Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Explorando o uso de análise estática e aprendizagem supervisionada de máquina para a identificação de códigos maliciosos em arquivos executáveis do sistema operacional Microsoft Windows

Alexandre J. Ribeiro

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Orientador
Prof. Dr. Rodrigo Bonifácio de Almeida

Brasília
2020

Ficha catalográfica elaborada automaticamente,
com os dados fornecidos pelo(a) autor(a)

RR484e

Ribeiro, Alexandre José
Explorando o uso de análise estática e aprendizagem supervisionada de máquina para a identificação de códigos maliciosos em arquivos executáveis do sistema operacional Microsoft Windows / Alexandre José Ribeiro; orientador Rodrigo Bonifácio Almeida. -- Brasília, 2020.
131 p.

Dissertação (Mestrado - Mestrado Profissional em Computação Aplicada) -- Universidade de Brasília, 2020.

1. Engenharia de Software. 2. Análise Estática de Malware. 3. Aprendizagem de Maquina. 4. Segurança de Computadores. I. Almeida, Rodrigo Bonifácio , orient. II. Título.

Universidade de Brasília
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Explorando o uso de análise estática e aprendizagem supervisionada de máquina para a identificação de códigos maliciosos em arquivos executáveis do sistema operacional Microsoft Windows

Alexandre J. Ribeiro

Dissertação apresentada como requisito parcial para conclusão do
Mestrado Profissional em Computação Aplicada

Prof. Dr. Rodrigo Bonifácio de Almeida (Orientador)
CIC/UnB

Prof. Dr. Rafael Timóteo de Sousa Júnior	Dr. André Luiz Peron Martins Lanna
ENE/UnB	FGA/UnB

Prof. Dr. Marcelo Ladeira
Coordenador do Programa de Pós-graduação em Computação Aplicada

Brasília, 10 de dezembro de 2020

Dedicatória

Este trabalho é dedicado a Deus, que tem sido meu esteio, dando-me a vida, saúde e paz, por meio de sua infinita graça e misericórdia. A ele, toda honra e glória, agora e para todo o sempre. Dedico também à minha maravilhosa e extraordinária família (esposa Janaína e filha Isabela), que me apoiou em todos os momentos desta caminhada.

Agradecimentos

Os agradecimentos principais são direcionados:

- Inicialmente, a Deus, nosso pai, senhor e criador, que me deu força e persistência durante toda a caminhada desse trabalho. A ele toda honra, glória e louvor.
- A minha esposa e filha, pela paciência, compreensão e irrestrito apoio, sem os quais, esse trabalho não seria possível.
- Aos meus pais João e Terezinha, pela base de valores que me sustenta.
- Ao meu orientador Prof. Dr. Rodrigo Bonifácio, por me guiar rumo a conclusão dessa pesquisa. Suas contribuições vão muito além da paciência, compreensão e orientação, mas sim como um farol a iluminar o rumo para conclusão dessa jornada. A ele, minha eterna gratidão.
- Aos Profs. Drs. Rafael Timóteo e André Lanna, componentes da banca de avaliação, por doarem seu tempo na avaliação deste trabalho.
- A todos os professores do PPCA, pelos ensinamentos fundamentais e base dessa caminhada.
- Aos Profs. Drs. Vinícius Borges e Fabiano Fernandes, por doarem seu tempo e compartilhar seu amplo conhecimento no aprendizado de máquina, os quais foram fundamentais à consecução desta pesquisa.
- Ao Sr. Chefe do CDS, Gen. Wolski, que me autorizou a dedicar parte do meu tempo de expediente de trabalho para participação nas aulas e na elaboração da pesquisa, e ao meu chefe imediato, Cel Castaño, pelo apoio e confiança irrestritos.
- A minha amiga Ana Cristina Fernandes Lima, pelo seu apoio, sugestões e contribuições na confecção dessa dissertação.
- A todos os amigos, em especial, Borges, Camargo, Gabriel, Yuri e Matheus, pelo apoio e contribuições na pesquisa, os quais foram a base para a conclusão e êxito desse trabalho.

- Ao projeto VirusShare.com[1], por manter e disponibilizar à comunidade um repositório de arquivos maliciosos, parte dos quais, foram utilizados na presente pesquisa.
- Ao serviço de análise *online* de códigos maliciosos Virustotal[2], o qual foi utilizado nesta pesquisa validar resultados.
- Ao serviço de análise *online* de códigos maliciosos Malshare[3], o qual foi utilizado nesta pesquisa para validar resultados.

[1]https://virusshare.com/
[2]https://www.virustotal.com
[3]https://www.malshare.com

Resumo

Malware tornou-se uma grande ameaça para governos, empresas e indivíduos. A forma clássica para a detecção de *malwares* é pela utilização de softwares como antivírus. No entanto, os produtos que oferecem esse tipo de contramedida estão se tornando cada vez mais ineficazes, devido ao surgimento de técnicas de evasão, tais como o polimorfismo, o que permite que centenas de milhares de exemplares surjam todos os dias. Para lidar com essa ameaça, métodos de aprendizado de máquina (ML) têm sido reportados como ferramentas promissoras na detecção de *malware*. Neste contexto, o presente trabalho explora técnicas de aprendizado supervisionado de máquina de maneira a produzir uma generalização prática de um preditor para aplicação em um determinado sistema de detecção de *malware*. Para tanto, utiliza atributos extraídos de arquivos executáveis do ambiente *Microsoft Windows*, também chamados de *Portable Executable* (PE), através da aplicação de ferramentas de análise estática, de código aberto, desenvolvidas na linguagem *Python*, e procedimentos relacionados à seleção de amostras, a coleta e tratamento dos dados coletados de repositórios disponíveis na Internet de aplicativos do sistema operacional *Microsoft Windows*. Dos algoritmos que compuseram os experimentos, o *Random Forest*, KNN e SVC apresentaram o melhor desempenho dentre aqueles utilizados. Como resultados, os experimentos atingiram uma precisão acima de 94% durante o treinamento dos modelos. Como contribuição, o trabalho proposto fornece uma evidência empírica da viabilidade da proposta, baseada nos experimentos realizados, cujos resultados foram embarcados em um protótipo de aplicação para a classificação entre arquivos maliciosos e benignos. Adicionalmente, oferece à comunidade científica, um conjunto de recursos extraídos de mais de 14.000 arquivos executáveis, entre arquivos maliciosos e benignos, como suporte para outros experimentos.

Palavras-chave: *malware*, análise estática, *python*, aprendizado supervisionado de máquina

Abstract

Malware has become a major threat to governments, businesses and individuals. The classic way to detect malwares is by using software such as antivirus. However, products that offer this type of protection against it are becoming increasingly ineffective, due to the emergence of advanced techniques, such as polymorphism, which allows hundreds of thousands of samples to appear every day. To address this threat, machine learning methods (ML) have been reported as promising tools for detecting malware. In this context, the present work explores supervised machine learning techniques in order to produce a practical generalization of a predictor for application in a given malware detection system. Our approach uses resources extracted from executable files in the Microsoft Windows environment, also called Portable Executables (PE), through the application of static analysis tools and open source codes, developed in language Python and procedures related to the samples' selection, the collection and treatment of data taken from repositories available on the Internet, from the Microsoft Windows operational system applications. Among the algorithms that composed the experiments, Random Forest, KNN and SVC showed the best performance among those used. As a result, the experiments reached accuracy above 94% during the training phase. As a contribution, the proposed work also provides empirical evidence based on experiments carried out, and by the construction of an application prototype for prediction between malicious and benign files. In addition, it offers to the scientific community a set of resources extracted from more than 14,000 samples, including malicious and benign files, and its related code as well, as support for other experiments.

Keywords: malware, static analysis, python, supervised machine learning

Sumário

1 Introdução **1**
1.1 Motivação 2
1.2 Definição do Problema 2
1.3 Objetivo 3
1.4 Justificativa 4
1.5 Contribuições 5
1.6 Metodologia 6
1.7 Organização do Trabalho 6

2 Fundamentação Teórica **8**
2.1 Definição de *Malware* 8
2.1.1 Classificação dos *Malware* 9
2.2 *Malwares* em Dispositivos Móveis 12
2.2.1 *Malwares* em Android 12
2.2.2 *Malwares* em IOS 12
2.3 Técnicas de Detecção de *Malwares* 13
2.3.1 Antivírus 13
2.3.2 Métodos de Análise 13
2.3.3 Ambientes de Análise 14
2.3.4 Técnicas de Evasão 18
2.4 Aprendizado de Máquina 20
2.4.1 Algoritmos de Classificação 21
2.4.2 Redução da Dimensionalidade 26
2.4.3 Mineração de Textos 27
2.5 Considerações Finais 28

3 Revisão Bibliográfica **29**
3.1 Objetivos da Revisão 29

3.2 Etapas da Revisão 30
3.2.1 1ª Etapa - Preparação da Pesquisa 30
3.2.2 2ª Etapa - Apresentação e Inter-relação dos Dados 30
3.2.3 3ª Etapa - Análise dos dados Coletados 33
3.3 Outros Trabalhos Relacionados à Pesquisa 35
3.4 Considerações Finais 38

4 Procedimentos **40**
4.1 Abordagem Proposta 40
4.2 Sequência de Atividades 41
4.2.1 Coleta dos Dados 42
4.2.2 Processamento dos Dados 43
4.2.3 Tratamento dos Dados 53
4.2.4 Treinamento e Avaliação dos Modelos 56
4.2.5 Funções para Medição da Performance 60
4.2.6 Seleção do Melhor Modelo 61
4.3 A Estrutura dos Arquivos Analisados 62
4.4 Principais Ferramentas Utilizadas 63
4.4.1 A Linguagem Python 64
4.4.2 Ferramentas Construídas e Customizadas 65
4.5 Considerações Finais 70

5 Experimentos **71**
5.1 Implementação 71
5.1.1 Ambiente de Experimentação 72
5.1.2 Experimento 1 - Modelo Inicial 72
5.1.3 Experimento 2 - PCA 74
5.1.4 Experimento 3 -TF-IDF 76
5.1.5 Persistência do Modelo 76
5.2 Avaliação dos Experimentos 77
5.3 Análise dos Resultados 80
5.3.1 Avaliação dos Modelos com Dados não Treinados 81
5.3.2 Validação com o *Dataset Clamp* 82
5.3.3 Discussão 84
5.4 Limitações da Pesquisa 85
5.5 Considerações Finais 86

6 Implementação do Protótipo **87**
6.1 Construção do Protótipo . . . 87
6.1.1 Principais Funcionalidades . . . 89
6.2 Componentes da Arquitetura . . . 90
6.3 Construção do Julgador . . . 94
6.4 Considerações Finais . . . 97

7 Conclusões e Trabalhos Futuros **98**

Referências **102**

Apêndice **113**

A Ferramenta de Extração dos Atributos **114**
A.1 Parser.py . . . 114

B Ferramenta de Identificação e Separação de Arquivos pelo Tipo **120**
B.1 Separador.py . . . 120
B.2 Persistência dos Modelos . . . 121

C Treinamento dos Modelos - Jupyter Notebook **122**
C.1 Código do Experimento 1 . . . 126
C.2 Código do Experimento 2 - PCA . . . 127
C.3 Experimento 3 -TF-IDF . . . 129

Lista de Figuras

2.1 Diferença entre VM e Container. Fonte: docker.com. 18
2.2 Principais abordagens de detecção de *malware* com aprendizado de máquina [1]. 21
2.3 Gráfico do SVM que mostra o hiperplano que separa um conjunto de dados em duas classes [2]. 23
2.4 Gráfico de árvore de decisão [3]. 25

3.1 Publicações por países. Fonte: *Scopus Elsevier* 31
3.2 Ocorrências de palavras. Fonte: https://tagcrowd.com/ 32
3.3 Autores com maior número de citações. Fonte: *Scopus Elsevier* 33

4.1 Diagrama de visão geral. Fonte: Adaptado pelo autor 41
4.2 Principais atividades. Fonte: o autor 42
4.3 PE *Magic Number*. 44
4.4 Frequência das Chamadas de Função. 47
4.5 Distribuição da entropia. 48
4.6 histograma do tamanho dos arquivos em bytes. 49
4.7 Presença de empacotadores. 50
4.8 Histograma que mostra a frequência de URLs. 51
4.9 Exemplo de Cross-validation [2]. 56
4.10 Matriz que mostra a correlação entre os atributos. 60
4.11 Estrutura dos arquivos PE [4]. 63
4.12 Estrutura básica do arquivo *PE* [5]. 65
4.13 Exemplo de extração de um binário com o peframe.py. 67
4.14 Exemplo de extração de um binário com o pecheck.py. Fonte: o autor. . . . 68
4.15 Exemplo da extração do binário com o PEScanner.py. Fonte: o autor. . . . 69
4.16 Extrato do *Dataset* resultante do processo de extração. Fonte: o autor. . . 69

5.1 Matriz de confusão que mostra o resultado da fase de treinamento do experimento 1 73

5.2 Variância dos dados em função da aplicação do PCA. 74
5.3 Resultados intermediários do Grid Search 75
5.4 Exemplo de predição com arquivo joblib 77
5.5 Boxplot com a acurácia média dos algoritmos no experimento 1 78
5.6 Boxplot com a acurácia média dos algoritmos no experimento com o PCA . 79
5.7 Boxplot com a acurácia média dos algoritmos no experimento o TF-IDF . . 80
5.8 Gráfico da Curva ROC - resultados 82
5.9 Gráfico de barras com os tempos gastos para treinamento dos algoritmos com melhor performance. 83
5.10 Gráfico resultante da aplicação do PCA no dataset Clamp 83

6.1 Principais ferramentas do protótipo. Fonte: o autor 88
6.2 Diagrama de visão geral. Fonte: o autor 90
6.3 Tela de submissão de arquivos binários. Fonte: o autor 91
6.4 Tela de resultados. Fonte: o autor. 93
6.5 Relatório do Virustotal dos arquivos binários. Fonte: Virustotal. 94
6.6 Bloxplot comparativo da acurácia entre os modelos e o *Voting Classifier* (VC). Fonte: o autor. 95
6.7 Diagrama de construção do modelo julgador. Fonte: o autor. 96

Lista de Tabelas

3.1 Total de publicações nas bases pesquisadas 31
3.2 Quadro resumo dos trabalhos relacionados 36

4.1 *Datasets* utilizados. 43
4.2 Conjunto de atributos selecionados. 46
4.3 Matriz de confusão para uma classificação binária. 59

5.1 Resultados do experimento 1 - valores médios 78
5.2 Resultados do experimento 2 - PCA 79
5.3 Resultados do experimento 3 - TF-IDF 80
5.4 Quadro resumo - acurácia dos modelos no treinamento 81
5.5 Quadro resumo da avaliação final dos modelos 81
5.6 Avaliação dos modelos 82
5.7 Avaliação com o dataset Clamp 84

6.1 Extrato de resultados do julgador 97

Lista de Abreviaturas e Siglas

API Application Program Interface.

AUC Area Under Curve.

AV Antivirus.

CDS Centro de Desenvolvimento de Sistemas.

CLI Command Line Interface.

CSV Comma-separated values files.

CTIR Gov Centro de Tratamento e Resposta a Incidentes Cibernéticos do Governo Federal.

CV Cross-Validation.

DDoS Distributed Denial of Service.

DL Deep Learning.

DLL Dynamic Link Library.

EB Exército Brasileiro.

IDS Intrusion Detection System.

IRC Internet Relay Chat.

JVM Java Virtual Machine.

KNN K-Nearest Neighbors.

ML Machine Learning.

NB Naive Bayes.

NIST National Institute of Standards and Technology.

PC Principal Component.

PCA Principal Component Analysis.

PCAP Packet Capture of Network traffic.

PE Portable Executables.

ROC Receiver Operating Characteristic.

RQ Research Question.

SO Sistema Operacional.

SVC C-Support Vector Classification.

TEMAC Teoria do Enfoque Meta Analítico.

URL Uniform Resource Locator.

US-CERT Cybersecurity and Infrastructure Security Agency (CISA) of United States of America.

VM Virtual Machine.

Capítulo 1

Introdução

A segurança em redes de computadores representa um grande desafio para governos e organizações na atualidade. Uma das mais sérias ameaças tem origem nos *malwares*, termo genérico que caracteriza diferentes tipos de códigos maliciosos [6]. Apesar do significativo aumento da eficiência dos mecanismos de segurança e de sua contínua evolução, códigos maliciosos continuam a se propagar pelas redes de computadores. Os objetivos são os mais variados, incluindo a evasão de dados, sequestro de informações pessoais e corporativas, ataques a sistemas industriais e espionagem (*spyware crime*).

Recentemente, os códigos maliciosos tornaram-se também uma arma de guerra entre nações, ou *cyberweapons*[1], com destaque para o mundialmente conhecido "*Stuxnet*", considerado na literatura como a primeira arma cibernética usada efetivamente contra uma nação [7]. Em 2017 ocorreu um exemplo da capacidade devastadora desses softwares, quando um ataque de proporções globais afetou diversas redes e sistemas computacionais ao redor do mundo, identificados como *malware* "*WannaCry*" e "*Petrwrape/Petya*", ambos classificados na categoria de *Ransomware*[2]. Nesses ataques, de acordo com o US-CERT [9], foram registradas dezenas de milhares de infecções em cerca de 150 países, incluindo os Estados Unidos, o Reino Unido, a Espanha, a Rússia, a França e Japão.

Destacam-se também as *botnets*, redes de computadores infectados por *malware*, que caracterizam-se por serem controlados remotamente e usados para ataques de Negação de Serviço Distribuído (DDoS), dentre outros [10].

[1] armas cibernéticas

[2] tipo de *malware*. Segundo o CTIR Gov [8], os *Ransomwares* geralmente encriptam ou bloqueiam os dados do sistema infectado, sendo comumente utilizados para extorsão

1.1 Motivação

Para enfrentar tais ameaças, o Estado brasileiro, com base na Estratégia Nacional de Defesa [11], e na Diretriz Ministerial nº 14/2009 [12], dentre outras medidas, atribuiu ao Exército Brasileiro (EB) a estruturação do setor de defesa cibernética do Brasil.

Segundo Mangialardo [11], a defesa cibernética pode ser definida como um "conjunto de ações defensivas, exploratórias e ofensivas, no contexto de um planejamento militar, realizadas no espaço cibernético, com a finalidade de proteger os sistemas de informação, obter dados para a produção do conhecimento de inteligência e causar prejuízos aos sistemas de informação do oponente".

Neste contexto e com o intuito de defender a rede corporativa da ameaça de *malwares*, o Exército Brasileiro lançou, em 2011, o projeto de uma solução comercial multiplataforma e corporativa de antivírus, por meio de um processo de licitação no valor de R$ 4.520.000,00. Tal montante, foi empregado na aquisição de uma solução comercial que permitisse o monitoramento em tempo real de incidentes de segurança decorrentes de arquivos e processos maliciosos, tendo sido tal projeto implantado a partir de 2016.

No entanto, o referido projeto tem apenas o aspecto preventivo e não abrange a análise e aquisição de conhecimento sobre as características dos códigos maliciosos. Tal conhecimento é extremamente relevante, considerando que não basta apenas sua detecção, mas sim compreender e identificar suas ações. Tal situação traz o risco de que *malwares* sofisticados, ou seja, artefatos especialmente desenvolvidos ou modificados, não sejam detectados, e adicionalmente, há o risco da ausência de conhecimentos adicionais sobre o *modus operandi* dos *malwares* eventualmente detectados.

Desta forma, e tendo por base o objetivo de contribuir com a estruturação do setor de defesa cibernética, o Centro de Desenvolvimento de Sistemas (CDS), órgão do Exército Brasileiro (EB), responsável pelo desenvolvimento de soluções de Tecnologia da Informação, recebeu a missão de implantar um ambiente especialmente preparado, com toda a infraestrutura necessária para hospedar, de forma segura e segregada, o estudo e análise de códigos maliciosos que afetem a rede corporativa do EB.

1.2 Definição do Problema

A forma clássica para se detectar um *malware* é pela utilização de softwares como antivírus, sistemas de detecção de intrusão baseados na análise de equipamentos (*hosts*), ou em análise de tráfego de redes tais como: *Intrusion Detection System* (IDS) e *Host Intrusion Detection System* (HIDS) [13]. Uma das primeiras formas de detecção de *malware* é baseada em heurística, em que se busca identificar características, tais como: fragmentos

de código, coleta de *hashes* e propriedades do arquivo, com o objetivo de se criar uma impressão digital, também chamada de assinatura [14]. No entanto, os produtos que oferecem esse tipo de proteção contra *malware* estão se tornando cada vez mais ineficazes [15], devido ao surgimento de técnicas de evasão, tais como o polimorfismo, o que permite que centenas de milhares de amostras surjam todos os dias, afetando assim, os sistemas de detecção tradicionais [14].

Por isso, as empresas *antimalwares* se voltaram para a aprendizagem de máquina, uma área da ciência da computação que tem sido usada com sucesso no reconhecimento de imagens, pesquisa e tomada de decisões, e mais recentemente, na detecção e classificação de *malware* [14].

Diante disso, considera-se que há, dentro da instituição Exército Brasileiro, a oportunidade de explorar o aprendizado de máquina na detecção de *malware*, a partir das seguintes questões de pesquisa (*Research Questions* - RQ):

(RQ1). *Qual é a eficácia do uso do ML para classificar arquivos como benignos ou maliciosos a partir de informações extraídas de arquivos executáveis do Windows?*

(RQ2). *Qual um conjunto mínimo de atributos a serem extraídos desse tipo de arquivo que podem ser usados para classificar ativos como benignos ou maliciosos, mantendo-se a acurácia?* e secundariamente responder:

- *Quais são as consequências de treinar um modelo de ML com uma base desequilibrada?*

(RQ3). *O uso de atributos textuais coletadas dos arquivos de binários contribui para a precisão do modelo final?*

1.3 Objetivo

O objetivo geral dessa pesquisa é investigar a efetividade dos algoritmos de aprendizado supervisionado de máquina aplicados em atributos extraídos por análise estática, de amostras de arquivos executáveis do sistema operacional MS-Windows, chamados *Portable Executáveis* (PE).

O objetivo geral, foi dividido nos seguintes objetivos específicos, que em seu conjunto, contribuirão para o êxito da pesquisa:

a) identificar na literatura, metodologias, práticas, técnicas e ferramentas, que tragam a compreensão sobre os pontos positivos e negativos das abordagens atuais de aprendizado de máquina para a identificação de códigos maliciosos;

b) extrair atributos dos arquivos PE a partir de ferramentas de análise estática existentes, para compor a base de dados de entrada para a aplicação nos experimentos de aprendizado de máquina;

c) Aplicar, através de experimentos empíricos, técnicas de aprendizado supervisionado de máquina para a identificação de códigos maliciosos;

d) Avaliar o desempenho da abordagem proposta, através de métricas referenciadas na literatura, para a mensuração de técnicas de ML; e

e) validar, como prova de conceito, a aplicabilidade da abordagem proposta em um protótipo de aplicação.

Para alcançar tais objetivos específicos, buscou-se experiências de trabalhos anteriores, dos quais foram selecionados um conjuntos de técnicas que foram aplicadas em atributos extraídos de arquivos executáveis PE. Para treinamento dos modelos, utilizou-se os seguintes algoritmos de classificação, referenciados na literatura, como relevantes para aprendizado supervisionado: KNN, SVC, *Gaussian* NB, *Random Forest* e *Decision Tree* [3, 16, 17].

1.4 Justificativa

Técnicas de aprendizado de máquina ou *Machine Learning* (ML), baseadas em modelos preditivos têm sido frequentemente investigados como uma ferramenta promissora para ajudar a detectar *malwares* [18, 15, 19]. Trabalhos recentes [20, 16, 21, 3, 22, 23, 15] exploram o uso do aprendizado de máquina para classificar artefatos (por exemplo, arquivos executáveis ou bibliotecas do Windows e aplicativos Android) como benignos ou malignos. Todos esses trabalhos usam conjuntos diversos de atributos, os quais podem ser coletados através da análise estática ou dinâmica. No entanto, apesar de um número considerável de estudos abordando a detecção de *malwares* por meio de técnicas de ML, ainda não são claros os benefícios de usar técnicas para encontrar um subconjunto menor de atributos, que permita reduzir o custo computacional da análise, sem comprometer o desempenho dos métodos para classificar arquivos como benignos ou malignos.

Segundo Ceron [24], as ferramentas comerciais existentes, em geral, fornecem um grande nível de detalhamento na análise desses códigos maliciosos. Um exemplo das principais ferramentas relacionadas à detecção de códigos maliciosos pode ser verificada pelo quadrante mágico do Gartner[3], o qual apresenta uma classificação de qualidade cri-

[3]https://www.gartner.com

ada por aquela organização baseada na avaliação de empresas e organizações que utilizam tais ferramentas.

Todavia, embora essas ferramentas de detecção de *malwares* sejam bastante confiáveis, o custo das ferramentas comerciais pode significar um alto impacto financeiro para as organizações, conforme pode ser verificado no portal *Cybersecurity Pricing*[4]. Ademais, tais ferramentas seguem uma arquitetura fechada, e, via de regra, não permitem a aquisição de conhecimento sobre a tecnologia embutida. Outro ponto relevante é a ausência da possibilidade de integração com tecnologias de outros fabricantes, bem como a dificuldade de adição e remoção de componentes.

Por fim, ressalta-se que a opção da presente pesquisa pelo formato de arquivos PE, baseia-se no fato de que os tipos de *malwares* mais detectados são direcionados aos arquivos executáveis portáteis de 32 bits, os quais são também capazes de infectar as plataformas de 64 bits, ambos dos sistemas operacionais *MS-Windows* [21].

1.5 Contribuições

Como contribuição, o presente trabalho fornece uma evidência empírica, baseada nos experimentos descritos na Seção 5.1, de uma abordagem capaz de distinguir entre arquivos maliciosos e benignos.

Adicionalmente, esta pesquisa apresenta as seguintes contribuições:

a) evidência empírica de que é possível reduzir a dimensionalidade do conjunto de dados, a partir da aplicação do método PCA (vide Seção 2.4.2), usados como entrada para treinamento dos algoritmos de aprendizado de máquina, sem afetar significativamente a precisão do modelo treinado para classificar os arquivos como benignos ou malignos;

b) evidência empírica que utilização de atributos textuais extraídos dos arquivos PE da base de pesquisa, não foram suficientes para assegurar o mesmo nível de precisão dos outros experimentos, quando aplicado a novos conjuntos de arquivos.

c) por fim, é oferecido à comunidade científica, como suporte a outros trabalhos, um *dataset* com um conjunto de atributos extraídos de mais de 14.000 amostras (entre arquivos maliciosos e benignos)[5], além das ferramentas construídas e utilizadas para a extração dos atributos dos arquivos analisados, e para o treinamento dos modelos de aprendizado de máquina.

Como produto final, foi desenvolvido um protótipo de aplicação Web, como parte de um sistema de detecção de *malwares*, para uso restrito ao ambiente organizacional

[4]https://cybersecuritypricing.org/
[5]https://github.com/alejr-git/Mal_Lab_ML_Project

pesquisado. Tal protótipo incorpora o modelo resultante do treinamento de aprendizado de máquina (descrito na Seção 6.1) e que poderá servir de base para o desenvolvimento e incorporação de outras funcionalidades.

1.6 Metodologia

Na pesquisa científica, o método pode ser entendido como um conjunto de atividades sistemáticas e racionais que permitem alcançar o objetivo de forma convincente [25]. Também pode ser definido como um conjunto de etapas e instrumentos pelo qual o pesquisador pode direcionar o seu projeto com critérios de caráter científico para alcançar dados que suportam ou não sua teoria inicial [26]. Dessa forma, a presente pesquisa foi constituída de duas partes. A primeira caracteriza-se como um revisão da literatura, com o objetivo de se obter um entendimento sobre o tema. A segunda parte é caracterizada pelo estudo de caso, a fim de proceder uma aproximação com o tema de pesquisa, a partir de uma série de procedimentos e experimentos relacionados ao aprendizado de máquina.

Após a revisão da literatura, foi criado e customizado um conjunto de ferramentas para a desmontagem (*disassembly*) dos arquivos binários e extração dos atributos para a montagem dos conjuntos de dados empregados nos experimentos de aprendizado de máquina.

Como experimentação prática, utilizaram-se três abordagens diferentes de aprendizado de máquina. No primeiro experimento, em uma abordagem direta, utilizaram-se 24 atributos numéricos extraídos de diversas amostras de *malwares* e arquivos benignos, com entrada para o processo de treinamento. Posteriormente, utilizou-se uma técnica para reduzir a dimensionalidade dos dados, a fim de verificar se houve e qual o impacto dessa técnica na acurácia dos modelos. Por fim, utilizaram-se atributos textuais, os quais não haviam sido utilizados nos experimentos anteriores, a fim de avaliar a sua relevância na precisão dos modelos. Esses experimentos resultaram em classificador de *malwares* que foi embarcado em um protótipo, chamado de Laboratório de Tratamento de Artefatos (LTA), construído como prova de conceito de uma aplicação de detecção de *malwares*.

1.7 Organização do Trabalho

O trabalho está organizado em 7 Capítulos. No Capítulo 2 será apresentado um referencial teórico para apoiar o leitor na compreensão de temas chave relacionadas à pesquisa, tais como a evolução dos *malwares*, as técnicas de análise de *malwares* e aspectos relacionados ao aprendizado de máquina. O Capítulo 3, por sua vez, trará uma revisão da literatura, onde serão apresentados alguns trabalhos utilizados como referência para o

desenvolvimento da solução proposta. No Capítulo 4 serão discutidos os procedimentos necessários à implementação dos experimentos. No Capítulo 5 são descritas um série de experimentações práticas, bem como a análise dos resultados alcançados. No Capítulo 6 será apresentado o protótipo de aplicação que embarca o classificador resultante dos experimentos, e por fim, no Capítulo 7 serão apresentadas as conclusões e trabalhos futuros.

Capítulo 2

Fundamentação Teórica

Este Capítulo apresenta os conceitos, técnicas e ferramentas relacionadas à essa dissertação, e objetiva facilitar o entendimento da pesquisa, sendo disposto em 5 seções: a primeira apresenta conceitos relacionados aos *malware*, e as suas diversas classificações, a segunda, explora a ação de *malwares* em dispositivos móveis, a terceira apresenta as diversas formas de detecção. A quarta apresenta as diversas técnicas de aprendizado de máquina utilizadas na pesquisa, e finalmente, uma breve retrospectiva é apresentada na última seção.

2.1 Definição de *Malware*

Segundo o NIST[1], *malware* é um programa ou software malicioso [27]. Tais programas são, em geral, inseridos em sistemas computacionais, normalmente de forma oculta, com o objetivo de comprometer a integridade, a confiabilidade e a disponibilidade de dados de vítimas, aplicações ou sistemas operacionais, ou simplesmente, para degradar os sistemas afetados [28].

Os *malware* exploram diferentes formas para comprometer os sistemas computacionais, quase sempre embutidos em programas legítimos. Tais formas incluem *malware* anexados em mensagens de e-mail, propagados automaticamente via rede, através da exploração de vulnerabilidades conhecidas e falhas de configuração[24]. Ademais, a grande quantidade de *malware* que são produzidos requerem um tratamento automatizado, inicialmente para sua detecção, seja para realizar a análise mais detalhada sobre seu comportamento [10]. Portanto, é de fundamental importância compreender as características dos diferentes tipos de *malware* para se aprimorar a segurança dos mecanismos de defesa. Dessa forma, após a detecção, uma próxima etapa é a classificação dos códigos maliciosos.

[1] National Institute of Standarts and Technology - Agência governamental dos Estados Unidos da América

2.1.1 Classificação dos *Malware*

Segundo o CERT.br [29], os *malware* podem ser classificados de acordo com o seu uso ou ação, mas não limitados a vírus de computador, *worms, trojans, rootkits, adware*, e *spyware*. Os principais tipos de códigos maliciosos e suas características serão apresentados a seguir.

Vírus de Computadores

Vírus é um programa ou parte de um programa de computador, normalmente malicioso, que se propaga inserindo cópias de si mesmo e se tornando parte de outros programas e arquivos[24]. Uma de suas principais características é a necessidade de ser executado para infectar ou executar sua ação maliciosa [30].

Sua origem remonta à década de 1970, quando, segundo a literatura, Robert Thomas Morris elaborou o primeiro vírus de computador chamado *Creeper*, que tinha a capacidade de infectar máquinas IBM 360 na ARPANET. Para eliminar este programa, foi desenvolvido um outro programa chamado *Reaper*, dando origem aos programas antivírus [11]. No entanto, a história dos vírus de computador, tal como a conhecemos, começou nos anos de 1980, quando essa terminologia foi estabelecida por Cohen em 1983 [31].

Segundo Arcoverde [32], a literatura reconhece vários tipos de vírus de computadores na atualidade, conforme exemplos a seguir:

a) Vírus de Boot Sector - São vírus que infectam o setor de inicialização de um disco, seja rígido ou flexível [32].

b) Vírus de MBR (*Master Boot Record*)- os vírus de MBR infectam o setor de boot dos discos, mas geralmente salvam uma cópia do MBR em um lugar distinto [32];

c) Vírus multipartidos - São vírus híbridos que tanto alteram o setor de boot quanto os programas executáveis do sistema operacional; e

d) Vírus de macro - São vírus que visam infectar arquivos de dados, tai como: arquivos do Microsoft Excel, Word, Access, Power Point, etc [32].

Worms

São uma variante específica de vírus, inicialmente replicados através de uma rede de computadores. Segundo Kaur [31], os *worms* possuem a característica de não necessitarem de interação humana para realizar sua autoinfecção, gerando um alto grau de propagação, ou cópias de si mesmo. Segundo De Melo *et al.* [28], a grande ameaça dos *worms* reside em sua capacidade de deteriorar a performance de um sistema, sobrecarregando serviços e gerando tráfego, que em alguns casos, pode provocar negação de serviço (DoS)[2].

[2]do inglês: *Denial of Service*

***Trojans* ou Cavalos de Troia**

Trojans são códigos maliciosos, através dos quais os atacantes, após invadirem um computador, alteram programas já existentes para que esses passem a executar também ações maliciosas, como por exemplo, permitir ao atacante o acesso efetivo ao sistema comprometido e extrair informações confidenciais, tais como: senhas (*passwords*) e dados bancários [30]. Nesta categoria, destacam-se também os RAT (*Remote Access Trojans*), também categorizados como *backdoors* [28].

Downloader

É um programa malicioso que se conecta à rede para obter e instalar um conjunto de outros programas maliciosos ou ferramentas que levem ao domínio da máquina comprometida [28]. Para evitar dispositivos de segurança instalados na máquina atacada, os *downloaders* podem vir anexados à mensagens de correio eletrônico para que, a partir de sua execução, obtenham de uma fonte externa a outra parte do código que contém o conteúdo malicioso [33].

Botnets

Segundo Ceron [24], *Bot* é um programa que dispõe de mecanismos de comunicação com o invasor o qual permitem que ele seja controlado remotamente. As máquinas comprometidas são configuradas para se comunicar com um terminal remoto, também chamados de servidor de "comando e controle" (C&C) [34], formando redes de computadores comprometidos ou zumbis [35]. Tais redes são chamadas "*botnets*", e são geralmente gerenciadas por um humano denominado "*botmaster*", o qual tem a capacidade de gerenciar toda a estrutura da rede maliciosa [10].

A comunicação entre o invasor e o computador infectado pelo *bot* pode ocorrer via canais de IRC[3], servidores Web e redes do tipo P2P (*peer to peer*)[4] [34]. Tais *bots* representam um dos tipos mais sofisticados e populares de crime cibernético na atualidade [36], sendo geralmente utilizadas para ataques de negação de serviço, propagação de códigos maliciosos (inclusive do próprio *bot*), coleta de informações de um grande número de computadores, envio de *spam* e camuflagem da identidade do atacante, através do uso de *proxies* instalados nos zumbis, dentre outros [30].

[3]Internet Relay Chat- protocolo de comunicação na Internet
[4]protocolo de redes distribuídas

Spywares

Códigos maliciosos que têm por objetivo capturar informações da máquina comprometida e enviar de volta para o atacante [30]. Tais códigos podem ser classificados como:

a) *keylogger*, tem a função de capturar e armazenar as teclas digitadas pelo usuário no teclado do computador [24];

b) *Screenlogger*, similares aos *keylogger*, são capazes de armazenar a posição do cursor e a tela apresentada no monitor, nos momentos em que o *mouse* é clicado [37]; e

c) *Adwares* é um tipo de software relacionado à publicidade, que traz de forma contínua anúncios para o computador afetado, e que também podem ser usados para fins maliciosos, quando as propagandas apresentadas são direcionadas, de acordo com a navegação do usuário e sem que o usuário saiba que está sendo monitorado [38].

Ransomwares

Pode ser entendido como um código malicioso que infecta dispositivos computacionais com o objetivo de sequestrar, capturar ou limitar o acesso aos dados ou informações de um sistema, geralmente através da utilização de algoritmos de encriptação (*crypto-ransomware*), para fins de extorsão [30], e dessa forma, restringe o acesso aos dados até que um resgate seja pago. Para obtenção da chave de decriptação, geralmente é exigido o pagamento (*ransom*) através de moedas eletrônicas (criptomoedas) tais como "*Bitcoins*".

Após a infecção inicial, o *ransomware* tenta se espalhar para unidades de armazenamento compartilhadas e para outros sistemas acessíveis [27]. O *ransomware* normalmente se espalha através de *e-mails* de *phishing* ou visitando inadvertidamente um site infectado, e explora vulnerabilidades não corrigidas em *softwares* [39].

Quanto à forma de ataque, embora existam outros métodos, a infecção por *ransomware* está frequentemente relacionada aos e-mails de *phishing* [39]. A recuperação pode ser um processo difícil que pode exigir os serviços de um especialista em recuperação de dados. Algumas vítimas pagam para recuperar seus arquivos, sem no entanto, a garantia de que os recuperem [8]. Dessa forma, a detecção automática de *malware* tem sido usada e adaptada por muitos algoritmos de aprendizado de máquina para minimizar este tipo de ameaça [40].

Rootkits

Segundo Arcoverde [32], *rootkit* é um *software*, ou conjunto de *softwares*, que possui o intuito de esconder determinadas atividades e comportamentos de um sistema. O termo *rootkit* origina-se da junção das palavras "*root*" (que corresponde à conta de superusuário ou administrador do computador em sistemas Unix) e "*kit*" (que corresponde ao conjunto

de programas usados para manter os privilégios de acesso desta conta) e técnicas para esconder e assegurar a presença de um invasor ou de outro código malicioso [30].

2.2 *Malwares* em Dispositivos Móveis

As questões de segurança estão crescendo rapidamente em termos de tecnologia móvel, principalmente em termos de ameaças de *malware*, segundo Ucci *et al.* [41]. Por isto vários métodos de análise de *malware* foram desenvolvidos para identificar, analisar e defender os dispositivos móveis contra a ameaça dos códigos maliciosos [31].

2.2.1 *Malwares* em Android

Segundo Bao *et al.* [42], o sistema operacional *Android* se tornou a plataforma móvel mais dominante atualmente. Tal afirmação poder ser confirmada pelo relatório do Gartner[5], que afirmou que no quarto trimestre de 2016, 81,7% dos dispositivos móveis utilizavam a plataforma *Android*. Infelizmente, os dispositivos móveis que utilizam o *Android* são cada vez mais alvo de ataques. De acordo com Tam *et al.* [43], cerca de 0,25% dos dispositivos *Android* foram infectados com *malware*, o que significa um grande número, considerando-se o número total de dispositivos *Android* no mundo.

Ainda segundo Bao *et al.* [42], os aplicativos *Android* são desenvolvidos principalmente na linguagem Java, em seguida compilados e, finalmente, convertidos em formato de arquivo de *bytecode*[6]. Segundo Kaur [31], os mais famosos *malware* para *Android* são "*Ackposts*", "*Acnetdoor*" e "*Adsms*".

2.2.2 *Malwares* em IOS

Segundo D'Orazio *et al.* [44], a *Apple*, empresa responsável pelo sistema operacional "IOS", é conhecida por ter um processo rigoroso de verificação para seus aplicativos, a fim de controlar a distribuição de aplicativos maliciosos. No entanto, diversos códigos maliciosos têm sido reportados na literatura, tais como o "*KeyRaider*", que em 2015 obteve com sucesso mais de 225.000 contas válidas de usuários da *Apple*, incluindo senhas, certificados e chaves privadas.

[5]https://www.gartner.com/

[6]arquivo compilado pela linguagem java para ser executado pelo *Java Virtual Machine* - JVM

2.3 Técnicas de Detecção de *Malwares*

Os métodos tradicionais de detecção de *malwares* são baseados em assinatura ou heurística, segundo Sihwail [30]. A assinatura é uma sequência de *bytes* ou "*hash*"[7] de arquivo que pode ser usado para identificar *malwares* específicos. No entanto, a detecção por assinaturas não é eficaz na detecção de *malwares* desconhecidos ou "*zero day*", e para *malwares* que usam técnicas de ofuscação [30]. As assinaturas são geralmente baseadas em características extraídas apenas por análise estática, enquanto as técnicas de detecção baseadas em heurística ou anomalias utilizam a análise estática e dinâmica. Como limitação da detecção por heurística, destaca-se a presença de alta taxa de "falsos positivo" [45].

2.3.1 Antivírus

As técnicas de detecção de códigos maliciosos, em geral estão relacionadas à utilização de programas antivírus, cuja eficiência pode estar relacionada à quantidade de assinaturas de *malwares* presente em sua base de dados [46]. Entretanto, desenvolver novas assinaturas, especialmente para os *malwares* mais recentes que empregam técnicas sofisticadas, tem sido uma tarefa complexa. Dessa forma, o desenvolvimento de técnicas de análise de *malware* tem contribuído sobremaneira com o aprimoramento da segurança dos sistemas computacionais [10].

2.3.2 Métodos de Análise

A análise de códigos maliciosos é composta por um conjunto de técnicas e procedimentos que visam investigar as características de arquivos suspeitos, segundo Cooke *et al.* apud Ceron [24]. Em termos gerais, a análise de *malware* pode ser dividida em análise estática e dinâmica, ou uma combinação de ambas (análise híbrida), cada uma oferecendo informações específicas sobre a natureza e a funcionalidade de um programa malicioso [47], as quais serão descritas a seguir.

Análise Estática

A análise estática consiste em avaliar o funcionamento de um programa sem a execução do mesmo, baseando-se apenas na análise de seu código. Em particular, a análise estática é pouco eficiente para programas que utilizam técnicas de ofuscação, tal como o Polimorfismo. Tais limitações da análise estática incitaram o surgimento de técnicas complementares, como é o caso da análise dinâmica [10].

[7]Resumo criptográfico obtido de um arquivo a partir da aplicação de uma função criptográfica

Análise Dinâmica

A análise dinâmica refere-se ao processo de execução e monitoramento de códigos maliciosos [48], e consiste em observar suas características funcionais através da sua execução em ambiente de teste controlado e contido chamado de "*sandbox*" [49]. A principal limitação da análise dinâmica é a possibilidade de executar apenas uma amostra de binário de cada vez [10]. As principais metodologias de análise dinâmica se baseiam na comparação do estado do sistema operacional antes e imediatamente após a execução do arquivo [48].

Análise Híbrida

Considerando que ambas, análise estática e dinâmica, tem vantagens e desvantagens [50], a análise híbrida busca obter o melhor de ambas. De acordo com Sihwail *et al.* [51], embora a análise estática apresente baixa complexidade e baixo custo operacional, a análise dinâmica oferece informações em tempo de execução que não podem ser obtidas apenas com análise estática.

2.3.3 Ambientes de Análise

O ambiente de análise é essencial para a abordagem da análise dinâmica, uma vez que o tipo de dado coletado depende tanto do ambiente como da tecnologia usada para capturar eventos do sistema utilizado, de modo que tais ambientes podem ser baseados em máquinas físicas, sistemas virtualizados ou sistemas emulados [47].

Máquinas Físicas

Máquinas físicas são computadores, também chamados de *bare metal*, onde os *malwares* são executados diretamente em seu sistema operacional pré-instalado [52]. Como limitações dessa abordagem, a reinstalação da máquina física é mais demorada do que de um ambiente virtual ou emulado, além de questões relacionadas à escalabilidade deste tipo de abordagem [47].

Sistemas Virtualizados

A virtualização é a simulação do software e ou hardware no qual outro software é executado, onde as instruções a serem executadas por uma aplicação são repassadas para o processador físico. Esse ambiente simulado, chamado de máquina virtual (VM), permite que ferramentas especializadas possam ser instaladas e os *malwares* executados em um ambiente isolado. Outras características são a velocidade de reinicialização, e principalmente a velocidade e a capacidade de restauração ao seu estado inicial[27]. O termo

guest se refere sistema virtualizado, enquanto o termo *host* se refere ao sistema que provê o ambiente de virtualização [52]. Existem muitas formas de virtualização, diferenciadas principalmente pela camada de arquitetura [53]. Como exemplo, a virtualização de aplicativos pode fornecer uma implementação virtual da interface de programação de aplicativos (API) para um aplicativo em execução, permitindo assim que os aplicativos desenvolvidos para uma plataforma sejam executados em outra sem modificar o próprio aplicativo. O *Java Virtual Machine* (JVM) é um exemplo de virtualização de aplicativos que atua como um intermediário entre o código do aplicativo Java e o sistema operacional [53].

Sistemas Emulados

Enquanto os simuladores buscam "imitar"algum sistema, arquitetura ou hardware, em cada aspecto interno de seu funcionamento, os emuladores apenas reproduzem o comportamento final de um determinado hardware. Dessa forma, é um software que simula uma funcionalidade ou uma peça de hardware, e que em geral não compartilha características físicas com o equipamento hospedeiro [49], ou seja, não há a dependência de que o sistema virtualizado seja da mesma arquitetura do sistema hospedeiro [52]. Isso possibilita a execução de um sistema operacional completo em hardware virtualizado que, ao contrário de uma VM, não está vinculado a nenhuma arquitetura específica. Como um exemplo de ambiente que se destina à emulação total do sistema, tem-se o QEMU[8], software de código aberto desenvolvido na linguagem *C* e que implementa uma emulação completa do processador, do hardware e de periféricos [47].

Sandboxes

É um tipo de sistema de análise dinâmica para executar um programa suspeito em um ambiente controlado [49]. Tanto os sistemas virtualizados, quanto os emulados, utilizam *sandboxes*, seja para obter um isolamento do ambiente externo, como também para obter o controle do ambiente de execução [54]. Desta forma, tem-se um ambiente no qual as ações dos aplicativos são restritas para que possam ser executados com segurança [42].

As *sandboxes* geralmente capturam as ações que ocorrem durante a execução do código, tais como: os arquivos criados ou modificados, o acesso ou modificações à chave de registros do sistema; as bibliotecas dinâmicas carregadas durante a execução; áreas de memória virtual utilizada; os processos instanciados; conexões de rede e dados transmitidos, dentre outros [47]. Esses dados podem ser apresentados por meio de relatórios de diferentes formatos [24].

[8]Disponível em https://www.qemu.org/

Os ambientes de *sandboxes* são construídos de maneiras diferentes, alguns são baseados em Virtualização ou *Virtual Machine* (VM), a exemplo do *CWsanbox, Cuckoo sandbox, Normam sandbox*; outros são baseados em máquinas físicas, tais como *barebox* e *bare-cloud* [49]. Há também *sandbox* baseados em sistemas emulados, tal como o QEMU [55]. As principais *sandboxes* são descritas a seguir:

Cuckoo sandbox, sistema de código aberto (*Open Source*) para análise dinâmica de *malwares*, que permite a execução do *malware* em um sistema operacional *Windows*, e fornece um relatório consolidado com as chamadas de API, detalhes sobre arquivos criados, excluídos, ou baixados pelo *malware* durante a sua execução [47], bem como os processos carregados em memória e em execução no sistema operacional, além do tráfego de rede no formato "PCAP" [56].

Limon sandbox[9], solução de código aberto (*General Public Licence*), desenvolvida como um projeto de pesquisa escrito em *Python*, que coleta, analisa e gera relatórios automaticamente sobre os indicadores de tempo de execução do *malware* do Linux. Permite ainda inspecionar *malwares* para *Linux* antes, durante e após a execução (análise *post-mortem*) através da execução de análise estática, dinâmica e de memória.

VMRay sandbox[10], solução automatizada de detecção e análise de *malware*, que permite que analistas e equipes de resposta a incidentes monitorem, analisem e identifiquem ameaças e extraiam indicadores de comprometimento (IOCs). O *VMRay* é um exemplo de ferramenta comercial que implementa *sandbox* baseada em nuvem, ou seja, sem necessidade da organização possuir a infraestrutura de hardware para hospedar o ambiente de análise.

Falcon Sandbox[11], outro exemplo ferramenta comercial para análise de *malware*, baseada em nuvem privada ou uma solução no local.

Containers

Basicamente um container é um sistema de virtualização a nível de sistema operacional, ou seja, o *kernel* (núcleo) permite que múltiplos processos sejam executados de forma isolada na mesma máquina. A ideia de *containers* surgiu inicialmente no ano 2.000 como "*jails*" do Sistema Operacional FreeBSD[12], uma tecnologia que permite particionar o sistema em vários subsistemas ou celas[13] [57].

[9]https://github.com/monnappa22/Limon
[10]https://www.vmray.com/malware-sandbox-products/
[11]https://www.crowdstrike.com/endpoint-security-products/falcon-sandbox-malware-analysis
[12]https://www.freebsd.org/doc/handbook/jails.html
[13]do inglês "*jails*"

Por definição um *container* é um conjunto de um ou mais processos organizados isoladamente do sistema operacional [27]. De uma maneira geral, representam uma abstração na camada de aplicativo que agrupa código e dependências, de modo que vários *containers* podem ser executados na mesma máquina e compartilhar o kernel do sistema operacional com outros *containers*, cada um sendo executado como processos isolados no espaço do usuário [58]

Todos os arquivos necessários à execução de tais processos são fornecidos por uma imagem distinta, enquanto as Máquinas Virtuais (VMs) são uma abstração de hardware físico, que é emulado por um servidor chamado de "*Hypervisor*" responsável por gerenciar vários outros servidores ou sistemas operacionais [27].

Uma das principais soluções de virtualização em *containers*, na atualidade, é o projeto "*Docker*"[14] baseado no LXC[15]. Outra possibilidade é a distribuição *Linux Container* (LXC)[16], projeto *open source* que oferece uma distribuição Linux para o desenvolvimento de tecnologias virtualização mais eficiente que máquinas virtuais [57].

Container versus Máquinas Virtuais

O *container* ocupa menos espaço que as VMs, ou seja, podem hospedar mais aplicativos e exigem menos recursos do hardware, enquanto as VMs incluem uma cópia completa do sistema operacional, aplicativos e binários necessários, ocasionando, além da lentidão do processo de inicialização, uso excessivo de recursos do hardware.

Dessa forma, o uso do *container* se traduz por um *boot* mais rápido e menor consumo de memória RAM. Outra grande vantagem é a portabilidade e controle de versão. Essa eficiência se mostra através de uma estrutura que permite maior escalabilidade e melhor uso dos recursos da máquina hospedeira.

Portanto, ao invés dos sistemas de virtualizações tradicionais,o presente trabalho propõe a utilização de um ambiente de virtualização baseada em "*containers*". A diferença entre a arquitetura de Máquinas Virtuais tradicionais e a arquitetura de *container* pode ser observada na Figura 2.1, onde se verifica que as aplicações *docker* compartilham o sistema operacional do hospedeiro (*host*). As vantagens e desvantagens dessa solução serão descritas a seguir:

- Vantagem - um maior paralelismo e uma maior interação com o ambiente de análise ambientes podem ser criados dinamicamente e recursos são alocados em tempo de execução, e comandos podem ser executados sem a interação de usuários;

[14]https://www.docker.com
[15]Linux Container - Projeto criado pela IBM em 2008
[16]https://linuxcontainers.org

- Desvantagem - existe uma fraca camada de isolamento entre as instâncias virtuais. No entanto, tal desvantagem será tratada com uma camada adicional de proteção através do sistema hospedeiro.

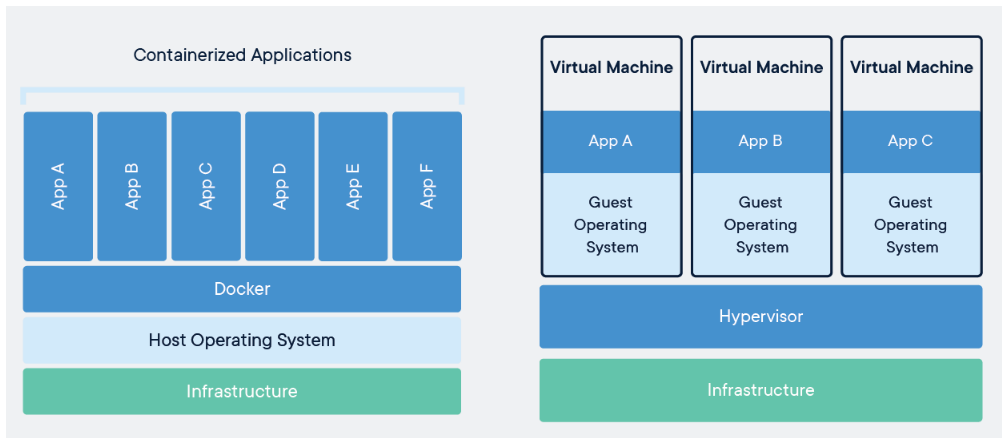

Figura 2.1: Diferença entre VM e Container. Fonte: docker.com.

2.3.4 Técnicas de Evasão

As técnicas de evasão compreendem uma série de técnicas empregadas pelo *malware* para permanecer invisível, e assim evitar a detecção ou dificultar esforços para sua análise [49]. A literatura descreve uma ampla gama de técnicas de anti-análise e de evasão [55]. Um exemplo bem conhecido é o "*Stuxnet*", considerado como a primeira arma cibernética que incorporou a tática de direcionamento como parte de suas táticas de evasão [49]. As principais técnicas de evasão são conhecidas como ofuscação, técnicas anti-debug e anti-VM.

Ofuscação

Os autores de *malwares* criaram um desafio para a abordagem baseada em assinatura usando técnicas de ofuscação para dificultar ou evitar sua detecção [45]. As características mais recentes que buscam evitar a identificação de *malware* pelos antivírus são implementadas através das técnicas de oligomorfismo, polimorfismo e metamorfismo [6], conforme descrição a seguir:

a) oligomorfismo - é uma variação da criptografia, onde o componente criptografado é escolhido de forma aleatória [59];

b) polimorfismo - o componente criptografado é modificado em cada infecção [60]; e

c) metaforfismo - é uma evolução do polimorfismo, com aplicação de técnicas de ofuscação em todo código do *malware*, ou seja todo o código é modificado através de técnicas, tais como troca de registradores, reordenação do código, seja de instrução ou de blocos do código. Tal técnica permite criar variantes muito diferentes do mesmo *malware*, porém com a mesma funcionalidade [59].

Técnicas *Antidebug*

Além das técnicas de ofuscação, os criadores de *malwares* empregam várias outras técnicas para evitar sua análise completa, tais como técnicas de empacotamento e *antidumping*, a fim de impedir que o *malware* seja encontrado através de programas em execução na memória [61]. Tais técnicas podem incluir a busca de valores de funções de API[17] do sistema operacional *MS-Windows*, que caso encontrados pelo *malware*, o mesmo pode encerrar a execução, a fim de evitar a sua identificação [61]. Umas das técnicas mais comuns de *antidebug* é a utilização de programas empacotadores (*packers*) para impedir ou dificultar a análise estática ou engenharia reversa do código malicioso [32].

Empacotadores

Um programa empacotador ou *packer* transforma um arquivo executável, por exemplo, um binário de *malware*, em uma representação sintática diferente, mas com semântica equivalente, cujo o objetivo é ofuscar ou criptografar o binário original, armazenando o resultado desse processo em um novo executável [48].

Técnicas *AntiMV*

Por fim, os *malwares* podem utilizar técnicas de evasão e ou detecção para verificar se o ambiente de execução é um sistema operacional emulado ou virtualizado, baseado em informações colhidas do ambiente de execução, e assim ocultar seu comportamento [47]. Tais técnicas, chamadas de *antiVM*, são recursos utilizados pelos atacantes para burlar sistemas de análise baseados em máquinas virtuais, modificando as estruturas usadas para rastrear o estado da máquina, por exemplo, modificando a estrutura do *kernel*, conforme Bulazel apud Bahram *et al.* [52].

[17]um conjunto de funções e procedimentos que permitem a criação de aplicativos que acessam os recursos, dados, aplicativos ou outro serviço

2.4 Aprendizado de Máquina

O aprendizado de máquina é um subcampo da inteligência artificial que visa capacitar os sistemas com a capacidade de usar dados para aprender e melhorar sem serem explicitamente programados [62], o que inclui uma ampla variedade de abordagens [14]. Os algoritmos de aprendizado de máquina podem ser categorizados como descoberta preditiva (aprendizado supervisionado) ou descoberta de padrões (aprendizado não supervisionado) [62].

O aprendizado supervisionado utiliza dados e metadados rotulados para ajustar o modelo e aplicar o modelo treinado a novas amostras e obter previsões [14]. Existem vários usos para os algoritmos de aprendizado supervisionado de máquina, tais como classificação, regressão, *clustering*, redução de dimensionalidade, seleção de modelo e pré-processamento [3].

Por sua vez, algoritmos de aprendizado não supervisionado identificam padrões ocultos em dados de entrada não rotulados [63] e não exigem que os dados de treinamento sejam rotulados. Eles também decidem as classificações para conjuntos de atributos de entrada por conta própria [3].

As abordagens de aprendizado de máquina aplicadas na classificação de *malware* utilizam, em geral, os atributos estruturais (análise estática) e comportamentais (análise dinâmica) de *malwares* e programas benignos [16]. O método de análise estática implica examinar o *malware*, analisando seus metadados, instruções de código de montagem, dentre outros dados [40]. Os atributos extraídos durante a análise estática podem incluir ainda, informações tais como tamanho do arquivo, número de seções, nomes de seções, etc [3].

Por outro lado, os atributos extraídos durante a análise dinâmica, ou comportamental, se referem aos dados capturados durante a execução do referidos códigos [16] e carregados em memória.

Mais recentemente, alguns trabalhos têm utilizado um método avançado de aprendizado de máquina que combinava aprendizado supervisionado e não supervisionado, chamado *Deep Learning* (DL), o que é considerado, por alguns pesquisadores, como uma nova fronteira no campo de mineração de dados e aprendizado de máquina [64].

A abordagem de *Deep Learning* facilita a extração de atributos em um alto nível de abstração de dados de baixo nível [14], pelo uso de redes neurais [65] tais como: Rede Neural Convolucional (CNN), Redes Neurais Artificiais (ANN), dentre outras [40]. A figura 2.2 mostra a taxonomia associada à detecção de *malware* e as principais algoritmos de classificação utilizados no aprendizado de máquina.

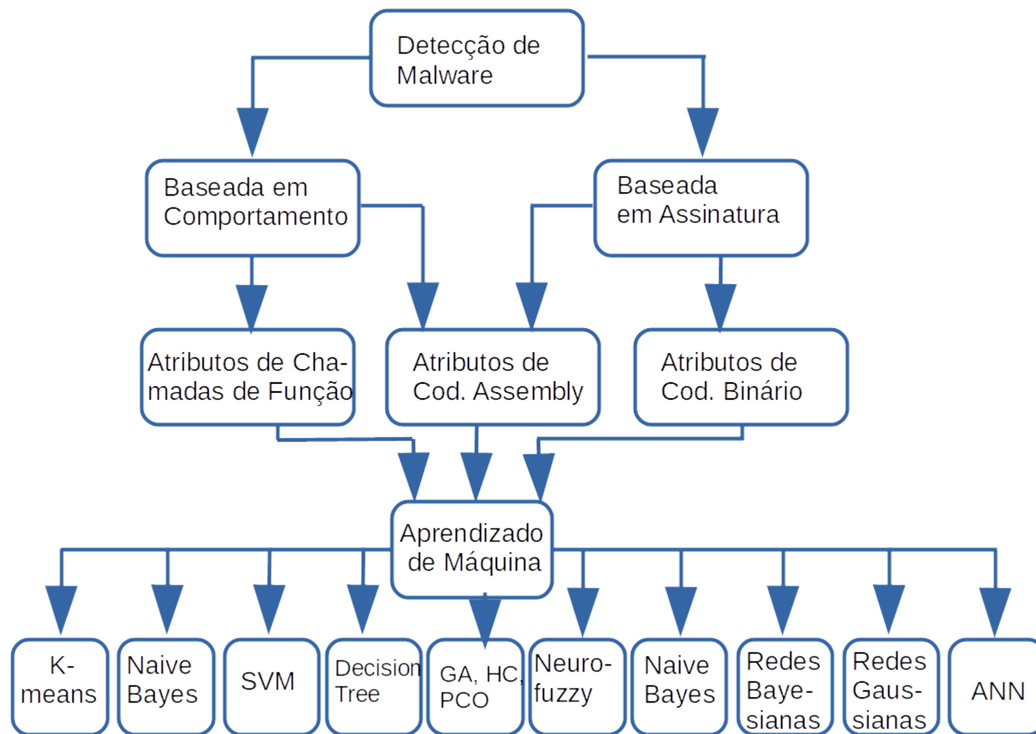

Figura 2.2: Principais abordagens de detecção de *malware* com aprendizado de máquina [1].

2.4.1 Algoritmos de Classificação

Existem vários usos para algoritmos de aprendizado de máquina, tais como: classificação, regressão, *clustering*, redução de dimensionalidade, seleção de modelo e pré-processamento [3]. Verifica-se que diversos algoritmos de aprendizado supervisionado têm sido utilizados para a classificação de *malware*, tais como: *Decision Tree*, *Random Forest*, *K-Nearest Neighbors* (KNN), *Logistic Regression*, *Linear Regression*, *Support Vector Machines* (SVMs), dentre outros [66, 16, 3, 50, 17, 67, 68].

Conforme o escopo da pesquisa, a aplicação da aprendizagem de máquina ficará restrito aos algoritmos de classificação, os quais consistem em modelos de reconhecimento de classes que são usados para predição da ocorrência dessas classes em dados ainda não classificados. Tais algoritmos são divididos em várias categorias, tais como redes bayesianas, como por exemplo o *Naive Bayes*; algoritmos de classificação estatística, como os SVMs; baseados em arvores de decisão, como o *Decision Tree* e *Random Forest*; e finalmente, os baseados em calculo de similaridade (*instance-based*) para a classificação de dados, com o KNN [69, 19]. Dessa forma, os algoritmos utilizados na presente pesquisa são listados a seguir:

- *K-Nearest Neighbors* (KNN)
- *C-Support Vector Classification* (SVC)
- *Gaussian Naive Bayes* (GaussianNB)
- *Decision Tree* (DT)
- *Random Forest* (RF)

A escolha dos referidos algoritmos pesquisa seguiu os seguintes critérios:

a) serem usados largamente em trabalhos correlatos;

b) serem representantes das diversas categorias acima mencionadas; e

c) estarem disponíveis na biblioteca *Scikit-Learn* [2].

Tais algoritmos são detalhados a seguir:

K-Nearest Neighbors **(KNN)**

O KNN é uma método, proposto por Cover e Hart (1967) [70], usado de classificação e regressão. O método é baseado em instâncias, ou seja atribuir uma classe a cada elemento, a partir da classe obtida de seus vizinhos mais próximos [71]. O algoritmo KNN assume que todas as instâncias correspondem a pontos no espaço n-dimensional [72]. Os vizinhos mais próximos de uma instância são definidos em termos da distância euclidiana padrão. A distância euclidiana é uma das medidas de similaridade mais referenciadas na literatura, e pode ser calculada pela seguinte fórmula (2.1) [71]:

$$d(x, y) = \sqrt{\sum_{i}^{n} (x_i - y_i)^2} \tag{2.1}$$

onde x_i é um exemplo de vetor de treino, n a dimensão do vetor, enquanto y é amostra de teste. O scikit-learn [2] implementa o *KNeighborsClassifier*, no qual o aprendizado é baseado no K vizinhos mais próximos de cada ponto de consulta, onde K é um valor inteiro especificado pelo usuário. O KNN é um dos algoritmos mais utilizados no aprendizado de máquina, devido em parte à sua idade e em parte à sua simplicidade [73]. No entanto, o KNN é afetado por problema relacionado a conjuntos de dados de alta dimensionalidade, chamado curso da dimensionalidade, pois os pontos tendem a não ser muito próximos um dos outros [2].

C-Support Vector Classification

O SVC (*C-Support Vector Classification*) é uma biblioteca do Scikit Learn [2] que implementa o algoritmo *Support Vector Machines* (SVM). O SVM é um dos mais utilizados para classificação, onde, dadas duas classes e um conjunto de instâncias de treinamento cujas amostras pertencem a essas classes, o SVM constrói um hiperplano que divide o espaço de características em duas regiões, maximizando a margem de separação entre as mesmas [71].

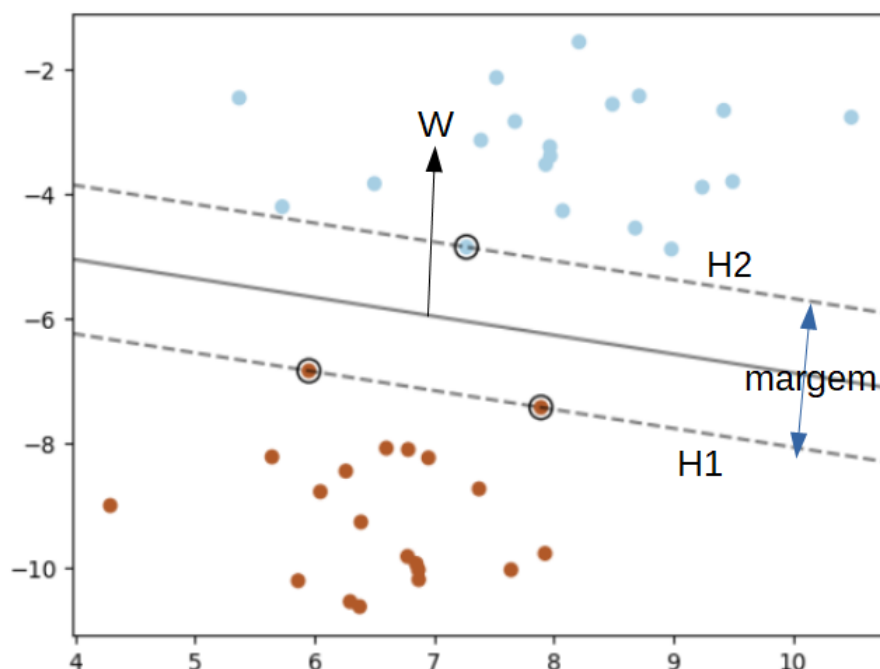

Figura 2.3: Gráfico do SVM que mostra o hiperplano que separa um conjunto de dados em duas classes [2].

Esse algoritmo é considerado um dos mais complexos algoritmos de aprendizado de máquina e contém muitos hiperparâmetros, como: *kernel, degree, gama, etc* [3], e que tem sido usado em muitos estudos de detecção de *malware* [21]. A Figura 2.3 apresenta um gráfico de limites de decisão do SVM para uma classificação bidimensional, onde mostra o hiperplano (W) de separação, com uma reta separadora, ou linha de decisão que maximiza margem entre dois conjuntos de pontos, e as as retas pontilhadas (H1 e H2) são um par de hiperplanos que geram as margem máxima pela minimização do vetor peso W. Os pontos interceptados em cada lados hiperplanos (H1) e (H2) são chamados de vetores de suporte (*support vectors*), para classificação de dados desconhecidos. Quando a dimensão dos dados é bidimensional, a equação da reta separadora é definida pela fórmula 2.2:

$$(w^T * x + b = 0) \tag{2.2}$$

Onde w e x são vetores bidimensionais: w é um vetor perpendicular, x é um vetor que pertence à reta, e b um número real [74].

Gaussian Naive Bayes **(GaussianNB)**

O *Gaussian* NB é uma implementação do algoritmo *Gaussian Naive Bayes*, o qual é um classificador probabilístico simples, baseado na teoria bayesiana [18], com suposições ingênuas sobre independência na correlação de diferentes características [21]. Tal classificação probabilística pode se expressa matematicamente pela equação 2.3:

$$P(A|B) = \frac{P(A|B) * P(A)}{P(B)} \tag{2.3}$$

que descreve a relação de probabilidades condicionais de quantidades estatísticas para dois eventos independentes A e B, dado $P(B) \neq 0$ ou seja a probabilidade condicionada de A, dada alguma característica observada em B [75].

Os algoritmos de classificação *Naive Bayes* são baseados na dependência entre os atributos (*features*) para determinar classificações [3]. O termo ingênuo vem da suposição de independência condicional entre cada par de atributos, dado o valor da variável de classe [2]. O *Naive Bayes* assume que todos os atributos são condicionalmente independentes; dessa forma, o cálculo da probabilidade é simplificado para o produto das probabilidades condicionais de observar atributos individuais, dado um rótulo de classe.

Inicialmente, são calculadas as probabilidades de cada classe. Em um exemplo de classificação binária com a mesma quantidade de elementos para cada classe a probabilidade seria de 50% [76]. A equação define o calculo das probabilidade do algoritmo *Gaussian Naive Bayes* [19] para classificação representada em 2, a partir de uma variável classe y e um vetor de atributos x_i, onde os paramentos σ_y e μ_y são estimados considerando-se a máxima probabilidade, conforme a equação 2.4 [2].

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right) \tag{2.4}$$

Decision Tree

O *Decision Tree* é um algoritmo de aprendizado supervisionado não paramétrico usado para classificação e regressão. O *Decision Tree* envolve uma série de expressões de decisão, conectados em forma de árvores [3], que são modelos de representação do conhecimento,

[18] http://www.dme.im.ufrj.br/ flavia/mad474/migonehedibert.pdf
[19] https://scikit-learn.org/stable/module/naive_bayes.html

nas quais cada nó representa uma decisão sobre um atributo que determina como os dados são divididos pelos nós filhos [74]. O objetivo é criar um modelo que possa predizer o valor de uma variável de destino, aprendendo regras simples de decisão inferidas a partir dos atributos (*features*) dos dados disponíveis [2].

Os algoritmos de árvore de decisão (*Decision Tree*) são os componentes fundamentais do *Random Forest*, os quais, apesar de sua simplicidade, estão entre os mais poderosos algoritmos de aprendizado de máquina disponíveis atualmente [66]. No entanto, como limitação, o *Decision Tree* tende ao *over-fitting* em conjuntos de dados com alta dimensionalidade. Portanto, é importante obter a proporção correta entre a quantidade amostras e o respectivo número de atributos [2]. A figura 2.4 mostra um exemplo de arvore de decisão, em forma de um grafo acíclico em que cada nó representa uma condição, teste de atributo ou regra de uma instância e cada ramo corresponde a um possível valor desse atributo.

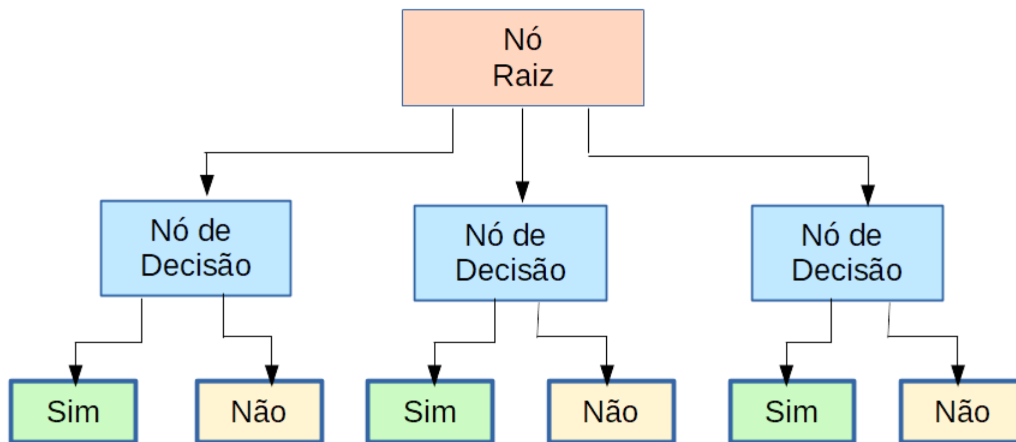

Figura 2.4: Gráfico de árvore de decisão [3].

Random Forest

O algoritmo *Random Forest* se compõe, basicamente, de uma combinação de preditores tipo *Decision Tree*, de modo que cada árvore depende dos valores de um vetor aleatório amostrado independentemente e com a mesma distribuição para todas as árvores que compõe a floresta [77].

Como uma limitação do algoritmo *Decision Tree* é sua tendência ao *over-fitting*, uma maneira de se evitar isto é pela construção de um conjunto de várias arvores de decisão, deixando que elas votem. Desta forma, a previsão do conjunto é dada como a previsão média dos classificadores individuais, de modo a diminuir os erros de predição [2]. Durante a construção da árvores, ao dividir cada nó, a melhor divisão (*split*) é encontrada, seja

de todos os atributos de entrada, ou seja pelo subconjunto aleatório dos melhores valores obtidos pelo parâmetro *max_features*.

A implementação do scikit-learn [2] combina os classificadores pela média probabilística de suas previsões, ao invés de permitir que cada classificador vote em uma única classe [2]. O *Random forest* (RF) também é representado como gráfico em forma de árvores, onde os nós testam determinadas condições em um conjunto específico de atributos e os ramos dividem a decisão em relação aos nós das folhas [76].

2.4.2 Redução da Dimensionalidade

A técnica de redução da dimensionalidade visa manter pequeno número de atributos, removendo aqueles considerados irrelevantes, redundantes ou que poderiam causar ruído no treinamento e diminuir a precisão na fase de treinamento dos modelos [78]. Além disso, pode ser usada como uma ferramenta de visualização, ao apresentar dados multivariados em uma forma humana acessível. Pode ser usada também para a extração de características e para a transformação preliminar aplicada aos dados antes do uso de outras ferramentas de análise, como agrupamento e classificação [79].

Apesar do surgimento de técnicas não lineares, as quais são aplicadas em dados não lineares e complexos [80], as técnicas de redução linear da dimensionalidade têm vantagens, tais como: a garantia de que os dados resultante são confiáveis, uma vez que apresenta a mesmas propriedades dos dados iniciais; a matriz de transformação pode ser armazenada em memória e ser usada em novos elementos; e principalmente a complexidade computacional é muito baixa, em termos de tempo e espaço [79].

As técnicas clássicas de redução de dimensionalidade incluem algoritmos não supervisionados, tais como: *Principal Component Analysis* (PCA) [81] e algoritmos supervisionados, como: *Linear discriminant analysis* (LDA) [82], Canonical *correlation analysis* (CCA) [83], dentre outros [78]. Na presente pesquisa, utilizou-se o PCA, o qual é uma da técnicas de redução linear, baseado na distância euclidiana [79], que reduz a dimensionalidade dos dados, enquanto mantém a variância dos dados, o mais próximo possível [78].

PCA

O PCA busca aumentar a eficiência computacional ao reter apenas informações úteis ao modelo, descartando as informações desnecessária à manutenção de uma boa performance [66]. Além disso, pode ser útil como ferramenta para visualização, filtragem de ruído, extração e engenharia de atributos [2].

O PCA produz combinações lineares das variáveis originais para gerar os eixos, também conhecidos como componentes principais (PCs) [84], e ao mesmo tempo que reduz

a dimensionalidade, retém os atributos (*features*) mais relevantes para classificação, ou seja, preserva a capacidade do modelo para distinguir entre arquivos benignos e maliciosos [50], produzindo ao final um novo conjunto de dados cujas variáveis são representativas da linearidade das variáveis no conjunto de dados original [85].

Os valores resultantes da aplicação do PCA são chamados de autovetores (*eigenvectors*) e autovalores (*igenvalues*) da matriz de covariância (ou correlação) e representam o seu núcleo (*core*). Os autovetores determinam a direção do novo espaço de atributos, enquanto os autovalores determinam a sua magnitude [76], e corresponde a variância resultante das transformações lineares aplicadas aos atributos originais [84].

Uma parte vital do uso do PCA na prática é a capacidade de estimar quantos componentes são necessários para descrever os dados [75]. Isso pode ser determinado a razão de variância explicada cumulativa em função do número de componentes.

Como ponto fraco do PCA, ressalta-se que ele tende a ser altamente afetado por *outliers*[20]. Por esse motivo, muitas variantes robustas de PCA foram desenvolvidas, muitas das quais agem para descartar iterativamente esses tipos de dados [75]. No entanto, tais variantes não foram utilizadas na presente pesquisa, uma vez que os *outliers* foram tratados com a normalização dos dados (vide Seção 4.2.3).

2.4.3 Mineração de Textos

A mineração de texto é uma área recente da ciência da computação que combina técnicas da mineração de dados, aprendizado de máquina, processamento de linguagem natural (PLN) e recuperação da informação (RI) em uma grande quantidade de dados [86]. Ela envolve várias etapas, tais como o pré-processamento, que visa transformar o documento em um formato que possa ser a entrada de algoritmos de aprendizado de máquina.

Uma maneira de processar o texto e transformá-lo na representação atributo-valor é utilizar a técnica conhecida como saco de palavras[21]. Esse pré-processamento pode ser realizado, a partir da aplicação de um algoritmo de categorização de textos [87], que converte um documento em um formato estruturado, ou seja, em um formato número numérico que reflete a importância de uma palavra naquele documento [88].

Os atributos estáticos extraídos de arquivos executáveis geralmente requerem um processamento adicional [89]. Nesse contexto, para conduzir às características textuais, alguns procedimentos adicionais são necessários, como a conversão de uma coleção de documentos de texto em uma matriz de contagens de tokens [2], e um processo de generalização do vetor de características [89].

[20]dados que se diferenciam muito dos outros dados
[21]do inglês *do inglês bag of words*

Em muitos documentos ou textos, certas palavras são mais comuns que outras. Nesse contexto, diversas abordagens de detecção de *malwares* utilizam métodos de classificação de textos [90]. Um exemplo de algoritmo de processamento de textos é o TF-IDF. O TF significa frequência de termo, enquanto TF-IDF significa frequência de termo vezes a frequência inversa do documento [2]. O TF-IDF um dos modelos mais populares para ponderar a frequência com que um termo que ocorre em um documento [88], sendo usado para converter características textuais em vetores numéricos antes de alimentá-los para os classificadores [3, 75].

O TF-IDF determina a frequência relativa de palavras em um documento específico em comparação com a proporção inversa dessa palavra em todo o documento [87]. A frequência inversa do documento atribui menor peso às palavras com maior frequência e maior peso às palavras menos frequentes [91]. O TF pode ser definido pela fórmula 2.5 a seguir:

$$tfidf(t,d) = tf(t,d) * idf(t) \tag{2.5}$$

enquanto o IDF pode ser computado pela equação 2.6:

$$idft(t) = \log \frac{n}{df(t)+1} \tag{2.6}$$

Onde n é o número total de documentos dentro de um conjunto (d) e o *df(t)* é a frequência do termo t [2].

2.5 Considerações Finais

Os conceitos apresentados neste capítulo visam facilitar o entendimento e dar embasamento sobre o tema, de modo a propiciar ao leitor uma melhor compreensão dos conceitos a serem abordados na revisão bibliográfica e dos elementos que compõem os experimentos realizados, sem entretanto, a pretensão de esgotar a literatura sobre o tema.

Capítulo 3

Revisão Bibliográfica

Neste capítulo serão apresentados os principais trabalhos relacionados à detecção de *malwares* com a utilização de técnicas de aprendizagem de máquina, buscando-se encontrar o estado da arte sobre o tema. a revisão bibliográfica foi dividida em quatro seções. A Seção 3.1 apresenta os objetivos e principais fontes da presente revisão. Na Seção 3.2, são elencadas as as diversas etapas da presente revisão. Na Seção 3.3, são apresentados os diversos trabalhos relacionados à pesquisa, e por fim, as considerações finais do capítulo.

3.1 Objetivos da Revisão

A presente revisão teve por objetivo identificar os principais e mais relevantes trabalhos relacionados ao tema da pesquisa. Considerando-se a constante evolução tecnológica associada à análise de *malware*, elencou-se o espaço temporal dos últimos 05 anos como adequado para se identificar as mais recentes pesquisas sobre o tema.

Para a revisão da literatura, foram adotados alguns conceitos preconizados na Teoria do Enfoque Meta Analítico Consolidado (TEMAC) proposta por Mariano e Rocha [92], o que possibilitou uma amplitude na análise de diferentes bases de dados e busca métodos e métricas bibliométricas para se encontrar pesquisas de qualidade.

Ainda segundo Mariano e Rocha *apud* Herrera e Herrera [92], na atualidade, os bancos de dados bibliográficos mais importantes são: ISI *Web Of Science* (WoS)[1], *Scopus Elsevier*[2] e *Google Scholar*[3].

Dessa forma, serão apresentados os principais trabalhos correlatos. Grande parte da bibliografia foi obtida a partir de consulta na biblioteca digital do portal de Periódicos

[1]http://www.webofknowledge.com
[2]http://www.scopus.com
[3]http://scholar.google.com

da CAPES[4], e posteriormente por uma busca mais refinada nas bases *Web Of Science* e *Scopus Elsevier*, considerando-se que essas bases englobariam as principais publicações afetas ao tema.

3.2 Etapas da Revisão

A presente revisão foi estruturada em três etapas, as quais serão descritas a seguir:

3.2.1 1ª Etapa - Preparação da Pesquisa

Inicialmente, foram selecionados trabalhos publicados e disponíveis em bases de dados científicas, publicados a partir de 2016, e que tenham relação com a análise de *malware*. Para tanto, buscaram-se os artigos considerados mais relevantes. Entende-se como relevantes, os artigos científicos associados ao tema de pesquisa, com maior número de citações dentro do universo pesquisado.

Uma vez selecionadas as bases, realizou-se então a busca dos trabalhos mais pertinentes. Esta etapa consistiu em definir as palavras-chave e as *strings* de pesquisa. Foi utilizado um conjunto de palavras chave, como *strings* de busca, conforme demonstrado na Tabela 3.1, limitando-se o espaço temporal de busca de 2016 até 2020.

3.2.2 2ª Etapa - Apresentação e Inter-relação dos Dados

Como resultado, no Portal de Periódicos da CAPES foram encontrados 6.089 artigos relacionados ao tema "*malware analysis*", publicados nos últimos 5 anos. A *Web Of Science*, retornou 1.891 artigos, enquanto a *Scopus* retornou 3.685. Aplicou-se então um filtro na busca com a combinação de "*Malware analysis and machine learning*", o resultado foi reduzido para 2.016 artigos no portal da Capes, 525 na base da *Web Of Science*, e 1.000 na base da *Scopus*. Posteriormente, aplicou-se a combinação de *"Malware and static analysis and machine learning"*, o resultado foi reduzido para apenas 734 ocorrências no Portal da Capes, *Web Of Science* 1.000 e *Scopus* 92. Por fim, aplicou-se a combinação diretamente relacionada ao tema da pesquisa: ***"Malware and static analysis and machine learning and Portable Executable"***. Como resultado, foram obtidos 21 trabalhos no Portal da Capes, 11 na base da *Scopus* e 8 no *Web Of Science*. A tabela 3.1 apresenta o sumário das publicações por bases e chaves de busca. A Figura 3.1 apresenta os países com maior número de publicações. Verifica-se que os países que mais publicaram trabalhos relacionados ao tema *"Malware analysis and machine learning"* foram Estados Unidos, seguidos da Índia e China. O Brasil aparece no *ranking* com 9 publicações no

[4]www.periodicos.capes.gov.br

Tabela 3.1: Total de publicações nas bases pesquisadas

Nr Ordem	**Chave de busca**	**Portal de periódicos CAPES/MEC**	***Scopus Elsevier***	**Web of Science**
1	*Malware Analysis*	6.089	3.685	1.891
2	*Malware analysis and Machine and Learning*	2.016	1.000	525
3	*Malware and static analysis and machine learning*	734	92	200
4	***Malware and static analysis and machine learning and Portable Executable***	**21**	**11**	**8**

período. No entanto, é importante ressaltar que esse número não contempla a quantidade de trabalhos de pesquisadores brasileiros em publicações internacionais, tampouco a quantidade de trabalhos de estrangeiros que por ventura tenham sido publicados no brasil.

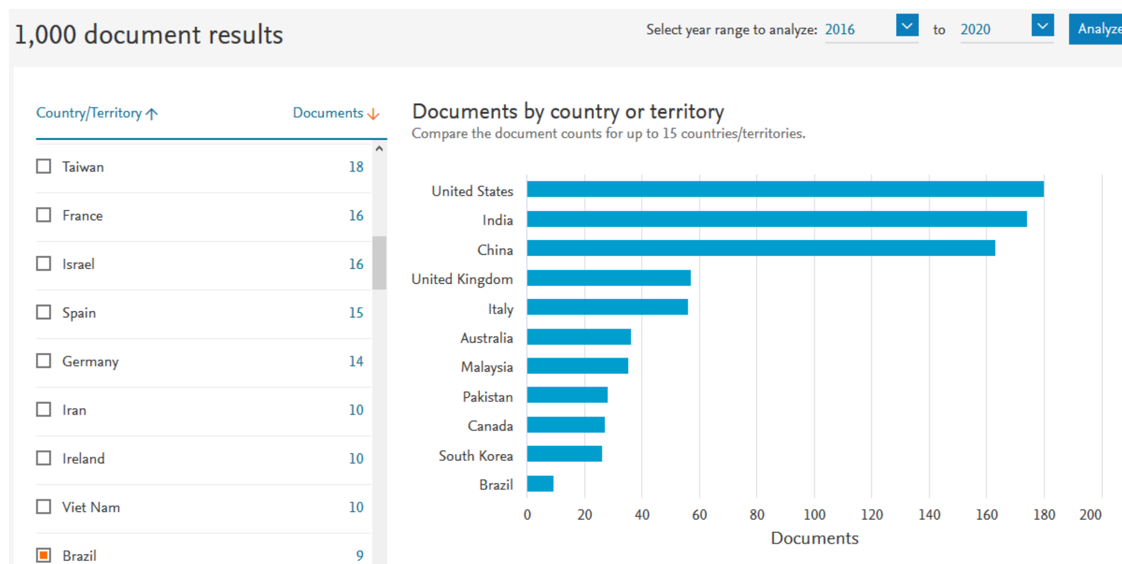

Figura 3.1: Publicações por países. Fonte: *Scopus Elsevier*

As instituições com o maior número de publicações, segundo a *Scopus* foram: *Beijing University*, da China, com 17; *Ben-Gurion University of the Negev*, de Israel com 15; *Amrita Vishwa Vidyapeetham* e *Amrita School of Engineering, Coimbatore*, da Índia, com 14 e 13 publicações respectivamente; *Chinese Academy of Sciences* da China, com 12, *Consiglio Nazionale delle Ricerche* e *Università degli Studi di Padova*, e *Università degli Studi di Cagliari*, da Itália, com 12 e 10 e 10 respectivamente, *Macquarie University*, da Austrália, com 10, e por fim, *Birla Institute of Technology and Science*, da Índia, com 9 publicações. Os países com maior quantidade de publicações no mesmo período, segundo a *Scopus Elsevier*, foram: os Estados Unidos com 180 publicações; seguidos da Índia com

175 e a China com 163, conforme mostra a Figura 3.1. Foram encontrados 98% dos trabalhos escritos na língua inglesa, e apenas 01 trabalho no idioma português.

Inter-relações

Como resultado da pesquisa, será apresentada a seguir a inter-relação dos dados encontrados:

a) Um cenário geral das palavras-chave é apresentado na Figura 3.2, na qual é exibido um panorama geral da divisão dos documentos levantados para o campo do conhecimento relacionado ao tema *"malware analysis and machine learning"*, através do sítio "tagcrowd [5]". As palavras-chave foram obtidas a partir de 1.000 *abstracts* de trabalhos coletados na base da *Scopus Elsevier*. Dessa forma, é possível verificar que a concentração dos documentos está mais voltada para as áreas relacionados à plataforma Android, *machine learning*, *detection*, dentre outras.

Figura 3.2: Ocorrências de palavras. Fonte: https://tagcrowd.com/

b) A Figura 3.3, mostra, através de um mapa de calor, os autores com o maior número de citações coletadas da base da *Scopus Elsevier*, no espaço temporal pesquisado.

[5] http://www.tagcrowd.com

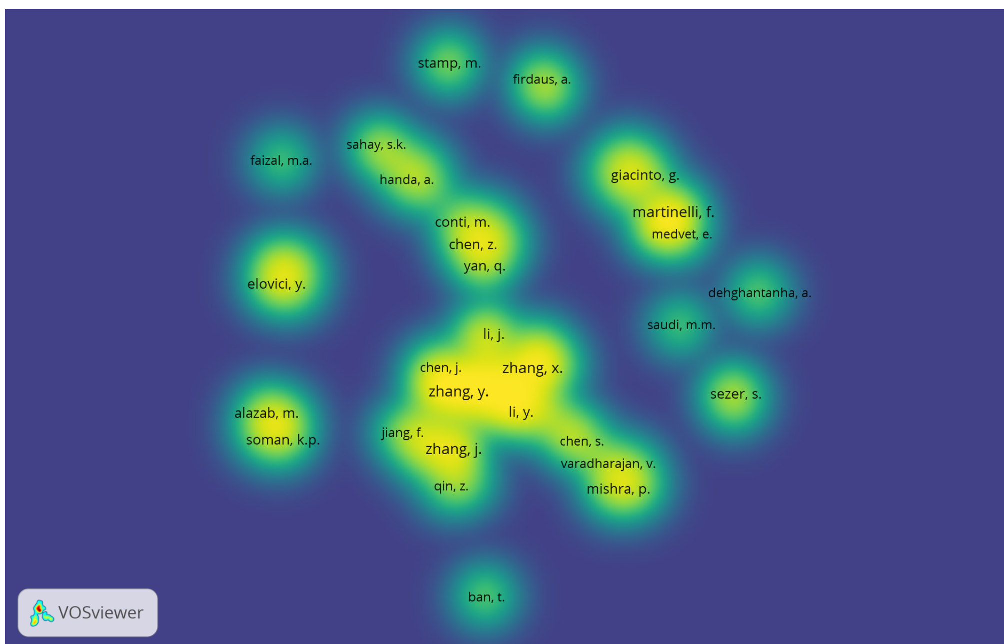

Figura 3.3: Autores com maior número de citações. Fonte: *Scopus Elsevier*

3.2.3 3ª Etapa - Análise dos dados Coletados

Nesta seção são apresentados os principais trabalhos que servirão de insumo para a pesquisa em fomento, evidenciados a partir das principais abordagens da literatura e das linhas de pesquisa sobre a temática escolhida pelo pesquisador. Com base no tema de pesquisa, verificou-se como relevantes os trabalhos descritos a seguir:

Como resultado da busca com o filtro "*Malware and static analysis and machine learning and Portable Executable*, objeto desta pesquisa, destacam-se os trabalhos de shalaginov *et al.* [21], "*Machine learning aided static malware analysis: A survey and tutorial*", com 13 citações, onde os autores apresentam uma visão geral sobre diferentes métodos de aprendizado de máquina para classificar *malwares* a partir de arquivos binários do *Windows* (PE32) com base em atributos estáticos extraídos do código-fonte, de um conjunto de dados de 131.072 arquivos obtidos do repositório Virusshare [93]. Foram extraídas as seguintes características estáticas: cabeçalho PE32, Bytes *n-grams*, *Opcode n-grams* e API *n-grams*. Os melhores resultados alcançados foram 97,63% de precisão, com base em todos os atributos extraídos dos arquivos de cabeçalho do PE32. Adicionalmente, o autor também apresenta um tutorial sobre como aplicar diferentes métodos de ML no conjunto de dados da classificação.

Destaque também para o trabalho de de Pham *et al.* [23], "*Static PE Malware Detection Using Gradient Boosting Decision Trees Algorithm*", com 3 citações, no qual os autores aplicam o método estático de detecção de malware pela análise de executável (PE) através do algoritmo *Gradient Boosting Decision Tree*". Os experimentos alcançaram uma

acurácia de 99,39%, com 1% de falso positivo, com um dataset de mais de 600.000 amostras para treino e 200.000 para teste oriundos do projeto *Endgame Malware Benchmark for research* (EMBER)[6].

Por fim, ressalta-se o trabalho de Poudyal [22], "*A Framework for Analyzing Ransomware using Machine Learning*", com 2 citações, no qual é proposta uma estrutura de engenharia reversa incorporando mecanismos de geração de atributos de aprendizado de máquina (ML) para detectar ransomware [27]. Para tanto, utilizou-se a ferramenta Linux *object-code dump tool* e um analisador (parser) para decodificar os binários (PE) a nível de instruções *assembly* e DLLs (*Dynamic Link Libraries*). Dos oito classificadores de ML testados, sete deles tiveram um bom desempenho com taxa de detecção de pelo menos 90%.

A busca na base da *Web Of Science* retornou 08 trabalhos, dos quais destaca-se o trabalho de Sun *et al.* [94], *"An Opcode Sequences Analysis Method For Unknown Malware Detection"*, com 3 citações, no qual os autores propõe um novo método para detectar características estáticas de arquivos *Windows Portable Executable* (PE), de 32 e 64 bits, por análise de sequências de *opcode*. Foi utilizado o *Term Frequency–Inverse Document Frequency* (TF–IDF) para calcular a frequência de cada sequência de código de operação (*opcode*). O método proposto mostrou resultados promissores com o algoritmo KNN e a rede neural de retropropagação (*backpropagation*) (BP). O dataset utilizado foi composto por mais de 20.000 amostras.

Trabalhos de Pesquisadores Brasileiros

A pesquisa no Brasil, associada à detecção e analise de *malwares*, tem entre os seus principais expoentes, o Prof. Paulo de Geus, da Universidade de Campinas, com 639 citações, seguido pelo Prof. André Grécio, da Universidade Federal do Paraná, com 438 citações e Marcus Botacim, com 69 citações no *Google Scholar*.

Especificamente na área de aprendizagem de máquina, destaca-se o trabalho elaborado pelo grupo de pesquisa liderado pelo Prof. Paulo de Geus e Marcus Botacin, com título, "Caracterização do Comportamento de Malware em Ambientes Linux - *Linux Malware Behavior Description*" [95], onde os autores exploram os desafios da classificação de *malwares* que afetam os sistemas operacionais baseados em distribuições Linux, e destacam os desafios desvantagens do desenvolvimento de classificadores de *malwares* baseados em ML para estes tipos de códigos maliciosos. Como resultados, aos autores descobriram que (i) os atributos dinâmicos superam os atributos estáticos; (ii) as características discretas apresentam menor variação de precisão; (iii) Conjuntos de dados apresentando características distintas impõem desafios de generalização aos modelos de ML; e (iv) que a análise

[6] https://github.com/elastic/ember

de atributos pode ser usada como informação de *feedback* para detecção de *malwares* e prevenção de infecções.

Outro trabalho associado à aprendizagem de máquina é apresentado pelo Prof. André Grégio *et al.* intitulado "*Shallow Security: on the Creation of Adversarial Variants to Evade Machine Learning-Based Malware Detectors*" [96], onde os autores abordam um método de exploração automática que consiste em mover as seções originais do código binário do *malwares* e inclui novos blocos de dados para criar amostras adversas que não apenas contornaram seus detectores de ML, mas também motores AV reais (com uma taxa de detecção mais baixa do que as amostras originais).

Segundo os autores, os ataques adversários se tornaram tão populares que uma grande empresa de Internet lançou um desafio público para encorajar os pesquisadores a contornar seus (três) detectores *malwares* baseados em análise estática de código e ML. Dessa forma, eles participaram desse desafio em agosto de 2019. Como resultado os autores reportam que conseguiram evitar a detecção em todos os 150 testes propostos pela empresa.

Por fim, outro trabalho que também explora o uso de aprendizagem de máquina para detecção de malware é apresentado por Silva *et al.* [97], intitulado "*Study on Machine Learning Techniques for Botnet Detection*", onde os autores apresentam um estudo sobre a utilização de técnicas da área de aprendizagem de máquina para a detecção de *botnets*, baseadas no protocolo IRC, através da análise de fluxos bidirecionais. os autores reportam terem obtidos bons resultados, tanto para estudos que utilizaram técnicas clássicas de ML, quanto para a detecção utilizando Redes Neurais Artificiais, onde obtiveram 98,8% de acurácia.

3.3 Outros Trabalhos Relacionados à Pesquisa

A literatura relata vários trabalhos baseados em ambos: algoritmos supervisionados e por algoritmos de aprendizado não supervisionado, utilizados para a classificação de malware tais como: *Decision Tree, Random Forest, K-Nearest Neighbors* (KNN), *Logistic Regression, Linear Regression, Support Vector Machines* (SVMs), entre outros [66, 16, 3, 50, 17, 67, 68].

Tabela 3.2: Quadro resumo dos trabalhos relacionados

Autor/ Ano	**Atributos**	**Classificadores**	***Dataset***	**Acurácia**
Kwon *et al.* (2020)	*Opcode* (n-gram)	*Deep Learning* (CNN *Networks*)	10.736 ASM *files*	98,74%
Jones (2019)	*Entropy*	KNN, DT, RF, SVM, NB e NC	60.000 *samples*	95%
Sun *et al.* (2019)	*op code sequence*	TF-IDF e Adaboost	20.000 *samples*	99,11%
Poudyal *et al.* (2019)	*Assembly instructions and* DLLs	*Random Forest*	9.339 *samples*	97,57%
Kumar *et al.* (2019)	68 features	LR, LDA, RF, DT, NB e KNN	5.200 *samples*	98,84%
Raff *et al.* (2018)	*Call Frequency, Opcode Frequency*	DT	2.011.786 *samples*	60%
Shalaginov *et al.* (2018)	PE32 *header, Bytes n-gram, Opcode n-gram,* API *Calls n-gram*	C4.5, KNN, ANN e SVM	131.072 *samples*	97,63%
Pham *et al.* (2018)	*Function Calls*	*Gradient boosting Decision trees*	800.000 records	97,57%
Cho *et al.* (2016)	API *calls*	*deep network* (SAE)	50.000 *samples*	95,64%
Singla *et al.* (2015)	*Opcode Frequency*	*Neural Networks*	1.230 *samples*	97,37%

Dessa forma, serão apresentados alguns trabalhos relacionados às propostas de detecção e classificação de *malware*, não elencados pela TEMAC, mas que servirão de subsídio para a experimentação prática, os quais são agrupados por análise dinâmica e estática.

Em relação à abordagem dinâmica, há muitos esforços para usar redes neurais para detecção e análise de malware, como por exemplo, o trabalho de Jones [3], que em sua tese de doutorado explora algoritmos de aprendizado de máquina supervisionados e aprendizado profundo (*deep learning*) para classificação de malware. Seu trabalho propõe um classificador chamado Malgazer, a partir do calculo entropia de aplicativos em execução. A pesquisa utilizou seis tipos de classificadores: *Decision Trees* (DT); *Random Forests* (RF); *Support Vector Machines* (SVM); *Naïve Bayes* (NB); *K-Nearest Neighbors* (KNN) e *Nearest Centroid* (NC). O conjunto de dados é composto por 60.000 amostras que foram

coletadas em virustotal[7]. A taxa de precisão alcançada pela Malgazer foi de aproximadamente 95%.

Do mesmo modo, Hardy *et al.* [64], abordam a arquitetura do *Deep Learning* para detecção de *malwares*, usando o modelo *stacked Auto Encoders* (SAEs). Os atributos são extraídos dos arquivos binários (PE) e com base nas chamadas de API. O conjunto de dados foi obtido da plataforma *Comodo Cloud Security Center*[8], composta por 50.000 arquivos de amostra (22.500 *malwares* e 22.500 arquivos benignos) e 5.000 arquivos desconhecidos, por isso compôs um conjunto de dados com 9.649 chamadas de API. A pesquisa atingiu uma precisão de 96,85%.

Existem também alguns trabalhos que propõem o uso de ferramentas de análise estática para extração de diferentes propriedades ou metadados (por exemplo, entropia, histogramas, comprimento da seção, etc) do código do *malware* [98]. Destaque para o trabalho de Kwon *et al.* [20], o qual propõe dois métodos de classificação de *malwares* conhecidos como métodos MCSP e MCSLT. No método de classificação *malware* usando codificação *Simhash* e método PCA (MCSP), os autores codificaram as sequências de *opcode* de arquivos ASM para a codificação *Simhash* e aplicaram o método PCA (usado para redução de dimensionalidade). Por outro, o método de transformação linear (MCSLT) aplica a camada LT à codificação Simhash. Os autores utilizaram, como *dataset*, o conjunto de dados do *Microsoft Challenge*[9], totalizando 10.868 arquivos ASM. Como resultado dos experimentos, o MCSP apresentou o melhor desempenho com um valor de 99,2% para 3 *n-grams*, enquanto o modelo MCSLT apresentou o melhor desempenho com um valor de 98,61% para 2 *n-grams*.

Outro exemplo é apresentado por Kumar [16], que propõe uma solução baseada na aplicação de uma combinação de campos de cabeçalho, valores brutos e valores derivados de arquivos executáveis (PE) extraídos do Clamp[10]. Um total de 2.488 amostras de *malware* e 2.722 amostras benignas foram divididas em dois conjuntos de dados (conjunto de atributos brutos e integrados). Vários algoritmos de aprendizado de máquina foram usados para classificação de *malwares*, tais como: *Decision Tree, Random Forest, K-Nearest Neighbors* (KNN), *Logistic Regression, Linear Regression, Support Vector Machines* (SVMs), juntamente com o método de validação cruzada (*Cross-Validation*). O melhor resultado foi obtido pelo algoritmo *Random Forest* com precisão de 97,47% e 98,78% no conjunto de atributos brutos e integrados, respectivamente.

Uma abordagem diferente é proposta por Raff *et al.* [15], que usam análise estática para obter bytes brutos do arquivo que são aplicados no sistema de detecção de *malwares*,

[7]https://www.virustotal.com/
[8]https://www.comodo.com/home/internet-security/updates/vdp/database.php
[9]https://www.kaggle.com/c/malware-classification/data
[10]www.github.com/urwithajit9/clamp

chamado “MalConv”. Ele usa modelos de rede neural para obter todo o código de bytes de arquivos executáveis, em vez de extrair e selecionar atributos. O conjunto de dados é composto por cerca de 400.000 arquivos. Os dados benignos originários são provenientes de uma instalação do *Microsoft Windows* e aplicativos relacionados, enquanto os arquivos maliciosos são provenientes do Virusshare[11]. Os melhores resultados alcançados foram uma precisão de 60% na fase de treinamento.

Por último, mas não menos importante, um exemplo simples de técnica de análise estática, com base na extração de atributos de arquivos do *Windows* PE, é apresentado por Singla [17]. Em sua proposta, os atributos extraídos são uma combinação da Frequência de Chamada de Função (*Function Call Frequency*) e da Frequência de Código (*Opcode Frequency*) para diferenciar *malwares* de arquivos benignos. O conjunto de dados é composto por 1.230 arquivos executáveis, dos quais 800 eram *malware* e 430 arquivos benignos. As amostras de *malwares* foram fornecidas pela *University of California*[12] e coletadas das fontes de dados online[13]. Enquanto os arquivos benignos foram coletados nos diretórios *System32* dos sistemas operacionais Windows 2000, Windows 2003, Windows XP e Windows 8. Foram utilizados os algoritmos de aprendizado de máquina disponíveis na biblioteca WEKA [14]. Como resultado, o classificador *Decison Tree* (DT) apresentou uma taxa de precisão mais alta de 97,37%, seguida pela *Forest Random* (J48), com uma precisão de 96,77%.

Embora todos os trabalhos apresentados tenham mostrado excelentes resultados, a presente pesquisa propõe uma abordagem diferente, que aplica uma técnica tradicionalmente usada em conjuntos de dados de alta dimensionalidade e abordagens *Deep Learning*, que será aplicada em conjuntos de dados de baixa dimensionalidade compostos de atributos de arquivos executáveis (PE), extraídos estaticamente. Tal a abordagem visa também reduzir a relevância individual dos atributos, e dessa forma, conceder melhores características de generalização ao modelo treinado, como uma forma de manter seu desempenho quando aplicado em um sistema de detecção de *malware* do mundo real. A Tabela 3.2 resume várias abordagens avançadas de detecção de *malware* com base em aprendizado de máquina para os arquivos PE do sistema operacional *Microsoft Windows*.

3.4 Considerações Finais

Neste capítulo foram apresentados os principais trabalhos na literatura sobre a análise de *malware*, coletados a partir da aplicação dos conceitos preconizados pela TEMAC. Além

[11]https://virusshare.com
[12]http://seclab.cs.ucsb.edu/
[13]http://www.nothink.org/honeypots/
[14]https://www.cs.waikato.ac.nz/ml/weka/

disso, foram elencados diversos outros trabalhos que servirão de subsídio para as experimentações práticas da abordagem proposta. Os trabalhos relacionados foram divididos em duas categorias de análise de *malware*: trabalhos relacionados ao aprendizado de máquina com análise dinâmica, e aprendizado de máquina com análise estática. Por fim, ressalta-se que, a quantidade trabalhos e os respectivos resultados alcançados demonstram que o tema é fortemente estudado na atualidade, e portanto, reforçam a relevância da pesquisa. No capítulo a seguir serão apresentados os procedimentos necessários aos experimentos realizados.

Capítulo 4

Procedimentos

Neste Capítulo são discutidos a abordagem proposta, a sequência de atividades, as ferramentas e as métricas de avaliação necessárias ao alcance dos objetivos da pesquisa. A visão geral da abordagem proposta é apresentada na Seção 4.1, e a sequência de atividades correlata, bem como as métricas de avaliação, são detalhadas na Seção 4.3. Na Seção 4.3 são descritas as ferramentas utilizadas e o motivo de cada escolha. A estrutura dos arquivos executáveis objetos da pesquisa é descrita na Seção 4.3, e as ferramentas utilizadas são detalhadas nas Seção 4.4. Por fim, a Seção 4.5 apresenta as considerações finais do Capítulo.

4.1 Abordagem Proposta

Nesta Seção é apresentada a visão geral do conjunto de procedimentos necessários à implementação dos experimentos de aprendizado supervisionado de máquina. Na Figura 4.1 é ilustrado o diagrama de visão geral do processo de aprendizado supervisionado de máquina da presente pesquisa. Nesta mesma Figura é possível observar uma sequência linear de atividades que tem inicio na coleção de amostras, passando por categorização em classes (malignos e benignos). Esse processo de categorização é fundamental para o aprendizado supervisionado de máquina.

Após a categorização, inicia-se o processo de extração de atributos, ou seja, de características relevantes dos arquivos para sua identificação como maliciosos ou benignos. Tais atributos comporão a entrada de dados para treinamento dos algoritmos de aprendizado de máquina. Uma vez composto o *dataset*, e antes de se iniciar o processo de treinamento, outra atividade importante é o processo de tratamento dos dados, o que inclui a limpeza dos dados e remoção de dados faltantes, nulos ou discrepantes.

A seguir, inicia-se o processo de treinamento com o algoritmos selecionados para a pesquisa, os quais são descritos na Seção 2.4.1. Dessa forma, o modelo de aprendizado

de maquina é construído, mediante o treinamento de algoritmos de classificação que tentam encontrar relações entre os atributos dos arquivos analisados e suas atribuições de classe [99].

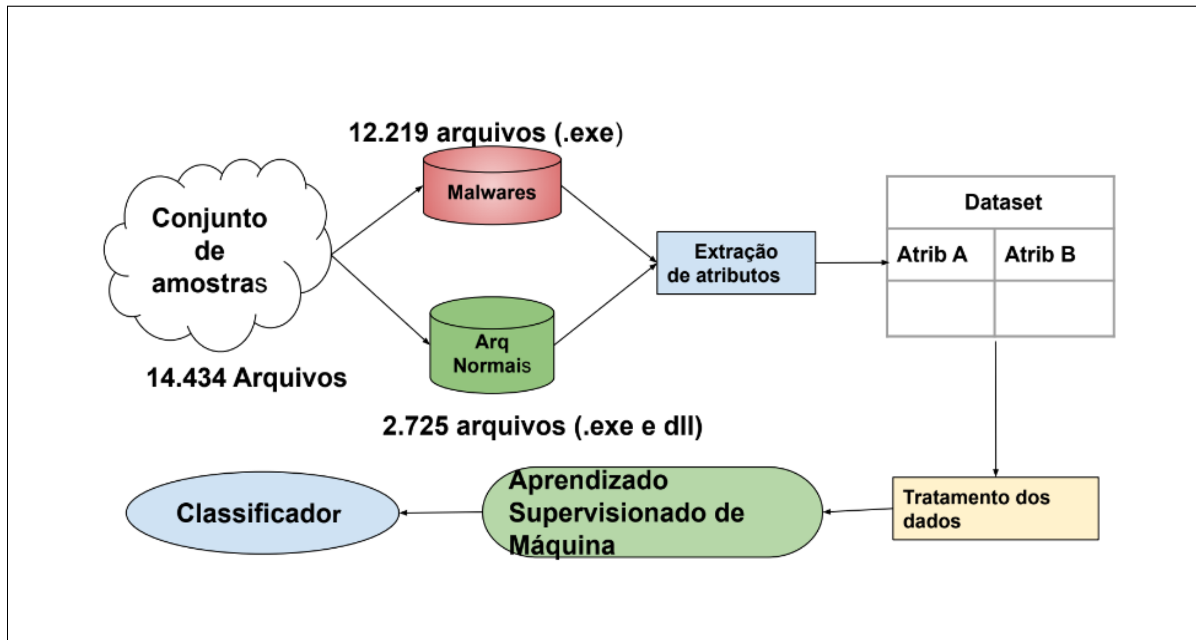

Figura 4.1: Diagrama de visão geral. Fonte: Adaptado pelo autor

4.2 Sequência de Atividades

Nesta seção detalhamos as atividades necessárias às experimentações práticas. Tais atividades são divididas em uma sequência de etapas, o que inclui os procedimentos de análise estática para minerar as características de arquivos binário, a coleta de dados para compor a base de treinamento em um banco de dados previamente rotulado. Além disso, identificamos algumas tarefas relacionadas e os procedimentos associados à extração e seleção de características estáticas de *malware* para a construção do modelo de aprendizado de máquina.

A figura 4.2 mostra as principais atividades referentes aos procedimentos desta pesquisa. Na primeira etapa, os arquivos executáveis (PE) são coletados para compor o conjunto de amostras. Tais amostras são então processadas para produzir um conjunto de dados brutos (*raw bytes*) em forma de relatório (vide Figura 4.14), através de um processo de desmontagem (*disassembly*). Para automatizar essa fase, usamos um conjunto de scripts *Python* (vide Seção 4.4). Em seguida, na segunda fase, esses relatórios são lidos por uma ferramenta (*parser*) para extrair os atributos elencados de cada relatório, os quais são armazenados em um arquivo "**.CSV*". Por fim, na última fase, implementou-se

um sistema de aprendizado supervisionado de máquina capaz de classificar os arquivos em duas classes: maliciosos ou benignos.

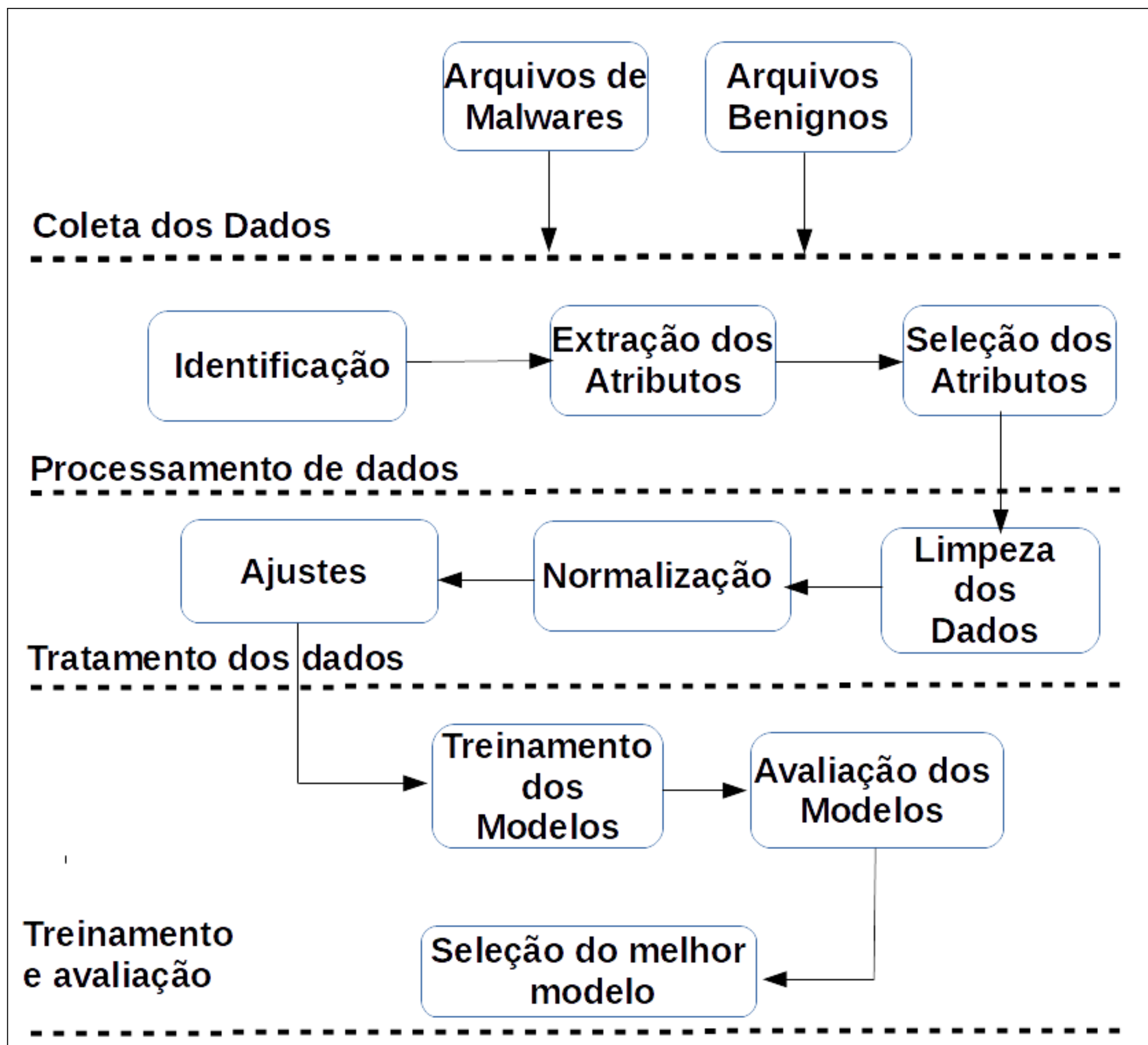

Figura 4.2: Principais atividades. Fonte: o autor

4.2.1 Coleta dos Dados

A presente pesquisa usa três conjuntos de dados para o treinamento do modelo, ambos compreendendo arquivos benignos e maliciosos. Estes *datasets* vêm de fontes diferentes. As amostras maliciosas foram obtidas do *corpus* do projeto Virusshare[1] [93], mediante uma solicitação prévia para acessar este repositório. Diferentemente, as amostras de arquivos benignos foram coletados de programas executáveis do sistema operacional *Windows* 7, procedimento semelhante ao adotado em outros estudos de pesquisa [17, 15].

[1] https://virusshare.com/

Também foram coletados arquivos benignos de diferentes repositórios de utilitários do sistema *Windows* (por exemplo, *Windows Sysnternal*) [2]. Nosso primeiro conjunto de dados de arquivos benignos contém 1066 ativos (arquivos .EXE ou .DLL) e 1075 malwares. De modo que o primeiro conjunto de dados é composto por 2.141 amostras, que representam um conjunto de dados balanceado ou seja, com uma distribuição uniforme entre as classes.

No entanto, para evitar o risco relacionado ao uso de um pequeno conjunto de dados, a pesquisa também explora um segundo *dataset* que compreende 10.829 ativos. Este é um *dataset* não balanceado que contém 9.600 *malwares* e 1.229 arquivos benignos que foram disponibilizados como um pacote de uma pesquisa anterior [61]. Ao final, foram coletados um terceiro conjunto de amostras, composto por 1.544 *malwares*, do projeto VirusShare, para avaliar se os modelos resultantes têm uma boa capacidade de generalização. Assim, nosso *dataset* disponível atinge um total de 14.434 amostras — 12.219 rotuladas como *malwares* e 2.725 rotuladas como benignos (ver Tabela 4.1).

Tabela 4.1: *Datasets* utilizados.

Dataset	Malwares	Benignos	Total
Dataset 1	1.075	1.066	2.141
Dataset 2	9.600	1229	10.298
Dataset 3	1.544	430	1.974
Total	12.219	2.725	14.934

4.2.2 Processamento dos Dados

Apos a coleta, algumas atividades são essenciais antes do início dos experimentos de aprendizado de máquina, a fim de se obter informações complementares, como por exemplo, a identificação do tipo de arquivo, além do processo de extração e seleção de seus atributos, os quais são detalhados a seguir, de acordo com o fluxo de execução.

Identificação

Quanto aos tipos de arquivos, a pesquisa ficou restrita à análise de arquivos PE do sistema operacional *Windows*. Dessa forma, o processo de coleta de dados visa também filtrar os dados de entrada, por meio da análise do tipo de arquivo, e identificar seu tipo, por meio de sua assinatura. Em um arquivo PE o campo de assinatura tem o valor “$x00004550$”, que traduzido para o código ASCII, é “$PE00$”. Cada arquivo PE começa com um pequeno executável MS-DOS, que pode ser identificado pelo campo “e_magic” de seu cabeçalho e

[2]https://docs.microsoft.com/en-us/sysinternals/

tem o valor “$0x5A4D$”. Em ASCII é conhecido como “*MZ*”, as iniciais de Mark Zbikowski, um dos criadores do MS-DOS [4].

O SO GNU/Linux implementa a biblioteca “*libmagic*”[3], via CLI, através do comando ‘*file*” que permite a identificação de tipos de arquivo. Por sua vez, *Python* implementa o módulo “*python-libmagic*”[4] para identificação de arquivos. Desta forma, construímos uma ferramenta chamada *separator.py*, que incorpora o ‘*libmagic*, para manter apenas os arquivos de formato PE no conjunto de dados. Um exemplo do *magic number* extraído pelo pecheck.py[5] pode ser visto na figura 4.3.

```
----------DOS_HEADER----------

[IMAGE_DOS_HEADER]
0x0        0x0    e_magic:                         0x5A4D
0x2        0x2    e_cblp:                          0x50
0x4        0x4    e_cp:                            0x2
```

Figura 4.3: PE *Magic Number*.

Extração de Atributos

Após coletar e identificar o arquivo binário, o próximo passo é extrair os atributos dos Arquivos PE, por meio de softwares especializados denominados ferramentas leitoras PE. Esses atributos são extraídos por meio de um processo de desmontagem que converte o código de máquina em um formato legível por humanos, denominado linguagem de código *assembly*, nossos dados brutos. Os atributos resultantes desse processo variam de chamadas de sistema e dependências de biblioteca a seccionamento e tempo de construção, entre outros. Uma ferramenta de linha de comando foi desenvolvida especialmente para esse fim. Ela incorpora três ferramentas pré-existentes (*PEframe.py*, *PEcheck.py* e *PEscanner.py*) para a desmontagem dos binários.

O processo de *disassembly* ou desmontagem, é processo inicial para quase todos os métodos de análise estática de códigos maliciosos. Desse processo normalmente resulta informações tais como: cálculo de seu resumo criptográfico (*hashes*) de arquivos e de suas seções, tempo de compilação (*timestamp*), sequências de caracteres, dependências de bibliotecas e chamadas de função, dentre outras. Além dos campos de cabeçalho, vários outros atributos estáticos podem ser extraídos de um executável binário, tais como chamadas de funções, entropia e tamanho das várias seções, etc. Dessa forma, para que sejam aplicadas no aprendizado de máquina, as informações devem ser primeiro extraídas do arquivo binário através de um software usado para converter o código da máquina em

[3]https://linux.die.net/man/3/libmagic
[4]https://pypi.org/project/python-libmagic/
[5]https://github.com/jivoi/pentest/blob/master/tools/pecheck.py

um formato legível por humanos, chamada de linguagem de montagem (*assembly language code*).

Após o processo de extração, as amostras binárias foram transformadas em arquivos brutos, dos quais, os atributos selecionados são capturados e armazenados em um arquivo CSV para compor os experimentos *dataset*. Durante este processo, foram selecionados 35 atributos ou características, entre os quais 26 eram numéricos: (*CreateFile, CreateProcess, CreateRemoteThread, CreateService, DeleteFile, GetModuleHandle, GetProcAddress, HttpSendRequest, InternetConnect, LoadLibrary, ReadProcessMemory, ShellExecute, StartService, URLDownloadToFile, imports, OpensProces, antidbg_len, code_entropy crypto, data_entropy, entropy, filesize, packer_len, sections, urls_len, yara_len*); 6 eram atributos textuais: (*antidbg, crypto, packer, filetype, url, yara*); e, finalmente, 3 atributos vinculados a valores de *hash* (*hash_md5, has _sha1, imphash*) foram descartados, porque são valores únicos para cada arquivo e não podem ser usados para extração de médias, frequência, etc. A tabela 4.2 mostra os atributos resultantes do processo de extração, agrupados por categorias definidas para a presente pesquisa.

Os dados coletados foram submetidos a um processo de análise exploratória para se avaliar a qualidade dos dados e a sua possível importância no processo de aprendizado do classificador. Os dados coletados foram agrupados nas seguintes categorias: informações do arquivo, chamadas de função (*Functions Calls*), campos de entropia, tamanho do arquivo, dentre outros. Para resumir, todos esses conjuntos de atributos podem ser agrupados nas seguintes categorias:

- Chamada de Função dos Sistema - também referenciadas como chamadas de API[6] (*System Functions Calls*) revelam o comportamento dos programas e podem ser consideradas uma marca essencial na detecção de malware [100, 22, 18]. Por exemplo, chamadas de API do Windows para as funções *WriteProcessMemory*, *LoadLibrary* e *CreateRemoteThread* são comportamentos suspeitos usados por malware para injeção de DLL em um processo [51]. Por isso, o agrupamento de chamadas de função em conjuntos lógicos podem fornecer informações detalhadas sobre o comportamento do programa malicioso, pois contém informações sobre as operações do programas. Dessa forma, foram selecionadas 14 chamadas de função, as quais são reportadas na literatura como relevantes na identificação de malware: *CreateFile, CreateProcess, CreateRemoteThread, CreateService, DeleteFile, GetModuleHandle, GetProcAddress, HttpSendRequest, InternetConnect, LoadLibrary, ReadProcessMemory, ShellExecute, StartService, URLDownloadToFile* [56, 17, 101, 102]. A Figura 4.4 apresenta o gráfico de frequência com a distribuição das chamada de função entre as amostras benignas e maliciosas, tendo como referência o *Dataset 1*. A diferença

[6] Application Programming Interface

Tabela 4.2: Conjunto de atributos selecionados.

Nr Ordem	Categoria	**Descrição**	**Tipo**
1	Chamadas de Função	CreateFile	cont
2		CreateProcess	cont
3		CreateRemoteThread	cont
4		CreateService	cont
5		DeleteFile	cont
6		GetModuleHandle	cont
7		GetProcAddress	cont
8		HttpSendRequest	cont
9		InternetConnect	cont
10		LoadLibrary	cont
11		OpensProcess	cont
12		ReadProcessMemory	cont
13		ShellExecute	cont
14		StartService	cont
15		URLDownloadToFile	cont
16	Técnicas Anti-debug	antidbg	list
17		antidbg_len	num
18	Campos de Entropia	code_entropy	num
19		data_entropy	num
20		entropy	num
21	Identificação do Arquivo	Filetype	string
22		filesize	num
23	Presença de Criptografia	crypto	cont
24	Cálculo de Hash	hash_md5	string
25		hash_sha1	string
26		imphash	string
27	Tabela de Importação	imports	cont
28	Empacotadores	paker	lista
29		packer_len	num
30	Quantidade de Seções	sections	cont
31	Presença de URLs	urls	list
32		urls_len	num
33	Presença de Regras Yara	yara	list
34		yara_len	num

da frequência entre amostras benignas e maliciosas é indicativo da importância das chamadas de API para o processo de treinamento dos modelos.

- Técnicas Anti-debug - certos malwares são codificados para se comportarem de maneira específica quando detectam um analisador no ambiente de execução, ou quando são executados em um ambiente virtualizado [101]. Tal característica é chamada de técnica *anti-debug*. Dessa forma, o atributo *antidbg* indica a quantidade dessas técnicas dentro do código do binário.

- Entropia (*entropy*) - se refere a medida de incerteza em uma série de números ou bytes, ou seja o nível de dificuldade ou da probabilidade de predição de um determinado dado [103, 3, 101]. A entropia é medida entre 0 e 8, medida que descreve a aleatoriedade dos dados [34]. A literatura reporta que um valor alto de entropia

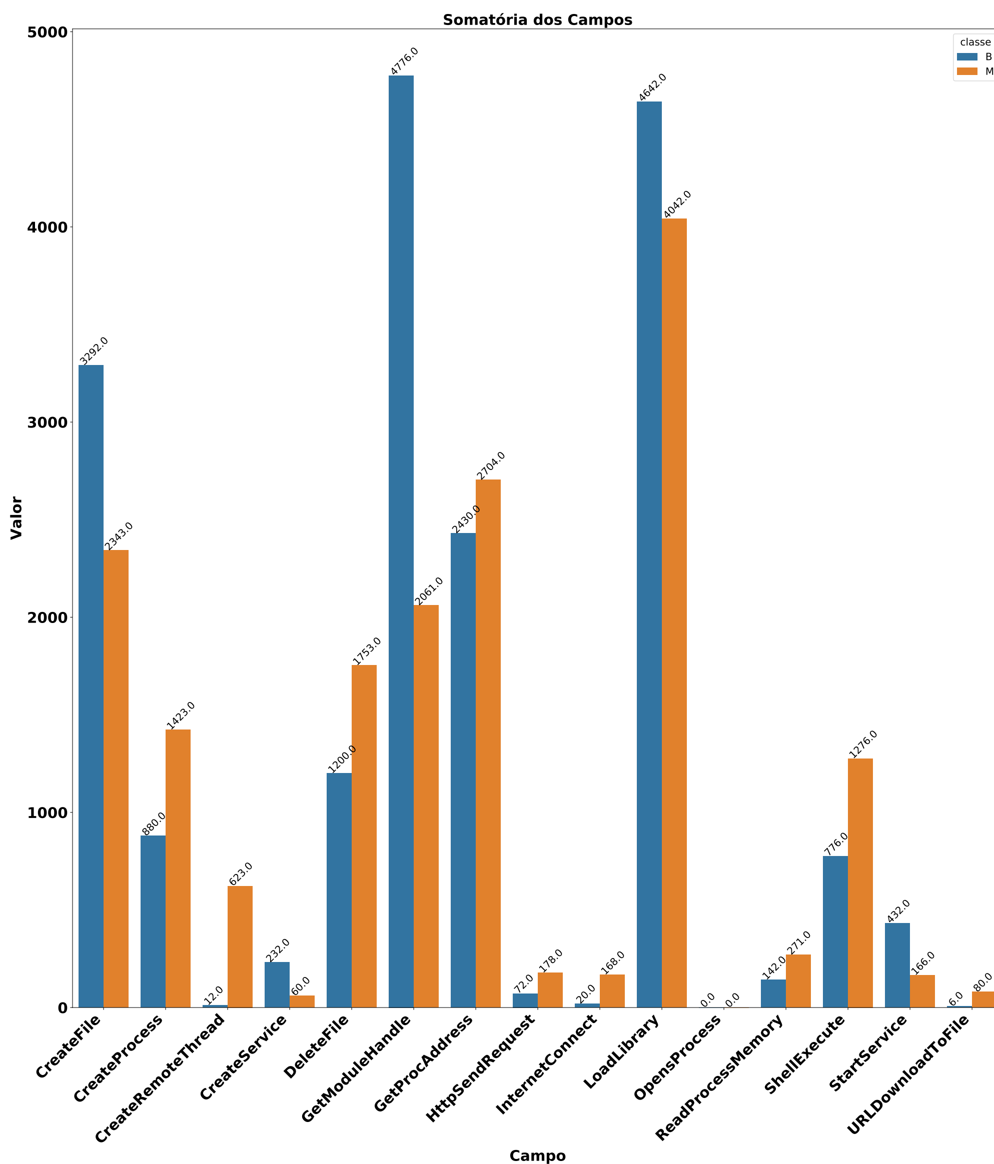

Figura 4.4: Frequência das Chamadas de Função.

indica uma maior probabilidade de dados codificados (presença de empacotadores) ou criptografados [104, 105, 34].

Com tais propriedades inerentes à medição do grau de compactação, a entropia é um recurso muito confiável para verificar o comportamento malicioso. Por exemplo, um arquivo com entropia maior que 7 é considerado suspeito [11]. Desa forma, a entropia é utilizada na literatura de diversas formas para a detecção de malware [3, 16, 21, 90, 106, 105, 50, 101]. A Figura 4.5 apresenta o gráfico de barras com a

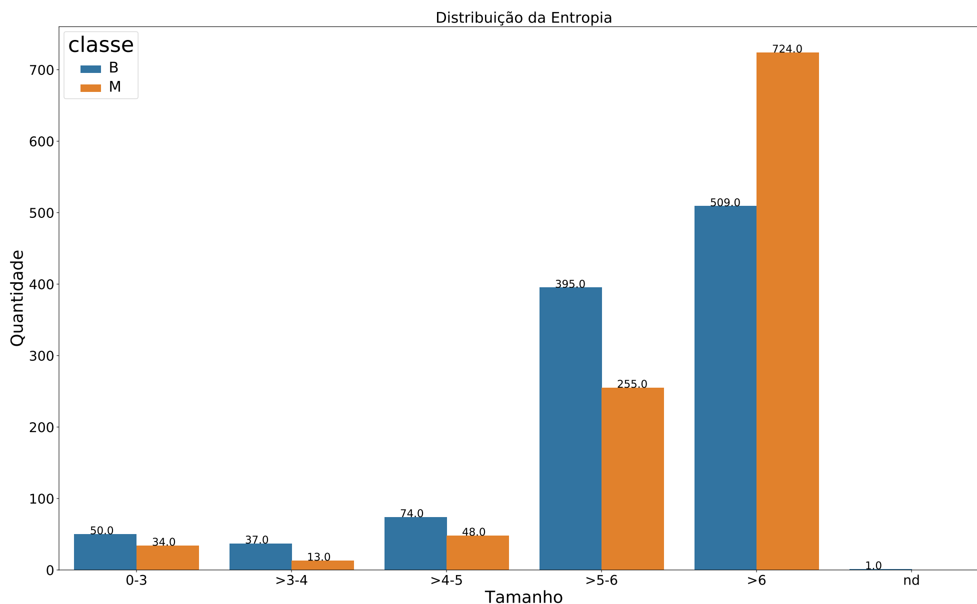

Figura 4.5: Distribuição da entropia.

distribuição da entropia de acordo com a sua classe (maliciosos ou benignos) em uma escala de 0 a 8, presente no *Dataset 1*. Na referida figura verifica-se uma maior distribuição da entropia de valores acima de 6 entre os arquivos maliciosos, representados na cor laranja, em relação ao arquivos benignos, representados na cor azul.

- Identificação do Arquivo (*File type*) - a identificação do tipo de arquivo é de fundamental importância no escopo da pesquisa, a fim de não gerar ruídos no conjunto dados de entrada para o treinamento dos modelos de aprendizado de máquina. Da mesma forma, o seu tamanho é usado como um atributo, considerando-se uma suposição inicial de que há uma grande diferença entre o tamanho do arquivo, de malware e programas benignos [16]. Dessa forma, o *file size* foi considerado um atributo para compor o conjunto de treinamento, apesar da similaridade de tamanho entre os arquivos benignos e maliciosos que compõe o *Dataset 1*, conforme se verifica no histograma apresentado na Figura 4.6.

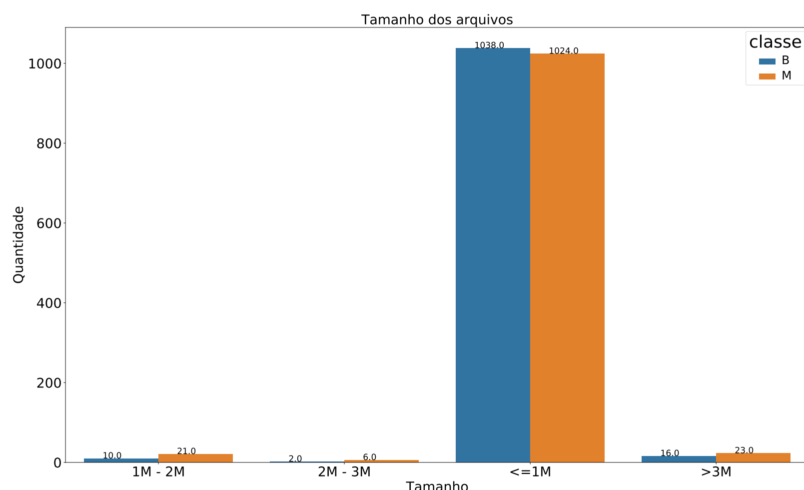

Figura 4.6: histograma do tamanho dos arquivos em bytes.

- Presença de Criptografia - a criptografia é outra ferramenta útil para ofuscação, e que pode ser usada pelo desenvolvedor do aplicativo antes da instalação ou em tempo de execução [54]. Desse modo, o objetivo do atributo *crypto* é identificar a presença de técnicas de criptografia em partes do código do binário analisado.

- Cálculo de *hash* - a aplicação da função *hash* é semelhante a uma soma de verificação de arquivo (*file checksum*), pois produz um pequeno valor relacionado e busca garantir a integridade de um arquivo [107]. A utilização de *hash* na análise de malware tem sido citado em diversos trabalhos [3, 108, 105, 41]. Na pesquisa foram extraídos *hash* a partir de três algoritmos diferentes: md5[7], sha1 e sha256 [8]. No entanto, os *hashes* extraídos não foram utilizados nos experimentos, por considerar que esses seriam adequados para uso em uma abordagem diferente do escopo desta pesquisa.

- Tabela de Importação - também referenciada como *Import Directory Table* [107], contém a tabela de endereços de informações, também chamada de *entry points*. Basicamente, a tabela de importação de diretórios consiste em uma matriz de entradas de diretório, com uma entrada para cada DLL carregada pelo arquivo executável e o seu respectivo endereço de memória. a Tabela de importação é representada pelo atributo *imports*, o qual contém a quantidade de entradas presente na referida tabela.

[7]https://tools.ietf.org/html/rfc1321
[8]https://csrc.nist.gov/Projects/Hash-Functions/NIST-Policy-on-Hash-Functions

- Empacotadores - são programas de compactação que geram um arquivo executável e geralmente são usados na compactação de malware [3], sendo o UPX[9] mais comumente encontrado segundo Kaur et al. [56]. Um programa empacotador ou "*Packer*" transforma um arquivo executável, por exemplo, um binário de malware, em uma representação sintática diferente, mais semanticamente equivalente, com o objetivo de ofuscar ou criptografar o binário original, armazenando o resultado desse processo em um novo executável [34].

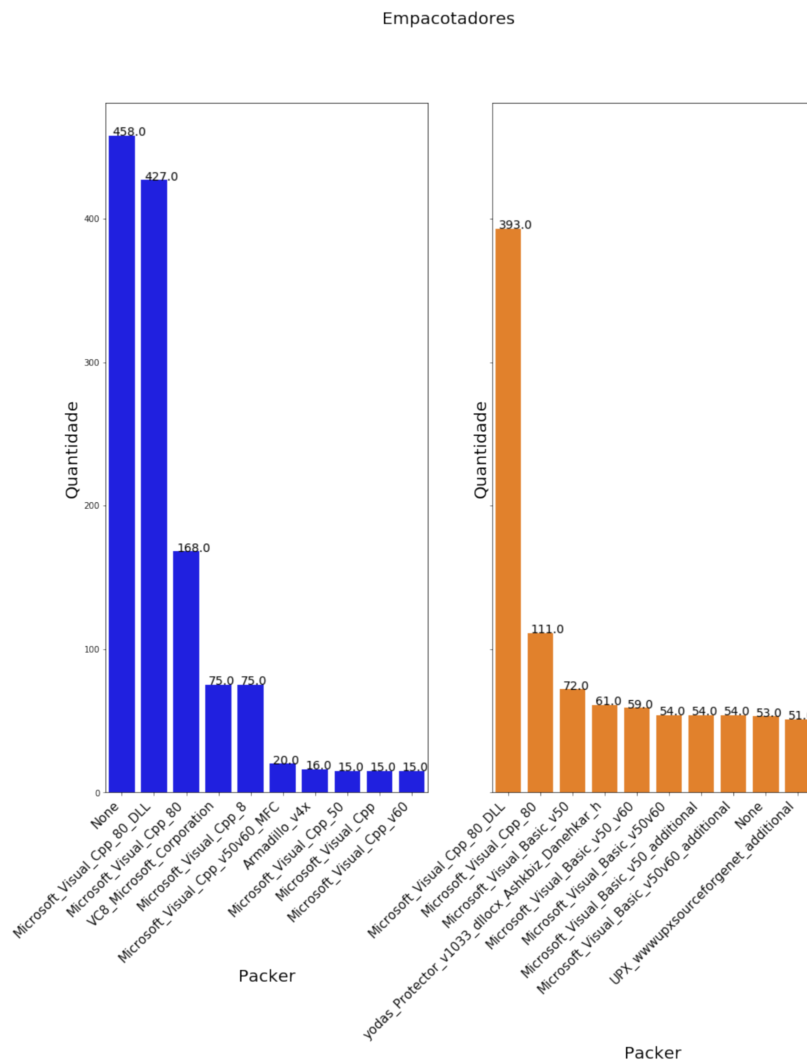

Figura 4.7: Presença de empacotadores.

A importância desse atributo tem sido reportado em diversos trabalhos [109, 105, 33, 110, 55, 16]. Na presente pesquisa foram elencados 02 atributos: os tipos de

[9]https://upx.github.io

empacotadores (*Packer*) e a quantidade (*Packer_len*) desses no código analisado. No entanto, nos experimentos 1 e 2 apenas o atributo *Packer_len*, que por ser um dado numérico foi utilizado, em detrimento ao atributo *Packer* que, por ser um atributo textual, foi apenas usado no experimento 3 (TF-IDF). A Figura 4.7 apresenta a distribuição da presença empacotadores nas duas classes de arquivos analisados (maliciosos e benignos) presentes no *Dataset 1*.

- Quantidade de Seções - as seções de um arquivo PE (*PE file section*) armazenam todos os códigos ou dados presentes no binários. Um arquivo binário tem um número variável de seções [100], devendo conter pelo menos 2 seções: a seção de código e a seção de dados [4]. Em um arquivo PE, há sempre um tipo de seção de código, porém, pode haver vários tipos de seção de dados. Na pesquisa foi analisada a quantidade de seções existentes, através do atributo *NumberofSections* de cada binário [16], extraído do cabeçalho do PE (*COFF File Header*).

- Presença de URL - pode indicar que o binário, quando executado, possa tentar estabelecer uma conexão com um servidor remoto para instalar outros códigos [111]. Tal característica é comum em malware conhecidos como *Trojan Downloader*[10]. Dessa forma, o atributo *url_len* se refere à quantidade de endereços de URL diferentes encontrados no arquivo binário [21]. A Figura 4.8 apresenta a distribuição da presença de URL nos arquivos maliciosos e benignos do *Dataset 1*, onde se verifica uma distribuição equilibrada na quantidade de URLs presentes nos arquivos benignos e maliciosos.

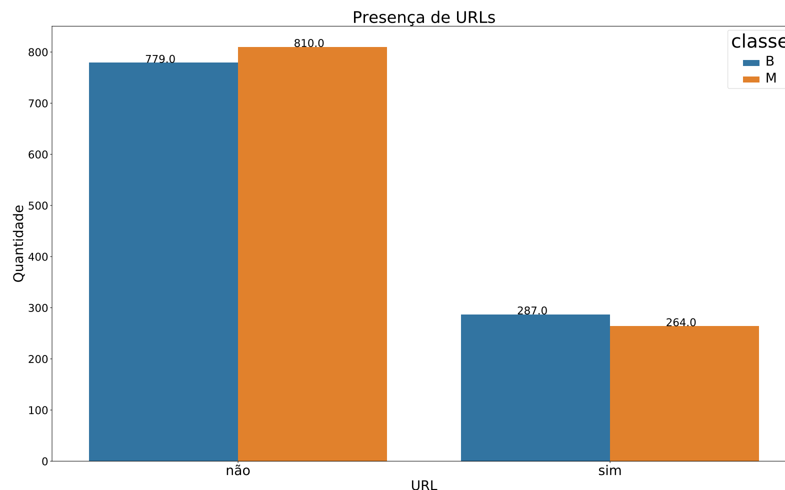

Figura 4.8: Histograma que mostra a frequência de URLs.

[10]https://cartilha.cert.br/malware/

- Presença de Regras YARA - YARA[11] é uma ferramenta multiplataforma, compatível com ambientes Windows, Linux e Mac OS X, e que também pode ser usada por linha de comando ou scripts em *Python*. a ferramenta YARA permite criar descrições (regras ou assinaturas) de famílias de malware ou de qualquer outra caraterísticas, sejam em padrões textuais ou binários. As regras YARA são muito usadas na comunidade de análise de malware como ferramenta de detecção [3], por isso, a presente pesquisa utiliza o atributo *yara_len* para quantificar as assinaturas encontradas nos binários analisados.

Seleção de Atributos

A seleção de atributos é o processo de agrupar o conjunto mais significativo de atributos a fim de reduzir a dimensionalidade e gerar maior resultado analítico [112]. Tal processo é fundamental, considerando que um sistema de *Machine Learning* só será capaz de aprender se os dados de treinamento contiverem atributos relevantes e suficientes [66]. Dessa forma, após as etapas de extração e limpeza de dados, é necessário eliminar os atributos redundantes, ou aqueles que não influenciam positivamente na classificação, ou que podem diminuir a precisão e qualidade do modelo [21].

Para isso, o conjunto de atributos é submetido a um processo de análise exploratória para avaliar a qualidade dos dados e sua possível importância no processo de aprendizagem do classificador. Por exemplo, na seleção de atributos, uma característica a ser observada é o seu grau de correção ou variância, o que designa a qualidade do atributo, ou seja, sua capacidade de distinguir entre executáveis maliciosos e benignos.

Adicionalmente, algumas medidas estatísticas podem ser tomadas para avaliar a importância dessas características, por exemplo o scikit-learn [2] possui a biblioteca "*VarianceThreshold*" para seleção de atributos, que permite a remoção daqueles que apresentarem uma baixa variância. Por padrão, tal método remove todos os atributos com zero variância. Outra possibilidade é a seleção dos melhores atributos com base em testes estatísticos univariados através do método "*transform*", o qual possui as seguintes opções:

- *SelectKBest* - o qual remove todos os atributos com escore menor que um k determinado;
- *SelectPercentile* - remove todos os atributos com k percentagem da variância de todos os atributos; e
- *GenericUnivariateSelect* - permite a seleção dos melhores atributos a partir do uso da opção "*hyper-parameter search estimator*".

[11] http://virustotal.github.io/yara/

O scikit-learn [2] oferece também a possibilidade de remoção dos atributos menos importantes, através de processo recursivo chamado de RFE (*Recursive feature elimination*), que utiliza o recurso de *cross-validation* para encontrar o número ideal de atributos.

Durante o processo de seleção, foram separadas 5 colunas cujo tratamento exige a aplicação de algoritmo de processamento de texto, tal como o TF-IDF (conforme Subseção 5.1.4). Deste modo, o conjunto de dados para os experimentos 1 e 2 foi composto de atributos quantitativos, sejam discretos (booleanos) ou contínuos. As colunas *hash e imphash* também foram removidas, pois, apesar de serem atributos numéricos, são valores únicos e que não poderiam ser aplicados em funções estatísticas, tais como extração de médias, sumarizações, etc.

Dessa forma, ao final do processo de seleção, de 35 atributos inicialmente coletados (como visto na Subseção 4.2.1), foram selecionados 28 dentre eles (24 numérico e 4 atributos textuais). Os demais atributos foram descartados por diferentes motivos, tais como os atributos *filetype e urls*.

4.2.3 Tratamento dos Dados

Algumas técnicas de tratamento dos dados mudam a forma como os dados são representados apenas para torná-los mais compatíveis com certos algoritmos de aprendizado de máquina, tal como normalização e limpeza [113]. Ambas as técnicas enfocam a transformação de uma característica individual de alguma forma. Dessa forma, antes de iniciar os experimentos, esses processos são extremamente importantes para a confiabilidade dos resultados, o primeiro é a limpeza dos dados e o segundo é a normalização.

Limpeza dos dados

Limpar os dados significa essencialmente remover do conjunto de treinamento as amostras que podem levar o classificador a apresentar resultados não confiáveis [99]. Em geral, os conjuntos de dados precisarão de uma limpeza, a fim de se manter apenas os atributos necessários [84].

A qualidade dos dados também pode ser afetada por atributos redundantes (valores duplicados, os quais devem ser removidos e também por dados faltantes (*missing values*) [114]. Uma das técnicas para lidar com dados faltantes é através da técnica de imputação de dados [113]. A imputação substitui os valores dos atributos ausentes por um valor estimado, por exemplo, uma medida da tendência central desse recurso, tal como media ou mediana.

Dessa forma, os dados são limpos com a substituição dos dados nulos e ausentes por um valor médio de cada atributo, e os dados foram preparados convertendo a criptografia da

coluna em booleano, atribuindo “1” para quando há um valor, e “0” para a ausência de um valor. Também foi aplicado um para substituir todos os registros pelos valores de “$None$” e “NaN” pela média da respectiva coluna. Essas médias são armazenadas nos objetos mean_malw e mean_benign para serem usadas na fase de treinamento dos modelos. Foram também removidos os atributos *code_entropy* e *data_entropy*, por apresentarem valores faltantes acima de 10%.

Por fim, considerando-se que o aprendizado supervisionado consiste em aprender o vínculo entre dois conjuntos de dados: os dados observados “x” e uma variável externa “y” que se está tentando prever, chamada de “*target*” ou “*labels*” [2]. Dessa forma, foi criado o atributo (coluna) “classe” em cada base para receber o rótulo (label “M” (*malwares*) ou “B” (benignos) respectivamente. Assim, o atributo “classe” foi povoado com valores “0” para arquivo de benignos e “1” para arquivos maliciosos respectivamente. Finalmente, os atributos numéricos são todos convertidos para *float*.

Normalização dos dados

Para evitar que a variação dos valores dos atributos possa afetar os resultados, é aplicada também uma técnica de normalização [84]. Normalização é o processo de escalar valores de modo a tornar os atributos igualmente significativos [50], ou seja, reduzir a variância de valores que podem afetar negativamente o treinamento do modelo.

Da mesma forma, é importante lidar com os chamados *outliers*[12], ou seja, valores que estão muito distantes da tendência central [113], ou dados com valores extremos, ou fora de uma distribuição normal. Um exemplo disso é a presença de *outliers* e variáveis fortemente inclinadas, as quais podem distorcer os resultados [84]. Alguns métodos de aprendizado de máquina, tais como o método redução de dimensionalidade PCA (descrito na Subseção 2.4.2), podem ser afetados pela presença de *outliers*. Portanto, sempre que possível, os *outliers* devem ser removidos do conjunto de dados [75].

Ajustes

Dessa forma, para o treinamento dos modelos, várias técnicas e ajustes de parâmetros foram aplicados aos modelos, de acordo com experimentos anteriores na literatura [66]. A estratégia de seleção de modelos para algoritmos de aprendizado de máquina geralmente envolve a otimização numérica de um critério que permite a seleção do modelo apropriado [115].

Dessa forma, alguns ajustes são necessários para um melhor desempenho e para se obter uma característica de generalização, além de outras correções necessárias. Neste

[12]https://mathworld.wolfram.com/Outlier.html

trabalho, são utilizadas varias técnicas para ajustes, tais como a validação cruzada (*Cross-Validation*) [3] e *grid search* [2], com ajustes de hiperparâmetros, para reduzir a possibilidade de *over-fitting* [116], as quais são detalhadas a seguir:

- ***Over-fitting***: Um algoritmo de ML é geralmente treinado com apenas uma parte do dados de treinamento, enquanto outra parte é reservada para a fase de testes [117]. No entanto, há o risco de que ruídos possam ser inseridos durante a fase treinamento fazendo com que o algoritmo apenas memorize características do conjunto de treinamento ao invés de encontrar uma regra geral de predição.

  Consequentemente, um modelo que apenas repete os rótulos das amostras de treinamento irá falhar na predição de novos dados [116], ou seja, as métricas de avaliação não se refletem na generalização do modelo (*over-fitting*) [2]. Para minimizar tal risco, várias técnicas têm sido referenciadas na literatura, tais como a divisão do dataset em 03 partes, uma para treinamento, outra para testes e a terceira para validação [2]. Outra possibilidade é aplicar a técnica chamada de *Cross-Validation* (CV).

- ***Cross-Validation***: Para se avaliar como os resultados obtidos podem ser generalizados em um dataset independente, os processos de aprendizado de máquina são geralmente apoiados em cenários de validação cruzada ou CV [99]. A validação cruzada serve para atenuar qualquer viés na partição dos dados, além de maximizar os resultados estatísticos obtidos nas interações. A Figura 4.9 mostra um exemplo do processo de *cross-validation*, com um 5-*fold*, ou seja, uma validação onde o conjunto de dados é particionado em cinco subconjuntos iguais, os primeiros 4 grupos de dados são alocados para o treinamento e o último para testes [3].

- **Hiperparâmetros**: Outro desafio do ML é encontrar os melhores parâmetros para o modelo, uma vez que cada algoritmo de aprendizado de máquina requer uma variedade de *constraints*, taxas de aprendizado, além de outros fatores que podem afetar a performance do modelo. Tais fatores são chamados de hiperparâmetros [115].

- ***Grid Search***: É implementado como uma pesquisa exaustiva, em contraste com a otimização aleatória de parâmetros, onde, depois que todas as combinações possíveis de parâmetros para um modelo forem avaliadas, os melhores parâmetros para o conjunto de dados serão mantidos [76]. Tal processo, chamado de *grid Searching* [3], é implementado no *scikit-learn* pela biblioteca "GridSearchCV" [2].

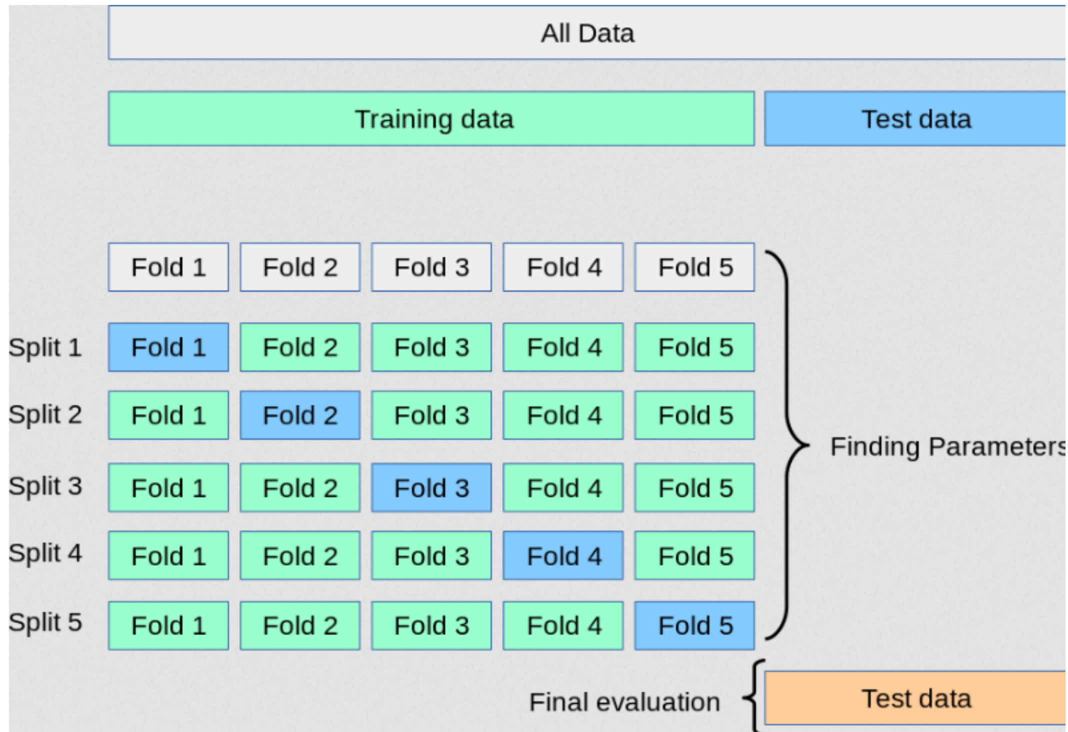

Figura 4.9: Exemplo de Cross-validation [2].

4.2.4 Treinamento e Avaliação dos Modelos

Após as fases coleta, processamento e tratamento dos dados, esta seção apresenta a fases de treinamento e avaliação dos modelos, para a construção de um classificador. Dessa forma, a tarefa de classificação consiste em descobrir uma função que mapeie um conjunto de registros em um conjunto de classes. Esse processo de treinamento e avaliação de desempenho dos modelos é descrito a seguir.

Treinamento dos Modelos

De acordo com o escopo desta pesquisa, para realizar os experimentos, treinamos e testamos nossas amostras usando 5 diferentes algoritmos de classificação de aprendizado de máquina: *K-Nearest Neighbors* (KNN), *C-Support Vector Classification* (SVC), *Gaussian Naive Bayes* (GaussianNB), *Decision Tree* (DT), e *Random Forest* (RF), referenciados em trabalhos correlatos como relevantes para o aprendizado supervisionado de máquina, conforme apresentado na Subseção 2.4.1.

Como os algoritmos aprendem com dados, eles encontram relacionamentos, desenvolvem compreensão, tomam decisões e avaliam sua confiança a partir dos dados de treinamento que recebem. Dessa forma, para o treinamento dos algoritmos, os dados são divididos em dois conjuntos, um com os atributos previsores e outro com o atributo alvo. Para cada valor do atributo alvo há uma classe, a qual normalmente possui um rótulo

categórico que diferencia entre malicioso ou benigno. Assim, esses algoritmos aprendem a ligação entre dois conjuntos de dados: os dados observados x e uma variável externa y que se está tentando prever, chamada de "*target*" ou "*labels*" [2], que recebeu o rótulo "1" (para malware) ou "0" (para benigno), respectivamente. Depois disso, outro passo importante é dividir seu conjunto de dados em dois subconjuntos: um conjunto de treinamento e um conjunto de teste.

Avaliação dos Modelos - Métricas e Ajustes

Como forma de se avaliar o processo de treinamento dos modelos, elencaram-se algumas métricas a serem aplicadas nos dados obtidos a partir da experimentação e da análise dos dados coletados. Kaner [118] define métrica como uma função de medição. Dessa forma, para mensurar os resultados e obter subsídios para avaliar os resultados obtidos, foram definidas métricas que estão relacionadas à mensuração de resultados referenciadas na literatura. Tais métricas foram coletadas dos diferentes experimentos que compõe a abordagem proposta, conforme descrito na Seção 5.1.

Para aplicação das métricas em uma classificação binária, os termos positivos e negativos se referem a predição do classificador, enquanto os termos verdadeiro e falso, se referem ao quanto a predição corresponde a um julgamento externo a partir da aplicação de métricas que são funções que avaliam o erro de uma predição [2]. Assim, ***True Positive*** **(TP)** é o número de arquivos maliciosos corretamente classificados como *malware*, ***False Positive*** **(FP)** é o número de arquivos benignos incorretamente classificados como *malware*, ***False Negative*** **(FN)** é o número de arquivos maliciosos classificados incorretamente como benignos e ***True Negative*** **(TN)** é o número de arquivos benignos corretamente classificados como benignos [66, 2, 75].

Considerando que um detector ou classificador de *malware* deve ter uma taxa de verdadeiro positivos (TPR) muito alta e uma baixa taxa de falso positivos (FPR) [90], o desempenho dos modelos serão avaliados a partir da combinação de 4 possíveis resultados:

1. Taxa de Verdadeiro Positivo (*True Positive Rate* - TPR), também referenciada como sensibilidade (*sensitivity*) [2], refere-se à proporção de arquivos maliciosos classificados corretamente como *malware* dentre todos aqueles que são maliciosos;

2. Taxa de Falso Positivo (*False Positive Rate* - FPR), refere-se à proporção de arquivos benignos classificados incorretamente como *malware* dentre todos aqueles que são benignos;

3. Taxa de Verdadeiro Negativos (*Negative Positive Rate* - TNR), também referenciada como especificidade (*specificity*) [2], refere-se à proporção de arquivos benignos classificados corretamente dentre todos aqueles que são benignos;

4. Taxa de Falso Negativos (*False Negative Rate* - FNR), refere-se à proporção de arquivos maliciosos classificados incorretamente como benignos dentre todos aqueles que são maliciosos.

Baseadas nas combinações acima, a avaliação de desempenho da presente pesquisa considera as seguintes métricas:

Accuracy - a métrica de avaliação chamada de *Accuracy*(Acc), corresponde ao número de previsões corretas dividido pelo número total de previsões, e que fornece a confiabilidade geral do classificador [2], podendo ser definida pela equação (4.1) a seguir:

$$Acc = \frac{TP + TN}{TP + FN + TN + FP} \tag{4.1}$$

Precision Score - se refere ao número de programas maliciosos identificados corretamente (Tp) sobre o número de verdadeiros positivos(Tp) somados aos falsos positivos(Fp) [2], conforme a equação (4.2) a seguir:

$$Precision = \frac{TP}{TP + FP} \tag{4.2}$$

Recall Score **(Rs)** - se refere ao número de verdadeiros positivos dividido pelo número de verdadeiros positivos mais os falsos negativos [2], conforme a equação (4.3) a seguir:

$$Recall = \frac{TP}{TP + FN} \tag{4.3}$$

F1 Score **(F1)** - é uma média ponderada entre *Precision* (Ps) e *Recall* (Rs). O F1 score mostra uma visão diferente da precisão de cada classificador, na qual os falsos positivos e os negativos são integrados [2], podendo ser representada pela equação (4.4) a seguir:

$$F\text{-}measure = 2 \times \frac{Precision * Recall}{Precision + Recall} \tag{4.4}$$

Matriz de Confusão - avalia a precisão da classificação calculando a matriz de confusão com cada linha correspondente à sua classe verdadeira [2]. Ela oferece um detalhamento do desempenho do modelo de classificação correspondente, ao mostrar para cada classe, o número de classificações corretas em relação ao número de classificações indicadas pelo modelo [74]. Medidas como F1 *score*, acurácia, precisão e abrangência de classificadores são retiradas da matriz de confusão [72]. Na presente pesquisa, como resultados, 0 indica arquivos benignos e 1 arquivos maliciosos. Cada linha representa a quantidade de TP , FP, FN e TP respectivamente, conforme apresentado na tabela 4.3.

Tabela 4.3: Matriz de confusão para uma classificação binária.

Classe	**Positivo**	**Negativo**
Benigno	TP	FN
Malware	FP	TN

Curva ROC - AUC - *Receiver Operating Characteristic Curve*, ou curva ROC é um gráfico que mostra o desempenho de um classificador binário, criado a partir da relação entre a taxa de *True Positive*(TPR), também chamada de *Recall* e a taxa de *False Positive* (FPR) [2]. Por sua vez, a *Area Under the ROC Curve* (AUC) é a métrica que sumariza a performance geral do classificador [99], ou seja, pode-se usar a AUC como uma medida de sucesso de um classificador [119], uma vez que uma pontuação alta na AUC corresponde a uma classificação bem-sucedida na identificação de malwares. Portanto, a AUC é uma métrica diretamente aplicável à avaliação da qualidade dos resultados do modelos [15].

Coeficiente de Correlação - é usado no aprendizado de máquina como uma medida de qualidade nas classificações binárias (duas classes). A correlação é a forma normalizada da covariância [113]. Ela representa a força da associação entre duas variáveis, a qual pode ser medida por um coeficiente de correlação, através de diferentes. O coeficiente de correlação de *Pearson* é o mais comum usado [108], e pode ser medido pela equação (4.5) a seguir:

$$r = \frac{n(\sum xy) - (\sum x)(\sum y)}{\sqrt{[n \sum x^2 - (\sum x)^2][n \sum y^2 - (y)^2]}} \tag{4.5}$$

onde n é o número de *malware*, x é um recurso específico e y é um destino (classe), que identifica um arquivo como malicioso ou benigno. Pode ser representada pela matriz de correlação que mostra o grau de similaridade entre o vetor característico, onde o valor calculado está em (0,1) [112]. Existem muitos tipos de correlação, mas neste artigo serão utilizados o coeficiente de correlação e o *Matthews Correlation Coeficient* (MCC).

A métrica MCC é utilizada no aprendizado de máquina como uma medida da qualidade nas classificações binárias (duas classes), sendo amplamente utilizado no campo da bioinformática como métrica de desempenho, tendo sido proposto por BBoughorbel et al. [120], para uso em bases desequilibradas. Dados desequilibrados significavam que havia mais amostras de uma categoria versus outras [3]. O MCC é um coeficiente de correlação em que o valor está em (-1, +1), onde +1 representa uma previsão perfeita, 0 representa uma previsão média e -1 representa uma previsão inversa [2]. O Scikit-learn implementa o MCC para classes binárias através da função *matthews_corrcoef* [2], que

pode ser descrita pela equação (4.6) a seguir:

$$MCC = \frac{tps * tn - fp * fn}{\sqrt{(tp + fp)(tp + fn)(tn + fp)(tn + fn)}} \tag{4.6}$$

Correlação de Atributos - é uma técnica para medir a força da associação entre duas variáveis e pode ser medida por um coeficiente de correlação, a partir da aplicação de diferentes métodos [108]. A Figura 4.10, apresenta, através de mapa de calor, uma matriz de correlação que apresenta o grau de correlação dos atributos entre si e entre cada um e a variável classe, a qual identifica o arquivo como malicioso ou benigno, a partir dos dados do *Dataset 1*.

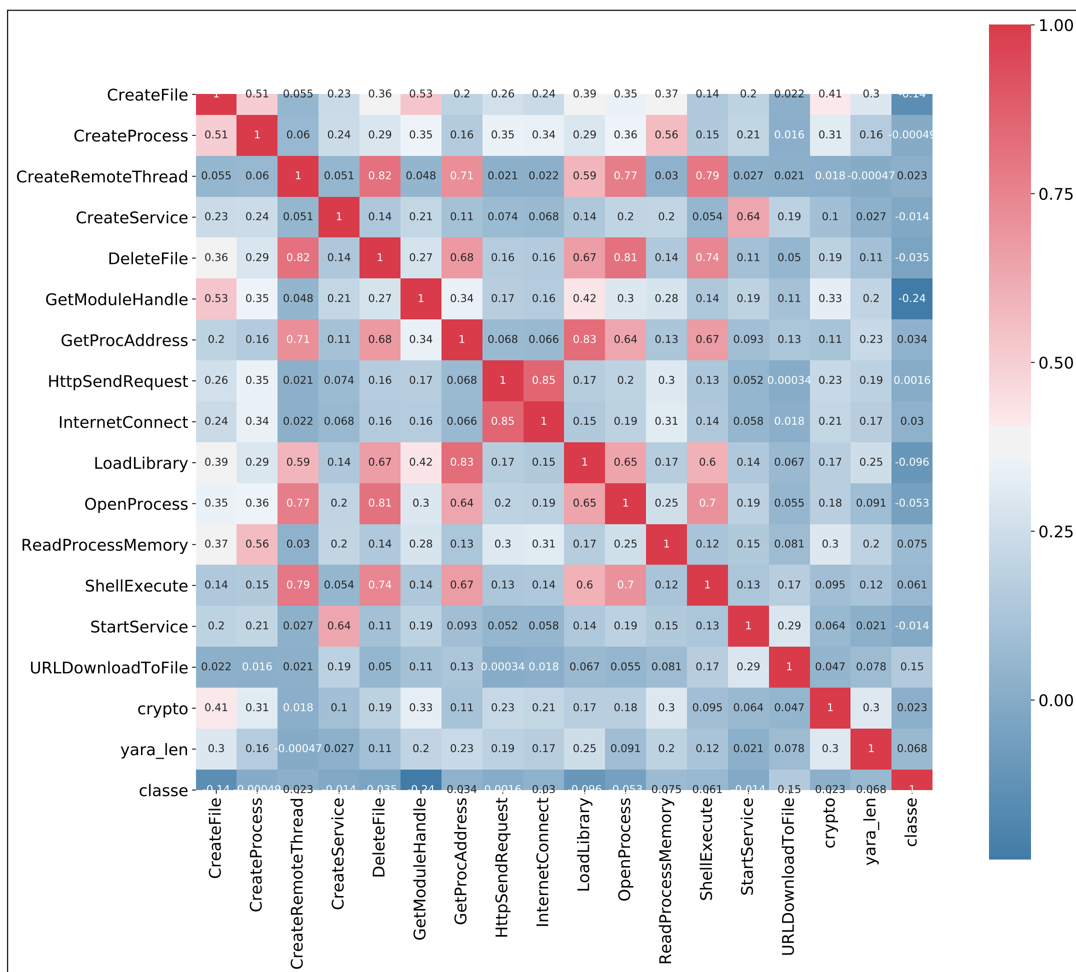

Figura 4.10: Matriz que mostra a correlação entre os atributos.

4.2.5 Funções para Medição da Performance

A biblioteca *sklearn.metrics* do scikit-learn [2] implementa varias funções para medir a performance do classificador. A presente pesquisa utiliza algumas dessa métricas listadas

a seguir:

- *precision_recall_curve* - calcula os pares *precision-recall* para diferentes limites de probabilidades.
- *roc_curve* - calcula a curva ROC.
- *confusion_matrix* - calcula a matriz de confusão e avalia a acurácia do classificador.
- *matthews_corrcoef* - Calcula o MCC.
- *roc_auc_score* - calcula a área abaixo da curva ROC (AUC).
- *accuracy_score* - calcula a acurácia.
- *f1_score* - Calcula o *f-score* ou*f-measure*.
- *precision_score* - Calcula a precisão.
- *recall_score* - Calcula o *Recall*.

4.2.6 Seleção do Melhor Modelo

Após as atividades de treinamento e avaliação, um importante processo é o de seleção do melhor modelo. Tal processo demanda escolhas que inclui: a avaliação de sua complexidade de implementação, além da performance, baseada nas métricas elencadas, as quais estão associadas com a eficiência computacional do modelo, o que ser medido pelo tempo gasto no processo de treinamento. Outra caraterística importante é a sua capacidade de generalização, ou seja, quão bem o modelo selecionado se comporta em dados novos.

No entanto, é importante ressaltar que, segundo a literatura [18, 121], que não é raro que um modelo de aprendizado de máquina que apresente uma boa performance no treinamento, apresente uma baixa precisão na predição de novos dados. Dessa forma, são listadas várias possibilidades para se obter bons resultados, combinando-se várias técnicas de aprendizado de máquina em um modelo preditivo [5]. Uma das formas é através do método *Voting Classifier*, que consiste em construir um modelo de aprendizado de máquina que treina em um conjunto com vários modelos e faz a predição baseada em uma combinação da maioria dos votos, ou seja, na média das predições do conjunto de modelos. A implementação do *Voting Classifier* do Scikit-learn [2] oferece duas opções:

- Voto da maioria (*Hard Voting*), a classe de saída do preditor é uma classe com a maioria de votos, ou seja, a classe que teve a maior probabilidade de ser prevista por cada um dos classificadores.
- Votação Suave (*Soft Voting*), que consiste em uma média ponderada das probabilidades, onde pesos diferentes podem ser atribuídos para cada modelo preditor.

4.3 A Estrutura dos Arquivos Analisados

Nesta Seção é descrita a estrutura dos arquivo executáveis para sistemas operacionais *Microsoft Windows* [13], objeto da presente pesquisa. A decisão de explorar este tipo de arquivo é baseada no fato de que maioria dos *malware* para a plataforma *Windows* são arquivos executáveis de 32 bits [21], soma-se o fato de que os *malwares* de 32 bits são capazes também de infectar as atuais plataformas de 64 bits. Por último, é importante considerar que ainda existem muitos sistemas operacionais *Windows* de 32 bits em uso.

Tais arquivos são também referenciados como arquivos PE, o que inclui os arquivos executáveis (.EXE), bibliotecas dinamicamente vinculadas (DLLs) e os arquivos fonte (binários). A distinção entre arquivos EXE e DLL é totalmente semântica, uma vez que ambos usam exatamente o mesmo formato PE. A única diferença é um bit que indica se o arquivo deve ser tratado como um EXE ou uma DLL. Pode-se ainda ter DLLs com extensões totalmente diferentes tais como ".OCX" e ".CPL" [4]. Arquivo PE contém um *Common Object File Format* (COFF) padrão, cabeçalho de seção, diretórios de dados, e a *Import Address Table* (IAT), dentre outros [21]. A Figura 4.11 mostra a estrutura de um arquivo PE.

[13] https://docs.microsoft.com/en-us/windows/win32/debug/pe-format

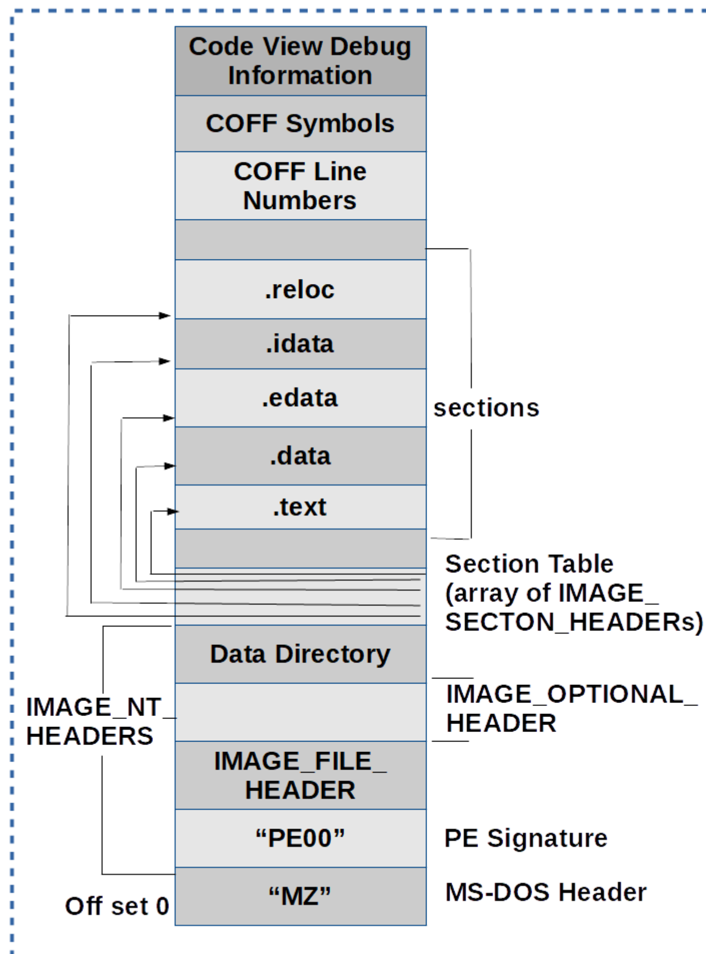

Figura 4.11: Estrutura dos arquivos PE [4].

Os padrões de detecção de *malwares* neste tipo de arquivo, podem ser baseados em dados extraídos através de técnicas de análise estática, tais como as chamadas de API[14] do *Windows*, padrões de assinaturas baseada em extração de *string*s, gráfico de fluxo de controle (CFG), frequência do código de operação (*op code*) e sequência de bytes (*n-grams*), dentre outros [51]. Tais características, referenciadas nesta pesquisa como atributos (vide Seção 4.2.2), foram extraídas e processadas pelo conjunto de ferramentas que são descritas na Seção a seguir.

4.4 Principais Ferramentas Utilizadas

Nesta seção serão descritas as ferramentas utilizadas na pesquisa para o pré-processamentos do binários, bem como, as ferramentas construídas e os módulos mais importantes utilizados na pesquisa. Uma das principais ferramentas comerciais referenciadas na literatura

[14]do inglês: *Application Programming Interface*

para a desmontagem de arquivos PE e análise estática, é o IDA Pro[15]. No entanto, a presente pesquisa utilizou apenas ferramentas de código aberto, e baseadas na linguagem *python*, para a desmontagem e *parsing* dos arquivos. Tais ferramentas foram selecionadas, a partir dos seguintes critérios:

- presença de funcionalidades necessárias à pesquisa;
- ausência de restrições de uso e de custos, por serem ferramentas de código aberto; e
- capacidade de operação multiplataforma, ou seja, a compatibilidade com diferentes sistemas operacionais.

4.4.1 A Linguagem Python

A linguagem *Python* é uma linguagem de programação criada por Guido Van Rossun em 1990 [122], sendo hoje, um projeto de código aberto administrado pela *Python Software Foundation*[16]. O *Python* é uma linguagem interpretada, multiplataforma e interativa o que facilita seu desenvolvimento, manutenção e testes. O *Python* foi selecionado como linguagem padrão para a pesquisa por diversos fatores tais como a sua simplicidade, mas principalmente, pela existência de várias ferramentas para análise de arquivos PE e de bibliotecas de aprendizado de máquina necessárias a pesquisa [21]. As principais ferramentas e módulos *Python* utilizados na presente pesquisa serão apresentados a seguir:

- ***Scikit-learn*** - o projeto *Scikit-learn*[17], teve inicio em 2007, como um projeto de David Cournapeau, chamado *Google Summer of Code project*, com sua primeira *release* lançada em 2010. Hoje é um projeto colaborativo, conduzido pela comunidade, mas que recebe doações institucionais e privadas que ajudam a garantir sua sustentabilidade [2].

  A ferramenta *Scikit-learn* é uma estrutura *python* para aprendizado de máquina, que possui métodos de seleção de atributos, algoritmos de classificação, bem como funcionalidades de validação e avaliação de desempenho [2]. Segundo Geron [66], o Scikit-Learn se destaca pela facilidade de uso, além de implementar com muita eficiência muitos algoritmos de *Machine Learning.* O *Scikit-learn* fornece ferramentas fáceis de usar e eficientes para análise de dados. Além disso, fornece rotinas prontas (*scripts*) para muitos conceitos importantes da Ciência de Dados, como aprendizado de máquina, validação cruzada, etc [123].

[15]https://www.hex-rays.com/products/ida/
[16]https://www.python.org/
[17]https://scikit-learn.org

- ***Jupyter Notebook*** é um aplicativo baseado em ambiente Web que permite estruturar e executar facilmente partes de código de uma maneira muito amigável [2]. Na presente pesquisa, o *Jupyter* foi utilizado para as fases de pré-processamento dos dados e treinamento dos algoritmos para os modelos de classificação.

- ***Django Framework***[18] é um framework para desenvolvimento rápido para web, escrito em *Python*, tornou-se um projeto de código aberto e foi publicado sob a licença BSD[19] em 2005. o Django foi utilizado como Servidor Web para hospedar o protótipo de aplicação (vide Seção 6.1).

4.4.2 Ferramentas Construídas e Customizadas

Além de se utilizar os módulos *Python* já existentes, foram também construídas e modificadas diversas ferramentas para atender aos objetivos da pesquisa. Dessa forma, a seguir serão apresentadas as ferramentas construídas e customizadas para a extração, *parsing* e manipulação dos atributos dos arquivos executáveis (*PE*). A figura 4.12 apresenta os principais componentes dos arquivos PE extraídos pelas ferramentas descritas a seguir.

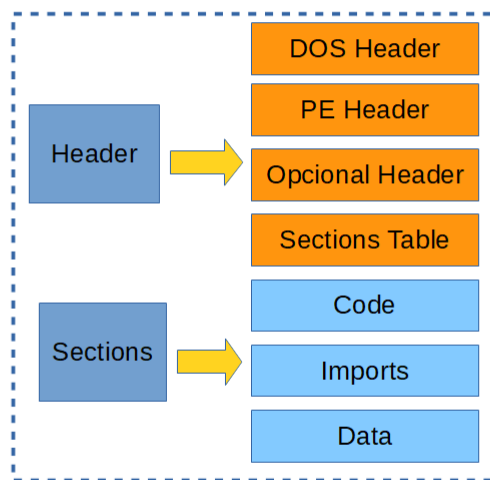

Figura 4.12: Estrutura básica do arquivo *PE* [5].

Ferramenta de Identificação e Separação de Arquivos

Para identificar e separar os arquivos de acordo com o seu tipo, foi criado o *script Separador.py*, o qual incorpora a biblioteca pré-existente "*libmagic*"[20]. A Listagem 4.1, mostra o extrato de código do *Separador.py*, criado para selecionar apenas os arquivos executáveis do MS-Windows. O código do *Separador.py* está disponível no Apêndice *B*.

[18]https://www.djangoproject.com/
[19]https://www.freebsd.org
[20]https://linux.die.net/man/3/libmagic

```
1 import os
2 import magic
3 import hashlib
4 #------------------------
5 elif os.path.isfile(path):
6     m = magic.Magic(mime=True)
7     mime = m.from_file(path)
```

Listagem 4.1: extrato do código do separador.py

Ferramentas de Extração e *Parsing*

Dessa forma, para a desmontagem dos binários extração e *parsing* dos arquivos foram construídas e modificadas ferramentas pré-existentes, conforme a seguir:

- **Parser.py** é uma ferramenta linha de comando que incorpora diversas bibliotecas, construída para a extração de atributos dos arquivos PE. A estrutura de dados resultante da extração é então gravada no arquivo de saída especificado, em formato **.CSV*, conforme exemplificado na Figura 4.16. A Listagem 4.2 apresenta o extrato do código da ferramenta *Parser.py*, disponível no Apêndice A.

```
1 import json
2 import os
3 from time import sleep
4 from argparse import ArgumentParser
5
6 from paths import *
7 from config import *
8
9 parser = ArgumentParser()
10
11 # parser.add_argument('-t', '--target', dest='target', default=None, required=False,
      ↪ help='(malwares|normais) Usa o diretorio padrao de Malwares ou Normais')
12 parser.add_argument('-b', '--base', dest='base', required=True, help='Nome␣da␣base␣a␣utilizar.'# O nome
      ↪ deve ter o prefixo "{}"'.format(BASE))
13 # parser.add_argument('-v', '--verbose', dest='verbose', required=False, help='Imprime o progresso para
      ↪ STDOUT')
14 args = parser.parse_args()
15
16 if BASE not in args.base:
17   raise Exception('O␣nome␣da␣base␣ter␣o␣prefixo␣BASE␣definido␣em␣paths.py.␣Veja␣--help␣para␣mais␣
      ↪ informacôes.')
18
19 for malware_ou_normais in [MALWARES, NORMAIS]:
20
21   input_ = '{}/{}'.format(args.base, malware_ou_normais)
22   output = input_.replace(BASE, RESULTS)
23   limit = 0
```

Listagem 4.2: extrato do código do parser.py

- **Config.py** é o responsável pela execução do *parsing*, e incorpora três ferramentas pré-existentes, desenvolvidas na linguagem *python*. Cada uma é responsável pela extração de atributos específicos. Tais ferramentas são: o PEframe.py[21]; PEcheck.py[22]; e PEscanner.py[23]. Tais ferramentas estão também embarcadas na distribuição Linux REMnux [24] e serão descritas a seguir:

- **O PEframe** é uma ferramenta desenvolvida por Gianni Amato[25], para realizar análises estáticas de malware executável portátil. Ele pode ajudar os pesquisadores de malware a detectar a presença de empacotadores, operações tipo "*XOR*"[26], assinatura digital, *mutex*[27], ações anti-debug e anti-VM, seções e funções suspeitas, presença de macro e muito mais informações sobre os arquivos suspeitos. A Figura 4.13 apresenta o resultado da extração do arquivo binário *brModelo.exe* com o *peframe.py*.

```
Short information
------------------------------------------------------------
File Name            BROTHER.exe
File Size            651411 byte
Compile Time         2010-04-16 03:47:33
DLL                  False
Sections             4
Hash MD5             0086b5717d5c2505cf8640ad8a6da8b5
Hash SHA-1           84bd96cf7a4f139a1afb68fcf5d5e9329275e668
Imphash              77b2e5e9b52fbef7638f64ab65f0c58c
Detected             Packer
Directory            Import, Resource

Packer matched [1]
------------------------------------------------------------
Packer               UPX -> www.upx.sourceforge.net

Suspicious API discovered [9]
------------------------------------------------------------
Function             EnumProcesses
Function             ExitProcess
Function             FtpOpenFileW
Function             GetProcAddress
Function             LoadLibraryA
Function             VirtualAlloc
Function             VirtualFree
Function             VirtualProtect
Function             recv
```

Figura 4.13: Exemplo de extração de um binário com o peframe.py.

[21]https://github.com/guelfoweb/peframe
[22]https://github.com/DidierStevens/DidierStevensSuite/blob/master/pecheck.py
[23]https://github.com/hiddenillusion/AnalyzePE/blob/master/pescanner.py
[24]https://zeltser.com/remnux-v6-release-for-malware-analysis/
[25]https://github.com/guelfoweb
[26]https://www.sans.org/blog/tools-for-examining-xor-obfuscation-for-malware-analysis/
[27]https://www.sans.org/blog/looking-at-mutex-objects-for-malware-discovery-indicators-of-compromise

- **O PEcheck** é uma ferramenta desenvolvida em *python* por Didier Stevens[28] para analisar arquivos PE, e obter informações sobre o arquivo analisado, tais como: número de seções, nome, tamanho, entropia, etc. O PEcheck pode ser definido como uma função Wrapper[29] que executa a ferramenta *PEfile.py*[30].

```
PE check for 'BROTHER.exe':
Entropy: 7.611011 (Min=0.0, Max=8.0)
MD5     hash: 0086b5717d5c2505cf8640ad8a6da8b5
SHA-1   hash: 84bd96cf7a4f139a1afb68fcf5d5e9329275e668
SHA-256 hash: 496c317022d5bc484dd21c79ac5b741d885e9dc107b9a9a4bcd8f15e3329f441
SHA-512 hash:
f4bbe32f1d8035bf14e39765719bced4f64252f96e265eba1f71deb20d91f9a720b6c9dc33598ccc
f90471966c12dfdc32ed44a131c277f2349948a26758390b
UPX0 entropy: 0.000000 (Min=0.0, Max=8.0)
UPX1 entropy: 7.928680 (Min=0.0, Max=8.0)
.rsrc entropy: 4.781176 (Min=0.0, Max=8.0)
.4UPX1 entropy: 0.000000 (Min=0.0, Max=8.0)
Dump Info:
----------Parsing Warnings----------

Suspicious flags set for section 0. Both IMAGE_SCN_MEM_WRITE and
IMAGE_SCN_MEM_EXECUTE are set. This might indicate a packed executable.

Suspicious flags set for section 1. Both IMAGE_SCN_MEM_WRITE and
IMAGE_SCN_MEM_EXECUTE are set. This might indicate a packed executable.

Suspicious flags set for section 3. Both IMAGE_SCN_MEM_WRITE and
IMAGE_SCN_MEM_EXECUTE are set. This might indicate a packed executable.

----------DOS_HEADER----------

[IMAGE_DOS_HEADER]
0x0        0x0   e_magic:                       0x5A4D
0x2        0x2   e_cblp:                        0x90
```

Figura 4.14: Exemplo de extração de um binário com o pecheck.py. Fonte: o autor.

O *PEfile.py*, criado por Ero Carrera [31], é um módulo *Python* desenvolvido como uma ferramenta multiplataforma para analisar e trabalhar com arquivos PE do *Windows*. O *PEfile.py* é independente, não possui dependências de outras bibliotecas e funciona nos ambientes OS [32], Windows[33] e Linux[34]. A Figura 4.14 apresenta o resultado da extração do arquivo binário *brModelo.exe* com o *PEcheck.py*.

- **O PEScanner.py**, é um analisador de PE também escrito em *Python* pelos autores do livro *Malware Analysts Cookbook* [104], para realizar análise estática do malware

[28]https://blog.didierstevens.com/2020/02/02/update-pecheck-py-version-0-7-9/
[29]wrapper é simplesmente uma função que existe para chamar outra função ou programa de computador
[30]https://pypi.org/project/pefile/
[31]http://blog.dkbza.org/
[32]https://www.apple.com/
[33]https://www.microsoft.com/
[34]https://www.linux.org/

antes de executá-lo em uma máquina virtual, e que pode ser aplicado para análise individual ou de grupo de arquivos, onde *pescanner.py* fará uma análise recursiva de todos os arquivos PE encontrados no diretório e subdiretórios alvo. A Figura 4.15 apresenta extrato do resultado da extração do arquivo binário *brother.exe* com o *PEScanner.py*.

```
[0] File: BROTHER.exe
################################################################################
##########

Meta-data
================================================================================
==========
Size          : 651411 bytes
Type          : PE32 executable (GUI) Intel 80386, for MS Windows, UPX compressed
Architecture      : 32 Bits binary
MD5           : 0086b5717d5c2505cf8640ad8a6da8b5hash
SHA1          : 84bd96cf7a4f139a1afb68fcf5d5e9329275e668hash
ssdeep             :
12288:ejkArEN249AyE/rbaMct4bO2/V3SikdCfdE/MPzA+dY6tZKdevN08:ZFE//Tct4bOsSg+MLBYQ
V08
imphash            : 77b2e5e9b52fbef7638f64ab65f0c58c
Date          : 0x4BC81615 [Fri Apr 16 07:47:33 2010 UTC]
Language      : ENGLISH
CRC:  (Claimed) : 0x0, (Actual): 0xaad45 [SUSPICIOUS]
Entry Point : 0x4b3b80 UPX1 1/4 [SUSPICIOUS]
================
Offset | Instructions
----------------------------------------
0      pusha
1      mov esi,0x472000
6      lea edi,[esi-0x71000]
12     push edi
13     jmp 0x4b3b9a
15     nop
```

Figura 4.15: Exemplo da extração do binário com o PEScanner.py. Fonte: o autor.

CreateProcess	CreateRemoteTh	CreateService	DeleteFile	GetModuleHandl	GetProcAddress	HttpSendReques	InternetConnect	LoadLibrary
0	0	0	0	3	3	0	0	6
3	0	0	3	3	3	0	0	6
0	0	0	0	0	0	0	0	0
0	0	0	0	0	3	0	0	3
4	0	0	1	5	7	4	4	9
3	0	0	0	0	0	0	0	0
0	0	0	0	3	0	0	0	0
0	1	0	0	3	4	0	0	7

Figura 4.16: Extrato do *Dataset* resultante do processo de extração. Fonte: o autor.

4.5 Considerações Finais

Neste capítulo foram apresentadas as atividades relacionadas ao processo de aprendizado supervisionado, bem como os principais procedimentos e atividades relacionadas, as ferramentas necessárias à extração e processamento dos atributos, obtidos da estrutura dos arquivos PE, e que compõem o conjunto de dados necessários à implementação prática dos experimentos a serem apresentados no capítulo a seguir.

Capítulo 5

Experimentos

Neste capítulo são descritas as implementações dos experimentos com aprendizado supervisionado de máquina, a partir dos procedimentos elencados no capítulo anterior, a fim de se avaliar os objetivos propostos na Seção 3.1, e responder as questões de pesquisa elencadas no Capítulo 1 (vide Seção 1.2), para tanto, o capítulo está dividido da seguinte forma: a Seção 5.1 apresenta o escopo dos diversos experimentos e o ambiente de experimentação. A Seção 5.2 apresenta uma avaliação dos diversos experimentos e os resultados alcançados, os quais são discutidos na Seção 5.3. Na Seção 5.4 são apresentadas as limitações da pesquisa, e por fim, na Seção 5.5, as considerações finais.

5.1 Implementação

Esta seção apresenta o escopo da implementação proposta da utilização do aprendizado supervisionado para a construção de um preditor de *malwares* e assim responder as questões de pesquisa (RQ) elencadas no Capítulo 1 (vide Seção 1.2). Em primeiro lugar, para responder à RQ1, investigou-se a eficácia do uso de algoritmos de ML para classificar ativos como benignos ou maliciosos, a partir das informações disponíveis no código-fonte dos arquivos do *Windows* PE. Adicionalmente, para abordar a RQ2, investigou-se também como reduzir a relevância individual dos atributos, através da experimentação do método PCA para redução da dimensionalidade. Por fim, no terceiro experimento, são introduzidos os atributos textuais aos conjuntos de dados de treinamento para verificar como eles contribuem com a precisão do modelo, e assim, abordar a RQ3.

Dessa forma, dentro do escopo desta pesquisa, utilizou-se o processo de aprendizado supervisionado de máquina para fazer previsões em dados novos e inéditos, a partir do treinamento em uma base de dados previamente rotulada. Os conjuntos de amostras utilizadas nos experimentos foram descritos na Seção 4.2.2. Tais amostras, conforme descrito na literatura, foram divididos dois subconjuntos: um conjunto de treinamento e

um conjunto de testes [117], que servirá para a avaliação do treinamento. Este conjunto de amostras compôs a entrada de dados para treinamento de cinco diferentes algoritmos de classificação: *k-Nearest Neighbors* (KNN), *Support Vector Classification* (SVC), *Gaussian Naive Bayes* (NB), *Random Forest* e *Decision Tree*. Esses algoritmos aprendem a ligação entre dois conjuntos de dados: os dados observados x e uma variável externa y que se está tentando prever, chamada de " destino " ou " rótulos " [2], que recebeu o rótulo " M " (para malware) ou " B " (para benigno), respectivamente. O resultado esperado desse processo de aprendizagem é um classificador para fazer previsões sobre dados novos e não vistos anteriormente. O ambiente de experimentação e os experimentos realizados são descritos ao longo desta Seção.

5.1.1 Ambiente de Experimentação

Esta seção descreve a configuração completa para análise e classificação de *malware*. Também inclui a análise experimental dos resultados. Os experimentos foram conduzidos em um Notebook com processador AMD A12 9720P RADEON R7, 12 Cores 4C+8G, 2.7 GHertz, com 08 GB de memória RAM, Sistema Operacional *Windows* 10 64 Bits, Sistema de virtualização baseado no *Virtualbox* versão 6.0.14, e máquina virtual (VM) com sistema operacional Linux Ubuntu 14.0, com 04 GB de RAM e 25 GB de disco rígido.

Tal ambiente hospedou todas as ferramentas utilizadas no experimentos, conforme Figura 4.2, sumarizadas nas seguintes atividades:

a) coleta de dados - aquisição dos binários;

b) processamento dos dados - desmontagem dos arquivos binários (PE) que compuseram os *datasets* de pesquisa e a extração dos atributos;

c) tratamento do dados - ajustes, normalização e limpezas dos dados obtidos do processo de extração;

d) treinamento e avaliação - experimentos de aprendizado supervisionado de máquinas realizados; e

e) por fim, a construção de um protótipo de aplicação para receber os modelos resultantes dos experimentos de aprendizado de máquina.

5.1.2 Experimento 1 - Modelo Inicial

Este primeiro experimento tem como objetivo responder ao RQ1 (*Qual a eficácia do uso do ML para classificar ativos como benignos ou malignos a partir de informações disponíveis no código-fonte dos arquivos do Windows PE?*).

Neste experimento, buscou-se a simplicidade do modelo, como uma prova de conceito de que os atributos extraídos da base de dados e sua aplicação direta, como entrada dos algoritmos, seriam suficientes para treinar os modelos, e se obter uma boa acurácia do classificador.

Para tanto, foram utilizados apenas atributos numéricos, pois considerando que uma parte crítica do sucesso de um projeto de *Machine Learning* é apresentar um bom conjunto de atributos para treinar [66], alguns atributos foram excluídos porque verificou-se, durante a fase da análise exploratória, que tais atributos poderiam ocasionar um ruído no processo de treinamento dos algoritmos. Foram também excluídos os atributos textuais, pois esses requerem um tratamento adicional (vide Subseção 5.1.4).

Deste modo, dos 34 atributos iniciais, restaram apenas 24 atributos numéricos, aos quais foi adicionado o atributo "*classe*", para rotular o dataset com designação de benignos ou malicioso, com valores 1 para arquivo de *malware* e 0 para os benignos, totalizando ao final 25 atributos (24 + 1 colunas) para o primeiro experimento.

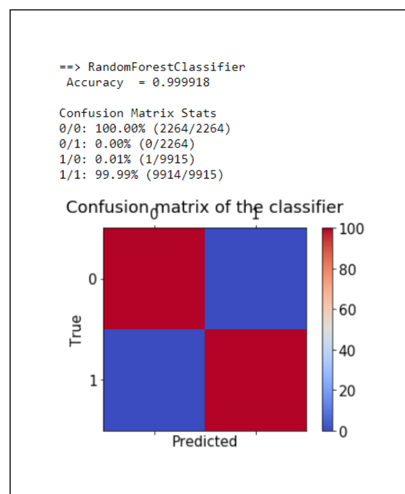

Figura 5.1: Matriz de confusão que mostra o resultado da fase de treinamento do experimento 1

Para o treinamento e teste dos algoritmos, os dados foram divididos na seguinte proporção: 80% para treino e 20% para teste (*train-test split method*) [16]. Em seguida, o comando "*fit*" foi aplicado, em conjunto com o parâmetro "*random_state*", que dividiu, de forma aleatória, o conjunto de dados em dois subconjuntos: um para treino e o outro para teste. O melhor algoritmo na fase de treinamento foi o *Random Forest*, como pode ser visto na Figura 5.1, que mostra os resultados da fase de treinamento e teste deste experimento.

Logo após a fase de treinamento e teste, o algoritmo com melhor desempenho foi aplicado a um processo de validação, para se verificar a persistência dos resultados obtidos nas fases de treinamento e testes.

Ao final do experimento, o modelo foi treinado novamente, agora com as duas bases, totalizando 12.970. Os mesmos procedimentos de limpeza e transformação aplicados nos dados de treinamento, foram também aplicados nessas bases. O código utilizado no presente experimento está listado no Apêndice C - Experimento 1.

Foram também separados 150 amostras de cada dataset, totalizando 300 registros separados especificamente para validar os resultados obtidos no treinamento. Logo após a fase de treinamento e teste, o algoritmo com melhor desempenho foi aplicado a outro processo de validação com o dataset 3, para se verificar a persistência da acurácia do modelo treinado.

Neste experimento também aplicou-se, um processo de verificação da relevância dos atributos, para verificar o seu poder discriminante, dentro do conjunto de dados. Verificou-se que os três atributos numéricos com maior relevância no conjunto de dados são: *entropy*, *antidbg_len* e *packer_len*.

5.1.3 Experimento 2 - PCA

O segundo experimento visa responder ao RQ2 (*qual é um conjunto mínimo de atributos a serem extraídos desse tipo de arquivo que ainda pode ser usado para classificar os ativos como benignos ou malignos, mantendo sua precisão?*), que busca saber como o número de atributos influencia o desempenho do modelo, eliminando-se algumas redundâncias. Assim, neste experimento foi aplicada a redução de dimensionalidade, ou seja, o número de atributos, também visa reduzir o impacto da correlação entre o número de atributos utilizados para discriminar *malwares*.

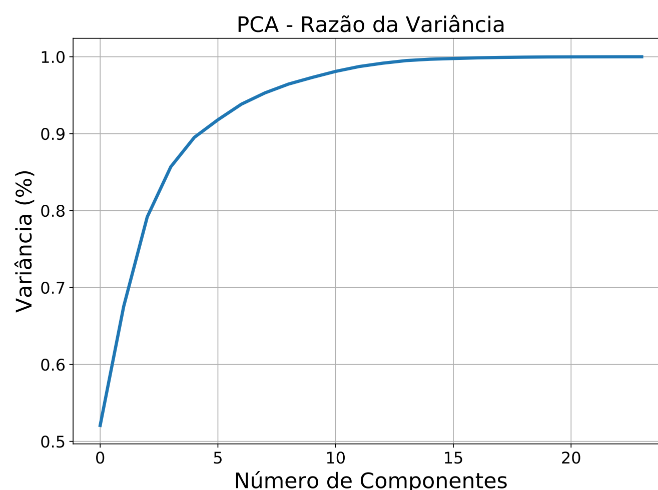

Figura 5.2: Variância dos dados em função da aplicação do PCA.

Ao aplicar função de redução da dimensionalidade, verificou-se que 12 componentes (PCA=12) seriam suficientes para preservar cerca de 99% de variância e, consequentemente, evitar a perda de informação com a redução da dimensionalidade, conforme pode ser verificado na Figura 5.2, que apresenta o gráfico da variância explicada e cumulativa resultante da aplicação do PCA no espaço N-dimensional definido pelo numero de atributos (N=24). O referido gráfico mostra que, com a aplicação do PCA, os 12 primeiros componentes correspondem à maior parte da variância dos dados, ou seja, o poder discriminante do conjunto de atributos.

Para treinamento dos modelos, foram também aplicados diversos ajustes de parâmetros nos algoritmos utilizados, de acordo com experimentos prévios da literatura [66]. Também utilizou-se o *grid search*, cujo objetivo é, por meio de uma análise exaustiva, encontrar a melhor combinação de parâmetros para cada modelo em questão. O *grid search* foi utilizado juntamente com a validação cruzada (CV) 5-*fold*, dessa forma cada combinação de parâmetros é treinada e testada dividindo-se o conjunto em 5 partes (4 para treino e 1 para validação alternando-se entre si). A Figura 5.3 mostra o gráfico de resultados versus a variação dos diversos parâmetros testados pelo método *grid search*. Na referida figura é possível observar os diversos resultados obtidos. O gráfico mostra o processo em que todas as combinações de possíveis valores da grade de parâmetros são avaliadas. Ao final desse processo, a melhor combinação é mantida.

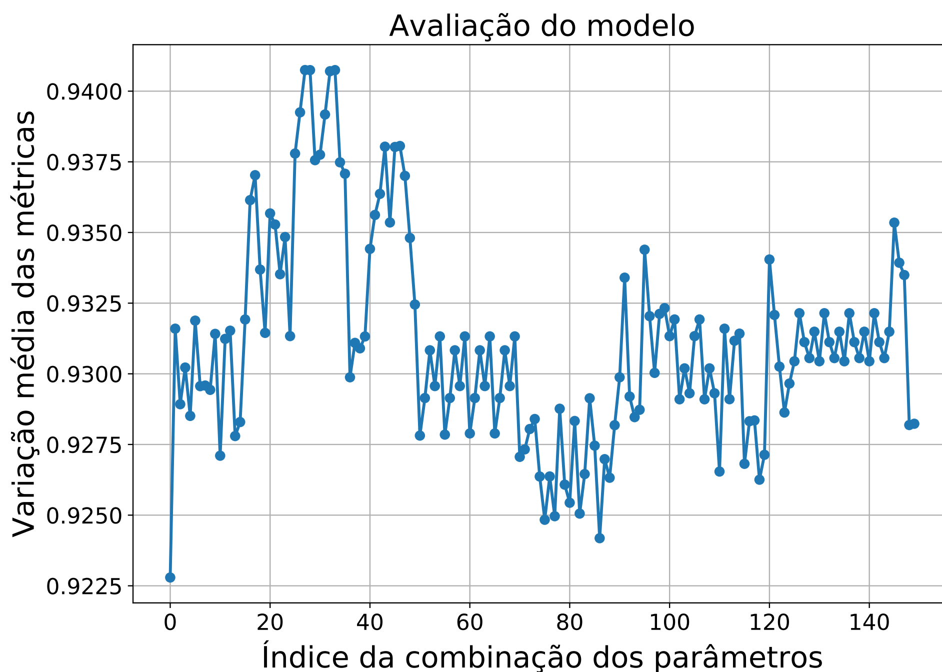

Figura 5.3: Resultados intermediários do Grid Search

Neste experimento todas as amostras disponíveis (conjunto de dados 1 + conjunto de dados 2) foram utilizadas para o treinamento e teste. No entanto, é importante ressaltar que o número de amostras de *malwares* corresponde a 81% do total de amostras, excedendo o número de amostras benignas em grande números. Dessa forma, considerando que o conjunto de dados é desequilibrado, esse experimento tenta responder também à questão secundária: *Quais são as consequências do treinamento do modelo ML em uma base desequilibrada?*

Para responder a essa pergunta, utilizou-se a métrica MCC (vide Subseção 4.2.4 para medir se o uso de uma base desequilibrada afetaria negativamente a precisão do modelo. Para verificar a performance dos modelos, foram utilizadas as seguintes métricas: *Recall* (r); F1 *score* (s); *Accuracy* (a); *Precision* (p) e para garantir um maior equilíbrio, foi criada também uma métrica "MMS", que é a média das anteriores ($MMS = (r + s + a + p)/4$). O código utilizado no presente experimento está listado no Apêndice - Experimento 2.

5.1.4 Experimento 3 -TF-IDF

No primeiro e no segundo experimentos, utilizaram-se apenas atributos com dados numéricos, descartando-se as colunas textuais. Diferentemente, neste experimento adotou-se uma abordagem diferente para responder ao RQ3 (*O uso de características textuais coletadas em arquivos binários irá contribuir para um modelo mais preciso?*).

Assim, apenas os atributos textuais extraídos das amostras (vide Seção 4.2.6) são utilizados, junto com a aplicação do TF-IDF, um algoritmo de categorização de textos [87], que converte um documento em um formato estruturado, ou seja, em formato numérico, e que reflete a importância das palavras constantes do referido documento [88]. Dessa forma, as colunas textuais *antidbg, crypto, packer* e *yara*, foram condensadas em uma única coluna denominada *texto*. Em seguida, aplicou-se um processo de vetorização [3]. Como consequência desse processo, o número de atributos subiu de 4 para 1.616.

5.1.5 Persistência do Modelo

Depois da fase de treinamento, é desejável ter uma maneira de persistir os modelos treinados para uso futuro sem ter que retreiná-los [2]. Dessa forma, verifica-se que encontrar o modelo com a melhor performance não é o fim do processo do aprendizado de máquina. O *scikit-learn* fornece a capacidade de salvar e recuperar os modelos treinados em arquivos, pelo métodos *pickle* (*dump* e *load*) e *joblib* (*joblib.dump* e *joblib.load*)[1] respectivamente. Por consequência, os modelos treinados são portáveis e assim, podem ser anexados como parte de um sistema de detecção de *malware* baseado em aprendizado de máquina. A

[1] https://joblib.readthedocs.io/en/latest/why.html

Figura 5.4 apresenta um exemplo de predição com modelo baseado no arquivo *joblib*. O código deste experimento está disponível no Apêndice B.

```
model_2_tfidf   = jb.load('model_rf_19set20.joblib')
############
model_rf = model_2_tfidf['model']
#model_rf

resultado_rf_2 = model_rf.predict(malw_clean.iloc[:,:-1])

print(pd.Series(resultado_rf_2).value_counts(),
      '\n\nACC: ' + str(accuracy_score(y_pred = resultado_rf_2,y_true=malw_clean.classe)))

1.0    1446
0.0      98
dtype: int64

ACC: 0.9365284974093264
```

Figura 5.4: Exemplo de predição com arquivo joblib

5.2 Avaliação dos Experimentos

Nesta seção, apresentamos os resultados experimentais das três abordagens da pesquisa: em primeiro lugar, no experimento 1 (na subseção 5.1.2), buscou-se o menor custo de processamento através de uma abordagem simples e direta de aprendizado de máquina supervisionado para verificar a eficiência da proposta de pesquisa.

Em seguida, no experimento 2 (na subseção 5.1.3), investigou-se como reduzir a relevância individual dos atributos, mantendo-se a capacidade de generalização do modelo, com a aplicação do método *Principal Component Analysis* (PCA) para seleção de atributos e redução de dimensionalidade.

Por fim, no terceiro experimento, utilizaram-se os atributos textuais ponderados pelo algoritmo TF-IDF (vide Seção 5.1.4), para verificar se esses seriam significantes para a precisão do modelo.

Depois do treinamento, os modelos são testados em um conjunto de dados de validação. Dessa forma, várias métricas foram utilizadas para avaliar o desempenho dos classificadores, as quais são descritas nas Seção 4.2.4.

A seguir são apresentados os resultados obtidos nos diversos experimentos realizados para responder as questões de pesquisa (RQ):

a) **Quanto à RQ1** - observou-se que, mesmo com uma quantidade reduzida de atributos selecionados, obteve-se uma acurácia acima de 97% no treinamento dos modelos (conforme demonstrado na tabela 5.1). Dentre os algoritmos de classificação utilizados, o de melhor desempenho foi o *Random Forest* que apresentou as seguintes métricas: *Accuracy*: 97,92; *Precision*: 96,97%; *Recall*: 99,99% e *F1 Score*: 98,98%.

Tabela 5.1: Resultados do experimento 1 - valores médios

Algoritmo	**Accuracy**	**Precision**	**Recall**	**F1 Score**
SVC	54,49	53,99	50,37	45,65
Random Forest	**97,92**	**96,97**	**99,99**	**98,98**
KNN	85,88	86,09	75,71	78,77
Decision Tree	95,26	95,39	99,94	99,98
Gaussian NB	52,54	43,03	51,22	48,03

Na Figura 5.5, é possível observar o gráfico *bloxplot* com a acurácia média (*mean score*), referente à aplicação do método *10-fold Cross-Validation* para treinamento dos modelos.

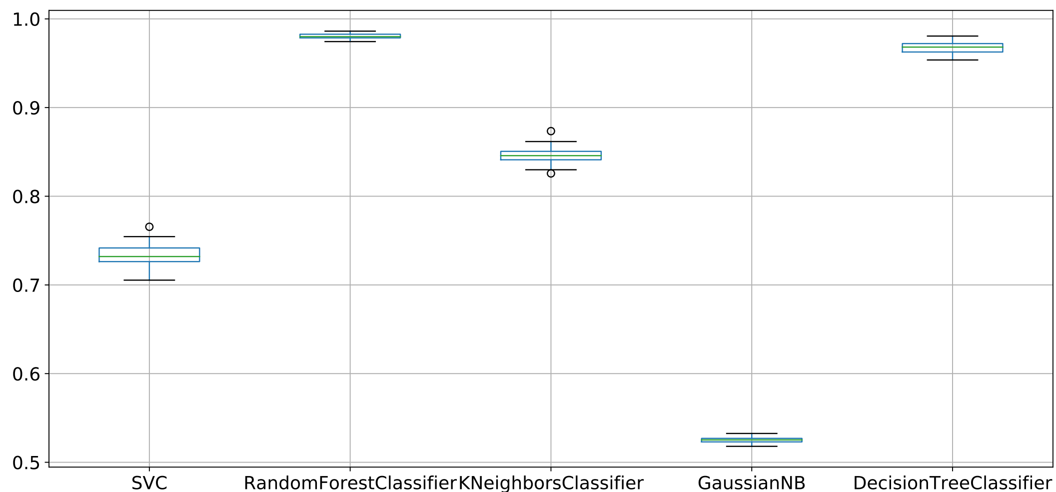

Figura 5.5: Boxplot com a acurácia média dos algoritmos no experimento 1

b) **Quanto à RQ2** - No experimento 2, buscou-se demonstrou que é possível reduzir o número de atributos, enquanto se mantém a precisão geral do modelo. Como resultado, o algoritmo que apresentou a melhor performance foi Random Forest, com os seguintes resultados: *Accuracy* 94,61%; *Precision* 97,25%; *Recall* 96,15%; *F1 Score* 96,55%; MMS 97,77% e MCC 84,91%. Verifica-se, através do resultado da métrica MMC, que o uso do conjunto de dados desbalanceado (número de amostras maliciosas muito superior ao numero de amostras benignas) apresentou uma variação negativa, quando comparada com as demais métricas.

Na Figura 5.6, é possível observar o gráfico *bloxplot* com a acurácia média (*mean score*) obtida do treinamento. No referido gráfico, verifica-se que o algoritmos (exceto o Gaussian NB) apresentaram uma acurácia média acima de 94% durante a fase de treinamento.

Tabela 5.2: Resultados do experimento 2 - PCA

Algorithm	Accuracy	Precision	Recall	F1 Score	MMS	MCC
SVC	92,97	97,06	92,11	95,28	94,10	81,57
Random Forest	**94,61**	**97,25**	**96,15**	**96,55**	**97,77**	**84,91**
KNN	93,64	97,72	94,45	95,79	94,63	83,66
Decision Tree	92,86	96,85	94,34	95,44	94,21	79,36
Gaussian NB	86,69	92,05	91,42	91,43	89,84	69,81

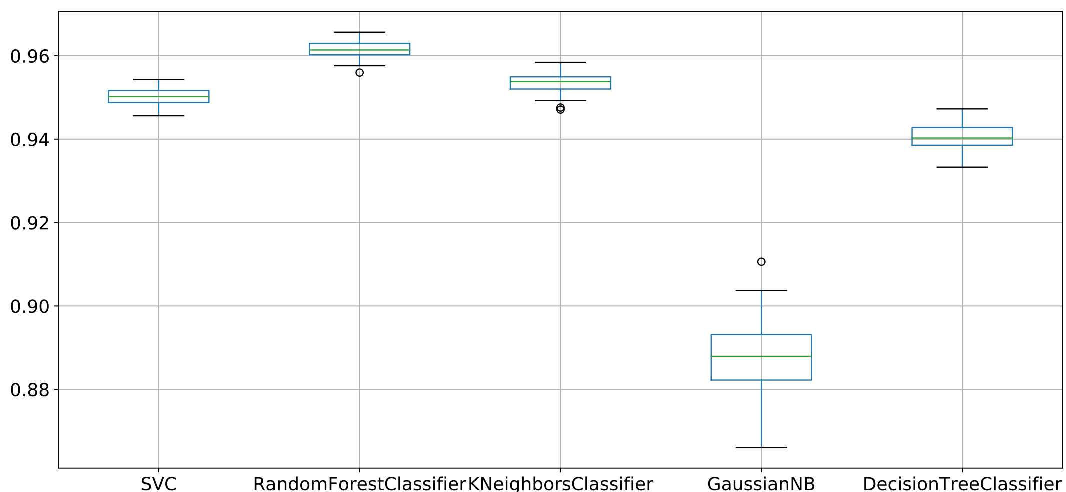

Figura 5.6: Boxplot com a acurácia média dos algoritmos no experimento com o PCA

c) **Quanto à RQ3** - para responder a essa pergunta, usaram-se os atributos textuais processados pelo algoritmo TF-IDF. Os resultados mostraram que o uso de apenas características textuais afetou negativamente a acurácia do modelo. Na primeira fase de treinamento, o algoritmo com melhor desempenho foi *Random Forest* com os seguintes valores: *Accuracy*: 98,66%; *Precision*: 98,50%; *Recall*: 97,14%; e *F1 Score*: 97,80%.

   A Figura 5.7 apresenta o gráfico *bloxplot* com a acurácia média (*mean score*) obtida no treinamento do modelos. No referido gráfico, verifica-se que o algoritmos (exceto o

Gaussian NB) apresentaram uma acurácia média equilibrada e acima de 95% durante a fase de treinamento.

Tabela 5.3: Resultados do experimento 3 - TF-IDF

Algoritmo	Accuracy	Precision	Recall	F1 Score
SVC	98,50	98,14	96,96	97,54
Random Forest	**98,66**	**98,50**	**97,14**	**97,80**
KNN	97,94	96,47	96,85	96,66
Decision Tree	98,62	98,31	97,19	97,74
Gaussian NB	70,35	69,30	81,38	66,78

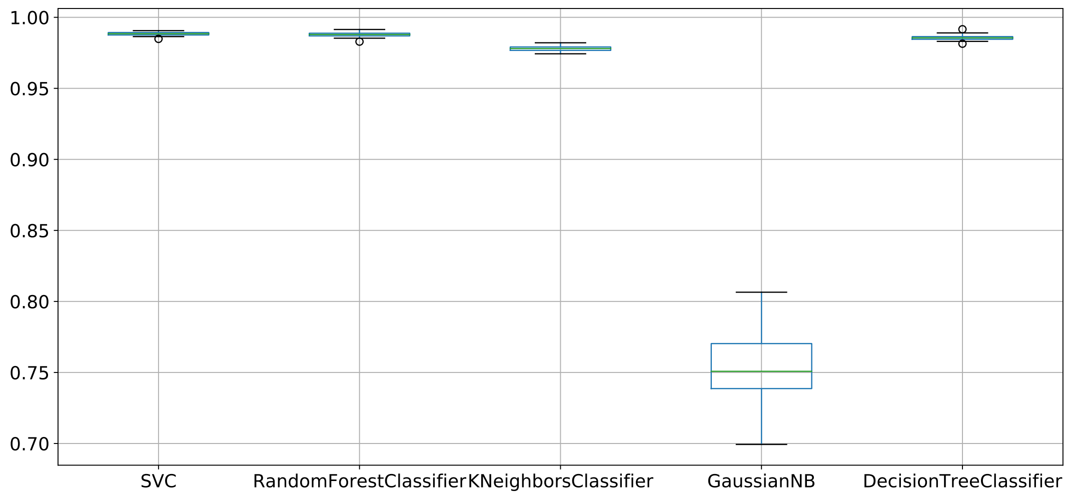

Figura 5.7: Boxplot com a acurácia média dos algoritmos no experimento o TF-IDF

5.3 Análise dos Resultados

Como avaliação dos resultados experimentais, o algoritmo *Random Forest* apresentou o melhor desempenho nos três experimentos durante a fase do treinamento. Quanto à aplicação da métrica MCC, conforme quadro resumo constante da tabela 5.2, verifica-se uma variação negativa quando comparado à outras métricas do modelo, podendo-se inferir que o uso da base desbalanceada afetou a acurácia dos modelos. A Tabela 5.4 apresenta um quadro comparativo entre os três experimentos com os resultados obtidos durante o treinamento dos modelos.

Tabela 5.4: Quadro resumo - acurácia dos modelos no treinamento

Experimentos	Modelo Inicial	PCA	TF-IDF
Melhor performance Algoritmo	Random Forest	Random Forest	Random Forest
Número de Atributos	25	12	1.616
Accuracy	97,92%	94,61%	98,66%
Precision	96,97%	97,25%	98,50%
Recall	99,99%	96,15%	97,14%
F-Score	98,98%	96,55%	97,80%

5.3.1 Avaliação dos Modelos com Dados não Treinados

A fim de se avaliar a capacidade de generalização dos classificadores, os modelos resultantes de cada experimento (1, 2 e 3), os melhores algoritmos foram aplicados a um novo conjunto de dados novos (*dataset 3*), composto de 1.544 *malwares* e 430 arquivos benignos.

Tabela 5.5: Quadro resumo da avaliação final dos modelos

Algoritmo	**Accuracy**	**Precision**	**Recall**	**F1 Score**
Exp.1 (Random Forest)	**85,56%**	**79,04%**	**81,11%**	**80,80%**
Exp.2 (KNN)	80,90%	77,21%	47,43%	47,48%
Exp.3 (SVC)	63,17%	64,04%	70,58%	60,34%

Para mensurar os resultados sobre o uso do aprendizado supervisionado de máquina para classificar arquivos como benignos ou maliciosos, foram utilizadas as seguintes métricas: *Accuracy*, *Precision*, *Recall*, *F1 Score* e *ROC Curve* e *AUC*, conforme resultados apresentados na Tabela 5.5 e descritos a seguir: Os resultados dessa avaliação são apresentados a seguir:

- o modelo do experimento 1, com classificador *Random Forest*, apresentou a melhor acurácia com 85,56%;
- modelo do experimento 2, com método PCA, o classificador KNN, apresentou a melhor performance, com 80,90% de acurácia;
- por fim, no modelo do experimento 3 (TF-IDF atributos textuais) o classificador SVC apresentou 63,17% acurácia.

Os resultados dessa de avaliação, indicam que os modelos (exceto o TF-IDF) mostraram um bom desempenho quando aplicados no novo conjunto de dados. A Figura 5.8 mostra, de forma gráfica, os diversos resultados obtidos na validação, apresentadas através da Curva ROC e AUC em relação aos experimentos 1, 2 e 3.

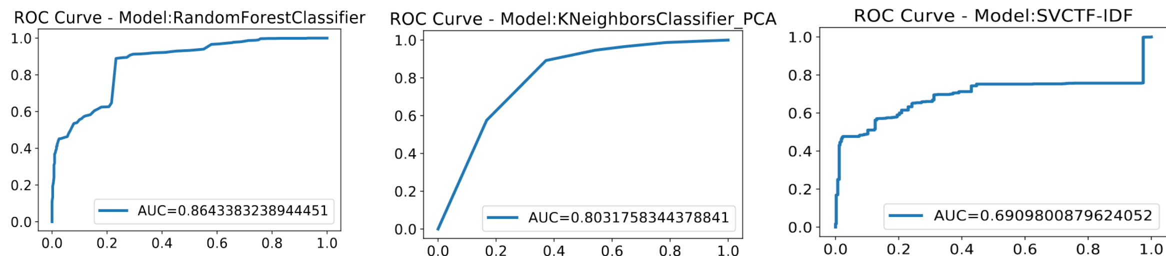

Figura 5.8: Gráfico da Curva ROC - resultados

Além da performance, outro fator a ser considerado para escolha do modelo a ser adotado é a sua complexidade, a qual pode ser medida pelo custo computacional ou seja, o tempo total para treinamento dos modelos. Dessa forma, é importante avaliar também qual das abordagens apresenta o menor custo computacional, a fim de se mensurar, não apenas a eficácia, mas também a eficiência dos modelos. Neste aspecto, verifica-se na Tabela 5.6 que o experimento 1 (Subseção 5.1.2) apresenta o menor *overhead* de processamento entre todos os experimentos.

Tabela 5.6: Avaliação dos modelos

Técnica	Número de atributos	Melhores Algoritmos	Tempo de Treinamento	Média da Acurácia	Resultado Validação
Cross-Validation (K=10)	25	Random Forest	4.03 Sec	97,92%	85,56%
PCA (Cross-Validation + grid search, K=5)	12	KNN	129,83 Sec	94,61%	80,90%
TF-IDF Cross-Validation (K=10)	1.616	SVC	530,54 Sec	98,66%	63,17%

5.3.2 Validação com o *Dataset Clamp*

Por fim, aplicou-se o método PCA, com os mesmos parâmetros usados no experimento 2 (Subseção 5.1.3), como uma abordagem de validação do método, agora aplicado a um conjuntos de dados usado por Kumar [16], e denominado Clamp[2]. Esse conjunto de da-

[2]https://github.com/urwithajit9/ClaMP

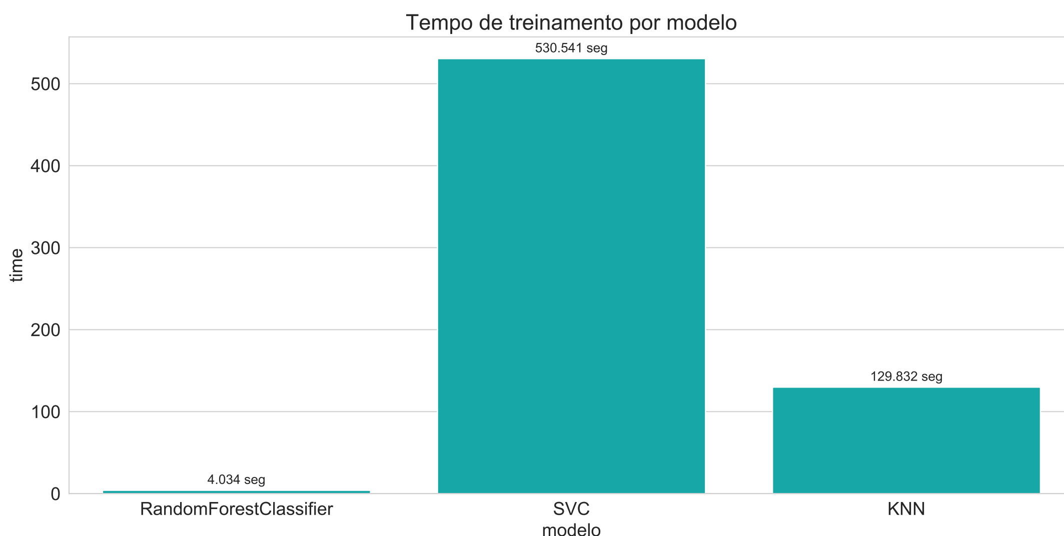

Figura 5.9: Gráfico de barras com os tempos gastos para treinamento dos algoritmos com melhor performance.

dos é composto por dois conjuntos de diferentes tipos de atributos: um deles denominado atributos integrados (*Clamp Integrated*), composto por 60 atributos, enquanto o outro chamado de atributos brutos (*Clamp Raw*), composto por 55 atributos, conforme descrito na tabela 5.7). A Figura 5.10 apresenta o gráfico da variância resultante da aplicação do PCA nesses conjuntos de dados. Na referida figura, verifica-se que se poderia reduzir a dimensionalidade dos dados do *Clamp Integrated* para 15 componentes principais (PCA=15), enquanto no *Clamp Raw*, a dimensionalidade poderia ser reduzida para 10 componentes principais (PCA=10), para se obter uma variância de 99%.

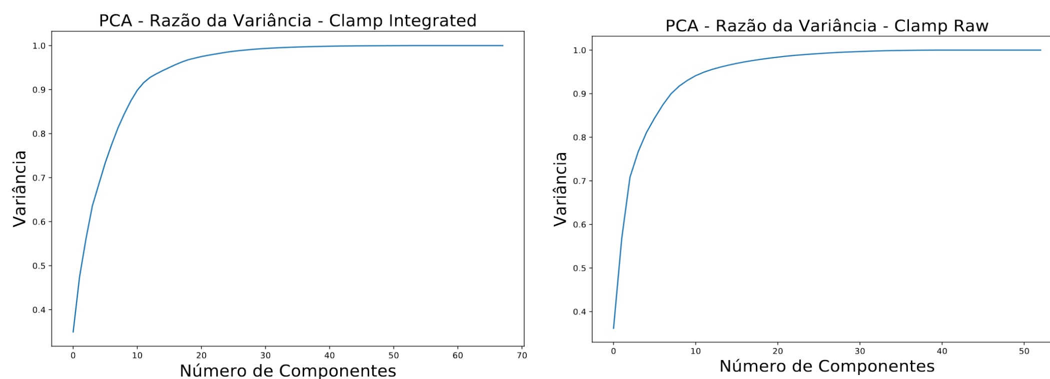

Figura 5.10: Gráfico resultante da aplicação do PCA no dataset Clamp

Os resultados apresentados na Tabela 5.7 indicam uma paridade de resultados entre aqueles relatados pelo autor e os resultados obtidos na presente pesquisa. Adicionalmente, não se verificou uma diferença significativa de precisão entre os atributos integrados e brutos com 97,29% e 97,25%, respectivamente. Como resultado final, verificou-se que modelo PCA resultante da presente pesquisa (vide Subseção 5.1.3), possui um número menor de atributos, entretanto, apresentou um resultado menor que o reportado por Kumar [16], conforme mostra a Tabela 5.7.

Tabela 5.7: Avaliação com o dataset Clamp

Datasets	**Número de atributos**	**PCA**	**Malwares**	**Benignos**	**Total de Amostras**	**Acurácia**	**Acurácia com PCA**
Clamp Raw	56	25	2.683	2.501	5.184	97,47%	**97,27%**
Clamp Integrated	70	28	2.488	2.774	5.262	98,79%	**97,28%**
Presente Pesquisa	**25**	**12**	**10.098**	**2.294**	**12.392**		**94.61%**

5.3.3 Discussão

Nesta seção são discutidos dos resultados do processo de avaliação, onde os modelos treinados são submetidos a um conjunto de dados não treinados, e especialmente separados para uma validação final dos modelos, na qual se buscou uma simulação de um ambiente real.

A avaliação dos dados foi aplicada uma base composta de 1.5440 *malwares*, que possuem uma distância temporal, ou seja, são mais recentes do que aqueles treinados pelo modelo. Dessa forma é razoável considerar que, pela rápida evolução dos *malwares*, associadas ao uso de técnicas avançadas de ofuscação e mudanças em sua estrutura interna a fim de evitar a sua detecção, haja uma menor acurácia dos modelos treinados com amostras diferentes.

Da mesma forma, os arquivos benignos utilizados na avaliação final foram extraídos unicamente do sistema operacional *Windows* 10, enquanto os modelos foram treinados em aplicativos de fontes variadas e de sistemas operacionais mais antigos, tais como o (*Windows* 7 e *Windows* 8). Dessa forma, esse processo de avaliação tentou simular o mundo real, onde os modelos recebem como entrada, arquivos muito diversos daquele vistos pelos algoritmos durante o treinamento.

Por consequência, conforme pode ser observado na Tabela 5.5, houve uma diminuição na performance, quando se compara com os resultados obtidos durante o treinamento. No entanto, é importante ressaltar que tal queda de performance, quando modelos de aprendizagem de máquina são aplicados em ambiente do mundo real, é reportada na

literatura [18, 121]. Segundo Allix *et al.* [18], durante o treinamento, quando se aplica 10-*fold Cross Validation*, como no caso da presente pesquisa, assume-se que 90% dos tipos *malwares* são previamente conhecidos. No entanto, essa situação não se verifica no mundo real, principalmente quando se compara o limitado numero de amostras de *malwares* utilizadas para treinamento dos modelos com a quantidade de novas amostras de *malwares* que surgem todos os dias. Ademais, a literatura reporta a utilização de técnicas de aprendizagem de máquina para modificar partes do código dos *malwares*, afim de degradar ou reduzir a taxa de detecção de modelos preditivos baseados em análise estática [96].

Ademais, outro aspecto a considerar é o impacto do uso de dados severamente desbalanceados para o treinamentos dos modelos [18, 65, 124], conforme utilizados na presente pesquisa, que contém cerca de 80% de amostras de *malwares* e apenas 20% de arquivos benignos. Para mitigar tal situação, a literatura reporta a utilização de diversas técnicas (*resampling methods*) [22, 75], que serão exploradas em trabalhos futuros.

5.4 Limitações da Pesquisa

A seguir, são apresentadas, as principais limitações relacionadas à presente pesquisa:

a) Os modelos de aprendizagem da presente pesquisa não têm o objetivo de substituir nenhuma solução de segurança, mas sim, complementar e reforçar aquelas já existentes.

b) Apenas os arquivos executáveis do *Microsoft Windows* (PE) foram examinados. Assim, são necessárias pesquisas adicionais para se detectar a presença de *malware* em outros tipos de arquivos.

c) O tamanho reduzido da base de amostras para o treinamento versus a representatividade de variantes de *malwares* existente no mundo real é insuficiente para se obter modelo com capacidade de generalização e assim apresentar uma boa performance quando aplicação em uma aplicação real. Desse modo, o conjunto de treinamento de dados deve ser mantido e atualizado ativamente ao longo do tempo com a adição de novas amostras.

d) Baseado no fato de que as características dos *malwares* evoluem muito rapidamente, não há garantia de que o modelo da proposta mantenha sua precisão quando aplicado em um novo conjunto de dados. Dessa forma, o modelo precisa ser treinado novamente toda vez que receber novos registros para manter sua precisão.

e) Por fim, o uso de dados de treinamento não representativos, incompletos ou adulterados, certamente resultará em predições imprecisas [125], afetando a confiabilidade do

modelo. Dessa forma, medidas apropriadas devem ser tomadas sobre a aquisição de novos dados, a fim de se evitar o envenenamento do conjunto de dados de treinamento com dados falsos ou tendenciosos.

5.5 Considerações Finais

Neste capítulo foram descritos os experimentos realizados para responder as questões de pesquisa elencadas no Capítulo 1. Para tanto, foram realizados três experimentos de aprendizado de máquina, a partir de um conjunto de dados composto por 14.343 amostras, das quais, 12.219 rotuladas como *malware* e 2.725 como benignos. Apresentou-se também uma discussão sobre os resultados alcançados, e por fim, as limitações da pesquisa. No capítulo a seguir, será apresentado o protótipo de aplicação prática criado para receber os modelos de classificador resultantes do processo de aprendizado de máquina obtidos nas experimentações práticas apresentadas ao longo deste capítulo.

Capítulo 6

Implementação do Protótipo

Neste capítulo é apresentado um protótipo de aplicação que embarca um modelo com os melhores resultados obtidos nos diversos experimentos apresentados no capítulo anterior. O objetivo é discorrer detalhes de sua arquitetura, apresentando seus principais módulos e funcionalidades, para tanto, está dividido da seguinte forma: na Seção 6.1 é apresentada detalhes técnicos de construção do protótipo. A Seção 6.2 apresenta seus principais componentes. Por fim, na 6.3 é elencada a proposta de implementação de um modelo julgador a fim de reduzir erros individuais de predição, utilizando-se para tal, os três modelos de aprendizado de máquina resultantes dos experimentos do Capitulo 5.

6.1 Construção do Protótipo

Esta seção descreve o LTA (Laboratório para Tratamento de Artefatos), um protótipo de aplicação de software para predição de códigos maliciosos, com o objetivo de apresentar, como prova de conceito, uma implementação prática do resultados dos experimentos de aprendizado de máquina.

O objetivo do aplicativo é permitir que usuários utilizem o classificador para detecção de arquivos maliciosos, sem a necessidade de conhecimento prévio sobre análise de *malwares*. Desta forma, o produto final é um protótipo de aplicativo que implementa o classificador resultante dos modelos de aprendizado de máquina treinados no Capítulo 5, para a detecção de códigos maliciosos.

Para a construção do protótipo, foi necessário também a construção e modificação de um conjunto de ferramentas, descritas na Seção 4.4, conforme demonstrado na estrutura de arquivos constante na Figura 6.1. Os softwares desenvolvidos para o protótipo, tratam-se em sua maioria, de programas *scripts* da linguagem *Python*, integrados a um ambiente Web, desenvolvido no *Django Web framework*. A escolha dessas tecnologias é aderente ao escopo da pesquisa, pelo uso de soluções baseadas em *Python*, e objetiva a

sua manutenibilidade, a compatibilidade entre seus principais componentes, bem como a sua futura evolução.

A Figura 6.1 apresenta os principais componentes de software do LTA. Nesta abstração, é possível notar os principais *scripts*, os quais estão divididos em três módulos principais: *Parsing*, *Preditions* e *Services*. Os *scripts* em destaque na cor verde foram especialmente construídos, enquanto os destacados na cor azul, são ferramentas pré-existentes, as quais foram especialmente modificadas para a presente pesquisa. Os principais componentes são descritos a seguir:

- os componentes do *Parsing*, incluem todos os arquivos necessários a identificação, desmontagem e *Parsing* dos binários, tais como **Separador.py, PECheck.py, PE-Frame.py** e **PEScanner.py**;
- os componentes do *Predictions*, incluem os *scripts* de predição **predict.py** e julgamento (**Referee.py**); e
- os componentes do *Services*, incluem os *scripts* de consulta aos serviços externos de análise de *malwares* Virustotal e Malshare, através de suas respectivas API: **VirusTotal.py** e **Malshare.py** (vide Listagem 6.2). Tal consulta objetiva oferecer uma maior robustez ao diagnóstico final, podendo também ser uma ferramenta de validação quanto à acurácia do preditor.

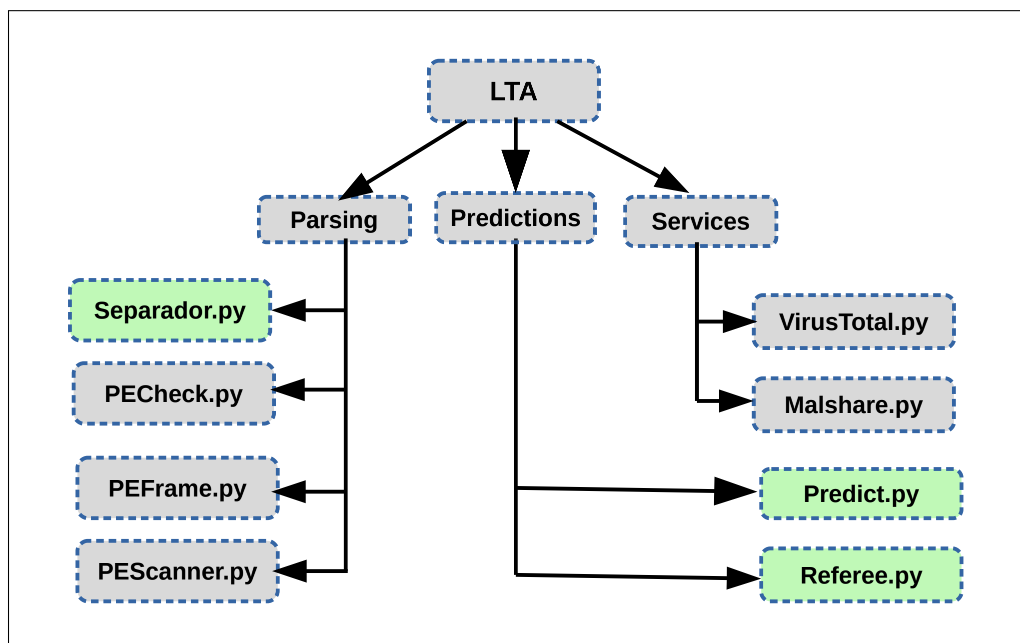

Figura 6.1: Principais ferramentas do protótipo. Fonte: o autor

Para uma maior portabilidade, o protótipo foi embarcado em um sistema de virtualização baseado na tecnologia de *containers Docker*[1]. Cada *container* opera com um espaço de memória virtualizado e isolado do espaço de memória da máquina hospedeira. Além do mais, todas as dependências estão encapsuladas no protótipo e podem ser executadas em praticamente qualquer plataforma, seja Microsoft Windows, Apple MacOS e GNU/Linux[2], conforme abordado na Seção 2.3.3. A Listagem 6.1 apresenta extrato do script de inicialização do *container* com o protótipo. Uma vez inicializado o *container*, o aplicativo pode ser utilizado em modo único ou de múltiplos usuários, através de sua interface Web.

```
1  ##########
2  FROM remnux
3  ENV USER docky
4  RUN pip install --upgrade pip
5  RUN pip install virtualenv
6  COPY docker/nopass /etc/sudoers.d/
7  RUN useradd -ms /bin/bash $USER
8  RUN usermod --password $USER $USER
9  WORKDIR /home/$USER/
10 COPY project/install install
11 RUN pwd && ls -lha
12 RUN chown -R $USER:$USER install
13 USER $USER
14 RUN pwd && ls -lha
15 RUN virtualenv --python=/usr/bin/python2.7 env2
16 RUN virtualenv --python=/usr/bin/python3.4 env3
17 RUN env2/bin/pip install -r install/requirements-2.txt
18 RUN env3/bin/pip install -r install/requirements-3.txt
19 EXPOSE 8080
20 CMD ["/bin/bash"]
21 #########
```

Listagem 6.1: script para inicializar o *container docker* com o protótipo

6.1.1 Principais Funcionalidades

O protótipo executa a seguintes funções principais:

- identificação do tipo de arquivo;
- desmontagem e *parsing* do arquivo a ser analisado;
- extração do *hash* e dos atributos referentes à amostra a ser analisada;
- busca em bases públicas de análise de *malware* pela existência do *hashs* extraído da amostra a ser analisada;

[1] https://www.docker.com/
[2] http://www.gnu.org

- aplicação de modelo de aprendizado de máquina na predição da amostra analisada; e
- predição final baseado nos resultados do aprendizado de máquina combinados com os resultados da consulta às bases públicas.

6.2 Componentes da Arquitetura

Esta seção descreve os principais componentes funcionais da arquitetura. Todos os componentes possuem função específica e serão descritos segundo o fluxo de execução, conforme o digrama de visão geral apresentada na Figura 6.2. Dessa forma, a arquitetura é composta pelos seguintes componentes funcionais: (1) módulo de submissão, (2) módulo de identificação, (3) módulo de consulta externa, (4) módulo de *parsing* e (5) módulo de resultados. Os componentes são descritos a seguir para um melhor entendimento da solução proposta.

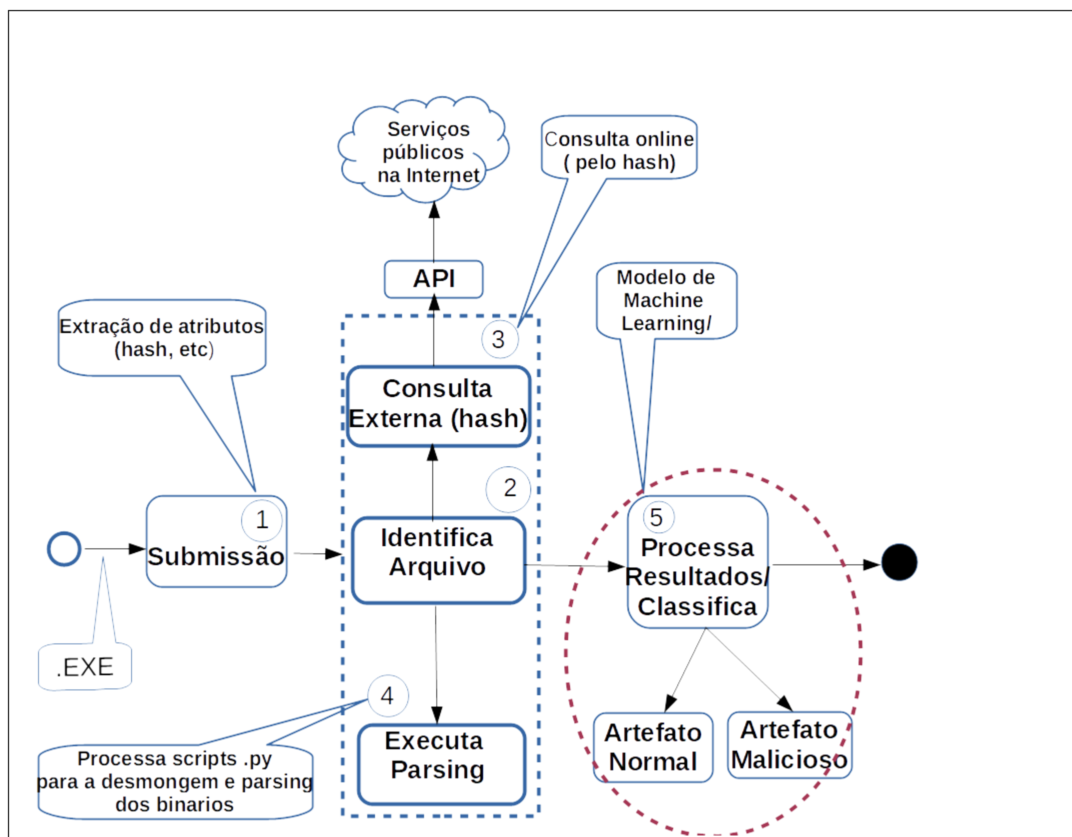

Figura 6.2: Diagrama de visão geral. Fonte: o autor

1) **Módulo de submissão**, tem a função de coleta de *malwares*, e pode ser feita de diversas formas, como por exemplo pela interface Web, na qual é possível realizar a

submissão direta de um arquivo a ser analisado, conforme figura 6.3, ou diretamente por meio das ferramentas (*scripts*) criadas para este fim. Um exemplo é a ferramenta *parser.py*, que além da desmontagem do binários, faz também a consulta via *hash*, à entidades externas. O processo de submissão é integrado com outras atividades, dessa forma, assim que um novo binário é inserido no módulo de submissão, vários processos serão disparados.

Figura 6.3: Tela de submissão de arquivos binários. Fonte: o autor

2) **Módulo de identificação**, impede que um arquivo diferente de um binário seja analisado e gere uma falso resultado, um vez que o módulo de *machine learning* foi treinado para analisar apenas os arquivos executáveis do *Microsoft Windows*. Os arquivos coletados são armazenados em um diretório chamado "*Upload*", e são renomeados segundo o seu próprio resumo criptográfico (*hash* MD5). Esse processo busca evitar que o mesmo arquivo seja armazenado de forma repetitiva, e serve de entrada para o módulo de consulta à entidades externas.

3) **Módulo de consulta à entidades externas**, é o componente responsável por submeter os *hash* dos arquivos aos serviços online Virustotal [3] [126], e o malshare[4] [127]. Essas ferramentas são extensivamente utilizadas para análise de malware [128, 41, 51, 24, 3, 11]. O processo de submissão à essas ferramentas utiliza APIs para a consulta ao Virustotal e ao Malshare, conforme apresentado na Listagem 6.2. A Listagem 6.3

[3]https://www.virustotal.com
[4]https://www.malshare.com/

apresenta as contas de correio especialmente criadas para consulta ao Virustotal. A versão gratuita desse serviço permite, no máximo, 15 consultas por minuto, enquanto o Malshare, em sua versão gratuita, permite ao público até 2.000 consultas por dia.

```
1 ########## Consulta Virustotal ###########
2 from interface import implements
3 from base import BaseChecker
4 import hashlib, sys, json, requests
5
6 VIRUS_TOTAL_URL = 'https://www.virustotal.com/vtapi/v2/file/report'
7 API_KEY = '85129abff09d44027871ba7f5166a50ff18fc925XXXXXXXXXXXXXXXXX'
8
9 class VirusTotalChecker(implements(BaseChecker)):
10   def __init__(self, path):
11   self.path = path
12     def process(self):
13     f = open(self.path,'rb')
14     content = f.read()
15     hash_md5 = hashlib.md5(content).hexdigest()
16
17     params = {
18       'apikey': API_KEY,
19       'resource': hash_md5
20     }
21     response = requests.get(VIRUS_TOTAL_URL, params=params)
22     j = response.json()
23     self.result = j
24     def is_malware(self):
25     return round( float(self.result['positives']) / float(self.result['total']), 4)
26 ########## Consulta Malshare ###########
27 from .base import ServiceChecker, ResultEnum
28 import hashlib, sys, json, requests, hmac
29 MALSHARE\_URL = 'https://malshare.com/api.php?api_key={}&action=search&query={}'
30 API\_KEY = '293f443abd0c6005fc27332709fc09d8b79XXXXXXXXXXXXXXXXXXXXXXXXXXX'
31 # Standard keys allow 2000 API calls per day
32 # 2000 samples per day
```

Listagem 6.2: Extrato do código de consulta ao Virustotal e ao Malshare

```
1 ###### contas de correio eletrônico para consulta ao Virustotal e Malwhare
2 Email: viruschecker001@gmail.com
3 Email: viruschecker002@gmail.com
4 Email: viruschecker003@gmail.com
5 ########
6 ##Chaves de acesso
7 001 4239b36810f7f8f2a0d2384a8bf6df0f1f2ae483XXXXXXXXXXXXXXXXXXXXXXXX
8 002 b5e95dfa2e97110f69c95adf0cff4cb49d5a9679XXXXXXXXXXXXXXXXXXXXXXXX
9 003 94c80b89da2cf6d12314642fad6dcaa7e00fdf67XXXXXXXXXXXXXXXXXXXXXXXX
```

Listagem 6.3: Lista de correio criadas para LTA

4) **Módulo de *Parsing***, é componente responsável pelo processamento dos binários. É composto por um conjunto de softwares, construídos e modificados, para a desmonta-

gem e *parsing* dos binários, conforme detalhamento constante das Seções 4.4 e 6.1. A Figura 6.1 apresenta os principais componente de software utilizados por este módulo.

5) **Módulo de Resultados**, implementa o modelo de aprendizado de máquina, em conjunto com os resultados obtidos das consultas externas. Ao final do processo de análise, é gerado um diagnóstico que aponta, se os atributos do artefato analisado, são compatíveis com as características de um arquivo malicioso, conforme os modelos treinados, combinado com os relatórios dos serviços externos consultados. Com base na predição e nas consultas externas, são elencados 3 resultados possíveis:

   - **normal** - se consta como negativo em todos os analisadores;
   - **suspeito** - se consta apareceu como positivo em pelo menos 1 analisador; e
   - **malicioso** - se consta como positivo em pelo menos 2 dos analisadores.

   Além do diagnóstico acima, o protótipo oferece dois tipos de relatórios, o primeiro contém relatórios obtidos dos Virustotal e Malshare (quanto houver), resultantes da consulta à entidades externas. O segundo tipo, resultante do processo de desmontagem (*disassembly*), permite o acesso aos arquivos de dados brutos (*raw bytes*), conforme apresenta a Figura 6.4. Dessa forma, o protótipo oferece também o acesso à informações para uma análise mais detalhada, disponibilizadas pelo Virustotal (conforme apresentada na figura 6.5) e pelas ferramentas de desmontagem utilizadas pelo protótipo, o que permitirá ao uma análise mais acurada do binário.

Figura 6.4: Tela de resultados. Fonte: o autor.

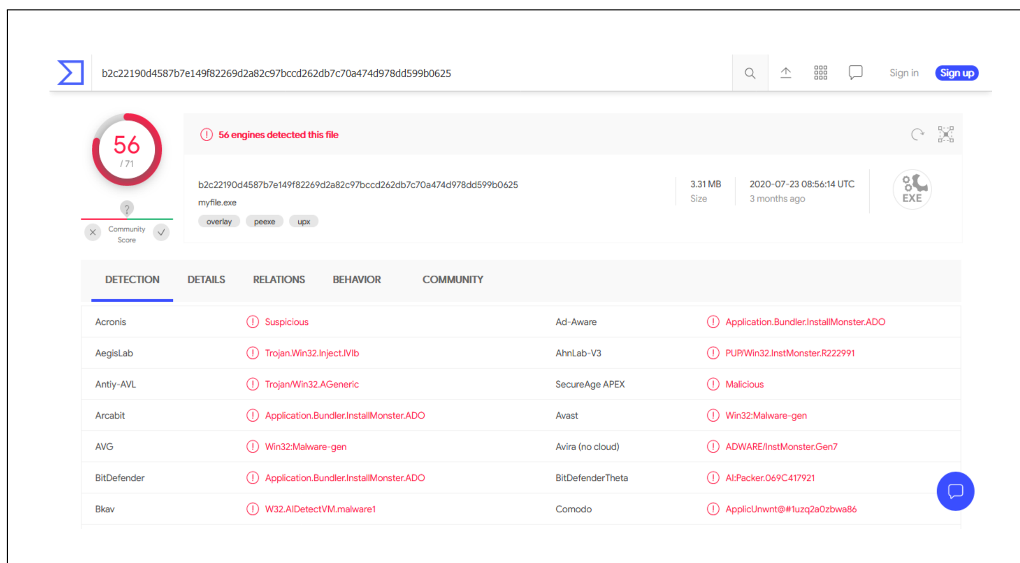

Figura 6.5: Relatório do Virustotal dos arquivos binários. Fonte: Virustotal.

6.3 Construção do Julgador

Nesta seção será abordada a proposta de combinar as predições de múltiplos classificadores, considerando que a combinação de predições de vários algoritmos poderia melhorar a capacidade de generalização e a robustez do modelo final em relação a um único preditor [76], conforme descrito na Seção 4.2.6.

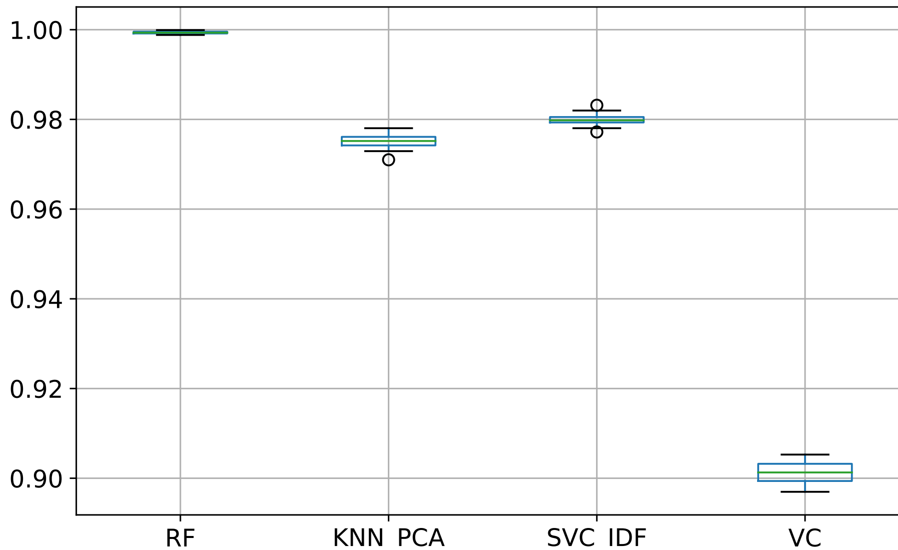

Figura 6.6: Bloxplot comparativo da acurácia entre os modelos e o *Voting Classifier* (VC). Fonte: o autor.

Avaliou-se então, o algoritmo *Voting Classifier* [2], o qual apresentou um acurácia menor que os modelos treinados, conforme pode ser observado na Figura 6.6. A referida figura apresenta um gráfico tipo "*boxplot*", com a comparação da acurácia obtida pelos três modelos vencedores nos experimentos realizados (vide Seção 5.3) e aplicação do algoritmo *Voting Classifier*[5] como um julgador dos diferentes resultados obtidos.

Dessa forma, e como uma forma alternativa de combinar os diferentes preditores, resolveu-se desenvolver um programa que permita utilizar os três modelos preditores no protótipo, e adicionalmente, construiu-se um quarto modelo, chamado julgador, ou árbitro (*Referee.joblib*), como forma de balancear resultados diferentes, e possivelmente minimizar a taxa de erros advinda de um único preditor.

```
1  ##########
2      clf_exp1 =joblib.load(self.JOBLIB_NORMAL)['model']
3      if 21 in exp:
4        clf_tfidf = joblib.load(self.JOBLIB_TFIDF_TEXT)['model']
5      if 22 in exp:
6        clf_tfidf = joblib.load(self.JOBLIB_TFIDF_NUM)['model']
7      clf_pca = joblib.load(self.JOBLIB_PCA)['model']
8      data_referee = pd.DataFrame()
9      data_referee["Random Forest"]=clf_exp1.predict(dados)
10     data_referee["SVC"]=clf_tfidf.predict(dados_tfidf)
11     data_referee["KNN"]=clf_pca.predict(df_pca)
12     results_referee = run_ml_gs(models_pca, data_referee, target)
13     best_clf_referee = max(results_referee, key=lambda x: x['best_score'])
```

[5]https://scikit-learn.org/stable/modules/ensemble.html/voting-classifier

```
14      clf_referee = best_clf_referee['best_model']
15 #########
```

Listagem 6.4: Extrato do programa que implementa o modelo julgador

A figura 6.7 apresenta o diagrama de componentes do modelo julgador (*Referee.py*). Na referida figura se verifica que o protótipo recebe como entrada, os métodos de transformações dos dados aplicados nos experimentos, e os arquivos *.joblib* com os modelos preditores resultantes. Para decisão final, o arbitro é treinado com os resultados da predição dos outros modelos, e apresenta sua predição com base nesse processo de aprendizado.

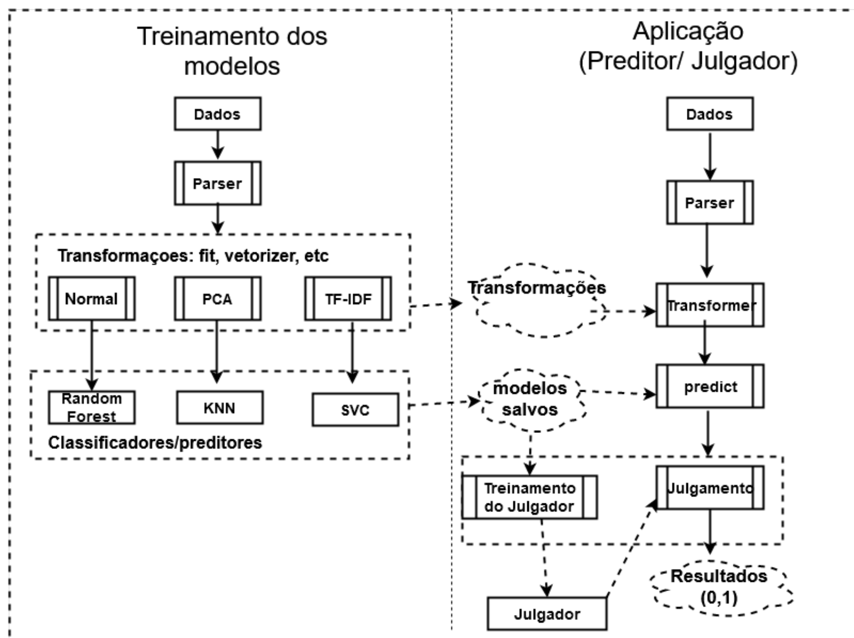

Figura 6.7: Diagrama de construção do modelo julgador. Fonte: o autor.

A Tabela 6.1 ilustra a saída do Julgador, com *hash* do binário analisado, e o resultado dos preditores, com 0 indicando que o binário foi classificado como um arquivo benigno, e 1, indicando que a binário foi classificado como malicioso. É possível observar que, em alguma predições, o SVC e o *Random Forest* identificaram as amostras como benignas (0), enquanto os demais classificadores (incluindo o julgador), classificaram a amostra como maliciosa (1). Neste caso, destaca-se a importância do Julgador para arbitrar a decisão final do preditor.

Tabela 6.1: Extrato de resultados do julgador

Hash	**Randon Forest**	**SVC**	**KNN**	**Referee**
09a2d2001f7c84f76407770a5efe0c19	1	1	1	1
00aeff6050e5f3b6cca059eff3a923f5	1	1	1	1
5e2328b765120806bc74658ef38a1e91	1	0	1	1
32f8d58f8d06afcb99038d935dfcf71b	0	1	1	1

6.4 Considerações Finais

Neste capítulo apresentou-se o protótipo de aplicação especialmente desenvolvido como prova de conceito da utilização prática dos resultados dos experimentos de aprendizado de máquina proposta pela presente pesquisa. Adicionalmente, abordou-se a proposta de implementação de um modelo julgador (árbitro), de modo que se minimize os erros individuais de predição dos modelos resultantes do processo de treinamento, e deste modo, obtenha-se uma melhor acurácia do protótipo.

Capítulo 7

Conclusões e Trabalhos Futuros

A segurança em redes de computadores representa um grande desafio para os governos e organizações na atualidade, e apesar da contínua evolução dos sistemas de segurança, centenas de milhares de amostras de códigos maliciosos, também referenciados na literatura como *malwares*, surgem todos os dias, afetando ou degradando os sistemas tradicionais de detecção tradicionais.

Neste contexto, técnicas de aprendizado de máquina ou *Machine Learning* (ML), baseados em modelos preditivos tem sido investigados na literatura como uma ferramenta promissora para a detecção de códigos maliciosos. Em geral, algoritmos de aprendizado de máquina podem ser categorizados como preditivos (ou aprendizado supervisionado) ou por descoberta de padrões (aprendizado não supervisionado). A aprendizagem supervisionada consiste em usar dados de treinamento rotulados para ajustar um modelo matemático, e em seguida, aplicar este modelo treinado para prever a ocorrência em dados novos, enquanto algoritmos de aprendizagem não supervisionados identificam padrões ocultos em dados não rotulados.

Dessa forma, o presente trabalho teve como objetivo explorar técnicas de aprendizagem supervisionada de máquina, para treinamento de um classificador a ser empregado em um sistema de detecção de *malwares*, baseado no processamento de características extraídas através de técnicas de análise estática de arquivos executáveis do sistema operacional *Windows*, conhecidos também como arquivos PE.

Inicialmente, e a fim de proporcionar uma contextualização sobre o tema, apresentou-se um breve introdução, e para uma melhor compreensão sobre os elementos que compõem os experimentos realizados, apresentaram-se os conceitos relacionados a *malwares*, bem como uma visão geral das principais características dos algoritmos de aprendizagem de máquina utilizados.

Em seguida, apresentou-se um conjunto de trabalhos correlatos, com o objetivo de posicionar a pesquisa em relação ao tema. Embora a revisão da literatura tenha reportado

trabalhos com resultados muito promissores, a presente pesquisa buscou uma abordagem diferente ao explorar um conjunto distinto de experimentos, que incluiu o emprego da técnica de redução da dimensionalidade e algoritmo de processamento de textos.

Como objetivo geral, a pesquisa buscou, a partir de pesquisas anteriores sobre o aprendizado de máquina na detecção de *malwares*, coletar subsídios suficientes para embasar os experimentos realizados, incluindo-se as técnicas de extração e seleção de atributos que fossem suficientemente relevantes para se treinar um classificador a ser embarcado em sistema de detecção de *malwares*. Para alcançar tal objetivo, a presente pesquisa, explorou, através de experimentações empíricas, um dataset de mais de 14.000 arquivos PE, que serviram como entrada para aplicação dos seguintes algoritmos de aprendizagem supervisionada: *K-Nearest Neighbors* (KNN), *C-Support Vector Classification* (SVC), *Gaussian Naive Bayes* (GaussianNB), *Decision Tree* (DT) e *Random Forest* (RF).

A pesquisa utilizou, como base, ferramentas na linguagem *Python*, a qual possui uma vasta quantidade de bibliotecas para manipulação de dados e aprendizado e máquina, tais como: *Notebooks*, *IPython*, *scikit-learn*, *NumPy*, *SciPy*, dentre outros. Adicionalmente, foram adaptadas ferramentas pré-existentes para desmontagem dos arquivos binários, extração e seleção de atributos relevantes, além da construção de *scripts* para manipulação, tratamento, limpeza dos dados, e treinamento dos classificadores para detecção de *malwares*.

Para testar a viabilidade da proposta, foram realizados três diferentes experimentos: no primeiro experimento, buscou-se a simplicidade do modelo, em uma abordagem com *Cross Validation 10-fold*. Em tal experimento, buscou-se uma prova de conceito de que os atributos extraídos, e sua aplicação direta como entrada dos algoritmos, seriam suficientes para treinar obter uma boa acurácia do classificador, enquanto o segundo experimento consistiu na avaliação da técnica de redução da dimensionalidade, ou seja, da redução do número de atributos, buscando-se também reduzir o impacto da importância individual de alguns atributos para discriminar *malwares*. Por fim, no terceiro experimento, avaliou-se a capacidade de detecção de *malwares* a partir do uso de atributos textuais (processados com o algoritmo TF-IDF) presentes nos conjuntos de dados. Como resultados experimentais, durante o treinamento, o algoritmo *Random Forest* apresentou o melhor desempenho nos três experimentos, com acurácia de 97,92%, 95,61% e 98,66%, respectivamente.

Para validar a capacidade de generalização dos modelos, ou seja para avaliação a capacidade de predição em dados não visto antes, simulando o mundo real, realizou-se uma uma avaliação com um *dataset* composto de 1.974 arquivos (vide Tabela 5.6). Como resultados, o algoritmo *Random Forest* apresentou o melhor desempenho nos experimentos 1, com 86,43%, enquanto o KNN apresentou o melhor desempenho no experimento 2, com 80,31% e no experimento 3, o SVC mostrou o melhor resultado, com 69,09%, de acordo

com a performance representada pelas Curvas ROC e AUC presente na Figura 5.8.

Adicionalmente, aplicou-se o método de redução de dimensionalidade utilizado na pesquisa em um conjunto de dados chamado de Clamp [16], a fim de se verificar a robustez do método PCA em outro conjunto de dados (vide Tabela 5.7).

Por fim, a pesquisa apresentou um protótipo de aplicação para um ambiente de detecção de *malwares*, que incorpora os classificadores resultantes do treinamento dos algoritmos de aprendizado de máquina. Deste modo, a pesquisa concluiu um ciclo completo da exploração de técnicas de análise estática em conjunto com técnicas de aprendizado de máquina como um método rápido para a construção de uma ferramenta de triagem e detecção de códigos maliciosos. Como contribuição, a pesquisa proporcionou para a instituição Exército Brasileiro, um protótipo de ferramenta a ser embarcado em uma aplicação de triagem e detecção de *malwares* em seus sistemas de proteção da rede corporativa. Espera-se, com base nos resultados empíricos alcançados, que a ferramenta mantenha a capacidade de identificar arquivos maliciosos, e que esta possa oferecer à instituição, a flexibilidade de expansão e adaptação necessárias a sua plena utilização.

Como contribuição adicional, para fins de reprodutibilidade e também em apoio a outras pesquisas relacionadas, os códigos e conjuntos de dados usados na presente pesquisa estão disponíveis à comunidade científica para download no GitHub [1].

Como limitações da pesquisa, ressalta-se que as características dos *malwares* evoluem muito rapidamente, e portanto, não há garantia de que o modelo proposto mantenha sua precisão, quando aplicado em um novo conjunto de dados. Dessa forma, o modelo precisa ser treinado periodicamente, e sempre que receber novos arquivos, a fim de manter sua precisão, além da necessidade de se manter o conjunto de dados atualizado com as variantes mais recentes de *malwares*.

Adicionalmente, ressalta-se também que a replicação desse método em ambiente de produção requer a análise de outras condicionantes, tais como a complexidade e custo computacional associados. Por fim, é importante destacar a existência de riscos associados ao processo de treinamento ou retroalimentação, pela utilização de bases tendenciosas ou maliciosamente adulteradas, o que certamente resultará em previsões não confiáveis, afetando a acurácia do modelo.

Como trabalhos futuros, à medida que bases de *malwares* maiores e confiáveis estejam disponíveis, planeja-se a ampliação da pesquisa com a incorporação de algoritmos de redes neurais, a fim de comparar com os resultados com os obtidos na presente pesquisa. Adicionalmente, pretende-se avançar na pesquisa para detecção de *malwares* em outros tipos de arquivos, tais como PDF, HTML, além de *malwares* para dispositivos móveis. Por fim, outro aprimoramento do modelo pode vir da combinação da análise estática com

[1] `https://github.com/alejr-git/Mal_Lab_ML_Project.git`

a dinâmica, associando-se dados obtidos de características presentes no código com dados extraídos em tempo de execução dos arquivos analisados.

Referências

[1] Souri, Alireza e Rahil Hosseini: *A state-of-the-art survey of malware detection approaches using data mining techniques.* Human-centric Computing and Information Sciences, 8(1):3, 2018. xii, 21

[2] Pedregosa, F., G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot e E. Duchesnay: *Scikit-learn: Machine learning in Python.* Journal of Machine Learning Research (JMLR), 12:2825–2830, 2011, ISSN 1532-4435. xii, 22, 23, 24, 25, 26, 27, 28, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 64, 65, 72, 76, 95

[3] Jones, K. J. e Y. Wang: *Malgazer: An automated malware classifier with running window entropy and machine learning.* Em *2020 Sixth International Conference on Mobile And Secure Services (MobiSecServ), Miami Beach, FL, USA, USA*, páginas 1–6. IEEE, 2020, ISBN 978-1-7281-5797-9. xii, 4, 20, 21, 23, 24, 25, 28, 35, 36, 46, 47, 49, 50, 52, 55, 59, 76, 91

[4] Pietrek, Matt: *An in-depth look into the win32 portable executable file format.* Disponível em: <https://docs.microsoft.com/en-us/archive/msdn-magazine/2002/february/inside-windows-win32-portable-executable-file-format-in-detail>. [Acesso em 08 set 2020], 2002. xii, 44, 51, 62, 63

[5] Chebbi, Chiheb: *Mastering Machine Learning for Penetration Testing.* Packt Publishing Limited, 2018, ISBN 9781788997409. xii, 61, 65

[6] Ceron, João M, Lisandro Granville e Liane Tarouco: *Taxonomia de malwares: Uma avaliação dos malwares automaticamente propagados na rede.* Anais do IX Simpósio Brasileiro em Segurança da Informação e de Sistemas Computacionais, 2009. 1, 18

[7] Lindsay, Jon R.: *Stuxnet and the limits of cyber warfare.* Security Studies, DOI: 10.1080/09636412.2013.816122, 22(3):365–404, 2013. 1

[8] BRASIL, CTIR Gov: *Alerta nº 04/2017–ataques de ransomware petrwrap/petya.* Disponível em: <https://www.ctir.gov.br/arquivos> [Acesso em: 08 set 2020], 2017. 1, 11

[9] USA, US Cert: *Indicators associated with WannaCry ransomware.* Disponivel em: https://www.us-cert.gov/ncas/alerts/TA17-132A. [Acesso em: 08 set 2020], 2017. 1

[10] Ceron, João Marcelo: *Arquitetura Distribuída e Automatizada para Mitigação de Botnet Baseada em Análise Dinâmica de Malwares*. Tese de Mestrado, Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2010. 1, 8, 10, 13, 14

[11] Mangialardo, RJ: *Integrando as análises estática de dinâmica na identificação de malwares utilizando aprendizado de máquina*. Master's thesis, Mestrado em Sistemas e Computação, Instituto Militar de Engenharia, Rio de Janeiro, página 105, 2015. 2, 9, 47, 91

[12] BRASIL, MD: *Diretriz md 14/2009*. Disponível em: <http://www.planalto.gov.br/>. [Acesso em 08 set 2020], 2008. 2

[13] Melo, Laerte Peotta de, Dino Macedo Amaral, Flavio Sakakibara, André Resende de Almeida, Rafael Timóteo de Sousa Jr e Anderson Nascimento: *Análise de malware: Investigaçao de códigos ma-liciosos através de uma abordagem prática*. SBSeg, 11:9–52, 2011. 2

[14] Lab, Kaspersky: *Machine learning methods for malware detection*. Disponível em: <https://media.kaspersky.com/en/enterprise-security/Kaspersky-Lab-Whitepaper-Machine-Learning.pdf>. [Acesso em 08 set 2020], 2020. 3, 20

[15] Raff, Edward, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro e Charles K Nicholas: *Malware detection by eating a whole exe*. Em *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*, 2018. 3, 4, 37, 42, 59

[16] Kumar, Ajit, KS Kuppusamy e G Aghila: *A learning model to detect maliciousness of portable executable using integrated feature set*. Journal of King Saud University-Computer and Information Sciences, 31(2):252–265, 2019, ISSN 1319-1578. 4, 20, 21, 35, 37, 47, 48, 50, 51, 73, 82, 84, 100

[17] Singla, Sanjam, Divya Bansal, Ekta Gandotra e Sanjeev Sofat: *A Novel Approach to Malware Detection using Static Classification*. International Journal of Computer Science and Information Security (IJCS), 13(3):5, 2015, ISSN 1947-5500. 4, 21, 35, 38, 42, 45

[18] Allix, Kevin, Tegawendé F Bissyandé, Quentin Jérome, Jacques Klein, Yves Le Traon *et al.*: *Empirical assessment of machine learning-based malware detectors for android*. Empirical Software Engineering, 21(1):183–211, 2016. 4, 45, 61, 85

[19] Milosevic, Nikola, Ali Dehghantanha e Kim Kwang Raymond Choo: *Machine learning aided Android malware classification*. Computers & Electrical Engineering, 61:266–274, julho 2017, ISSN 00457906. 4, 21

[20] Kwon, Young Man, Jae Ju An, Myung Jae Lim, Seongsoo Cho e Won Mo Gal: *Malware classification using simhash encoding and pca (mcsp)*. Symmetry, https://doi.org/10.3390/sym12050830, 12:830, maio 2020. 4, 37

[21] Shalaginov, Andrii, Sergii Banin, Ali Dehghantanha e Katrin Franke: *Machine learning aided static malware analysis: A survey and tutorial.* Em *Cyber Threat Intelligence*, páginas 7–45. Springer, 2018. 4, 5, 23, 24, 33, 47, 51, 52, 62, 64

[22] Poudyal, Subash, Kul Prasad Subedi e Dipankar Dasgupta: *A framework for analyzing ransomware using machine learning.* Em *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, páginas 1692–1699. IEEE, 2018. 4, 34, 45, 85

[23] Pham, Huu Danh, Tuan Dinh Le e Thanh Nguyen Vu: *Static pe malware detection using gradient boosting decision trees algorithm.* Em *International Conference on Future Data and Security Engineering*, páginas 228–236. Springer, 2018. 4, 33

[24] Ceron, João Marcelo: *MARS: uma arquitetura para análise de malwares utilizando SDN.* Tese de Doutoramento, (Doutorado em Sistemas Digitais) - Escola Politécnica, University of São Paulo, São Paulo., 2017. 4, 8, 9, 10, 11, 13, 15, 91

[25] Wazlawick, Raul: *Metodologia de pesquisa para ciência da computação*, volume 2. Elsevier Brasil, 2014, ISBN 9788535277821. 6

[26] Ciribelli, Marilda Corrêa: *Como elaborar uma dissertação de mestrado através da pesquisa científica.* página 66. Rio de Janeiro : 7Letras, 2003, ISBN 857577000810. 6

[27] Souppaya, Murugiah e Karen Scarfone: *Guide to malware incident prevention and handling for desktops and laptops*, 2013. 8, 11, 14, 17, 34

[28] Melo, Laerte Peotta de, Dino Macedo Amaral, Flavio Sakakibara, André Resende de Almeida, Rafael Timóteo de Sousa Jr e Anderson Nascimento: *Análise de malware: Investigaçao de códigos ma-liciosos através de uma abordagem prática.* SBSeg, 11:9–52, 2011. 8, 9, 10

[29] CERT.br: *Códigos maliciosos (malware).* Disponível em: <https://cartilha.cert.br/malware/>. [Acesso em 08 set 2020], 2017. 9

[30] Sihwail, Rami, Khairuddin Omar e Khairul Akram Zainol Ariffin: *A survey on malware analysis techniques: Static, dynamic, hybrid and memory analysis.* International Journal on Advanced Science, Engineering and Information Technology, arXiv:1710.09435, 8:1662–1671, 2018, ISSN 2088-5334. 9, 10, 11, 12, 13

[31] Kaur, Simarleen e Arvinder Kaur: *Detection of Malware of Code Clone using String Pattern Back Propagation Neural Network Algorithm.* Indian Journal of Science and Technology. doi: 10.17485/ijst/2016/v9i33/95880, 9(33), setembro 2016, ISSN 0974-5645, 0974-6846. 9, 12

[32] ARCOVERDE, Henrique Ferraz: *Malwares brasileiros: técnicas, alvos e tendências.* Tese de Mestrado, Dissertação (mestrado) - UFPE, Centro de Informática, Programa de Pós-graduação em Ciência da Computação, Universidade Federal de Pernambuco, 2013. `<https://repositorio.ufpe.br/handle/123456789/11832>`, acesso em 2020-01-12. 9, 11, 19

[33] Andrade, C: *Análise Automática de Malwares Utilizando as Técnicas de Sandbox e Aprendizado de Máquina*. Tese de Mestrado, Mestrado em Sistemas e Computação, Instituto Militar de Engenharia, Rio de Janeiro, jan 2013. `<http://www.comp.ime.eb.br/pos/arquivos/publicacoes/dissertacoes/2013/2013-Cesar.pdf>`, acesso em 2019-02-01. 10, 50

[34] Melo, Daniel Araújo: *ARCA - Alerts root cause analysis framework*. Tese de Mestrado, Ciência da Computação, Universidade Federal de Pernambuco, <https://repositorio.ufpe.br/handle/123456789/13946>, 2014. 10, 46, 47

[35] Barbara, Santa, Clemens Kolbitsch, Paolo Milani Comparetti e Ludovico Cavedon: *(54) Methods and Systemis Formalware Detection Based on Environment Dependent behavior*. página 12, 2016. 10

[36] Demertzis, Kostantinos: *Real-time Computational Intelligence Protection Framework Against Advanced Persistent Threats*. Cyber-Security and Information Warfare, Series: Cybercrime and Cybersecurity Research, Nova Science Publihsers, Inc. New York, ISBN 978-1-53614-386-7. 10

[37] Leite, Lindeberg Pessoa: *Agrupamento de malware por comportamento de execução usando lógica fuzzy*. Tese de Mestrado, ENE - Mestrado em Engenharia Elétrica (Dissertações), Universidade de Brasília, UnB, 2017. `<http://repositorio.unb.br/handle/10482/23118>`, acesso em 2020-02-12. 11

[38] Silva, Raphael Campos: *Malflow: um framework para geração automatizada de assinaturas de malwares baseado em fluxo de dados de rede*. Tese de Mestrado, Pós-Graduação em Ciência da Computação, Universidade Estadual Paulista, Unesp, 2017. 11

[39] USA, US Cert: *Ransomware*. Disponível em: <https://www.us-cert.gov/security-publications/Ransomware>, [Acesso em 08 set 2020]. 11

[40] Cakir, Bugra e Erdogan Dogdu: *Malware classification using deep learning methods*. Em *Proceedings of the ACMSE 2018 Conference*, páginas 1–5, 2018. 11, 20

[41] Ucci, Daniele, Leonardo Aniello e Roberto Baldoni: *Survey of Machine Learning Techniques for Malware Analysis*. Computers & Security, 81:123–147, março 2019, ISSN 01674048. 12, 49, 91

[42] Bao, Lingfeng, Tien Duy B. Le e David Lo: *Mining sandboxes: Are we there yet?* Em *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), doi:10.1109/SANER.2018.8330231*, páginas 445–455, Campobasso, 2018. ISBN 978-1-5386-4969-5. 12, 15

[43] Tam, Kimberly, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh e Lorenzo Cavallaro: *The Evolution of Android Malware and Android Analysis Techniques*. ACM Comput. Surv., 49(4):76:1–76:41, janeiro 2017, ISSN 0360-0300. `http://doi.acm.org/10.1145/3017427`, acesso em 2019-03-10. 12

[44] D'Orazio, Christian J, Kim Kwang Raymond Choo e Laurence T Yang: *Data exfiltration from internet of things devices: ios devices as case studies*. IEEE Internet of Things Journal, 4(2):524–535, 2017. 12

[45] Melo, Laerte Peotta de, Dino Macedo Amaral, Flavio Sakakibara, André Resende de Almeida, Rafael Timóteo de Sousa Jr e Anderson Nascimento: *Análise de malware: Investigaçao de códigos maliciosos através de uma abordagem prática*. SBSeg, 11:9–52, 2011. 13, 18

[46] Nath, Hiran V. e Babu M. Mehtre: *Static Malware Analysis Using Machine Learning Methods*. Communications in Computer and Information Science,vol 420. Springer, Berlin, Heidelberg, páginas 440–450, 2014. 13

[47] Wagner, Markus, Fabian Fischer, Robert Luh, Andrea Haberson, Alexander Rind, Daniel A. Keim e Wolfgang Aigner: *A survey of visualization systems for malware analysis*. The Eurographics Association, 2015. 13, 14, 15, 16, 19

[48] Egele, Manuel, Theodoor Scholte, Engin Kirda e Christopher Kruegel: *A survey on automated dynamic malware-analysis techniques and tools*. ACM Computing Surveys, 44(2):1–42, fevereiro 2012, ISSN 03600300. 14, 19

[49] Afianian, Amir, Salman Niksefat, Babak Sadeghiyan e David Baptiste: *Malware dynamic analysis evasion techniques: A survey*. 52(6), novembro 2019, ISSN 0360-0300. 14, 15, 16, 18

[50] Ács, Jakub: *Static detection of malicious pe files*. B.S. thesis, Czech Technical University, 2018. `<https://dspace.cvut.cz/bitstream/handle/10467/77271/F8-BP-2018-Acs-Jakub-thesis.pdf>`, acesso em 2020-01-10. 14, 21, 27, 35, 47, 54

[51] Sihwail, Rami, Khairuddin Omar e K A Z Ariffin: *A Survey on Malware Analysis Techniques: Static, Dynamic, Hybrid and Memory Analysis*. International Journal on Advanced Science, Engineering and Information Technology,8(4–2), página 10, 2018. 14, 45, 63, 91

[52] Bulazel, Alexei e Bülent Yener: *A Survey On Automated Dynamic Malware Analysis Evasion and Counter-Evasion: PC, Mobile, and Web*. Em *Proceedings of the 1st Reversing and Offensive-oriented Trends Symposium on - ROOTS*, páginas 1–21, Vienna, Austria, 2017. ACM Press, ISBN 978-1-4503-5321-2. 14, 15, 19

[53] Scarfone, K A, M P Souppaya e P Hoffman: *Guide to security for full virtualization technologies*. National Institute of Standards and Technology, NIST, SP 800-125, 2011. 15

[54] Tam, Kimberly, Ali Feizollah, Nor Badrul Anuar, Rosli Salleh e Lorenzo Cavallaro: *The evolution of android malware and android analysis techniques*. ACM Computing Surveys (CSUR), 49(4):1–41, 2017. 15, 49

[55] Botacin, Marcus Felipe: *Hardware-Assisted Malware Analysis Análise de Malware com Suporte de Hardware*. Tese de Mestrado, recurso online. Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação, Campinas, SP, 2017. 16, 18, 50

[56] Kaur, Navroop e Amit Kumar: *A Complete Dynamic Malware Analysis*. International Journal of Computer Applications, 135(4):20–25, fevereiro 2016, ISSN 09758887. 16, 45, 50

[57] Puig, Francesc Xavier, J.J. Villalobos, Ivan Rodero e Manish Parashar: *Exploring the potential of freebsd virtualization in containerized environments*. UCC '17, página 191–192, New York, NY, USA, 2017. Association for Computing Machinery, ISBN 9781450351492. `https://doi.org/10.1145/3147213.3149210`. 16, 17

[58] Sayadi, H., H. Mohammadi Makrani, O. Randive, S. M. P.D., S. Rafatirad e H. Homayoun: *Customized machine learning-based hardware-assisted malware detection in embedded devices*. Em *2018 17th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/ 12th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, páginas 1685–1688, 2018. 17

[59] Cozzolino, Marcelo Freire: *Detecção de Variantes Metamórficas de Malware por Comparação de Códigos Normalizados*. Tese de Mestrado, (Dissertação). Universidade de Brasília, Faculdade de Tecnologia, Departamento de Engenharia Elétrica, 2012. 18, 19

[60] Karapoola, Sareena, Chester Rebeiro, Unnati Parekh e Kamakoti Veezhinathan: *Towards identifying early indicators of a malware infection*. Asia CCS '19, página 679–681, New York, NY, USA, 2019. Association for Computing Machinery, ISBN 9781450367523. `https://doi.org/10.1145/3321705.3331006`. 18

[61] Andrade, Cesar Augusto Borges de, Claudio Gomes de Mello e Julio Cesar Duarte: *Malware Automatic Analysis*. Em *2013 BRICS Congress on Computational Intelligence and 11th Brazilian Congress on Computational Intelligence, doi: 10.1109/BRICS-CCI-CBIC.2013.119*, páginas 681–686, Ipojuca, Brazil, setembro 2013. IEEE, ISBN 978-1-4799-3194-1. 19, 43

[62] Rege, Manjeet e Raymond Blanch K Mbah: *Machine learning for cyber defense and attack*. The Seventh International Conference on Data Analytics. DATA ANALYTICS 2018.Athens, Greece. ISBN: 978-1-61208-681-1, página 83, 2018. 20

[63] Sathya, R e Annamma Abraham: *Comparison of supervised and unsupervised learning algorithms for pattern classification*. (IJARAI) International Journal of Advanced Research in Artificial Intelligence, 2(2):34–38, 2013, ISSN 2165-4069. 20

[64] Hardy, William, Lingwei Chen, Shifu Hou, Yanfang Ye e Xin Li: *Dl4md: A deep learning framework for intelligent malware detection*. Em *Proceedings of the International Conference on Data Mining (DMIN)*, página 61. The Steering Committee of The World Congress in Computer Science, Computer, 2016. `<http://www.dmin-2016.com/>`. 20, 37

[65] Le, Quan, Oisín Boydell, Brian Mac Namee e Mark Scanlon: *Deep learning at the shallow end: Malware classification for non-domain experts*. Digital Investigation, Elsevier, 26:S118–S126, 2018, ISSN 1742-2876. 20, 85

[66] Géron, Aurélien: *Hands-on machine learning with Scikit-Learn and TensorFlow: concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, Inc., 2019, ISBN 9781492032649. 21, 25, 26, 35, 52, 54, 57, 64, 73, 75

[67] Altaher, Altyeb, Sureswaran Ramadass e Ammar ALmomani: *An intelligent approach for malware detection in dual stack ipv4/ipv6 networks*. International Journal of Physical Sciences, 7(10):1607–1612, 2012, ISSN 1992-1950. 21, 35

[68] Ravi, Chandrasekar e R Manoharan: *Malware detection using windows api sequence and machine learning*. (IJCA)International Journal of Computer Applications, Citeseer, 43(17):12–16, 2012, ISSN 0975-8887. 21, 35

[69] Martins, João Antonio, Iago Sestrem Ochôa, Luis Augusto Silva, André Filipe Sales Mendes, Gabriel Villarrubia Gonzalez, Juan Francisco De Paz e Valderi Reis Quietinho Leithardt: *Pripro: A comparison of classification algorithms for managing receiving notifications in smart environments*, 2019. 21

[70] Cover, T. e P. Hart: *Nearest neighbor pattern classification*. IEEE Transactions on Information Theory, 13(1):21–27, 1967. 22

[71] Henke, Márcia, Clayton Santos, Eduardo Nunan, Eduardo Feitosa, Eulanda dos Santos e Eduardo Souto: *Aprendizagem de maquina para seguranca em redes de computadores: Metodos e aplicacoes*. Minicursos do XI Simpósio Brasileiro de Segurança da Informação e de Sistemas Computacionais (SBSeg 2011), 1:53–103, 2011, ISSN 2176-0063. 22, 23

[72] CAMARGO, Octavio: *Impacto de malwares reais em sistemas industriais com classificação supervisionada usando aprendizado de máquina*. Tese de Mestrado, (Dissertação) Mestrado em Sistemas e Computação, Instituto Militar de Engenharia, Rio de Janeiro, 2019. 22, 58

[73] Mitchell, Tom M *et al.*: *Machine learning*. McGraw-hill New York, 1997, ISBN 978-0-07-042807-2. 22

[74] Goldschmidt, Ronaldo, Emmanuel Passos e Eduardo Bezerra: *Data Mining: Conceitos, técnicas, algoritmos, orientações e aplicações*. Elsevier Brasil, 2ª edição, 2015, ISBN 978-85-352-7822-4. 24, 25, 58

[75] VanderPlas, Jake: *Python data science handbook: Essential tools for working with data*. " O'Reilly Media, Inc.", 2016, ISBN 1491912138. 24, 27, 28, 54, 57, 85

[76] Raschka, Sebastian e S RASCHKA: *Predictive modeling, supervised machine learning, and pattern classification: the big picture*, 2014. <https://sebastianraschka.com/Articles/2014_intro_supervised_learning.html>, acesso em 2020-09-08. 24, 26, 27, 55, 94

[77] Breiman, Leo: *Random forests.* Machine Learning, Springer Link. https://doi.org/10.1023/A:1010933404324, 45(1):5–32, 2001, ISSN 0885-6125. 25

[78] Ji, Shuiwang e Jieping Ye: *Linear dimensionality reduction for multi-label classification.* Em *IJCAI-09 - Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI International Joint Conference on Artificial Intelligence, páginas 1077–1082. International Joint Conferences on Artificial Intelligence, jan 2009, ISBN 9781577354260. 26

[79] Koren, Y. e L. Carmel: *Robust linear dimensionality reduction.* IEEE Transactions on Visualization and Computer Graphics. https://doi.org/10.1145/212094.212114, 10(4):459–470, july 2004. 26

[80] Maaten, Laurens Van Der, Eric Postma e Jaap Van Den Herik: *Dimensionality reduction: A comparative review.* Relatório Técnico, Tilburg University, 2009. `<http://citeseerx.ist.psu.edu/viewdoc/summary>,doi:10.1.1.364.6824>`, acesso em 2020-03-08. 26

[81] Jolliffe, Ian T. e Jorge Cadima: *Principal component analysis: a review and recent developments.* Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences, doi: 10.1098/rsta.2015.0202, 374, 2016. 26

[82] Fukunaga, K: *Introduction to statistical pattern classification academic press.* Academic Press, San Diego, California, USA, 2ª edição, 2013, ISBN 9780080478654. 26

[83] Hotelling, Harold: *Relations between two sets of variates.* Springer, New York, NY. https://doi.org/10.1007/978-1-4612-4380-9$_1$4, 1992, ISBN 978-1-4612-4380-9. 26

[84] Holland, Steven M: *Principal components analysis (pca).* Relatório Técnico, Department of Geology, University of Georgia, Athens, GA, 2008. `<http://strata.uga.edu/8370/handouts/pcaTutorial.pdf>`, acesso em 2020-02-13. 26, 27, 53, 54

[85] Bari, Anasse, Mohamed Chaouchi e Tommy Jung: *Predictive analytics for dummies.* John Wiley & Sons,Honoken, New Jersey, USA, 2016, ISBN 978-1-118-72920-5. 27

[86] Feldman, Ronen e James Sanger: *The Text Mining Handbook.* Cambridge University Press, https://doi.org/10.1017/CBO9780511546914, 2006, ISBN 9780511546914. 27

[87] Ramos, Juan *et al.*: *Using tf-idf to determine word relevance in document queries.* Em *Proceedings of the first instructional conference on machine learning*, volume 242, páginas 133–142. Piscataway, NJ, 2003. 27, 28, 76

[88] Bafna, Prafulla, Dhanya Pramod e Anagha Vaidya: *Document clustering: Tf-idf approach.* Em *2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT). doi: 10.1109/ICEEOT.2016.7754750*, páginas 61–66. IEEE, 2016. 27, 28, 76

[89] Killam, Richard, Paul Cook e Natalia Stakhanova: *Android malware classification through analysis of string literals.* Text Analytics for Cybersecurity and Online Safety (TA-COS), 2016. 27

[90] Kuriakose, Jikku e P Vinod: *Unknown metamorphic malware detection: Modelling with fewer relevant features and robust feature selection techniques.* IAENG International Journal of Computer Science, 42(2):139–151, 2015, ISSN 1819-9224. 28, 47, 57

[91] Qaiser, Shahzad e Ramsha Ali: *Text mining: use of tf-idf to examine the relevance of words to documents.* (IJCA) International Journal of Computer Applications, 181(1):25–29, 2018, ISSN 0975-8887. 28

[92] Mariano, Ari Melo e Maíra ROCHA Santos: *Revisão da Literatura: Apresentação de uma Abordagem Integradora.* XXVI Congreso Internacional AEDEM, página 17, 2017. 29

[93] virusshare: *Repository of malware samples to provide security researchers, incident responders, forensic analysts, and the morbidly curious access to samples of live malicious code.* Disponível em: <https://virusshare.com/> [Acesso em: 10 jan 2020]. 33, 42

[94] Sun, Zhi, Zhihong Rao, Jianfeng Chen, Rui Xu, Da He, Hui Yang e Jie Liu: *An opcode sequences analysis method for unknown malware detection.* Em *Proceedings of the 2019 2nd international conference on geoinformatics and data analysis*, páginas 15–19, 2019. 34

[95] Galante, Lucas, Paulo de Geus e Marcus Botacin: *Linux malware behavior description.* Revista dos Trabalhos de Iniciação Científica da UNICAMP, (27):1–1, nov. 2019. `https://econtents.bc.unicamp.br/eventos/index.php/pibic/article/view/2426`. 34

[96] Ceschin, Fabrício, Marcus Botacin, Heitor Murilo Gomes, Luiz S. Oliveira e André Grégio: *Shallow security: On the creation of adversarial variants to evade machine learning-based malware detectors.* Em *Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium*, ROOTS'19, New York, NY, USA, 2019. Association for Computing Machinery, ISBN 9781450377751. `https://doi.org/10.1145/3375894.3375898`. 35, 85

[97] Silva, Luis Felipe Bueno, Luan Nunes Utimura, Kelton Augusto Pontara da Costa, Márcia Aparecida Zanoli Meira e Silva e Simone das Graças Domingues Prado: *Study on machine learning techniques for botnet detection.* IEEE Latin America Transactions, 18(05):881–888, 2020. 35

[98] Kolosnjaji, Bojan, Apostolis Zarras, George Webster e Claudia Eckert: *Deep learning for classification of malware system call sequences.* Em *Australasian Joint Conference on Artificial Intelligence*, páginas 137–149. Springer, 2016. 37

[99] Allix, Kevin: *Challenges and Outlook in Machine Learning-based Malware Detection for Android.* Tese de Doutoramento, University of Luxembourg, Luxembourg, Luxembourg, 2015. 41, 53, 55, 59

[100] De Paola, Alessandra, Salvatore Gaglio, Giuseppe Lo Re e Marco Morana: *A hybrid system for malware detection on big data.* Em *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS). DOI: 10.1109/INFCOMW.2018.8406963*, páginas 45–50. IEEE, 2018. 45, 51

[101] Nath, Hiran V. e Babu M. Mehtre: *Static Malware Analysis Using Machine Learning Methods.* Communications in Computer and Information Science, vol 420. Springer, Berlin, Heidelberg, páginas 440–450, 2014. 45, 46, 47

[102] Saeed, Imtithal A, Ali Selamat e Ali MA Abuagoub: *A survey on malware and malware detection systems.* International Journal of Computer Applications. IJCA Journal, 67(16), 2013. 45

[103] Kyeom Cho, Tae Guen Kim, Yu Jin Shim Minsoo Ryu Eul Gyu Im: *Malware Analysis and Classification Using Sequence Alignments: Intelligent Automation & Soft Computing: Vol 22, No 3.* Intelligent Automation Soft Computing. DOI: 10.1080/10798587.2015.1118916, páginas 371–377, 2016. 46

[104] Ligh, Michael, Steven Adair, Blake Hartstein e Matthew Richard: *Malware analyst's cookbook and DVD: tools and techniques for fighting malicious code.* Wiley Publishing, 2010, ISBN 978-0-470-61303-0. 47, 68

[105] Kumar, Ajit, KS Kuppusamy e G Aghila: *Features for Detecting Malware on Computing Environments.* International Journal of Engineering and Applied Computer Science, doi: 10.24032/ijeacs/0102/01, 01(02):31–36, dezembro 2016. 47, 49, 50

[106] Ki, Youngjoon, Eunjin Kim e Huy Kang Kim: *A novel approach to detect malware based on api call sequence analysis.* International Journal of Distributed Sensor Networks, 11(6):659101, 2015. `<https://journals.sagepub.com/doi/pdf/10.1155/2015/659101>`, acesso em 2020-01-15. 47

[107] Pietrek, Matt: *Peering inside the pe: a tour of the win32 (r) portable executable file format.* Microsoft Systems Journal-US Edition, 9(3):15–38, 1994. 49

[108] Idrees, Fauzia, Muttukrishnan Rajarajan, Mauro Conti, Thomas M Chen e Yogachandran Rahulamathavan: *Pindroid: A novel android malware detection system using ensemble learning methods.* Computers & Security, 68:36–46, 2017. `<https://openaccess.city.ac.uk/id/eprint/17316/1/>`, acesso em 2020-01-31. 49, 59, 60

[109] Lyda, Robert e James Hamrock: *Using entropy analysis to find encrypted and packed malware.* IEEE Security & Privacy. DOI: 10.1109/MSP.2007.48, 5(2):40–45, 2007. 50

[110] Kirat, Dhilung, Lakshmanan Nataraj, Giovanni Vigna e BS Manjunath: *Sigmal: A static signal processing based malware triage.* Em *Proceedings of the 29th Annual Computer Security Applications Conference. Association for Computing Machinery. https://doi.org/10.1145/2523649.2523682*, páginas 89–98, New York, NY, USA, 2013. ISBN 9781450320153. 50

[111] Patil, Dharmaraj R. e J. B. Patil: *Malicious urls detection using decision tree classifiers and majority voting technique.* Cybernetics and Information Technologies, 18(1):11 – 29, 01 Mar. 2018. `https://content.sciendo.com/view/journals/cait/18/1/article-p11.xml`. 51

[112] Bhattacharya, Abhishek e Radha Tamal Goswami: *A hybrid community based rough set feature selection technique in android malware detection.* Em *Smart Trends in Systems, Security and Sustainability*, páginas 249–258. Springer, 2018. 52, 59

[113] Kelleher, John D, Brian Mac Namee e Aoife D'arcy: *Fundamentals of machine learning for predictive data analytics: algorithms, worked examples, and case studies.* MIT press, 2015. 53, 54, 59

[114] John, George H, Ron Kohavi e Karl Pfleger: *Irrelevant features and the subset selection problem.* Em *Machine Learning Proceedings 1994*, páginas 121–129. Elsevier, 1994, ISBN 9781483298184. 53

[115] Cawley, Gavin C e Nicola LC Talbot: *On over-fitting in model selection and subsequent selection bias in performance evaluation.* Journal of Machine Learning Research, 11(Jul):2079–2107, 2010. 54, 55

[116] Dietterich, Tom: *Overfitting and undercomputing in machine learning.* ACM computing surveys (CSUR). https://doi.org/10.1145/212094.212114, 27(3):326–327, 1995. 55

[117] Google: *Google's fast-paced, practical introduction to machine learning.* Disponível em: <https://developers.google.com/machine-learning/crash-course/classification>, [Acesso em 08 set 2020], 2020. 55, 72

[118] Kaner, Cem *et al.*: *Software engineering metrics: What do they measure and how do we know?* Em *10th International Symposium on Software Metrics, 2004. Proceedings.* IEEE CS, 2004. 57

[119] Damodaran, Anusha, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H. Austin e Mark Stamp: *A comparison of static, dynamic, and hybrid analysis for malware detection.* Journal of Computer Virology and Hacking Techniques, 13(1):1–12, fevereiro 2017, ISSN 2263-8733. 59

[120] Boughorbel, Sabri, Fethi Jarray e Mohammed El-Anbari: *Optimal classifier for imbalanced data using matthews correlation coefficient metric.* PloS one. e0177678. https://doi.org/10.1371/journal.pone.0177678, 12(6), 2017. 59

[121] Jimenez, Matthieu, Renaud Rwemalika, Mike Papadakis, Federica Sarro, Yves Le Traon e Mark Harman: *The importance of accounting for real-world labelling when predicting software vulnerabilities.* ESEC/FSE 2019, página 695–705, New York, NY, USA, 2019. Association for Computing Machinery, ISBN 9781450355728. https://doi.org/10.1145/3338906.3338941. 61, 85

[122] Van Rossum, Guido e Fred L Drake Jr: *Python tutorial.* Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995. 64

[123] Paffenroth, Randy e Xiangnan Kong: *Python in data science research and education.* Em *Proceedings of the 14th Python in Science Conference, doi: 10.25080/Majora-7b98e3ed-019*, páginas 164 – 170, 2015. 64

[124] Gandotra, Ekta, Divya Bansal e Sanjeev Sofat: *Malware analysis and classification a survey.* Journal of Information Security, 05(02):56–64, 2014, ISSN 2153-1234, 2153-1242. http://www.scirp.org/journal/doi.aspx?DOI=10.4236/jis.2014.52006, acesso em 2019-01-12. 85

[125] Economy, US: *Algorithmic bias detection and mitigation: Best practices and policies to reduce consumer harms.* Web site: https://www.brookings.edu/research/algorithmic-bias-detection-and-mitigation-best-practices-and-policies// -to-reduce-consumer-harms/>, [Acesso em 05 de Julho de 2020], 2019. 85

[126] Virustotal: *Analyze suspicious files and urls to detect types of malware, automatically.* Disponível em: <https://www.virustotal.com> [Acesso em: 08 set 2020]. 91

[127] Malshare: *The malshare project is a collaborative effort to create a community driven public malware repository.* Disponível em: <https://malshare.com/> [Acesso em: 08 set 2020]. 91

[128] Bulazel, Alexei e Bülent Yener: *A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web.* Em *Proceedings of the 1st Reversing and Offensive-Oriented Trends Symposium, doi:10.1145/3150376.3150378*, ROOTS, New York, NY, USA, 2017. Association for Computing Machinery, ISBN 9781450353212. 91

Apêndice A

Ferramenta de Extração dos Atributos

A.1 Parser.py

```python
  ### importação das bibliotecas
  import hashlib
  import os
  import subprocess
  import json
from os.path import expanduser
from argparse import ArgumentParser

HOME = expanduser("~")
BIN = HOME + '/virus/bin'
RENMUX = '/opt/remnux-scripts'

FILES = HOME + '/virus/files'
TEMP = FILES + '/temp'
ERROS = FILES + '/results/erros'

def clean_spaces(s):
    while '  ' in s:
        s = s.replace('  ', ' ')
    s = s.strip()
    return s

def search_dict(d, l):
    if not d: return None

    rv = d

    for key in l:
        if key in rv:
            rv = rv[key]
        else:
            return None
```

```
33
34          return rv
35
36      def len_or_none(l):
37          try:
38              return len(l)
39          except:
40              return None
41
42      def print_dict(d):
43          for k, v in d.items():
44              print(k, v)
45
46      class Check:
47          name = ''
48
49          def __init__(self, inputpath, cache=False):
50              self.cache = cache
51              self.hascache = False
52              self.inputpath = inputpath
53              self.hashname = hashlib.md5(open(self.inputpath,'rb').read()).hexdigest()
54              self.logpath = '{temp}/{name}/{hashname}.log'.format(
55                  temp = TEMP,
56                  name = self.name,
57                  hashname = self.hashname
58              )
59              self.parsed = {}
60
61          def run(self):
62              cmd = '{checker} {inputpath} >| {logpath}'.format(
63                  checker = self.checker,
64                  inputpath = self.inputpath,
65                  logpath = self.logpath
66              )
67              print('\n>>>', cmd)
68
69              #try to parse cached log files
70              # try:
71              #     open(self.logpath, 'r+')
72              #     self.hascache = True
73              # except:
74              #     self.hascache = False
75
76              # create new log files
77              try:
78
79                  if not self.cache or not self.hascache:
80                      result = subprocess.check_output(cmd, stderr=subprocess.STDOUT, shell=True) \\
81                      #, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
82
83                  with open(self.logpath, 'r', errors='ignore') as f:
84                      return self.parse(f)
85
86              except subprocess.CalledProcessError as e:
87
88                  path = '{erros}/{hashname}.log'.format(erros=ERROS, hashname=self.hashname)
89                  with open(path, 'w+') as f:
```

```
90                  line = '{name} {code}\n\n {error}'.format(name=self.name, code=e.returncode, error=e.output)
91                  f.write(line)
92
93              return self.parse(None)
94
95      def parse(self, f):
96          pass
97
98
99  class PeCheck(Check):
100     name = 'pecheck'
101
102     def __init__(self, filepath, cache=True):
103         self.checker = 'python {bin_}/pecheck.py'.format(bin_ = BIN)
104         super().__init__(filepath, cache)
105
106     def parse(self, f):
107         entropy_found = False
108
109         self.parsed.update({
110             'entropy': None,
111             'code_entropy': None,
112             'data_entropy': None
113         })
114
115         if not f:
116             return self.parsed
117
118         for line in f.readlines():
119             if not entropy_found and 'Entropy' in line:
120                 entropy_found = True
121                 self.parsed['entropy'] = line.split(' ')[1]
122
123             if 'CODE entropy' in line:
124                 self.parsed['code_entropy'] = line.split(' ')[2]
125
126             if 'DATA entropy' in line:
127                 self.parsed['data_entropy'] = line.split(' ')[2]
128
129         return self.parsed
130
131 class PeScanner(Check):
132     name = 'pescanner'
133         def __init__(self, filepath, cache=True):
134         # self.checker = 'python {renmux}/pescanner.py'.format(renmux = RENMUX)
135         self.checker = 'pescanner'
136         super().__init__(filepath, cache)
137
138 class PeFrame(Check):
139     name = 'peframe'
140         def __init__(self, filepath, cache=True):
141         self.checker = 'python3 {bin}/peframe/peframe/peframecli.py --json'.format(bin=BIN)
142         super().__init__(filepath, cache)
143
144     def parse(self, f):
145         d1 = self.parse_json(f)
146         d2 = self.parse_text(f)
```

```
147             self.parsed.update(d1)
148             self.parsed.update(d2)
149             return self.parsed
150 
151         def parse_json(self, f):
152             d = json.loads(f.read()) if f else None
153             parsed = {}
154 
155             # parse json file
156             parsed.update({
157                 'hash_md5':  search_dict(d, ['hashes', 'md5']),
158                 'hash_sha1': search_dict(d, ['hashes', 'sha1']),
159                 'filesize':  search_dict(d, ['filesize']),
160                 'filetype':  search_dict(d, ['filetype']),
161                 'packer':    search_dict(d, ['peinfo', 'features', 'packer']),
162                 'antidbg':   search_dict(d, ['peinfo', 'features', 'antidbg']),
163                 'crypto':    search_dict(d, ['peinfo', 'features', 'crypto']),
164                 'sections':  search_dict(d, ['peinfo', 'sections', 'count']),
165                 'imphash':   search_dict(d, ['peinfo', 'imphash']),
166                 'imports':   len_or_none(search_dict(d, ['peinfo', 'directories', 'import'])),
167                 'urls':      search_dict(d, ['strings', 'url']),
168                 'yara':      [],
169                 'imports':   0
170             })
171 
172             if d:
173                 for yp in d['yara_plugins']:
174                     for k, v in yp.items():
175                         parsed['yara'].append(v)
176 
177             # add lens
178             parsed.update({
179                 'packer_len':  len_or_none(parsed['packer']),
180                 'antidbg_len': len_or_none(parsed['antidbg']),
181                 'urls_len':    len_or_none(parsed['urls']),
182                 'yara_len':    len_or_none(parsed['yara']),
183             })
184 
185             return parsed
186                 def parse_text(self, f):
187                 keywords = [
188                 'OpensProcess',
189                 'CreateRemoteThread',
190                 'ReadProcessMemory',
191                 'CreateProcess',
192                 'ShellExecute',
193                 'HttpSendRequest',
194                 'InternetConnect',
195                 'CreateService',
196                 'StartService',
197                 'CreateFile',
198                 'GetProcAddress',
199                 'LoadLibrary',
200                 'DeleteFile',
201                 'URLDownloadToFile',
202                 'GetModuleHandle'
203             ]
```

```
204
205         parsed = {}
206         for k in keywords:
207             parsed.update({k:0})
208
209         if not f: return parsed
210         # else
211         f.seek(0)
212         for line in f.readlines():
213
214             for k in keywords:
215                 if k in line: parsed[k] += 1
216
217         return parsed
218
219 if __name__ == '__main__':
220         # parse arguments
221     parser = ArgumentParser()
222     parser.add_argument('-i', '--input', dest='input', help='Input file or directory')
223     parser.add_argument('-o', '--output', dest='output', help='Output directory')
224     parser.add_argument('-c', '--cache', dest='cache', help='Use cached outputs')
225     parser.add_argument('-l', '--limit', dest='limit', default='0', help='Limit number of files')
226     parser.add_argument('-v', '--verbose', dest='verbose', help='Print progress to STDOUT')
227     args = parser.parse_args()
228     input_ = os.path.abspath(args.input)
229     output = os.path.abspath(args.output)
230     limit = int(args.limit)
231
232     if not os.path.isdir(args.output):
233         raise Exception('OUTPUT must be a directory!')
234
235     inputpaths = []
236     if os.path.isdir(input_):
237         for f in os.listdir(args.input):
238             if not os.path.isdir(f):
239                 inputpaths.append(os.path.join(input_, f))
240     else:
241         inputpaths.append(input_)
242
243
244     # csv - clear file if exists and write headers
245     outputpath_csv = '{path}/0_output.csv'.format(path=output)
246
247     for i, inputpath in enumerate(inputpaths):
248
249         if limit != 0 and i >= limit:
250             break
251
252         print('---\nRunning', i+1, 'of', len(inputpaths), inputpath)
253
254         d = {}
255          pecheck = PeCheck(inputpath)
256         dc = pecheck.run()
257         # print_dict(dc)
258         d.update(dc)
259
260         # pescanner = PeScanner(inputpath)
```

```
261             # d.update(pescanner.run())
262
263             peframe = PeFrame(inputpath)
264             df = peframe.run()
265             # print_dict(df)
266             d.update(df)
267
268             # print('\n---', d)
269
270             # json
271             outputpath = '{path}/{file}.log'.format(path=output, file=pecheck.hashname)
272             with open(outputpath, 'w+') as f:
273                 f.write(json.dumps(d))
274
275             # csv
276             k = list(d.keys())
277             k.sort()
278
279             # write header
280             if i == 0:
281                 with open(outputpath_csv, 'w+') as f:
282                     line = ''
283                     for i in k:
284                         line += '"{}";'.format(str(i).replace('"', "'"))
285                     line += '\n'
286                     f.write(line)
287
288             with open(outputpath_csv, 'a+') as f:
289                 # write lines
290                 line = ''
291                 for i in k:
292                     line += '"{}";'.format(str(d[i]).replace('"', "'"))
293                 line += '\n'
294                 f.write(line)
```

Apêndice B

Ferramenta de Identificação e Separação de Arquivos pelo Tipo

B.1 Separador.py

```
1  ############### Separador.py  ########################3
2  import os
3  import magic
4  import hashlib
5
6  from shutil import copyfile
7  from argparse import ArgumentParser
8
9  from paths import *
10 from config import *
11
12 parser = ArgumentParser()
13 parser.add_argument('-d', '--directory', dest='directory', default=None, required=False, help='Diretorio a separar')
14 args = parser.parse_args()
15
16 if not args.directory:
17     raise Exception('Voce deve passar o argumento --directory ou -d')
18
19 for fname in os.listdir(args.directory):
20     path = os.path.join(args.directory, fname)
21
22     if os.path.isdir(path):
23         print('Pulando diretorio:', path)
24
25     elif os.path.isfile(path):
26         m = magic.Magic(mime=True)
27         mime = m.from_file(path)
28
29         # fazer uma copia
30         mime_slug = mime.replace('/', '_').replace('-', '_')
31         bkp_dir = args.directory.replace(BASE, BKP)
32         bkp_mime_dir = os.path.join(bkp_dir, mime_slug)
```

```
33
34          if not os.path.exists(bkp_mime_dir):
35              print('Criando', bkp_mime_dir)
36              os.makedirs(bkp_mime_dir)
37
38          new_path = os.path.join(bkp_mime_dir, fname)
39          # print('Copiando', path, new_path)
40          copyfile(path, new_path)
41
42          if 'application' in mime:
43              md5 = hashlib.md5(open(path,'rb').read()).hexdigest()
44              # print(mime, md5)
45              os.rename(path, path.replace(fname, md5))
46          else:
47              # print('Deletando', mime, fname)
48              os.remove(path)
49
50
51      else:
52          print('Erro:', path)
```

B.2 Persistência dos Modelos

```
1  ######
2  predicted_scale = clf_scale.predict(nova_base_scale)
3  metric_scale  = get_metrics(predicted_scale, nova_base.classe)
4  print(metric_scale)
5
6  desc = modeloRF para classificar malwares com dados normalizados, foi utilizado a técnica \
7  MinMaxScaler(feature_range=[0, 1]) para normalizacâo dos dados e foram utilizadas todas as colunas
8
9  save_model(clf_scale, description=desc, date='25/03/2020', version='', metrics=metric_scale,
       ↪ name_model='model_rf_scale')
10 -----------------
11 {'ACC': 0.9999051952976867, 'PRECISION': 0.9999479220914488, 'RECALL': 0.9994725738396624, 'FSCORE':
       ↪ 0.9997101074468555}
12 ['model_rf_scale.joblib']
13 ### carrega modelo
14
15 modelo = joblib.load('model_rf_scale.joblib')
```

Listagem B.1: salvando o modelo em um arquivo

Apêndice C

Treinamento dos Modelos - Jupyter Notebook

```
#%string title = "Treinamento dos Modelos "
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import preprocessing
from collections import Counter
import joblib

import warnings
warnings.filterwarnings('ignore')

#%matplotlib inline

pd.set_option('display.max_columns', 500)

## LIBS PARA MACHINE LEARNING
from sklearn.model_selection import  GridSearchCV

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import RandomForestClassifier
from sklearn.tree import DecisionTreeClassifier

from sklearn.model_selection import cross_val_score
from sklearn.metrics import confusion_matrix
from sklearn.pipeline import Pipeline

import sklearn.metrics as metrics
from sklearn.metrics import make_scorer,f1_score,recall_score,accuracy_score,precision_score
import pprint

from sklearn.feature_extraction.text import TfidfVectorizer
```

```
37 from sklearn.feature_extraction.text import CountVectorizer
38
39 from sklearn.decomposition import PCA
40 from sklearn.preprocessing import MinMaxScaler
41 import pickle as pk
42
43 ##### FUNCTIONS #####
44
45 ## função para plotar matriz de confusão
46 def plot_cm(cm, labels):
47
48     # calcula porcentagens
49     percent = (cm*100.0)/np.array(np.matrix(cm.sum(axis=1)).T)
50
51     print('Confusion Matrix Stats')
52     for i, label_i in enumerate(labels):
53         for j, label_j in enumerate(labels):
54             print("%s/%s: %.2f%% (%d/%d)" % (label_i, label_j, (percent[i][j]), cm[i][j], cm[i].sum()) )
55
56     # Show confusion matrix
57     fig = plt.figure()
58     ax = fig.add_subplot(111)
59     ax.grid(b=False)
60     cax = ax.matshow(percent, cmap='coolwarm')
61     plt.title('Confusion matrix of the classifier')
62     fig.colorbar(cax)
63     ax.set_xticklabels([''] + labels)
64     ax.set_yticklabels([''] + labels)
65     plt.xlabel('Predicted')
66     plt.ylabel('True')
67     plt.show()
68
69 ## função para plotar AUC Curve
70 def plot_auc_curve(clf, X_test, y_test):
71     y_pred_proba = clf.predict_proba(X_test)[::,1]
72     fpr, tpr, _  = metrics.roc_curve(y_test,  y_pred_proba)
73     auc          = metrics.roc_auc_score(y_test, y_pred_proba)
74
75     plt.plot(fpr,tpr,label="data 1, auc="+str(auc))
76     plt.legend(loc=4)
77     plt.show()
78
79 ## função para executar modelos de ML
80 def run_ml_gs(models,dataset,target):
81
82     for model in models:
83         indx = models.index(model)
84         grid_search = GridSearchCV(model['model'],
85                     model['param_grid'],
86                     cv=5,
87                     verbose=1,
88                     n_jobs=10,
89                     refit='MMS',
90                     scoring={'RC':'recall',
91                             'AC':'accuracy',
92                             'F1':'f1',
93                             'PC':'precision',
```

```
94                              'MMS':make_scorer(mean_metrics_score, greater_is_better= True)}
95                      )
96
97          grid_search.fit(dataset, target)
98          i = grid_search.cv_results_['params'].index(grid_search.best_params_)
99          models[indx]['all_results'] = grid_search.cv_results_
100         models[indx]['best_results'] = { k:grid_search.cv_results_[k][i] for k in  grid_search.cv_results_.keys()  }
101         models[indx]['best_params'] = grid_search.best_params_
102         models[indx]['best_score'] = grid_search.best_score_
103         models[indx]['best_model'] = grid_search.best_estimator_
104     return models
105
106 ## função para executar modelos de ML
107 def run_ml(modelos, X_train,X_test,y_train,y_test):
108     lista_metrics = list()
109
110     for m in modelos:
111
112         print("\n==>",m.__class__.__name__)
113
114         clf = Pipeline([('clf', m )])
115
116         clf       = clf.fit(np.asarray(X_train), y_train)
117
118         predicted = clf.predict(np.asarray(X_test))
119         print(" Accuracy   = %f \n" % accuracy_score(y_test, predicted))
120
121         labels = list(np.unique(y_train)) #[0,1]
122         cm      = confusion_matrix(y_test, predicted,labels=labels)
123
124         if plotCm:
125             plot_cm(cm,labels)
126
127         if hasattr(clf, 'predict_proba') and plotAuc:
128             plot_auc_curve(clf, X_test, y_test)
129
130         metric = get_metrics(predicted, y_test)
131         metric['modelo'] = m.__class__.__name__
132
133
134         etc_eval     = cross_val_score(estimator = clf, X = X_train, y = y_train, cv = 10)
135         metric["CV"] = etc_eval.mean()
136
137         print(metric)
138
139         lista_metrics.append(metric)
140
141     return lista_metrics
142
143 def get_metrics(predicted, y_test):
144     metric = dict()
145
146     metric["ACC"] = accuracy_score(y_pred=predicted,y_true=y_test)
147
148     metric["PRECISION"] = np.mean(metrics.precision_score(y_pred=predicted,y_true=y_test,average=None))
149     metric["RECALL"]    = np.mean(metrics.recall_score(y_pred=predicted,y_true=y_test,average=None))
150     metric["FSCORE"]    = np.mean(metrics.f1_score(y_pred=predicted,y_true=y_test,average=None,pos_label='H'))
```

```
151
152     return metric
153
154 def mean_metrics_score(y_true, y_pred):
155     f1 = f1_score(y_true, y_pred)
156     r = recall_score(y_true,y_pred)
157     a = accuracy_score(y_true, y_pred)
158     p = precision_score(y_true, y_pred)
159     mms = (f1 + a + r)/3
160     return mms
161
162
163 def clean_data(data, replace_zero = True, value_replace = None):
164     print('Cleaning Data...')
165     ## convertendo coluna cryto para boolean
166     data.crypto = [1 if len(c) > 2 else 0 for c in data.crypto ]
167
168     if replace_zero:
169         print('\t by zero')
170         data = data.replace(to_replace='None', value=0)
171         data = data.replace(to_replace=np.NaN, value=0)
172     else:
173         ## loop para substituir valores None e NaN pela média
174         mean = 0
175
176         ## IF apenas para print do texto correto
177         if value_replace != None:
178             print('\t by specific value')
179         else:
180             print('\t by mean')
181
182         for i,c in data.iteritems():
183             if value_replace != None:
184
185                 mean = value_replace[i]
186             else:
187                 m = data.loc[(data[i] != 'None') & (data[i] != np.NaN),i]
188                 m = m.astype(float)
189                 mean_malw[i] = m.mean()
190                 mean = m.mean()
191
192             data.loc[:,i] = data.loc[:,i].replace(to_replace='None',value=mean)
193             data.loc[:,i] = data.loc[:,i].replace(to_replace=np.NaN,value=mean)
194
195     return data
196
197 def save_model(modelo, description='',date='',version='', metrics='',name_model='modelo'):
198     dic_save_model = dict({
199         'model': modelo,
200         'metadata': {
201             'description': description,
202             'author': 'autor',
203             'date': date,
204             'source_code_version': version,
205             'metrics': metrics
206         }
207     })
```

```
208     print(joblib.dump(dic_save_model, name_model+'.joblib'))
209
210 ## função para eliminar os caracteres
211 def clear_string(column):
212     column = column.replace(regex='\[|\]|\'|\,',value=' ')
213     return column#.replace(to_replace='None',value=' ')
214
215 ## LISTA COM OS ALGORITMOS QUE SERÃO UTILIZADOS
216 modelos = [SVC(kernel='rbf'),
217            RandomForestClassifier(random_state=42),
218            KNeighborsClassifier(),
219            GaussianNB(),
220            DecisionTreeClassifier(random_state=42)]
221
222 ## VARIAVEIS GLOBAIS QUE SÃO UTILIZADAS NAS FUNÇÕES
223 global plotAu, plotCm, feat_import
224 plotAuc = False ## indica se vai plotar a curva AUC (utilizada apenas para classificador binário)
225
226 plotCm  = True ## indica se vai plotar a matriz de confusão
```

C.1 Código do Experimento 1

```
1 ##########  Experimento 1 ################
2
3 ########## # Modelos com Dados Normais
4 colunas = ['filetype', 'packer','urls','yara','antidbg','hash_md5','hash_sha1','imphash','code_entropy','data_entropy']
5 global mean_malw
6 mean_malw = dict()
7
8  ## IMPORTANDO OS DADOS COM OS MALWARES
9 malw = pd.read_csv('Malwares_output.csv')
10
11 ## criando uma cópia dos registros para fazer os tratamentos
12 malw_clean = malw.copy()
13
14 ## excluindo colunas que não serão utilizadas
15 malw_clean = malw_clean.drop(colunas, axis=1)
16 malw_clean = clean_data(malw_clean)
17 malw_clean['classe'] = 1
18
19 normal = pd.read_csv('Normais.csv',sep=',')
20
21 ## criando uma cópia dos registros para fazer os tratamentos
22 normal_clean = normal.copy()
23
24 ## excluindo colunas que não serão utilziadas
25 normal_clean = normal_clean.drop(colunas, axis=1)
26 normal_clean = clean_data(normal_clean,replace_zero=False)
27 normal_clean['classe'] = 0
28
29 ##CRIANDO BASE DE DADOS DE TREINO E TESTE
30
31 ## concatenandos as duas bases em uma só
```

```
32 dados = pd.concat([normal_clean,malw_clean])
33
34 ## convertendo os dados para tipo float
35 dados = dados.applymap(float)
36 dados.classe.value_counts()
37 x = dados.iloc[:,:-1]
38
39 y = dados["classe"]
40
41 X_train, X_test, y_train, y_test = train_test_split(x,y, test_size=0.2, random_state=42)
42
43 ## CARREGANDO AS NOVAS BASES DE DADOS
44 novo_malw       = pd.read_csv('output-fit-2.csv',sep=';')#,nrows=900)
45 novo_malw_clean = novo_malw.drop(columns = ['Unnamed: 34'])# colunas)
46 novo_malw_clean = novo_malw_clean.drop(columns = colunas)
47 novo_malw_clean = clean_data(novo_malw_clean,True)
48 novo_malw_clean['classe'] = 1
49
50 novo_norm = pd.read_csv('novabase.csv',sep=';')
51 novo_norm_clean = novo_norm.drop(columns = ['Unnamed: 34'])# colunas)
52 novo_norm_clean = novo_norm_clean.drop(colunas, axis=1)
53 novo_norm_clean = clean_data(novo_norm_clean, False, mean_malw)
54 novo_norm_clean['classe'] = 0
55
56 nova_base = pd.concat([novo_malw_clean, novo_norm_clean])
57 nova_base = nova_base.reset_index(drop=True)
58 ###################
```

C.2 Código do Experimento 2 - PCA

```
1 # MODELOS COM PCA, CV e GRID SEARCH
2 ## LISTA COM OS ALGORITMOS E SEUS PARÂMETROS QUE SERÃO UTILIZADOS
3 models = [
4 {
5 "name":"SVC",
6 "model":SVC(),
7 "param_grid": {'C':[0.001, 0.01, 0.1, 1, 10],
8                'gamma' : [0.001, 0.01, 0.1, 1],
9                'kernel':['rbf']}
10 },
11 {
12 "name":"Random Forest",
13 "model":RandomForestClassifier(),
14 "param_grid" : {'criterion':['gini','entropy'],
15           'n_estimators':[10,15,20,25,30],
16           'min_samples_leaf':[1,2,3],
17           'min_samples_split':[3,4,5,6,7],
18           'random_state':[123],
19           'n_jobs':[-1]}
20 },
21 {
22 "name":"KNN",
23 "model":KNeighborsClassifier(),
```

```
24 "param_grid" : {'n_neighbors':[5,6,7,8,9,10],
25           'leaf_size':[1,2,3,5],
26           'weights':['uniform', 'distance'],
27           'algorithm':['auto', 'ball_tree','kd_tree','brute'],
28           'n_jobs':[-1],
29           'metric':['minkowski','euclidean','manhattan']}
30
31 },
32 {
33 "name":"Decision Tree",
34  "model":DecisionTreeClassifier(),
35 "param_grid" : {'max_features': ['auto', 'sqrt', 'log2'],
36           'min_samples_split': [2,3,4,5,6,7,8,9,10,11,12,13,14,15],
37           'min_samples_leaf':[1,2,3,4,5,6,7,8,9,10,11],
38           'random_state':[123]}
39 },
40 {
41 "name":"GaussianNB",
42  "model":GaussianNB(),
43  "param_grid":{}
44 }
45 ]
46
47 models_tsne = [
48 {
49 "name":"SVC",
50 "model":SVC(),
51 "param_grid": {'C':[0.001, 0.01, 0.1, 1, 10],
52                'gamma' : [0.001, 0.01, 0.1, 1],
53                'kernel':['rbf']}
54 },
55 {
56 "name":"Random Forest",
57 "model":RandomForestClassifier(),
58 "param_grid" : {'criterion':['gini','entropy'],
59           'n_estimators':[10,15,20,25,30],
60           'min_samples_leaf':[1,2,3],
61           'min_samples_split':[3,4,5,6,7],
62           'random_state':[123],
63           'n_jobs':[-1]}
64 },
65 {
66 "name":"KNN",
67 "model":KNeighborsClassifier(),
68 "param_grid" : {'n_neighbors':[5,6,7,8,9,10],
69           'leaf_size':[1,2,3,5],
70           'weights':['uniform', 'distance'],
71           'algorithm':['auto', 'ball_tree','kd_tree','brute'],
72           'n_jobs':[-1],
73           'metric':['minkowski','euclidean','manhattan']}
74
75 },
76 {
77 "name":"Decision Tree",
78  "model":DecisionTreeClassifier(),
79 "param_grid" : {'max_features': ['auto', 'sqrt', 'log2'],
80           'min_samples_split': [2,3,4,5,6,7,8,9,10,11,12,13,14,15],
```

```
81             'min_samples_leaf':[1,2,3,4,5,6,7,8,9,10,11],
82             'random_state':[123]}
83 },
84 {
85 "name":"GaussianNB",
86  "model":GaussianNB(),
87  "param_grid":{}
88 }
89 ]
90 ###############
91 scaler = MinMaxScaler(feature_range=[0, 1])
92 data_rescaled   = scaler.fit_transform(x)
93 nova_base_scale = scaler.transform(nova_base.iloc[:,:-1])
94
95 pca = PCA(n_components=12)
96
97 dataset       = pca.fit_transform(data_rescaled)
98 nova_base_pca = pca.transform(nova_base_scale)
99 #dataset_tsne =tsne.fit_transform(data_rescaled)
100 pk.dump(pca, open("pca.pkl","wb"))
101
102 df_pca        = pd.DataFrame(dataset,columns=["Col1","Col2","Col3","Col4","Col5","Col6", "Col7","Col8","Col9","Col10,"Col11"
103 nova_base_pca = pd.DataFrame(nova_base_pca,columns=["Col1","Col2","Col3","Col4","Col5","Col6", "Col7","Col8","Col9","Col10,"
104
105 ##Executa grid search
106 results_pca = run_ml_gs(models, df_pca, y)
107 best_clf_pca = max(results_pca, key=lambda x: x['best_score'])
108
109 clf_pca      = best_clf_pca['best_model']
110 clf_pca
111 predicted_pca = clf_pca.predict(nova_base_pca)
112 metric_pca = get_metrics(predicted_nova_base, nova_base.classe)
113
114 desc = 'modelo KNNA para classificar malwares com PCA'
115 save_model(clf_pca, description=desc, date='25/03/2020', version='', metrics=metric_pca, name_model='model_knn_pca')
```

C.3 Experimento 3 -TF-IDF

```
1 ########## Treinamento com TF-IDF #################
2 #malw_clean = malw.copy()
3 #normal_clean = normal_clean.drop(colunas, axis=1)
4 ## aplicando a função para eliminar os caracteres somente nas respectivas colunas
5 malw_clean_texto = malw.loc[:,['antidbg','crypto', 'packer', 'yara']].apply(clear_string)
6
7 ## concatenando as colunas e salvando em uma única coluna
8 malw_clean_texto['text']=malw_clean_texto[['antidbg','crypto','packer', 'yara']].apply(lambda x:' '.join(x), axis=1)
9 malw_clean_texto['text']=malw_clean_texto['text'].str.lower()
10
11 malw_clean_texto['classe'] = 1
12
13 # criando uma cópia dos registros para fazer os tratamentos
14 #normal_clean = normal.copy()
15
```

```
16 ## aplicando a função para eliminar os caracteres somente nas respectivas colunas
17 normal_clean_texto= ormal.loc[:,['antidbg', 'crypto', 'packer', 'yara']].apply(clear_string)
18 ## concatenando as colunas e salvando em uma única coluna
19 normal_clean_texto['text']=normal_clean_texto[['antidbg','crypto','packer','yara']].apply(lambda x:' '.join(x),axis= 1)
20 normal_clean_texto['text']=normal_clean_texto['text'].str.lower()
21
22 normal_clean_texto["classe"] = 0
23 ## concatenandos as duas bases em uma só
24 dados_text = pd.concat([normal_clean_texto.loc[:,['text','classe']],
25                         malw_clean_texto.loc[:,['text','classe']]])
26
27 dados_text = dados_text.reset_index(drop=True)
28
29 print('Tamanho base de malware: ', len(malw))
30 print('Tamanho base de benigmo: ', len(normal))
31 print('Tamanho base final: ', len(dados))
32
33 #######Vetorizando ########
34 vectorizer    = TfidfVectorizer()
35 vectors       = vectorizer.fit_transform(dados_text.text).toarray()
36 feature_names = vectorizer.get_feature_names()
37 dados_idf     = pd.DataFrame(vectors, columns=feature_names)
38
39 ## criando uma cópia dos registros para fazer os tratamentos
40 #novo_malw_clean = malw.copy()
41
42 ## aplicando a função para eliminar os caracteres somente nas respectivas colunas
43 novo_malw_clean_text= novo_malw.loc[:,['antidbg', 'crypto', 'packer', 'yara']].apply(clear_string)
44 ## concatenando as colunas e salvando em uma única coluna
45 novo_malw_clean_text['text']=
46 novo_malw_clean_text[['antidbg','crypto','packer','yara']].apply(lambda x:''.join(x),axis = 1)
47 novo_malw_clean_text['text']=novo_malw_clean_text['text'].str.lower()
48
49 novo_malw_clean_text["classe"] = 1
50
51 ## criando uma cópia dos registros para fazer os tratamentos
52 #novo_malw_clean = malw.copy()
53
54 ## aplicando a função para eliminar os caracteres somente nas respectivas colunas
55 novo_norm_clean_text=novo_norm.loc[:,['antidbg', 'crypto', 'packer', 'yara']].apply(clear_string)
56 ## concatenando as colunas e salvando em uma única coluna
57 novo_norm_clean_text['text']=
58 novo_norm_clean_text[['antidbg','crypto','packer','yara']].apply(lambda x:''.join(x), axis = 1)
59 novo_norm_clean_text['text'] = novo_norm_clean_text['text'].str.lower()
60
61 novo_norm_clean_text["classe"] = 0
62
63 nova_base_texto = pd.concat([novo_malw_clean_text, novo_norm_clean_text])
64 nova_base_texto = nova_base_texto.reset_index(drop=True)
65
66 ## aplicando a função para eliminar os caracteres somente nas respectivas colunas
67 nova_base_texto.loc[:,['antidbg', 'crypto', 'packer', 'yara']] = \
68     nova_base_texto.loc[:,['antidbg', 'crypto', 'packer', 'yara']].apply(clear_string)
69
70 nova_base_texto['text']=nova_base_texto[['antidbg','crypto','packer','yara']].apply(lambda x:'.join(x),axis=1)
71 ## concatenando as colunas e salvando em uma única coluna
72 vectors          = vectorizer.transform(nova_base_texto.text)
```

```
73 feature_names     = vectorizer.get_feature_names()
74
75 denselist = vectors.toarray()
76 novo_idf  = pd.DataFrame(denselist, columns=feature_names)
77
78 novo_idf['classe']  = nova_base.classe
79
80 ###Executando o modelos com CV, 10 K-fold.
81 metricas_tf = run_ml(modelos, X_train=dados_idf, X_test=novo_idf.iloc[:,:-1],
82                      y_train=dados_text.classe,
83                      y_test=novo_idf.classe)
84
85 clf_tf = RandomForestClassifier(random_state=42)
86 clf_tf.fit(dados_idf , dados_text.classe)
87
88 predicted_nova_base_text = clf_tf.predict(novo_idf.iloc[:,:-1])
89 metric_tf = get_metrics(predicted_nova_base_text, novo_idf.classe)
90
91 desc = 'RandomForest para classificar malwares com TF-IDF com as colunas antidbg,crypto,packer,yara'
92
93 ###### Salvando o melhor modelo em arquivo
94 save_model(clf_tf, description=desc,date='25/10/2020',version='',metrics=metric_tf, name_model='model_rf_tf')
```