficiente de *Silhouette* do

estudo. Os pontos correspondem aos perfis médios de cada binário, a distância entre eles reflete a similaridade sintática medida pelo vetor de características gerado. Observa-se um agrupamento compacto dos utilitários `groupadd`, `groupmod` e `groupdel`, responsáveis por criar, alterar e remover grupos de usuários. Também aparecem próximos `chpasswd` e `chgpasswd`, voltados à atualização de senhas, além de `mke2fs`, `tune2fs` e `e2fsck`, do conjunto de ferramentas do ext4. Essa proximidade sugere que os vetores capturam instruções comuns a tarefas da mesma família funcional, em linha com o coeficiente de *Silhouette* 0,6957 da Tabela 2.

**Figura 1. Projeção *t-SNE* dos vetores médios (512 dimensões) em 25 aplicações com bigramas. *Silhouette*: 0,6937 (Tabela 2).**




Embora o foco principal do estudo não tenha sido a detecção direta de alterações maliciosas, os agrupamentos obtidos servem como uma linha de base sintática para cada aplicação. A ideia é que, ao introduzir modificações estruturais em um binário, como uma tentativa de trojanização, a distribuição de seus opcodes se altere a ponto de distanciá-lo dos demais binários com função semelhante. Esse afastamento pode ser interpretado como um *outlier*, funcionando como um primeiro sinal de alerta para análises manuais mais aprofundadas.

## 5. Conclusão

Neste artigo, propôs-se um fluxo para cálculo de assinaturas sintáticas de executáveis privilegiados no Linux, tendo como caso de estudo 75 binários presentes nas versões 22.10, 23.04, 23.10, 24.04 e 24.10 do Ubuntu. Foram testadas diferentes configurações para os vetores de características: tamanhos de $n$-gramas (2, 3 e 4), dimensionalidades (64, 128, 256 e 512) e quantidades de aplicações ($|A| = 25, 50, 75$). O objetivo foi investigar em que condições essas assinaturas permitem distinguir binários com base em seu perfil sintático, medido pelo coeficiente de *Silhouette* após a clusterização com *k-means*. Os testes mostraram que os executáveis puderam ser agrupados com base em padrões sintáticos ex-

traídos de instruções de máquina, com coerência funcional nos grupos formados. Em conjuntos menores, bigramas com vetores de alta dimensionalidade geraram separações mais nítidas. Já em conjuntos maiores, a qualidade da separação caiu, exigindo vetores mais informativos. Os resultados sugerem o potencial das assinaturas sintáticas para detectar alterações estruturais em binários, podendo apoiar estratégias de detecção de executáveis maliciosos. Dentre os resultados obtidos, é possível destacar as seguintes conclusões: (I) Como esperado, o aumento no número de aplicações analisadas dificultou a separação dos grupos. Com 25 aplicações, foi possível formar agrupamentos sintaticamente coesos, como o conjunto `groupadd`, `groupmod` e `groupdel`, que compartilham funções administrativas semelhantes. Nessa configuração, bigramas de 512 dimensões alcançaram o maior coeficiente de *Silhouette* ($s = 0{,}6937$). À medida que o conjunto cresceu para 50 e 75 binários, a separação entre os *clusters* se tornou menos nítida (caindo para $s = 0{,}6083$ e $s = 0{,}5135$, respectivamente); (II) A dimensionalidade dos vetores teve impacto na qualidade dos agrupamentos. Nas três amostras ($|A| = 25, 50, 75$), configurações com 128 a 512 dimensões produziram melhores separações do que vetores com apenas 64 dimensões, que apresentaram um aumento na sobreposição entre *clusters*; (III) A projeção *t-SNE* reforçou essas observações, evidenciando agrupamentos distintos nas configurações menores e uma maior sobreposição à medida que a complexidade dos dados aumentava. Trabalhos futuros podem explorar esquemas de ponderação como o BM25 e técnicas de projeção não lineares mais recentes (UMAP, PaCMAP), visando obter separações eficazes com vetores mais compactos. Por fim, aplicar o *pipeline* a distribuições com atualizações quase diárias, permitirá testar sua resiliência em cenários contínuos, aproximando-o de um sistema de monitoramento em produção.

## Agradecimentos

Este trabalho foi apoiado pelo Centro de Computação Científica e Software Livre (C3SL) da UFPR, em parceria com o Ministério da Saúde e o Ministério da Educação, bem como pelo CNPq por meio de bolsa de produtividade em pesquisa.

## Referências

Alan Lacerda (2021). O formato elf (executable and linking format). `https://alacerda.github.io/posts/o-formato-elf/`. Acesso em: maio 2025.

Arthur, D. and Vassilvitskii, S. (2006). k-means++: The advantages of careful seeding. Technical report, Stanford.

Bureau, P.-M., Étienne M. Léveillé, M., and Bilodeau, O. (2014). Operation windigo: The vivisection of a large linux server-side credential stealing malware campaign. Technical report, ESET.

Canonical Ltd. (2025). Ubuntu release notes. `https://releases.ubuntu.com/`.

Chen, W., Zhao, S., and Zhang, L. (2021). Static malware detection via opcode unigram frequency and $k$-nn. *Security and Communication Networks*, 2021:1–13.

Debian Project (2024). Debian policy manual: Checksums in `/var/lib/dpkg/status`. `https://www.debian.org/doc/debian-policy/ch-files.html#s-checksums`.

Docker Documentation (2025). *Docker Engine Reference*. Docker, Inc. Versão 26.1.3. Disponível em: `https://docs.docker.com/engine/`.

Edge, J. (2024). Backdoor discovered in xz utils compression library. `https://lwn.net/Articles/965307/`. Publicado em: 29 mar. 2024. Acesso em: maio 2025.

Free Software Foundation (2024). *GNU Binutils Manual*. GNU Project. Disponível em: `https://www.gnu.org/software/binutils/manual/`.

Gray, J., Sgandurra, D., Cavallaro, L., and Alis, J. B. (2024). Identifying authorship in malicious binaries: Features, challenges & datasets. *ACM Computing Surveys*.

Greenberg, A. (2023). The huge 3cx breach was actually 2 linked supply chain attacks. *WIRED*.

Han, J., Kamber, M., and Pei, J. (2011). *Data Mining: Concepts and Techniques*. Morgan Kaufmann, Waltham, MA, 3rd edition.

Jalilian, A., Narimani, Z., and Ansari, E. (2020). Static signature-based malware detection using opcode and binary information. In *Data Science: From Research to Application*, volume 45 of *Lecture Notes on Data Engineering and Communications Technologies*, pages 24–35. Springer.

Kaufman, L. and Rousseeuw, P. J. (2009). *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons.

Linux Foundation (2015). Filesystem hierarchy standard, version 3.0. `https://refspecs.linuxfoundation.org/FHS_3.0/fhs-3.0.html`. Seções 4.13 (/usr/bin) e 4.14 (/usr/sbin). Acesso em: maio 2025.

Romanov, A., Kurtukova, A., Fedotova, A., and Shelupanov, A. (2023). Authorship identification of binary and disassembled codes using nlp methods. *Information*, 14(7):361.

Saini, V., Gupta, R., and Soni, N. (2025). Opcode-based malware classification using machine learning and deep learning techniques. *arXiv preprint arXiv:2504.13408*.

Salton, G. and Yang, C.-S. (1973). On the specification of term values in automatic indexing. *Journal of documentation*, 29(4):351–372.

Santos, I., Brezo, F., Ugarte-Pedrero, X., and Bringas, P. G. (2013). Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231:64–82.

van der Maaten, L. and Hinton, G. (2008). Visualizing data using t-sne. *Journal of Machine Learning Research*, 9:2579–2605.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

W3Techs (2025). Usage statistics of linux for websites. `https://w3techs.com/technologies/details/os-linux`.

Zhang, B., Xiao, W., Xiao, X., Sangaiah, A. K., Zhang, W., and Zhang, J. (2020). Ransomware classification using patch-based cnn and self-attention network on embedded n-grams of opcodes. *Future Generation Computer Systems*, 110:708–720.