Sistema de Detecção de Ameaças em Pacotes Linux via Rust

Pedro Kroning Correa Balen, Alexandre De Oliveira Zamberlan
Curso de Ciência da Computação
UFN - Universidade Franciscana
Santa Maria - RS
p.balen@ufn.edu.br, alexz@ufn.edu.br

***Resumo*—Os sistemas Linux têm se tornado alvos crescentes de softwares maliciosos que visam comprometer a segurança e a integridade dos dados. As aplicações de antivírus tradicionais, muitas vezes falham em detectar ameaças novas ou modificadas, criando uma janela de vulnerabilidade. Este trabalho visa o desenvolvimento de um protótipo de ferramenta para detecção de *malwares* em Linux, que integra a análise por assinaturas com a análise preditiva por heurísticas. Para isso, foi utilizada a linguagem Rust, combinando regras YARA para ameaças conhecidas e *Machine Learning* para a detecção de padrões maliciosos desconhecidos. Registra-se que, ao final deste projeto, a ferramenta analisou pacotes Linux, exibiu um veredito no terminal e gerou um relatório em formato texto, permitindo uma análise de segurança mais ágil e completa. Para isso, resultados de avaliação foram exibidos e discutidos.**

***Palavras-chave*: Antivírus; Linux; Rust; YARA; Machine Learning.**

I. Introdução

A segurança da informação passa por constantes provações nos dias atuais[1], sendo uma aliada fundamental da tecnologia da informação. A frequência das manifestações de softwares maliciosos, popularmente conhecidos como vírus, que englobam uma vasta categoria de ameaças como *worms*, *trojans*, *spyware* e *ransomware*, tem aumentado significativamente [1]. Esses softwares são geralmente desenvolvidos com o propósito de comprometer a integridade, a confidencialidade e a disponibilidade de sistemas computacionais. A evolução dessas ameaças que empregam técnicas avançadas para se esquivar da detecção, representa um risco substancial para todos os sistemas operacionais, incluindo o ambiente Linux, que tem se tornado um alvo crescente para ataques especializados, como cita Raffa, Sgandurra e O'Keeffe [1].

As ferramentas de detecção, amplamente conhecidas como antivírus (AV), adotam diferentes métodos na busca por arquivos maliciosos [1], [2]. Uma das abordagens é a busca por elementos presentes dentro do arquivo que podem ser julgados como maliciosos, entre os quais enquadram sequências de valores hexadecimais, binários, entre outros. Essas sequências são rotuladas como assinaturas e podem ser definidas como a identidade de um *malware* [2], [3], esses valores são previamente conhecidos e catalogados. Embora seja uma estratégia de alta precisão para identificar ameaças conhecidas, esse modelo só é eficaz contra ameaças já catalogadas. Ele pode falhar em proteger contra *malwares* novos ou variantes modificadas recém descobertas, conhecidas como ameaças de Dia Zero [2], criando uma perigosa janela de vulnerabilidade entre o surgimento da ameaça e a criação e distribuição de uma nova assinatura.

A pesquisa em cibersegurança adota, complementarmente a busca por assinaturas, abordagens inspiradas na Inteligência Artificial (IA), como a análise heurística. Conforme descrito por Aslan e Samet [2], esta abordagem não depende de correspondências exatas com ameaças conhecidas. Em vez disso, ela utiliza técnicas diversas, como *Machine Learning*, para identificar padrões de comportamento suspeitos que indicam algo malicioso [2].

Portanto, o objetivo geral é desenvolver um protótipo de ferramenta para detecção de *malware* em sistemas operacionais Linux, que integra análise baseada em assinaturas e análise preditiva por heurísticas. Como objetivos específicos, há: i) desenvolver uma aplicação baseada em regras de assinatura para detectar *malwares* já conhecidos, assim como em análise heurística para prever se um pacote Linux é malicioso com base em suas características; ii) definir quais métricas serão utilizadas nos testes; iii) testar a ferramenta com amostras de *malwares* e programas legítimos para medir sua eficácia.

II. Referencial teórico

Nesta seção, busca-se apresentar e discutir os conceitos, ferramentas e trabalhos relacionados que dão suporte para este estudo.

A. Contexto: Linux, Pacote Linux, Malwares, Linha de Comando, Virus

Linux é um sistema operacional de código aberto, baseado no kernel Linux. Diferente de sistemas proprietários, seu código-fonte é publicamente acessível e pode ser modificado e distribuído livremente [1], [4].

Em distribuições Linux, um pacote é a unidade básica de distribuição e gerenciamento de software. De forma geral, um pacote Linux é um arquivo que contém os binários compilados de um programa, bibliotecas associadas, arquivos de configuração, documentação e metadados que descrevem o software, suas versões e suas dependências [5]. Os binários executáveis contidos nesses pacotes seguem o formato ELF

[1]Este trabalho foi realizado em 2025-2026.

(*Executable and Linkable Format*), que é o formato padrão de executáveis em sistemas Linux.

Malware é qualquer software que executa intencionalmente cargas maliciosas em máquinas vítimas, como computadores, celulares, e redes de computadores [2]. O principal propósito do *malware* é comprometer a integridade, a confidencialidade ou a disponibilidade de um sistema computacional.

O terminal, também conhecido como linha de comando ou console, é uma interface textual que permite a comunicação direta entre o usuário e o sistema operacional por meio de comandos digitados, possibilitando executar programas, manipular arquivos e gerenciar processos. Uma Text User Interface (TUI) é um tipo de interface que permite a interação com o sistema por meio de elementos visuais exibidos em modo texto, geralmente dentro de um terminal. Em aplicações de linha de comando, o termo parâmetro refere-se às informações fornecidas pelo usuário ao chamar um programa, passadas junto ao nome da aplicação no terminal, com o objetivo de definir como ele deve ser executado ou qual dado deve ser processado.

De acordo com Aslan e Samet [2], o vírus é um tipo específico de *malware*. Sua característica principal é a capacidade de se replicar ao se anexar a outros programas ou arquivos legítimos, que atuam como hospedeiros. Os autores afirmam que muitos vírus modernos utilizam técnicas para não serem detectados por sistemas de segurança, como os antivírus.

O MalwareBazaar é uma plataforma pública, voltada para o armazenamento, compartilhamento e análise de amostras de *malware*. O seu principal objetivo é auxiliar pesquisadores, analistas de segurança e desenvolvedores de ferramentas antivírus no estudo de ameaças digitais, fornecendo acesso a um amplo repositório de arquivos maliciosos classificados e documentados.

Os autores Raffa, Sgandurra e O'Keeffe [1] categorizam antivírus como um software projetado especificamente para detectar software malicioso e é uma das medidas de segurança preventivas mais amplamente adotadas. Ele também pode ser implementado como um controle reativo, sendo, na maioria dos casos, capaz de remover o código malicioso em que o mecanismo de deteção tenha sido acionado com sucesso. A mesma fonte também reforça que o software antivírus desempenha um papel importante na proteção de usuários finais e redes contra vários tipos de *malware*.

O VirusTotal é uma plataforma online gratuita voltada para a análise e detecção de arquivos e URLs potencialmente maliciosos. O serviço permite que usuários enviem arquivos, programas, pacotes e links suspeitos, os quais são analisados simultaneamente por dezenas de motores antivírus e mecanismos de detecção de ameaças.

B. Teoria básica: Machine Learning, Assinatura, Hashing

Ömer Aslan e Refik Samet [2] definem *Machine Learning* como um subcampo da Inteligência Artificial, que consiste em um conjunto de algoritmos que estima corretamente os resultados das aplicações sem ser explicitamente programado.

Assinatura é uma característica do *malware* que encapsula a estrutura do programa e identifica exclusivamente cada *malware* [2], essa identificação é feita a partir de uma comparação da assinatura do arquivo a ser investigado com as assinaturas já conhecidas por serem maliciosas [3].

O termo *hashing* refere-se ao processo de transformar dados de tamanho variável em uma sequência fixa de caracteres, por meio de uma função matemática denominada função *hash*. O resultado dessa operação é conhecido como valor *hash* ou resumo criptográfico, e serve como uma "impressão digital" única do conteúdo original. O SSDEEP é uma ferramenta utilizada para a identificação de similaridade entre arquivos, por meio de um método de *hashing*.

Conforme explicado por Aslan e Samet [2], a detecção baseada em heurísticas é uma abordagem que utiliza sistemas baseados em técnicas de *Machine Learning*, para identificar *malware*. Diferentemente da detecção por assinatura, que busca por uma característica única e exatas de ameaças já conhecidas, a abordagem heurística analisa o código ou o comportamento de um programa para determinar se é malicioso.

No ecossistema Python, de acordo com o site de referência [6], o scikit-learn destaca-se como uma das bibliotecas mais robustas para *Machine Learning*, oferecendo recursos essenciais para o reconhecimento de padrões em textos e imagens. Sua arquitetura é fundamentada em interfaces padronizadas, como *Estimators* (para o treinamento de modelos), *Transformers* (para o pré-processamento de dados) e *Predictors* (para a geração de previsões). Além de facilitar a integração com diferentes aplicações, a biblioteca otimiza o fluxo de trabalho por meio de classes como o *Pipeline*, que automatiza a preparação e o treinamento, permitindo a exportação de modelos prontos para execução em ambientes de produção.

C. Tecnologias e metodologias aplicadas: Matplotlib, Rust, YARA, Máquina Virtual, Scrum, GitHub

Complementarmente ao treinamento, a biblioteca Matplotlib atua como a ferramenta essencial para a visualização e comunicação dos resultados gerados pelos modelos de reconhecimento. Em uma aplicação baseada no scikit-learn, o Matplotlib é responsável por transformar as predições em relatórios gráficos detalhados, permitindo quantificar o desempenho do sistema. Por meio de recursos como matrizes de confusão e gráficos de barras, a biblioteca evidencia a eficácia do reconhecimento, distinguindo claramente os acertos fidedignos dos erros cometidos [7]. Essa integração garante que o usuário final tenha uma compreensão visual e precisa de como o modelo interpretou os padrões processados.

Rust é uma linguagem de programação que, apesar de jovem, tem sido frequentemente classificada positivamente na pesquisa de desenvolvedores do Stack Overflow desde 2016 [8]. Bugden e Alahmar também afirmam que o sucesso vem de seus principais destaques, segurança de memória e o desempenho. Isso ocorre pela maneira a qual a linguagem gerencia a memória em tempo de compilação, por meio de um sistema de propriedade, eliminando a necessidade de um coletor de lixo. A memória é alocada quando uma variável é declarada e

desalocada assim que ela sai de escopo. Este sistema, segundo os autores [8], impede vulnerabilidades comuns como use after free e double free, pois o programador não gerencia a memória manualmente, o que define uma regra fundamental da linguagem, que só pode haver um proprietário da memória por vez.

YARA, conforme definido por Lockett [3], é uma ferramenta que permite a criação de regras para identificar e classificar *malware* com base em padrões de dados textuais ou binários. O autor a descreve como uma solução mais flexível para a detecção baseada em assinatura em comparação com métodos como o hashing criptográfico, que tenta buscar por identidade exata e pouco flexível. Também se destaca pelo alto grau de personalização, permitindo criar expressões regulares, que consistem de sequência de caracteres que formam um padrão de busca.

Uma máquina virtual (VM) é um ambiente de computação isolado que emula um sistema físico dentro de outro sistema operacional. Ela permite executar sistemas e aplicações de forma independente do hardware físico, proporcionando um ambiente controlado e seguro para testes e desenvolvimento.

O Scrum é uma metodologia ágil de gerenciamento de projetos amplamente utilizada no desenvolvimento de software. Seu principal objetivo é organizar e otimizar o processo de desenvolvimento por meio de ciclos curtos de trabalho e entregas contínuas de valor.

O GitHub é uma plataforma online voltada para o armazenamento, controle de versão e colaboração em projetos de softwares. A ferramenta permite que desenvolvedores de diferentes locais trabalhem de forma integrada, acompanhando as modificações realizadas no código-fonte e mantendo um histórico detalhado de todas as versões do projeto.

D. Trabalhos correlatos

Jaiswal [4] conduziu uma avaliação comparativa entre Linux, Windows 10 e macOS, baseada em seis fatores: segurança, eficiência, processamento, acessibilidade, controle de funcionalidades e usabilidade geral. O estudo conclui que o Linux é uma alternativa superior aos sistemas proprietários em termos de segurança, desempenho e customização.

Gherardi et al. [5] analisam a evolução dos tamanhos de pacotes na distribuição Ubuntu ao longo de suas versões consecutivas, tratando o sistema operacional como um ecossistema de pacotes interdependentes. A partir de aproximadamente 370 mil mudanças de tamanho, os autores construíram um modelo matemático capaz de prever com alta precisão a distribuição de tamanhos dos pacotes. Os resultados indicam um tamanho mínimo de pacote de aproximadamente 741 bytes, imposto pelo sistema de gerenciamento, e que pacotes grandes têm menor probabilidade de sofrer grandes aumentos entre versões. Embora o foco não seja segurança, o trabalho evidencia padrões estatísticos bem definidos na evolução estrutural de pacotes Linux.

Bugden e Alahmar [8] conduziram um estudo comparativo de *benchmarking* entre seis linguagens C, C++, Go, Java, Python e Rust, com foco em segurança e desempenho. Os resultados demonstraram que Rust superou Go, Java e Python em desempenho e se manteve competitiva em comparação com C e C++. Quanto à segurança, Rust foi considerada a mais segura entre as linguagens analisadas, especialmente em ambientes concorrentes.

Raffa, Sgandurra e O'Keeffe [1] realizaram um estudo sistemático sobre o estado dos antivírus para Linux, testando quatro AVs instalados localmente ao longo de dez meses e 58 AVs por meio do VirusTotal com 4000 amostras de *malware*. A análise de capacidades foi realizada com a ferramenta CAPA. Os resultados revelaram que, embora três dos quatro AVs locais exibissem taxas de detecção acima de 90%, dois apresentaram regressão no nível do arquivo. No VirusTotal, um terço dos AVs teve taxa de detecção de no máximo 30%, e 24 dos 58 foram afetados por regressão. A análise sugeriu que os autores de *malware* estão focando em especializar abordagens existentes para evadir antivírus. Essas descobertas levaram os autores a concluir que as bases de dados de assinaturas de AVs Linux não são bem mantidas pelos fornecedores.

Aslan e Samet [2] publicaram uma revisão abrangente sobre as diversas abordagens de detecção de *malware*. O estudo analisou múltiplas abordagens, incluindo detecção baseada em assinaturas, heurística, comportamento, aprendizado profundo, nuvem e IoT. A detecção por assinaturas é rápida para *malwares* conhecidos, mas falha com desconhecidos. A abordagem heurística tem bom desempenho para variantes conhecidas e algumas desconhecidas, mas pode gerar altas taxas de falsos positivos. A detecção baseada em comportamento é eficaz para *malwares* desconhecidos, pois os programas tendem a manter comportamentos similares mesmo com alterações no código. As abordagens mais recentes, como aprendizado profundo e nuvem, melhoram as taxas de detecção, mas apresentam desafios de resistência a ataques e privacidade de dados, respectivamente.

Lockett [3] avaliou a eficácia das regras YARA comparando-as com o hashing criptográfico SHA-256 e o hashing difuso SSDEEP, utilizando dois conjuntos de 15 amostras de *malwares*: amostras conhecidas e ofuscadas do GitHub e amostras novas do VirusTotal. O hashing criptográfico detectou todas as amostras conhecidas, mas falhou com as novas. O SSDEEP demonstrou capacidade de identificar similaridades em amostras ofuscadas. As regras YARA apresentaram a maior taxa de correspondência (73%) para classificação de *malware* ofuscado e novo, superando o SSDEEP (33%), com uma taxa de detecção geral de 57%. O estudo concluiu que as regras YARA são a técnica mais eficaz para detecção baseada em assinaturas, mas destacou a necessidade de complementá-las com métodos baseados em comportamento.

Finalmente, esses trabalhos colaboraram para definições de teorias e de ferramentas. Jaiswal [4] e Gherardi et al. [5] ajudaram na escolha do escopo de Sistema Operacional. Bugden e Alahmar [8] indicou a linguagem Rust e suas bibliotecas. Raffa, Sgandurra e O'Keeffe [1] mostrou que uso de assinaturas estáticas em AV podem apresentar falhas, justificando o uso de *Machine Learning*. Aslan e Samet [2] deram suporte para o uso de *Machine Learning* em questões

comportamentais e heurísticas. E Lockett [3] auxiliou na construção de regras YARA em métodos estáticos.

III. Metodologia

Para o desenvolvimento deste software, foi utilizada a metodologia ágil Scrum, que possibilita um processo colaborativo, favorecendo a comunicação entre os envolvidos e a organização das tarefas. Essa abordagem baseou-se em entregas incrementais de resultados e funcionalidades, realizadas semanalmente entre o aluno e o orientador, garantindo o acompanhamento contínuo da evolução do projeto. Para o versionamento do projeto, foi utilizada a ferramenta GitHub. O desenvolvimento do sistema fundamentou-se em um ecossistema multi-linguagem, utilizando Python para a gestão da lógica de *machine learning* com scikit-learn e para a geração de relatórios gráficos via Matplotlib. O desempenho e a segurança de baixo nível foram garantidas pela linguagem Rust, enquanto a análise especializada de artefatos e a extração de capacidades de arquivos estáticos foram implementadas por meio das ferramentas YARA e Capa.

Para validar a eficácia do protótipo e atender aos objetivos específicos, foi adotada uma abordagem de pesquisa experimental. O desenvolvimento foi dividido em duas fases principais de implementação, alinhadas com a abordagem híbrida do sistema: **Fase 1: Detecção por Assinatura.** Desenvolvimento do motor de análise estática utilizando a biblioteca YARA. O objetivo desta fase foi validar a capacidade da ferramenta em detectar *malwares* já conhecidos e catalogados, utilizando regras de repositórios públicos; **Fase 2: Detecção Preditiva.** Desenvolvimento do módulo de análise heurística utilizando técnicas de *Machine Learning*. O foco desta fase foi implementar a capacidade de prever ameaças desconhecidas baseando-se em características suspeitas extraídas dos arquivos.

Para validar a eficácia do protótipo, foi adotada uma metodologia de testes baseada na definição de métricas claras e na utilização de um conjunto de dados controlado para os experimentos.

a) Métricas de Avaliação: Para medir a eficácia da ferramenta foram utilizadas duas métricas principais: a taxa de detecção, caracterizada pelo percentual de amostras maliciosas corretamente identificadas como tal, e a taxa de falsos positivos, percentual de software benigno incorretamente classificado como malicioso. Adicionalmente, o tempo de execução da análise foi cronometrado.

b) Conjunto de Dados para Teste: Os testes foram executados em um ambiente controlado, utilizando uma máquina virtual. Foram utilizados dois conjuntos de amostras distintos:

- **Amostras Maliciosas:** Um conjunto de pacotes Linux obtidos de repositórios públicos de ameaças, como o MalwareBazaar, para testar a taxa de detecção.
- **Amostras Benignas:** Um conjunto de pacotes de software legítimo e de código aberto, obtidos de repositórios oficiais, para avaliar a taxa de falsos positivos.

A. Modelagem

Para organização do sistema, foram definidos os requisitos funcionais (RF) e Não Funcionais (RNF) que podem ser encontrados no Apêndice A deste trabalho.

O principal resultado deste trabalho é um protótipo de software funcional, desenvolvido na linguagem Rust, que atenda as funcionalidades definidas. Este protótipo aplicou a metodologia de análise híbrida proposta, combinando a velocidade da detecção por assinaturas com a capacidade preditiva do *Machine Learning*. A Figura 1 ilustra as principais funcionalidades do sistema.

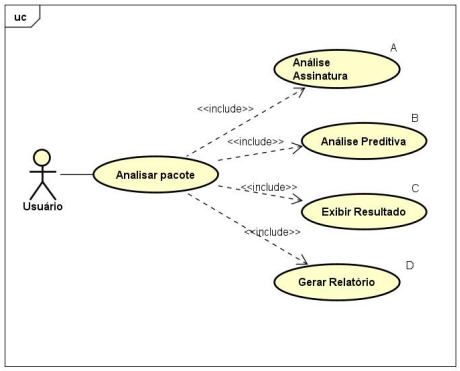

Figura 1: Diagrama de Casos de Uso do Sistema.

O fluxo começa quando o usuário inicia a aplicação via terminal e terá como principal capacidade a subsequente análise de pacotes Linux. O resultado final desta análise será entregue ao usuário por meio de dois artefatos principais:

- Um veredito imediato na saída padrão do terminal, informando se o pacote foi classificado como "Benigno" ou "Malicioso";
- Um relatório persistente em formato de arquivo de texto. Este relatório, gerado automaticamente ao final do processo, servirá como um registro detalhado da análise, contendo o veredito final e os indicadores de comprometimento encontrados.

Uma vez definidas as funcionalidades do sistema, pode-se visualizar os aspectos estruturais desta proposta na Figura 2, por meio de um Diagrama de Componentes. Nele, destacam-se os ambientes modulares do ecossistema de aprendizado de máquina Scikit-learn e o motor de varredura da aplicação.

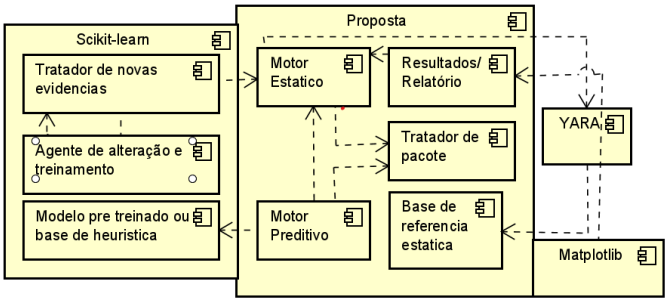

Figura 2: Diagrama de Componentes.

O componente “Tratador de pacote” é responsável por descompactar o conteúdo interno dos arquivos .deb em um diretório temporário. Os componentes “Motor Estático” e “Motor Preditivo”, que executam, respectivamente, a detecção estática baseada em assinaturas e a análise comportamental de ameaças, dependem do “Tratador de pacote” porque necessitam do conteúdo previamente extraído para aplicar suas varreduras.

Para operar, o “Motor Estático” depende estruturalmente da integração com a biblioteca externa YARA e com a “Base de referência estática” para a análise de assinaturas. Paralelamente, o “Motor Preditivo” depende estritamente do “Modelo pré-treinado” para classificar as heurísticas desconhecidas. O diagrama também ilustra um ciclo de retroalimentação, no qual ameaças confirmadas pelo “Motor Estático” são enviadas ao “Tratador de novas evidências” para enriquecer o treinamento contínuo da Inteligência Artificial. Por fim, os dados processados fluem para o componente de “Resultados/Relatório”, que depende da biblioteca externa Matplotlib para a consolidação visual das métricas de eficácia. Dessa forma, com esse diagrama, é possível compreender a arquitetura do sistema, a integração de recursos e suas dependências. Finalmente, registra-se que os componentes “Motor Estático”, “YARA” e “Base de referência estática” compõem a funcionalidade “Análise Assinatura”; os componentes “Motor Preditivo”, “Modelo pré treinado ou base de heurística”, “Tratador de pacote”, fazem parte da funcionalidade “Motor Preditivo”; assim como os componentes “Matplotlib”, “Resultado/Relatório” representam a funcionalidade “Exibir Resultados” e “Gerar Relatório” todas funcionalidades mostradas na Figura 1.

IV. Resultados de Implementação

Esta seção descreve a implementação dos módulos que concretizam as funcionalidades previstas no Diagrama de Casos de Uso (Figura 1): a análise de pacotes (A e B), a exibição de resultados no terminal (C) e a geração de relatório (D). Os códigos podem ser encontrados no GitHub[2] do autor.

Para que os motores de detecção possam analisar os pacotes suspeitos, foi implementado um módulo de extração em duas etapas (Figura 2), responsável por desconstruir o pacote original de forma segura. A lógica principal pode ser observada através de duas funções fundamentais: a `extrair_deb` e a `extrair_tar` da Figura 3.

Na Figura 3, o primeiro trecho (linhas 23 a 53) atua como o ponto de entrada da análise. O pacote Debian (`.deb`) é, estruturalmente, um arquivo do tipo `ar`. A função `extrair_deb` abre esse contêiner e itera sobre seus componentes internos buscando exclusivamente pela carga útil principal: o arquivo `data.tar`, o qual armazena os executáveis e bibliotecas do programa. Ao encontrá-lo, o código identifica o algoritmo de compressão utilizado e delega os bytes brutos para as funções decodificadoras correspondentes. Se o pacote não possuir essa estrutura esperada, a execução falha rapidamente.

[2]https://github.com/pedrobalen/tcc

```rust
23  pub fn extrair_deb(caminho: &Path, dir_pai: &Path) -> Result<PacoteExtraido, ExtratorErro> {
24      let dir_temp = tempfile::tempdir_in(dir_pai)?;
25      let arquivo = File::open(caminho)?;
26      let mut ar = ar::Archive::new(arquivo);
27
28      while let Some(entrada) = ar.next_entry() {
29          let mut entrada = entrada?;
30          let identificador = String::from_utf8_lossy(entrada.header().identifier())
31              .trim_end_matches('/')
32              .to_string();
33
34          if !identificador.starts_with("data.tar") {
35              continue;
36          }
37
38          let mut dados = Vec::new();
39          entrada.read_to_end(&mut dados)?;
40
41          let arquivos = if identificador.ends_with(".gz") {
42              descompactar_tar_gz(&dados, dir_temp.path())?
43          } else if identificador.ends_with(".xz") {
44              descompactar_tar_xz(&dados, dir_temp.path())?
45          } else {
46              return Err(ExtratorErro::CompressaoNaoSuportada(identificador));
47          };
48
49          return Ok(PacoteExtraido { dir_temp, arquivos });
50      }
51
52      Err(ExtratorErro::DataTarNaoEncontrado)
53  }
```

Figura 3: Função `extrair_deb` do Tratador de Pacotes.

A Figura 4, por sua vez, no trecho entre as linhas 65 a 87, realiza a extração final dos arquivos. A função `extrair_tar` lê os dados descompactados e os salva no diretório temporário. Para garantir a segurança da operação, a rotina ignora qualquer tipo de atalho. Isso protege contra ataques, impedindo que um pacote malicioso consiga escapar da pasta de análise e sobrescrever arquivos do sistema operacional. Por fim, a função retorna os caminhos de todos os arquivos extraídos com segurança.

```rust
65  fn extrair_tar<R: Read>(leitor: R, destino: &Path) -> Result<Vec<PathBuf>, ExtratorErro> {
66      let mut tar = tar::Archive::new(leitor);
67      let mut arquivos = Vec::new();
68
69      for entrada in tar.entries()? {
70          let mut entrada = entrada?;
71          let tipo = entrada.header().entry_type();
72
73          if tipo.is_symlink() || tipo.is_hard_link() {
74              continue;
75          }
76
77          if tipo.is_file() {
78              entrada.unpack_in(destino)?;
79              let caminho = destino.join(entrada.path()?);
80              arquivos.push(caminho);
81          } else if tipo.is_dir() {
82              entrada.unpack_in(destino)?;
83          }
84      }
85
86      Ok(arquivos)
87  }
```

Figura 4: Função `extrair_tar` do Tratador de Pacotes.

Após a extração dos pacotes, o Motor Estático está apto para executar a varredura. A Figura 5 ilustra a função `compilar_regras`, responsável por preparar o motor de análise estática. Ela percorre o diretório especificado, lê o conteúdo dos arquivos com extensão `.yar` e os compila em um único objeto de regras. Essa abordagem otimiza a performance do sistema, garantindo que a compilação ocorra apenas uma vez e o resultado possa ser reutilizado em múltiplas varreduras contínuas.

A Figura 6 detalha a função `varrer_arquivos`, que executa a inspeção dos binários contra as ameaças conhecidas. Utilizando o objeto de regras previamente compilado, a rotina analisa a lista de arquivos e, caso haja uma correspondência (*match*), extrai a *string* de severidade dos metadados da regra.

Depois que implementado o Motor Estático, podemos popular o arquivo base.yar com as regras de varredura. Cada

```rust
// Compila todas as regras .yar encontradas no diretorio informado e
// retorna o objeto Rules pronto para uso pelo Scanner. A compilacao e
// feita uma unica vez e o resultado pode ser reutilizado em multiplas
// varreduras, evitando retrabalho em modo benchmark.b
pub fn compilar_regras(diretorio: &Path) -> Result<yara_x::Rules, AssinaturasErro> {
    let mut compiler = yara_x::Compiler::new();

    for entrada in fs::read_dir(diretorio)? {
        let caminho = entrada?.path();
        if caminho.extension().is_some_and(|ext| ext == "yar") {
            let fonte = fs::read_to_string(&caminho)?;
            compiler
                .add_source(fonte.as_str())
                .map_err(|e| AssinaturasErro::Compilacao(e.to_string()))?;
        }
    }

    Ok(compiler.build())
}
```

Figura 5: Função `compilar_regras` do Motor Estático.

```rust
// Varre uma lista de arquivos extraidos contra as regras YARA compiladas.
// Para cada arquivo, instancia um Scanner, executa o scan e coleta as regras
// que casaram junto com seus metadados de severidade. Segue a logica Fast-Fail:
// retorna as deteccoes encontradas para que o chamador decida se deve abortar
// o pipeline antes do estagio de ML.
pub fn varrer_arquivos(
    regras: &yara_x::Rules,
    arquivos: &[PathBuf],
) -> Result<Vec<Deteccao>, AssinaturasErro> {
    let mut deteccoes = Vec::new();
    let mut scanner = yara_x::Scanner::new(regras);

    for arquivo in arquivos {
        let resultados = scanner.scan_file(arquivo)?;

        for regra in resultados.matching_rules() {
            let severidade = regra
                .metadata()
                .find(|(chave, _)| *chave == "severidade")
                .and_then(|(_, valor)| match valor {
                    yara_x::MetaValue::String(s) => Some(s.to_string()),
                    _ => None,
                })
                .unwrap_or_else(|| "desconhecida".to_string());

            deteccoes.push(Deteccao {
                regra: regra.identifier().to_string(),
                severidade,
                arquivo: arquivo.clone(),
            });
        }
    }

    Ok(deteccoes)
}
```

Figura 6: Função `varrer_arquivos` do Motor Estático.

regra é composta por três partes: metadados descritivos (nome, descrição e severidade), padrões a serem buscados (strings de texto, sequências hexadecimais ou expressões regulares) e uma condição lógica que define quando a regra deve disparar. O arquivo contém nove regras que cobrem categorias distintas de ameaças comuns em malware para Linux: conexões reversas (reverse shells), download e execução direta de scripts remotos, coleta de credenciais, mecanismos de persistência no sistema, binários com empacotadores, técnicas anti-análise em binários ELF, dados codificados em base64 e mineração de criptomoedas. Cada regra utiliza condições calibradas para reduzir falsos positivos, por exemplo, algumas exigem a presença simultânea de múltiplos indicadores em vez de um único padrão isolado, como exemplificado na Figura 7.

Após a aplicação das regras estáticas, o sistema também precisa transformar os arquivos analisados em dados numéricos que possam ser utilizados por um modelo de classificação (Figura 8). Para isso, é necessário examinar os arquivos extraídos, selecionar apenas aqueles que correspondem ao formato padrão de executáveis em sistemas Linux e coletar informações estruturais relevantes sobre cada um deles. Essa etapa é implementada no arquivo features.rs, que atua como uma ponte entre o Motor Estático e o Motor Preditivo.

Inicialmente, o arquivo verifica quais amostras correspondem a binários ELF, formato utilizado por executáveis no Linux (Figura 8). Em seguida, para cada binário identificado, são extraídas características capazes de representar o arquivo de maneira numérica. Essas informações são organizadas na estrutura `FeaturesElf` e posteriormente convertidas em um vetor compatível com o modelo preditivo.

```
rule crypto_miner
{
    meta:
        descricao = "Detecta indicadores de mineracao de criptomoedas"
        severidade = "critica"

    strings:
        $stratum    = "stratum+tcp://" ascii
        $xmrig      = "xmrig" ascii nocase
        $monero     = "monero" ascii nocase
        $pool       = "pool." ascii
        $wallet     = /[48][0-9AB][1-9A-HJ-NP-Za-km-z]{93}/ ascii
        $hashrate   = "hashrate" ascii nocase
        $mining     = "mining" ascii nocase

    condition:
        2 of them
}
```

Figura 7: Exemplo de regra YARA da Base de Referência Estática.

```rust
// Extrai as features de todos os binários ELF encontrados na lista de
// arquivos. Arquivos que não são ELF são silenciosamente ignorados, já
// que pacotes .deb contêm uma mistura de binários, scripts e configs.
pub fn extrair_features(arquivos: &[PathBuf]) -> Result<Vec<FeaturesElf>, FeaturesErro> {
    let mut resultados = Vec::new();

    for caminho in arquivos {
        let dados = fs::read(caminho)?;

        if let Some(elf) = parsear_elf(&dados) {
            resultados.push(FeaturesElf {
                arquivo: caminho.clone(),
                tamanho: dados.len() as u64,
                entropia: calcular_entropia(&dados),
                num_secoes: elf.section_headers.len(),
                num_importacoes: elf.dynsyms.iter().filter(|s| s.is_import()).count(),
            });
        }
    }

    Ok(resultados)
}
```

Figura 8: Função `extrair_feature` do Motor Preditivo.

Entre essas características, uma delas mede o grau de aleatoriedade dos dados presentes no arquivo. Valores elevados podem indicar conteúdo comprimido ou ofuscado, comportamento frequentemente associado a malwares empacotados. Essa medida é conhecida como entropia de Shannon, cuja implementação é apresentada na Figura 9.

Além da entropia, o sistema também considera a quantidade de divisões internas presentes no executável, utilizadas para organizar código, dados e metadados. Essas divisões são chamadas de seções ELF. Por fim, também é avaliada a quantidade de funções externas utilizadas pelo programa, isto é, chamadas a recursos fornecidos por bibliotecas dinâmicas do sistema. Esse atributo corresponde ao número de importações dinâmicas.

Dessa forma, as funções `extrair_features` (Figura 8) e `calcular_entropia` (Figura 9) transformam cada binário ELF analisado em um conjunto de quatro atributos principais: tamanho bruto, entropia de Shannon, número de seções e quantidade de importações dinâmicas. Esse conjunto representa numericamente o arquivo e permite que suas características sejam utilizadas pelo Motor Preditivo durante a etapa de classificação.

```rust
// Calcula a entropia de Shannon sobre os bytes brutos do arquivo.
// Valores próximos de 0 indicam conteúdo repetitivo (ex: seções zeradas),
// enquanto valores próximos de 8 indicam alta aleatoriedade, típica de
// dados cifrados, comprimidos ou ofuscados — característica comum em malware.
fn calcular_entropia(dados: &[u8]) -> f64 {
    if dados.is_empty() {
        return 0.0;
    }

    let mut frequencias = [0u64; 256];
    for &byte in dados {
        frequencias[byte as usize] += 1;
    }

    let total = dados.len() as f64;
    let mut entropia = 0.0;

    for &contagem in &frequencias {
        if contagem > 0 {
            let probabilidade = contagem as f64 / total;
            entropia -= probabilidade * probabilidade.log2();
        }
    }

    entropia
}
```

Figura 9: Função `calcular_entropia` do Motor Preditivo.

Quando o Motor Estático não identifica correspondências nas regras YARA, a análise é encaminhada para o Motor Preditivo. Para que o modelo treinado possa ser utilizado dentro da aplicação em Rust, ele é carregado a partir de um arquivo no formato ONNX (Figura 10), que permite transportar modelos criados em Python para diferentes ambientes de execução.

```rust
// Carrega o modelo ONNX do disco e cria uma sessão reutilizável.
// A sessão pode ser passada para múltiplas chamadas de `classificar`
// sem reinicializar o runtime, o que é eficiente em modo benchmark.
pub fn carregar_modelo(caminho: &Path) -> Result<Session, InferenciaErro> {
    Ok(Session::builder()?.commit_from_file(caminho)?)
}
```

Figura 10: Função `carregar_modelo` do Motor Preditivo.

Conforme apresentado na Figura 10, a função carregar modelo recebe o caminho do arquivo .onnx e cria uma sessão de inferência. Essa sessão é posteriormente reutilizada pela função de classificação, tornando o processo mais eficiente durante a análise de múltiplos binários. Após o carregamento do modelo, cada binário ELF é representado por um vetor contendo as características extraídas anteriormente. Esse vetor é convertido para o formato esperado pelo runtime onnx e enviado ao modelo para classificação. A função responsável por executar esse processo recebe o nome de classificar.

Na Figura 11, cada conjunto de características é convertido em um tensor bidimensional e enviado para a sessão ONNX, que retorna o rótulo previsto e as probabilidades associadas às classes possíveis. A classe 0 representa um arquivo benigno, enquanto a classe 1 representa um arquivo malicioso. A probabilidade correspondente à classe predita é armazenada como valor de confiança da inferência.

Dessa forma, o Motor Preditivo complementa a análise baseada em regras ao permitir a classificação de binários que não foram detectados diretamente pelo Motor Estático. Em vez de depender apenas de assinaturas conhecidas, essa etapa utiliza características estruturais e estatísticas dos arquivos para apoiar a tomada de decisão do sistema.

```rust
// Classifica cada binário ELF injetando seu vetor de features [tamanho,
// entropia, num_secoes, num_importacoes] na sessão ONNX e coletando a
// predição binária com a confiança associada.
//
// Assume exportação com zipmap=False no skl2onnx, de modo que
// "output_probability" seja um tensor float32 linearizado [n_amostras × n_classes].
pub fn classificar(
    sessao: &mut Session,
    features: &[FeaturesElf],
) -> Result<Vec<ResultadoInferencia>, InferenciaErro> {
    let mut resultados = Vec::new();

    for feat in features {
        let vetor = feat.como_vetor();
        // Tensor 2D [1, 4]: uma amostra com as quatro features ELF
        let tensor = Tensor::<f32>::from_array(([1usize, 4], vetor.to_vec()))?;
        let saidas = sessao.run(ort::inputs!["X" => tensor])?;

        // "label": int64 [n_amostras] — rótulo predito (0 ou 1)
        let (_, rotulos) = saidas["label"].try_extract_tensor::<i64>()?;
        let predicao = rotulos[0];

        // "probabilities": float32 [n_amostras × n_classes], linearizado.
        // Para uma amostra, índice 0 = P(benigno), índice 1 = P(malicioso).
        let (_, probs) = saidas["probabilities"].try_extract_tensor::<f32>()?;
        let confianca = probs[predicao as usize];

        resultados.push(ResultadoInferencia {
            arquivo: feat.arquivo.clone(),
            predicao,
            confianca,
        });
    }

    Ok(resultados)
}
```

Figura 11: Função `classificar` do Motor Preditivo.

Com todos os módulos de análise implementados, o componente “Resultados/Relatório” (Figura 2) é responsável por consolidar e apresentar ao usuário os vereditos produzidos pelos motores. O pipeline classifica o veredito final em três níveis: alerta vermelho, quando ao menos uma regra YARA identificou uma ameaça conhecida; alerta laranja, quando o modelo de *Machine Learning* classificou ao menos um binário como malicioso; e sinal verde, quando o pacote foi considerado limpo.

Na Figura 12, a função `executar_pipeline` conecta todos os módulos previamente descritos em um fluxo sequencial. Primeiramente, o pacote é extraído pelo Tratador de Pacote. Em seguida, o conteúdo é submetido à varredura YARA. Se houver correspondência, o pipeline é imediatamente interrompido e o resultado retorna como alerta vermelho, sem executar o estágio de *Machine Learning*. Caso nenhuma assinatura seja detectada, as *features* dos binários ELF são extraídas e submetidas ao modelo preditivo. Se ao menos um binário for classificado como malicioso, o resultado é alerta laranja; caso contrário, sinal verde.

Após a classificação, os resultados são exibidos diretamente no terminal de forma estruturada, diferenciando a saída conforme o nível de alerta: para ameaças por assinatura, são listadas as regras acionadas e suas severidades; para detecções heurísticas, são exibidos os binários suspeitos com o grau de confiança do modelo. O sistema também oferece um modo de varredura em lote, que analisa todos os pacotes de um diretório e exporta os resultados em um arquivo CSV, registrando o veredito, o tempo de execução e os detalhes de cada análise.

O componente “Agente de alteração e treinamento” (Figura 2), localizado dentro do ecossistema Scikit-learn, é responsável por transformar amostras brutas em dados numéricos e treinar o modelo de classificação. Sua implementação é dividida em dois *scripts* Python.

Na Figura 13, o *script* `extrator_features.py` atua como a contraparte em Python do módulo de extração de *features* implementado em Rust. A função `extrair_features` utiliza a biblioteca `pyelftools` para parsear cada binário ELF e coletar as características desejadas. Os resultados são

salvos em um CSV com a coluna adicional de rótulo, distinguindo amostras benignas de maliciosas.

```rust
fn executar_pipeline(
    alvo: &Path,
    regras: &Rules,
    sessao: &mut Session,
) -> anyhow::Result<RelatorioAnalise> {
    let dir_base = Path::new("temp");
    fs::create_dir_all(dir_base)?;
    let pacote = extrator::extrair_deb(alvo, dir_base)?;
    let especificacoes = pacote.especificacoes.clone();

    // Estágio 1 – varredura YARA (Fast-Fail)
    let deteccoes = assinaturas::varrer_arquivos(regras, &pacote.arquivos)?;
    if !deteccoes.is_empty() {
        return Ok(RelatorioAnalise {
            especificacoes,
            resultado: ResultadoPipeline::AlertaVermelho { deteccoes },
            features: Vec::new(),
            inferencias: Vec::new(),
        });
    }

    // Estágio 2 – extração de features ELF e inferência ML
    let features = features::extrair_features(&pacote.arquivos)?;
    if features.is_empty() {
        return Ok(RelatorioAnalise {
            especificacoes,
            resultado: ResultadoPipeline::SinalVerde { tem_elf: false },
            features,
            inferencias: Vec::new(),
        });
    }

    let resultados = inferencia::classificar(sessao, &features)?;
    let ameacas: Vec<_> = resultados
        .iter()
        .filter(|r| r.predicao == 1)
        .cloned()
        .collect();

    let resultado = if ameacas.is_empty() {
        ResultadoPipeline::SinalVerde { tem_elf: true }
    } else {
        ResultadoPipeline::AlertaLaranja { ameacas }
    };

    Ok(RelatorioAnalise {
        especificacoes,
        resultado,
        features,
        inferencias: resultados,
    })
}
```

Figura 12: Função `executar_pipeline` do Resultados/Relatório.

```python
def extrair_features(caminho: Path) -> list[float] | None:
    """Retorna [tamanho, entropia, num_secoes, num_importacoes] ou None."""
    dados = caminho.read_bytes()
    if dados[:4] != MAGIC_ELF:
        return None
    try:
        with caminho.open("rb") as f:
            elf = ELFFile(f)
            num_secoes = elf.num_sections()

            # Importacoes dinamicas: simbolos com shndx indefinido em .dynsym,
            # equivalente a goblin's is_import() no motor Rust.
            secao_dynsym = elf.get_section_by_name(".dynsym")
            num_importacoes = (
                sum(1 for s in secao_dynsym.iter_symbols() if s.entry["st_shndx"] == "SHN_UNDEF")
                if secao_dynsym
                else 0
            )

        return [
            float(len(dados)),
            calcular_entropia(dados),
            float(num_secoes),
            float(num_importacoes),
        ]
    except Exception:
        return None
```

Figura 13: Função `extrair_features` do Agente de Alteração e Treinamento.

O segundo *script*, `treinar.py` (Figura 14), executa o pipeline de treinamento em quatro etapas: carregamento e concatenação dos CSVs gerados pelo extrator, divisão em 80% treino e 20% teste com estratificação por classe, treinamento do modelo e avaliação com métricas de desempenho. A função `treinar_modelo` instancia um classificador *Random Forest* com 100 árvores de decisão e profundidade máxima de 10 níveis. A semente fixa assegura a reprodutibilidade dos resultados. Após o treinamento, o modelo é avaliado por meio de acurácia, precisão, *recall* e F1-Score.

```python
def treinar_modelo(X_treino, y_treino) -> RandomForestClassifier:
    modelo = RandomForestClassifier(
        n_estimators=N_ESTIMATORS,
        max_depth=MAX_DEPTH,
        random_state=RANDOM_STATE,
        n_jobs=-1,
    )
    modelo.fit(X_treino, y_treino)
    return modelo
```

Figura 14: Função `treinar_modelo` do Agente de Alteração e Treinamento.

O componente "Modelo pré-treinado ou base de heurística" (Figura 2) representa o artefato produzido pelo Agente de Alteração e Treinamento: o arquivo `random_forest.onnx`, consumido pelo Motor Preditivo em tempo de execução. Na Figura 15, a função `exportar_onnx` utiliza a biblioteca `skl2onnx` para converter o modelo *Random Forest* treinado em um formato portável. A entrada é definida como um tensor com as mesmas quatro *features* extraídas pelo scanner Rust, e a configuração de saída garante que as probabilidades sejam retornadas como tensor linearizado, formato compatível com o módulo de inferência previamente descrito. O componente Matplotlib, representado como dependência externa no diagrama de Componentes (Figura 2), é utilizado dentro do *script* de treinamento para a geração de relatórios gráficos que permitem avaliar visualmente o desempenho do modelo. São gerados dois artefatos principais: uma matriz de confusão, que permite visualizar a distribuição de verdadeiros positivos, verdadeiros negativos, falsos positivos e falsos negativos; e um gráfico de importância relativa das *features*, que quantifica a contribuição de cada atributo na decisão de classificação. Ambos os gráficos são analisados na seção de Resultados e Discussões. O componente "Tratador de novas evidências" (Figura 2), ilustrado dentro do ecossistema Scikit-learn, representa o ciclo de retroalimentação entre o "Motor Estático" e o pipeline de treinamento. Quando o "Motor Estático" identifica uma ameaça por assinatura, o binário confirmado como malicioso constitui uma nova evidência rotulada que pode ser incorporada ao conjunto de dados de treinamento. Essa amostra é então submetida ao "Agente de Alteração e Treinamento", que extrai suas *features* e retreina o modelo *Random Forest* com o *dataset* enriquecido. Dessa forma, ameaças conhecidas detectadas por regras YARA fortalecem progressivamente a capacidade do "Motor Preditivo" de reconhecer ameaças desconhecidas, estabelecendo um processo iterativo de melhoria contínua do sistema.

```python
def exportar_onnx(modelo: RandomForestClassifier) -> None:
    """Exporta para ONNX com entrada 'X' float32 [None, 4] e probabilidades
    como tensor float32 (zipmap=False), compativel com inferencia.rs."""
    MODELO_DIR.mkdir(exist_ok=True)

    initial_types = [("X", FloatTensorType([None, 4]))]
    options = {RandomForestClassifier: {"zipmap": False}}

    modelo_onnx = convert_sklearn(
        modelo,
        initial_types=initial_types,
        options=options,
        target_opset=15,
    )

    caminho = MODELO_DIR / "random_forest.onnx"
    caminho.write_bytes(modelo_onnx.SerializeToString())
    print(f"  Modelo ONNX salvo em {caminho}")
```

Figura 15: Função `exportar_onnx` do Modelo Pré-treinado.

V. Resultados e Discussões

Esta seção apresenta os resultados obtidos para cada fase do sistema, avaliados com as métricas e o conjunto de dados definidos na metodologia.

A. *Cenário 1: Avaliação da Fase 2 — Motor Preditivo*

O primeiro cenário avalia o desempenho da Fase 2 da implementação: o Motor Preditivo baseado em *Machine Learning*.

O classificador *Random Forest* foi treinado com 100 árvores de decisão e profundidade máxima de 10 níveis. Os resultados, mensurados pelas métricas de taxa de detecção e taxa de falsos positivos definidas na metodologia, são apresentados a seguir.

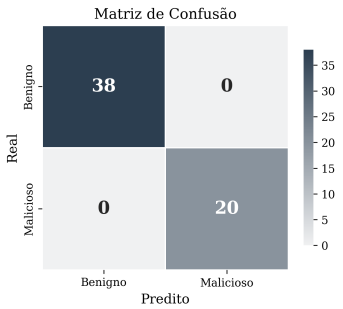

Figura 16: Matriz de confusão do Motor Preditivo.

A matriz de confusão (Figura 16) confirma que o modelo classificou corretamente todas as 58 amostras[3] do conjunto de teste: 38 verdadeiros negativos e 20 verdadeiros positivos, sem registrar falsos positivos ou falsos negativos. A taxa de detecção para a classe maliciosa foi de 100%, e a taxa de falsos positivos foi de 0%.

A análise da importância relativa das *features* (Figura 17) revela que o número de seções (47,1%) e a quantidade de importações dinâmicas (41,8%) respondem juntas por 88,9% da capacidade de classificação do modelo. A entropia contribui com 9,6% e o tamanho do arquivo com apenas 1,5%. Conforme demonstrado na Figura 18, a classificação em tempo real de um binário ELF pelo Motor Preditivo exibe as *features* extraídas e a confiança atribuída pelo modelo à predição.

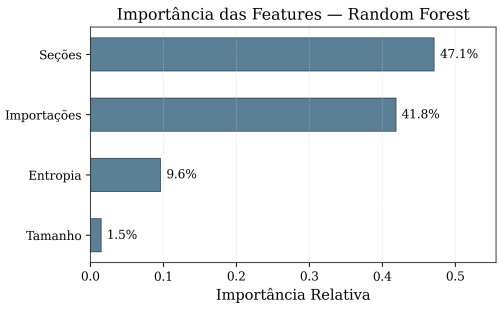

Figura 17: Importância relativa das *features* no modelo *Random Forest*.

[3]A divisão dos conjuntos de treinamento e teste foi realizada de forma aleatória. Destaca-se que as amostras provêm de repositórios mantidos por instituições oficiais, o que assegura a qualidade dos dados.

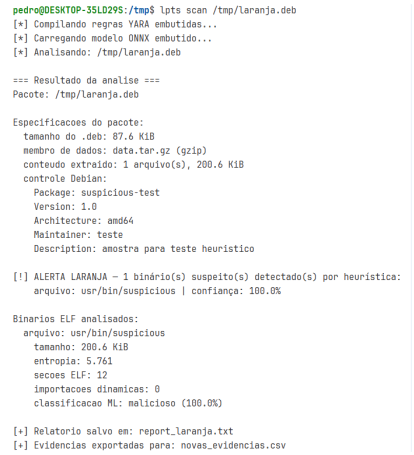

Figura 18: Saída do sistema para um pacote com alerta laranja (heurística ML).

B. Cenário 2: Avaliação da Fase 1 — Motor Estático

O segundo cenário avalia os resultados da Fase 1 da implementação: o Motor Estático baseado em regras YARA. A Tabela I sumariza as nove regras implementadas, suas severidades e condições de disparo. A estratégia de calibração adotada é coerente com as conclusões de Lockett [3], que obteve taxa de detecção geral de 57% com regras YARA, reforçando que a eficácia desta abordagem depende diretamente da qualidade e especificidade das regras definidas.

Tabela I: Regras YARA implementadas no Motor Estático

Regra	**Severidade**	**Condição**
`reverse_shell`	Crítica	Qualquer padrão
`download_and_execute`	Crítica	Download + pipe
`credential_harvesting`	Alta	2 ou mais padrões
`persistence_mechanism`	Alta	Qualquer padrão
`upx_packed_elf`	Média	ELF + assinatura UPX
`suspicious_elf_techniques`	Alta	ELF + 3 técnicas
`base64_payload`	Média	Qualquer padrão
`network_exfiltration`	Alta	IP + rede + supressão
`crypto_miner`	Crítica	2 indicadores

Conforme demonstrado na Figura 19, a execução real do Motor Estático sobre uma amostra maliciosa exibe a regra acionada, sua severidade e o arquivo responsável pela correspondência, validando o funcionamento do estágio de detecção por assinatura.

C. Discussão

O modelo atingiu bons resultados no conjunto de teste avaliado. Entretanto, esse resultado deve ser interpretado com cautela. O *dataset* é relativamente pequeno (289 amostras, das quais apenas 58 no conjunto de teste), e a separação entre as classes é bastante nítida. Isso sugere que as amostras maliciosas utilizadas seguem um perfil predominante binários empacotados ou ofuscados e podem não refletir toda a diversidade de ameaças encontradas em cenários reais.

```
pedro@DESKTOP-35LD29S:/tmp$ lpts scan /tmp/vermelho.deb
[*] Compilando regras YARA embutidas...
[*] Carregando modelo ONNX embutido...
[*] Analisando: /tmp/vermelho.deb

=== Resultado da analise ===
Pacote: /tmp/vermelho.deb

Especificacoes do pacote:
  tamanho do .deb: 6.5 MiB
  membro de dados: data.tar.gz (gzip)
  conteudo extraido: 1 arquivo(s), 14.1 MiB
  controle Debian:
    Package: malware-test
    Version: 1.0
    Architecture: amd64
    Maintainer: teste
    Description: amostra maliciosa para teste

[!] ALERTA VERMELHO — 1 ameaça(s) detectada(s) por assinatura:
    regra: crypto_miner | Detecta indicadores de mineracao de criptomoedas | severidade: critica | arquivo: usr/bin/miner
```

Figura 19: Saída do sistema para um pacote com alerta vermelho (assinatura YARA).

Apesar dessas limitações, os resultados são consistentes com a fundamentação teórica do trabalho. Raffa, Sgandurra e O'Keeffe [1] demonstram que as bases de assinaturas de antivírus Linux não são bem mantidas, com um terço dos antivírus testados no VirusTotal atingindo taxa de detecção de no máximo 30%. Nesse cenário, a complementação por análise heurística representa uma estratégia relevante para cobrir a lacuna da detecção exclusivamente por assinaturas, como também recomendado por Aslan e Samet [2].

A arquitetura híbrida do sistema oferece duas vantagens: a confiabilidade da detecção por assinatura para ameaças conhecidas e a capacidade de generalização do *Machine Learning* para padrões desconhecidos. Além disso, o ciclo de retroalimentação, em que ameaças confirmadas pelo Motor Estático enriquecem o *dataset* de treinamento, tende a fortalecer o Motor Preditivo ao longo do tempo. A Figura 20 demonstra a execução sobre um pacote legítimo, resultando em sinal verde.

```
pedro@DESKTOP-35LD29S:/tmp$   lpts scan /tmp/verde.deb
[*] Compilando regras YARA embutidas...
[*] Carregando modelo ONNX embutido...
[*] Analisando: /tmp/verde.deb

=== Resultado da analise ===
Pacote: /tmp/verde.deb

Especificacoes do pacote:
  tamanho do .deb: 329.9 KiB
  membro de dados: data.tar.gz (gzip)
  conteudo extraido: 1 arquivo(s), 754.7 KiB
  controle Debian:
    Package: iproute2-fake
    Version: 1.0
    Architecture: amd64
    Maintainer: teste
    Description: pacote de teste benigno

[+] SINAL VERDE — /tmp/verde.deb está aparentemente limpo.

Binarios ELF analisados:
  arquivo: usr/bin/ip
    tamanho: 754.7 KiB
    entropia: 6.180
    secoes ELF: 31
    importacoes dinamicas: 201
    classificacao ML: benigno (100.0%)

[+] Relatorio salvo em: report_verde.txt
```

Figura 20: Saída do sistema para um pacote classificado como limpo.

VI. Conclusões

A literatura demonstra que os antivírus para Linux possuem bases de assinaturas mal mantidas e que a detecção exclusivamente por assinaturas é incapaz de lidar com ameaças de Dia Zero. A complementação com *Machine Learning* apresenta-se como estratégia promissora para cobrir essa lacuna, enquanto as regras YARA se destacam como a técnica mais eficaz de detecção estática, conforme apresentado e discutido.

Este trabalho alcançou seu objetivo geral ao entregar um protótipo funcional que integra ambas as abordagens: estática e preditiva. O Motor Estático, com nove regras YARA, identifica ameaças conhecidas, e o Motor Preditivo, baseado em *Random Forest* com quatro *features* de binários ELF, classifica amostras desconhecidas alcançando 100% de acurácia no conjunto de teste de 58 amostras, sem falsos positivos. Esse resultado, porém, deve ser interpretado com cautela devido ao tamanho reduzido do *dataset* (289 amostras). O protótipo atende às funcionalidades e aos requisitos identificados: analisa pacotes `.deb`; exibe o veredito em três níveis de alerta; e gera relatórios em formato texto.

Como trabalhos futuros, recomenda-se a ampliação do *dataset* com amostras mais diversas, a inclusão de novas *features* como chamadas de sistema e grafos de fluxo de controle, a extensão do suporte a outros formatos de pacote (`.rpm`, `.pkg.tar`) e a exploração do ciclo de retroalimentação já previsto na arquitetura para melhoria contínua do modelo.

Referências

[1] G. Raffa, D. Sgandurra e D. O'Keeffe. "A Comprehensive Evaluation of Modern Linux Antiviruses". Em: *2022 IEEE International Conference on Big Data (Big Data)*. Osaka, Japan, 2022. DOI: 10.1109/BigData55660.2022.10020475.

[2] Ö. Aslan e R. Samet. "A Comprehensive Review on Malware Detection Approaches". Em: *IEEE Access* 8 (2020), pp. 1109–1141. DOI: 10.1109/ACCESS.2019.2963724.

[3] A. Lockett. *Assessing the Effectiveness of YARA Rules for Signature-Based Malware Detection and Classification*. 2021.

[4] Ashish Jaiswal. "Linux—the Operating System". Em: *Journal of Advances in Shell Programming* (). ISSN: 2395-6690. DOI: 10.37591/JoASP.

[5] Marco Gherardi et al. "Hard and soft bounds in the evolution of Ubuntu packages. A lesson for species body masses?" Em: *arXiv preprint arXiv:1303.0011* (2013). arXiv:1303.0011 [physics.soc-ph]. URL: https://arxiv.org/abs/1303.0011.

[6] Scikit-learn Developers. *Scikit-learn: Machine Learning in Python*. https://scikit-learn.org. Acesso em: 24 mar. 2026. 2025.

[7] The Matplotlib Development Team. *Matplotlib: Visualization with Python*. https://matplotlib.org/. Acesso em: 24 mar. 2026. 2025.

[8] W. Bugden e A. Alahmar. *Rust: The Programming Language for Safety and Performance," in 2. Uluslararası Lisansüstü Çalışmalar Kongresi*. 2nd International Graduate Studies Congress (IGSCONG'22), 2022.

VII. Apêndice

A. Requisitos Funcionais

RF1: **Entrada de Pacotes** – Por meio do terminal, o usuário escreve o nome da aplicação e então como parâmetro irá passar o diretório o qual contém o pacote no sistema.

RF2: **Análise por Leitura do Pacote** – O software deve ter capacidade de avaliar o pacote introduzido pelo usuário (Figura 1.A e B), avaliando se o pacote possui natureza benigna ou maligna.

RF3: **Exibição de Resultados no Terminal** – A aplicação deve reportar o veredito da análise diretamente na tela do terminal. A exibição deve incluir o resultado final (benigno/malicioso) e, em caso de detecção, a lista de arquivos suspeitos e as heurísticas ou regras YARA acionadas (Figura 1.C).

RF4: **Geração de Relatório** – Ao concluir a análise, a aplicação deve gerar um arquivo de relatório (Figura 1.D) em formato texto detalhando os resultados da varredura.

B. Apêndice - Requisitos Não Funcionais

RNF1: **Leveza** – O sistema deve ser projetado para ser leve e eficiente, garantindo baixo consumo de recursos computacionais durante sua execução. Essa abordagem permite que a aplicação seja executada rapidamente em diferentes ambientes Linux, ocupando pouco espaço em disco e mantendo o desempenho mesmo em máquinas com configurações modestas.

RNF2: **Desempenho** – O sistema deve ser desenvolvido com foco em alto desempenho, garantindo rapidez na execução das análises e na geração dos relatórios. A utilização da linguagem Rust contribui diretamente para essa característica, por oferecer compilação nativa e otimizações de baixo nível que resultam em uma aplicação eficiente e responsiva.

RNF3: **Segurança** – A própria ferramenta de análise não deve introduzir riscos de segurança. A escolha da linguagem Rust suporta este requisito, pois ela é projetada para prevenir automaticamente vulnerabilidades comuns de gerenciamento de memória, que são uma causa principal de falhas de segurança em outros sistemas.

RNF4: **Usabilidade** – Sendo uma ferramenta de linha de comando, sua operação deve ser simples e intuitiva. Os comandos para iniciar uma varredura devem ser diretos e a saída no console deve ser clara, concisa e legível.

RNF5: **Manutenibilidade** – O código-fonte da aplicação deve ser bem documentado e seguir os padrões da linguagem Rust.

RNF6: **Escalabilidade** – A arquitetura do software deve ser projetada de forma modular. Isso garante que, no futuro, seja possível adicionar novas funcionalidades, como diferentes métodos de análise ou novos formatos de relatório.